II. ORSZÁGOS OBJEKTUMORIENTÁLT KONFERENCIA

Objektumorientált technológia

Microsoft

ORACLE

NETSCAPE

EUROPEAN INSTITUTE
FOR RESEARCH AND STRATEGIC STUDIES
IN TELECOMMUNICATIONS GMBH

1997. OKTÓBER 28-29.

VISEGRÁD

ITA 492

II. ORSZÁGOS OBJEKTUMORIENTÁLT KONFERENCIA

Objektumorientált technológia

NOKIA

Microsoft

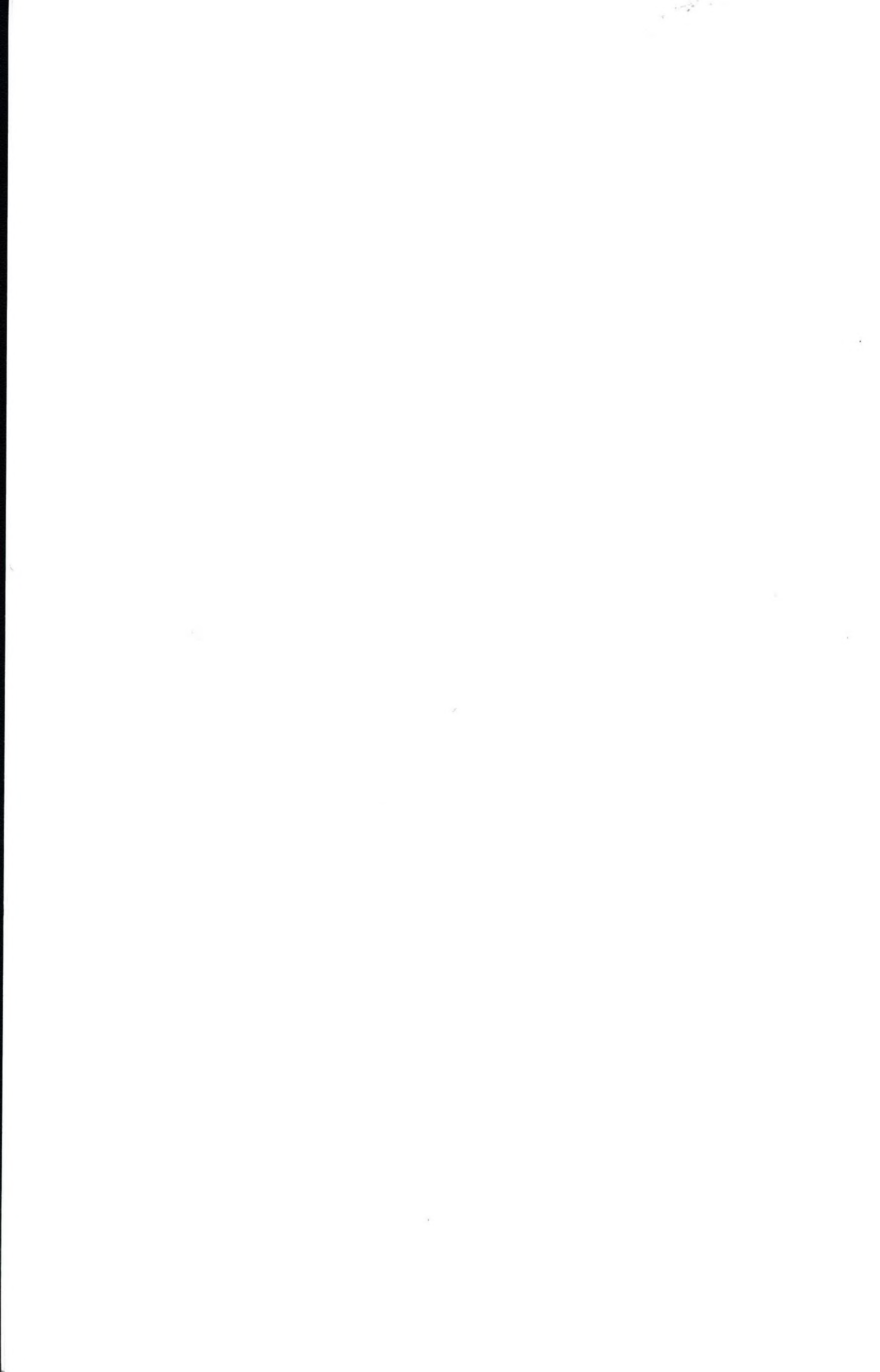
ORACLE

NETSCAPE

EUROPEAN INSTITUTE FOR RESEARCH AND STRATEGIC STUDIES IN TELECOMMUNICATIONS GMBH

1997. OKTÓBER 28-29.

VISEGRÁD



Tartalomjegyzék

1. **Sdrjan Kovacevic**, AONIX - Scaleable a lifecycles for Object-Oriented Development 7.

2. **Aleksandar Ivankovic** - A SUN objektumorientált technológiája: hálózati objektumok 9.

3. **Paul Donnelly**, IONA Technologies - Virtual Homogenity-So Many Pieces, So Little Glue 11.

4. **Dr. Raffai Mária**, Széchenyi István Főiskola - OOA&D versus SA/SD avagy objektumorientált szemlélet a fejlesztés teljes életciklusában 13.

5. **Boér Tamás**, Andersen Consulting - Objektum-orientált stratégiák 23.

6. **Porkoláb Zoltán**, X-Close Kft. - C++Standard Template Library 31.

7. **Vég Csaba**, KLTE - Az OO szemlélet architektúrális támogatása 39.

8. **Frigó József, Kelen András, Szomolányi Márton** TRIAD - Tervezés, implementálás és rendszerfelügyelet, Software Trough Pictures 53.

9. **Egyed László -Nick János**, IQSOFT - Elektronikus áruház az Interneten 55.

10. **Tóthné Bokor Judit**, MNB - **Németh Miklós**, IQSOFT SzIBilla az MNB statisztikai-célú Információ Bázisa 57.

11. **Németh Miklós** IQSOFT - Korszerű WEB technológiák 61.

12. **Nick János** IQSOFT - ObjectStore ODBMS Internet/Intranet alkalmazásokban 63.

13. **Szalontai Zoltán** - Unisoftware - Komponens alapú fejlesztés Uniface7 Open 4GL fejlesztőeszközzel 65.

14. **Nyikes Tamás**, IBM - IBM Component Broker 71.

15. **Frigó József, Kelen András, Szomolányi Márton** -TRIAD Elosztott OO rendszerek tervezése, implementálása és felügyelete 79.

16. **Huba Zoltán** Hungária Számitástechn.Kft., **Kelen A.-Szomolányi Márton** TRIAD Vállalati modellezés és BPR a Rummler-Brache módszertannal 81.

17. **Frigó József, Kelen András, Szomolányi Márton** -TRIAD
Nagy OO rendszerek kooperativ tervezése 87.

18. **Gajdics György** - Megatrend Kft. - Osztálykönyvtárra alapozott alkalmazásfejlesztés a Megatrend Kft. keretében 89.

19. **Sebestyén Zsolt, Takáts Tamás** - AXIS Kft. - Elosztott OO technológiák támogatása tranzakció szerverekkel 97.

20. **Porkoláb Zoltán** - ELTE - Object Oriented Techniques Implementing Distributed Corba Objects in Java 103.

21. **Firnága László** - IQSOFT - WEB alkalmazások fejlesztése 113.

22. **Tihanyi Péter** - IQSOFT - Workflow és az Internet 115.

23. **Pokó István** AXIS Kft. - A komponens alapú architektúra 4GL szemszögből 117.

24. **Klotz Tamás** - Oracle Hungary - OO technológiák az Oracle stratégiai fejlesztéseiben 125.

25. **Balogh Kálmán** Informix - Az Informix objektum-relációs architektúrája 135.

26. **Lukovics László, Szabó József** - MOL Rt. - Adatbázistervezési talapasztalatok. a MOL RT. Főnix projektjében 137.

27. **Gáspár András** - Harang Bt. - Vasúti menetrendek újraütemezése szimulációs úton II. Folyamatorientált szimulációs ütemezés 143.

Kelen Andras

Srdjan Kovacevic, Aonix:

TITLE

Scaleable Lifecycles for Object-Oriented Development

ABSTRACT

One of the problems in software development was, and still is, that we try to use standard software processes/lifecycles for a variety of projects. A lifecycle is defined in terms of four major interrelated components: methods, processes, tools and organization models, and each component has to be selected in accordance with a specific project and its scope and goals. For instance, an OO lifecycle that defines requirements, analysis, design, implementation, testing and maintenance phases might be appropriate for a large, traditional project; but it is definitely not appropriate to force-fit every software project into this heavy lifecycle. For example, a small OO project using an OO 4GL like Visual Basic still has to keep track of requirements and an overall domain architecture, but a detailed objects model is usually not necessary. So a lighter lifecycle can be used.

What is needed is a mix-and-match framework of different lifecycles where we choose the relevant activities and build a process that fits the problem we are trying to solve. With this approach, we focus our attention around individual activities and deliverables/results and not around a traditional phased approach. By carefully selecting lifecycle activities depending on a given problem, OO technology can deliver promissed productivity and quality improvements to software development.

A SUN objektumorientált technológiája: hálózati objektumok

Aleksandar Ivankovic

A SUN NEO technológiája biztosítja az objektumok együttműködését a heterogén elosztott számítástechnikai környezetekben. Az objektumtechnológia nagy jövője a szoftverek "megkomponálásában" rejlik, azaz abban, hogy a szoftverrendszerek jól ismert komponensekből állíthatók össze. A SUN NEO környzete az alábbi fő elemekből áll:

*** Solaris NEO**

A Solaris Neo 2.0 a SUN elosztott objektum környezete, amely lehetővé teszi a vállalatok számára az üzletmenet szempontjából létfontosságú, közösen használt szoltáltatások elérését egy vállalati Web-en keresztül. A létrehozatalt követően, ezen közösen használt szolgáltatások a vállalaton belül, illetve az Interneten keresztül a legkülönbözőbb kliens platformokról elérhetővé válnak.

*** Joe 2.0**

A Joe 2.0 a normál, Web böngészőkön futó Java programokat összekapcsolja a nagy teljesítményigényű, robosztus üzleti alkalmazásokkal a vállalati Intraneten és az Interneten keresztül. A Joe olyan modell, amelynek segítségével a különböző szolgáltatások eljuttathatók az alkalmazottak, illetve a vállalati ügyfelek részére. Mindemellett rugalmas környzetet biztosít a gyorsan változó és fejlődő vállalati igények számára.

*** Java IDL 1.1**

A Java IDL, a SUN 100 %-os Pure Java Object Request Broker rendszere biztosítja azt a szoftver alapot, amely révén a vállalati kliens/szerver alkalmazások továbbíthatók az Internetre. Azon alkalmazások, amelyek a Java IDL-t használják, zökkenőmentesen integrálhatók más szoftverrendszerek nem Javaban írt komponenseihez. A Java IDL rendszer a Corba 2.0-n és az IIOP ipari szabványon alapul.

Java alapú menedzsment eszköz, amely segítségével adminisztrálni, kezelni lehet az elosztott objektum rendszert.

Virtual Homogenity – So Many Pieces, So Little Glue

Paul Donnelly - Senior product manager, IONA Technologies

Two contrasting trends dominate much of software development today. The first is the need to enable new styles of application construction. These applications are network oriented with an emphasis on keeping the logic in one place. They are described as three-tier or n-tier applications. They offer a replacement for the traditional client-server view. These applications make extensive use of emerging internet technologies such as Java and CORBA. The other trend is the desire, based on sound business principles, to preserve exisiting investment by reusing, integrating and wrappering and extending installed, working software solutions.

The key to enabling new applications while simultaneously preserving investment lies in the provision of a software infrastructure that creates the illusion of a virtual homogenity over an underlying heterogenous reality. The technical basis for this illusion is Component Technology and the various infrastructures to glue these components together.

This presentation gives a view of where Component Technology is today, be it Java or ActiveX and looks at the underlying techniques to glue these componentes together, be it CORBA or DCOM. Then looking to the future, we can start to see how the various pieces will come together to reshape the application landscape.

OOA&D versus SA/SD

avagy objektumorientált szemlélet a fejlesztés teljes életciklusában

dr. Raffai Mária
E-Mail: raffai@rs1.szif.hu
Széchenyi István Főiskola, Informatikai és Villamosmérnöki Fakultás
Informatika Tanszék
✉ 9026 Győr, Hédervári út 3.

OOA&D versus SA/SD

Object-Oriented Paradigm in the Hole Life Cycle

Abstract

In the 50 year's history of computer science the effective new principles were first applied through several programming languages, as they are the most mature areas for challenge. After the introduction of these new languages developers needed methods and tools for software analysis and design that could reflect the capabilities of these new languages. Today the OO programming activity is already in application, but the system designers doesn't use this new concept in the hole life cycle. The projects work with hybrid processes, where the traditional, structured and the object oriented methods and principles are mixed in the three phases of the software life cycle. This leads to lots of problems, because the conversion between the different methods suffers losses of information. In my presentation I am dealing with these problems, and I point out the best approach for optimizing the development activity. The only paradigm which is able to assure better productivity, better management of user requirements, and better reuse of analysis/design results and source code is the unified OO principle in the hole life cycle.

Az informatika fél évszázados történetében a különböző új fejlesztési módszerek iránti igény és ezek bevezetése először mindig a programozás területén jelentkezett. A programtervezésben kipróbált és bevált módszereket később átültették a szoftverfejlesztési életciklus teljes fázisára is, így biztosítva egy, az egész folyamatban egységes szemléletet.

A 70-es évek elején új, strukturált programozási technikák és nyelvek jelentek meg, amelyekből kizárták a bonyolult, áttekinthetetlen programokat eredményező **go to** utasítást, és amelyekkel biztosították a programok moduláris felépítését. Ezek a programnyelvek, mint például a N.WIRTH által kifejlesztett Pascal is, egy újfajta szemléletet honosítottak meg, számos előnnyel rendelkeznek, többek között egyszerű és gyors programmódosítást tesznek lehetővé, és tiszta, világos, adat-, és funkciószerkezetben való gondolkodást követelnek meg a programozótól.

Az elemzési/tervezési munkát végző rendszerszervezők hamar átlátták, hogy fel kell oldani azt az ellentmondást, ami a fejlesztési ciklus egyes fázisainak végrehajtásában alkalmazott eltérő módszerek miatt mutatkozik. A strukturált programnyelvek megjelenése után a strukturált elemzési/tervezési módszerekre vonatkozóan is hamarosan napvilágot láttak COAD, YOURDON, GANE és SARSON, DEMARCO és mások munkái, az első publikációk. Azt mondhatjuk, hogy ezek a sikeres módszertani ajánlások szinte napjainkig fennmaradt és alkalmazott technikákká váltak.

A fejlődés azonban nem állt meg! A grafikus elemek kezelését is lehetővé tevő, egyre nagyobb teljesítményű PC-k megjelenésével a programtervezők megvalósíthatónak látták a 60-as évek végén, az objektum-kezeléssel kapcsolatban felmerült gondolatot. A 70-es évek elején kifejlesztett, OO kiterjesztésű LISP, vagy az objektumorientált Smalltalk az első, sikeres kezdeményezés volt, ám a forradalmi gondolat csupán a 80-as évek második felében tudott meghonosodni az alkalmazásokban.

Az új elképzelés szerint a programtervezők komplexebb elemekben gondolkodnak, olyanokban, amelyek az adatszerkezeteken és az ezekkel végezhető műveleteken túl egységesen képesek kezelni a valóság objektumait, a köztük fennálló kapcsolatokat pedig sokkal magasabb szinten valósítják meg. Az egymás tulajdonságait öröklő, többszörösen felhasználható, egymással kommunikáló objektumok sokkal inkább hasonlítanak a valós világra, mint a J.D.WARNIER, az M.JACKSON féle adatszerkezet-lebontáson, vagy a CONSTANTINE-MEYERS nevével fémjelzett funkciólebontáson alapuló tervezés komponensei.

A *hibrid*, majd a *tiszta OO programnyelvek* megjelenése és gyors terjedése óriási lökést adott az objektumtechnológia széleskörű alkalmazásának, és csupán néhány évre volt szükség ahhoz, hogy bebizonyosodjon, az elemzési/tervezési munkában is szükség van az újfajta gondolkodásmód bevezetésére.

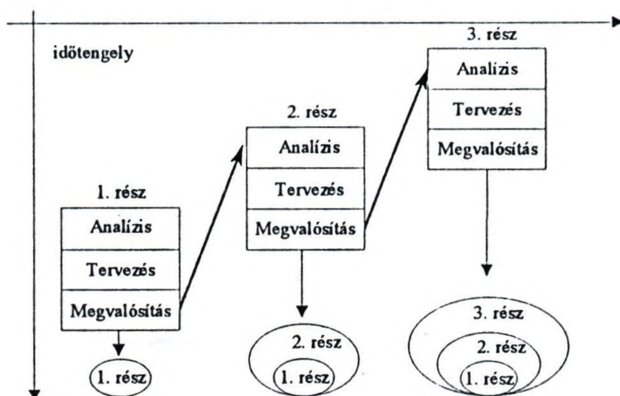
Objektumorientált elemzési/tervezési módszerek		
A módszer	**Fejlesztők**	**Év**
Lorensen	W.Lorensen	1986
OSMOSYS	Winter Partners	1988
RDD/CRC	Bech	1989
HOOD	Hood	1989
OORASS	Reenskang	1989
OOSD	Wasserman	1990
OOA&D	P.Coad, E.Yourdon	1990
Booch	G.Booch	1991
OMT	J.Rumbaugh	1991
OOSE	I.Jacobson	1992
OOIE	J.Martin és J. Odell	1992
UML	G.Booch, I.Jacobson, J.Rumbaugh	1997

Az egyre gyorsabban terjedő OO implementációs módszerek és programnyelvek, az objektumorientált alkalmazásfejlesztő 4G nyelvek és CASE eszközök mellett egyre nagyobb igény jelentkezett a fejlesztés teljes életciklusát egységesen kezelő OO elemzési és tervezési módszerek kidolgozására és alkalmazására. Évtizedünk első felében több, a gyakorlati projektekben eredményesen használt módszer vált ismertté [1]-[6], [12].

1. A fejlesztés klasszikus életciklusa

Egy rendszerfejlesztési projekt megvalósítási folyamatát, mint tudjuk a fejlesztés életciklus modellje írja le, amely valójában a feladatokat, azok egymásutániságát, a fejlesztési tevékenység iteratív módon történő finomítását jelenti. A cél egy olyan cselekvés-sor meghatározása, amellyel minimálisra csökkenthető a fejlesztés kockázata, a szoftverrel szembeni elvárások és az eredmény közötti különbség.

A legtöbb modell a fejlesztési folyamatot fázisokra bontja. Az egyik legáltalánosabban ismert és alkalmazott modell az ún. *vízesés modell*, amely egymástól jól elkülöníthető, egymást lineárisan követő diszkrét tevékenységekből áll. Ebben a folyamatban egy tevékenység csak akkor kezdődhet el, ha az előtte lévőt már befejezték. Vannak azonban olyan modellek is, amelyek a fejlesztési lépéseket nem szigorúan kötött sorrendben végzik, mint például az *evolúciós*, az *inkrementális fejlesztés* vagy a *spirál modell.*



Ábra 1. Az inkrementális fejlesztési modell

Ismeretes a három alapvető fázis: (1) *analízis*, (2) *tervezés* és az (3) *implementáció*, amelyek a különböző modellek esetében további finomított lépésekből állnak. A fázisok feladatai egymástól meglehetősen eltérő tevékenységeket jelentenek, amelyeket a fejlesztési projekt által meghatározott *paradigmák* szerint, a kiválasztott *módszerekkel* és *technikákkal* végeznek [9], [10]. Tudnunk kell azonban, hogy az elemzési, tervezési és programozási/tesztelési feladatokhoz más és más módszerek és technikák állnak rendelkezésre, amelyeket azonban célszerű úgy megválasztani, hogy az a fejlesztendő szoftver termék minősége szempontjából a lehető leghatékonyabb legyen.

<u>Számtalan kérdés merül fel:</u>

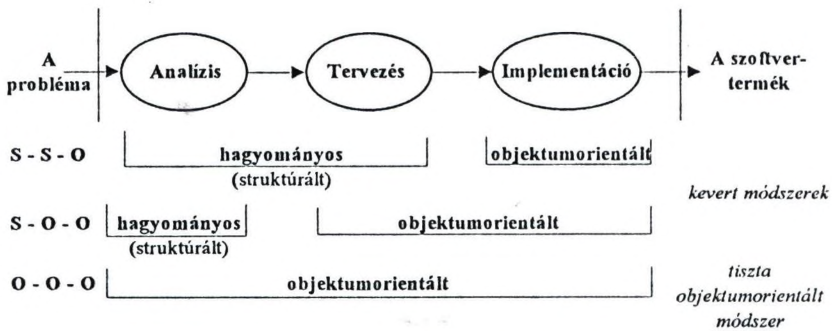
? Milyen módszereket alkalmazzunk a fejlesztési munka során?
? Milyen különbségek vannak a strukturált és objektumorientált elemzési/tervezési módszerekben?
? Milyen problémák merülnek fel, ha nem egységes szemléletben fejlesztünk?
? Javasolt-e, szükséges-e a teljes életciklusban objektumorientált szemlélet szerint dolgozni? stb.
? Mennyiben befolyásolja a szoftver minőségét a különböző módszerek alkalmazásából adódó konverzió? stb.

Mivel úgy látom, hogy még nem általános az objektumorientált elemzési/tervezési módszerek használata, az egységes OO szemlélet, sőt, sokan vitatják is ennek előnyeit a strukturálttal szemben ezért előadásomban éppen ezekre, a módszerek kevert alkalmazásából adódó problémákra szeretnék rávilágítani!

2. A rendszerfejlesztés hibrid módszerei

A napjainkra már szinte általánossá vált objektumorientált programozási munkát az esetek többségében még hagyományos elemzési/tervezési módszerek előzik meg. Jóllehet egyre szélesedik azoknak a fejlesztőknek a köre, akik az üzleti problémák megoldását szolgáló szoftver kifejlesztését objektumorientált szemléletben végzik, mégis egyenlőre sok olyan elemző/tervező munkatárs van, aki a szoftverfejlesztésben *hibrid megoldásokat* alkalmaz. A különböző módszerekkel készített eredmények azonban csak akkor használhatók a következő fázis bemeneteként, ha azt megfelelő módon átalakítjuk, a másik módszer számára érthetővé tesszük. Ez azonban időt vesz igénybe, és információvesztéshez vezet. Nem is szólva arról a hátrányról, ami a korszerűbb metodika lehetőségeinek a hagyományos korlátja miatti kihasználatlanságához vezet. Azt tehát egyértelműen megállapíthatjuk, hogy az a módszer lehet a leghatékonyabb, amely *a teljes életciklusban egységes szemléletet* valósít meg.

A különböző fázisokban a strukturált és objektumorientált módszereket elvileg 8 féleképpen variálhatjuk [13], ezek közül azonban csak a legjellemzőbb lehetséges előfordulások elemzésével foglalkozom. Az 1. sz. ábra a leggyakoribb változatokat szemlélteti.



Ábra 2. A probléma megoldása hagyományos és objektumorientált módszereket alkalmazva

Már az 1. ábrából is láthatjuk, hogyan keverhetők az egyes módszerek, egy másik aspektus (lásd 3. sz. ábra) azonban azt is képes szemléltetni, milyen útvonalak járhatók be a fázisok között, sőt, hogy milyen technológiák kapcsolhatók egymáshoz.

2.1. S -S -O fejlesztés
Strukturált elemzés/tervezés, objektumorientált megvalósítás

A fejlesztésnek ez a változata csupán az implementációhoz, a programozási munkához használ tiszta vagy objektumorientált technológiákat, az elemzési és tervezési munkát strukturáltan végzi.

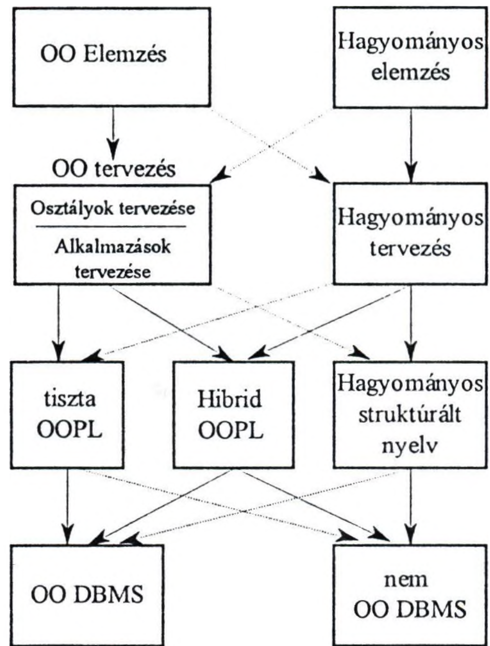
Az a fejlesztési projekt, amelyben *tiszta objektumorientált programozási nyelveket* alkalmaznak, így például Eiffel-t vagy Smalltalk-ot valójában megköveteli, hogy a tervezést is hasonlóan, objektumorientált szemléletben végezzük el. Strukturált tervezés esetében ugyanis szükségünk van a strukturált szerkezetnek és elemeknek objektumorientált komponensekké történő átalakítására.

A *hibrid nyelvek*, mint például a C++ használata jól illeszkedik a struktúra szerkezetekhez, azokat kiválóan implementálja, azonban az eredmény nem lesz objektumorientált. Használatuk mégis javasolt, hiszen a procedurális programkódok így könnyebben alakíthatók át OO kódokká, és így a program rugalmasabban illeszkedik a meglévő, sok szempontból objektumorientált hardver-szoftver környezethez.

2.2. S -O -O fejlesztés Strukturált elemzés, OO tervezés és megvalósítás

Felismerve a tervezési és programozási módszerek különbözőségének problémáját egyre gyakoribbá válik, hogy a fejlesztők a tervezési munkát is objektumorientált módon végzik. Marad az elemzési tevékenység, amelynél sok fejlesztő még ma is azt gondolja, hogy nem lényeges, milyen módszert alkalmaz. Mivel azonban az egyes fázisok eredménye a következő fázis inputja, ezért az elemzési eredményeket is alá kell vetni egy átalakító procedúrának kell alávetnünk annak érdekében, hogy a strukturált elemekből objektumokat kell generálni. Ez azonban nem egyszerű feladat, számtalan probléma merül fel.

A hagyományos, strukturált elemzési eredménye többek között az (1) *egyed-kapcsolat diagram*, valamint az (2) *adatfolyam-diagram*. Ezzel szemben az objektumorientált elemzési/tervezési módszerek alkalmazásakor osztályokat definiálunk, osztályhierarchia és tranzakció-elemző diagramokat készítünk. Ha most egy strukturált egyedből akarunk osztályt konvertálni, akkor egy sor tulajdonságot, jellemzőt, vagy éppen a kapcsolat módját nem tudjuk meghatározni. Ennek alapvető oka, hogy az elemzés során nem erre koncentrálunk, az absztrakció során nem objektumokat definiálunk, hanem csak attribútumokkal leírt egyedeket.



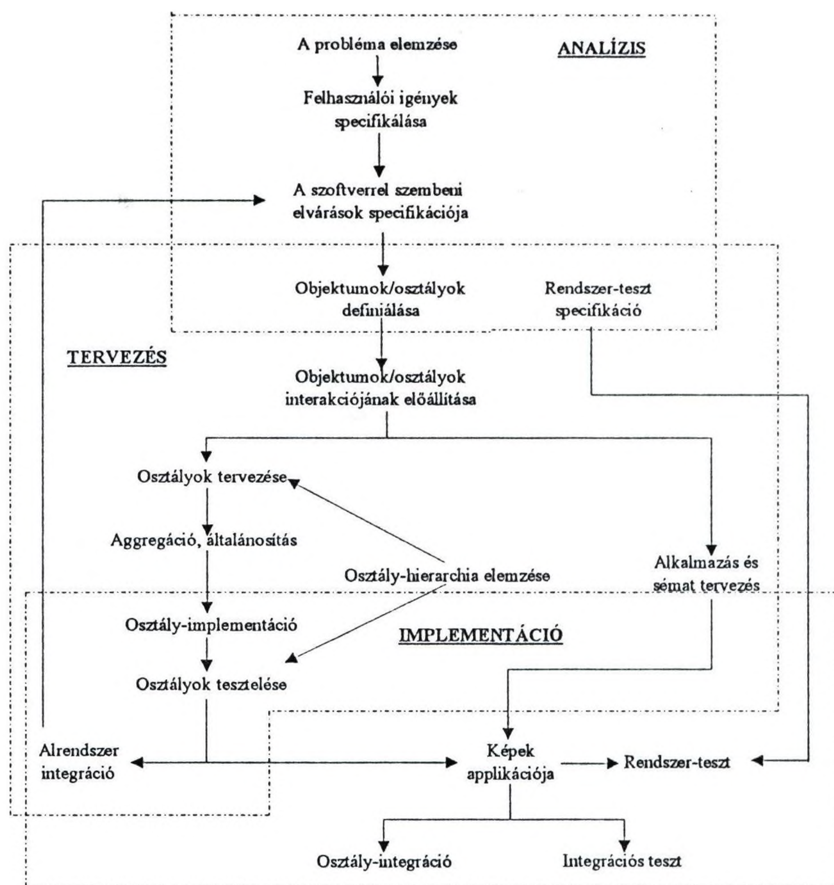
Ábra 3.
Objektumorientált versus strukturált technikák

Az osztálydiagram megmutatja az osztályok közötti aggregációs kapcsolatot, az öröklődést, szemléltetni tudja a hármas kapcsolatot, de arra is alkalmas, hogy ábrázolja, milyen funkció szerint kapcsolódik az egyik osztály a másikhoz. Ezeket a jellemzőket az ER modellből nem tudjuk meghatározni, hiszen valójában az osztályt abból az egyedből vezetjük le, amelyre vonatkozóan nem ismerjük a hozzárendelhető műveleteket.

Többen foglalkoztak a különböző tervezési módok outpujainak transzformálásával, így például ALABISO (1989), WASSERMAN (1990) vagy WARD, aki az ER modellből próbálta levezetni az osztályok közötti öröklődési kapcsolatokat. Ezek a kísérletek azonban nem vezettek az elvárt eredményre.

3. Objektumorientált fejlesztés előnyei a strukturálttal szemben

Az objektumorientált fejlesztési életciklus, mint azt a 4. sz. ábra is mutatja eltér a hagyományostól, bár alapvetően ugyanúgy három fázist tartalmaz: (1) *analízis*, (2) *tervezés* és (3) *implementáció*, de jól látható, hogy vannak olyan feladatok, amelyek fázisba sorolása nem egyértelmű. Ha a hagyományos életciklus modelleket tekintjük, akkor a fejlesztés fokozatos finomítása, iterativitása miatt az objektumorientált fejlesztés életciklusát. leginkább talán az inkrementális modellhez hasonlíthatjuk

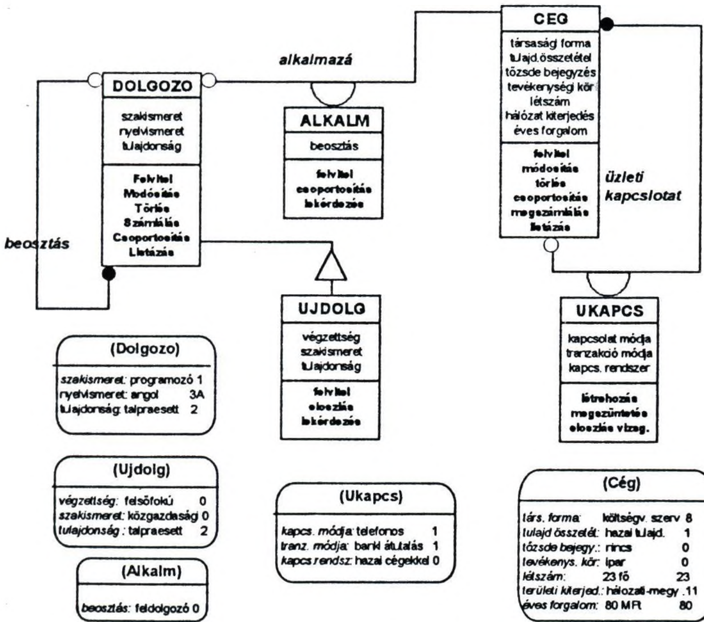


Ábra 4. Az objektumorientált fejlesztési modell

Mint tudjuk, az információrendszerek strukturált fejlesztése során a logikai tervezésnél élesen különválik az adatok és folyamatok tervezési szakasza. Az objektumorientált szemléletben ez valójában egységesen kezelhető, hiszen az objektumokhoz attribútumokat és műveleteket egyaránt hozzárendelünk. Az objektumorientált elv alkalmazásának legfontosabb előnye, hogy a fejlesztési ciklus minden fázisában ugyanazt az objektum modellt használjuk. Jóllehet több olyan közös technika létezik, amely a különböző fejlesztési fázisokban egyaránt alkalmazható, mégis különbség van az egyes fázisok között. Míg az elemzési fázisban egy szervezeten belüli üzleti folyamatok, szerepek és szabályok legmagasabb szintű objektum-modelljével foglalkozunk, addig a tervezési fázisban ezek finomítására kerül sor, olyan újabb objektumok bevezetésére, amelyek a felhasználói kommunikációt (dialógusok, outputok), a tranzakciókat valósítják meg. Az ilyen objektumok és alkalmazások közvetlen módon implementálhatók OOP nyelvekkel.

3.1. Elemzési fázis

Az objektum modell technikái lehetővé teszik, hogy a valóságnak egy a korábbinál sokkal mélyebb absztrakcióját végezzük el, konzisztensebb modelljét hozzuk létre. Az objektum-osztályok, amelyeket attribútumaikkal és viselkedésükkel együtt kezelünk sokkal közelebb állnak az emberi gondolkodáshoz, a dolgokról alkotott képhez, mint az adat- és funkcióstruktúrák. Ha megvizsgáljuk az 5. sz. ábrán szemléltetett objektum modell-részletet, akkor láthatjuk, hogy az instanciákkal kiegészített objektumkapcsolati diagram lényegesen több információt nyújt a cégeknél dolgozó alkalmazottakról, mint egy egyed-kapcsolat diagram [11].



Ábra 5. Objektum-modell instanciákkal

A létrehozott osztályok és ezek hierarchiája biztosítja, hogy egy későbbi időpontban, amikor a rendszer valamelyik pontján változás következik be, újabb felhasználói igény jelentkezik, vagy más alrendszert fejlesztünk, akkor ezek a meglévő objektumok és a hierarchiában alatta lévő, a felsőbb osztályok attribútumait öröklő osztályok egyszerűen használhatók minden tulajdonságukkal együtt már alkalmazásokhoz is, vagyis *újra felhasználhatók*. És végül a GUI környezet segítségével könnyen lehetővé válik a felhasználói interfészek tervezése és szükség szerinti módosítása is.

3.2. Tervezési fázis

Minden fejlesztési módszer és munka célja olyan tervet készíteni, amelynek eredménye egy, a problémát és annak környezetét reálisan leíró, egyértelműen specifikált modell, amely könnyen implementálható és a későbbiekben egyszerűen karbantartható. Bár eddig három alapvető életciklus fázisról beszéltünk a tiszta OO szemlélet alkalmazásánál mégis látnunk kell, hogy nem lehet olyan éles határt húzni az elemzési és tervezési munka fázisai közé, mint a strukturált fejlesztésnél. Már az elemzés során osztályokat specifikálunk, ami valójában tervezési feladat, ugyanakkor a tervezési fázisban létrehozott osztály-hierarchiát további elemzések után újabbakkal bővítjük. Ezzel valójában az inkrementális fejlesztési jelleg domborodik ki, a rendszer részekből való felépítése, a modell fokozatos finomítása, az *iteráció*.

A tervezési fázisban a strukturált módszereknél is nagyobb hangsúlyt kap a *modularitás* valamint a rendszer moduljainak egymástól való függősége. Az objektum-modellnél a modulkapcsolatok alapvetően kétféleképpen realizálódnak az osztályok között: (1) egy osztály magában foglal egy másikat, vagy (2) két osztály egy interfésszel kapcsolódik egymáshoz, vagyis egy objektumra mutató pointer paramétere lehet egy másik objektumot hívó eljárásnak.

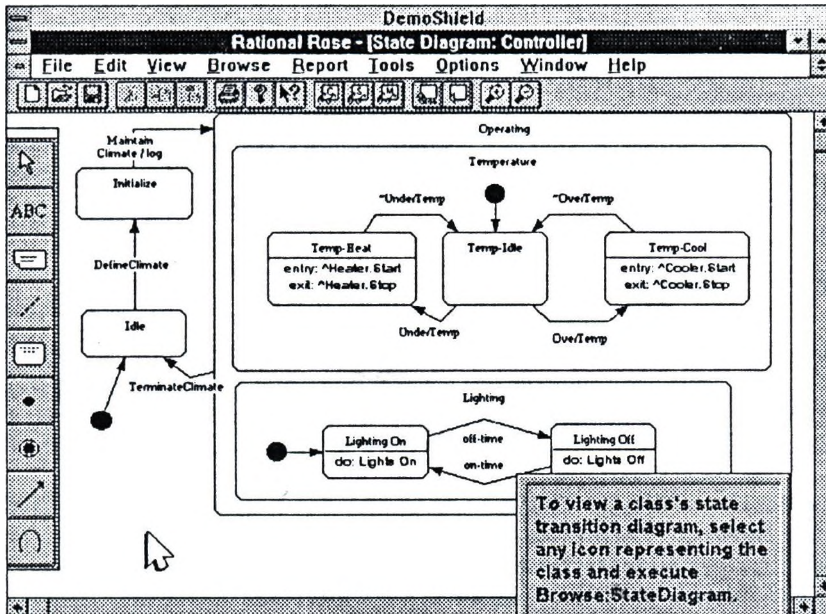
Fontos megemlíteni a másik nagyon fontos jellemzőt, a *kohéziót*. Az OO tervezés nagy hangsúlyt helyez a rendszerelemek belső kötésének szorosságát meghatározó kohézióra, és a szekvenciális kötődés mellett a legerősebb kohéziót, a *funkcionális összetartozást* preferálja.

4. Számítógép a fejlesztésben - CASE eszközök

A elemzés/tervezés feladatait, a fejlesztési munka eredményét a verbális leírás mellett különböző ábrázolási technikákkal szemléltetjük, amelyek könnyen áttekinthetőek, és rengeteg információt hordoznak. Vannak azonban olyan problémák, amelyek csak verbálisan kezelhetők. Az OO fejlesztésnél bevezettek új koncepciókat (polimorfizmus, dinamikus kötés), amelyek szimbolikus ábrázolása még nehezebb. Szükség van tehát olyan eszközökre, amelyek hatékonyan támogatják az elemzés/tervezés feladatait, kezelik és nyilvántartják a fejlesztési eredményeket, irányítják a projekt munkát. Sajnos azonban nem minden OOA&D módszerhez áll rendelkezésre megfelelő CASE eszköz, vagy olyan technikák segítik az elemzési/tervezési munkát, amelyek eredetileg strukturált tervezéshez készültek.

Szerencsés, ha rendelkezésünkre áll egy CASE eszköz, amely oly módon segít a teljes életciklusban, hogy a problémamegoldást legjobban támogató módszereket és technikákat alkalmazhatjuk. Ilyen CASE eszközök az OMT, Fusion, Booch, OOCL, Martin/Odell, Shlaer/Mellor, Coad/Yourdon

módszereket valamint az UML-t támogató *Paradigm Plus*, a Martin/Odell féle módszertant támogató *PTECH CASE* eszköz vagy a Booch'93, OOSE, OMT, UML módszereket felajánló *Rational Rose Family* szoftvercsalád. Bár egy CASE eszköz megvásárlása meglehetősen magas ráfordítást igényel hosszú távon megéri a befektetés, hiszen gyorsabban, a fejlesztés során elkövethető hibák minimálisra szorításával, a felhasználói igényekhez való jobb alkalmazkodással könnyen karbantartható, hatékony szoftvert fejleszthetünk.



Ábra 6. A Rational Rose CASE eszközével készített állapotdiagram

5. Következtetések

Az objektumorientált elemző/tervező módszerek alkalmazása, eredményeinek OO programnyelvekkel és OO adatbázis-kezelő rendszerekkel történő realizálása a szoftverfejlesztés teljes életciklusára nézve egységes szemlélet biztosít. Az OO fejlesztési életciklusban az egyes lépések többszörösen iterálódnak, az osztályok tervezésének top-down és bottom-up kevert módja lehetővé teszi nagy és rugalmas rendszerek tervezését. Az így létrehozott osztályok a meglévő és a jövőbeli projektekben is többszörösen felhasználhatók.

A fejlesztési munka különböző lépéseinek végrehajtásával párhuzamosan végezhető el az osztályok implementációja, ami lehetővé teszi, hogy a programozási munka lényegesen korábban elkezdődhet, mint a teljesen kész terv. A projekt munkájában több feladat hárul a rendszerelemzőkre és - tervezőkre, az implementációs fázis ugyanis jól előkészített és magas szinten automatizálható. Különös gondot kell fordítani az osztály-könyvtár fenntartására és karbantartására. Ezt a feladatot rend-

szerint egy felelős könyvtár-menedzser végzi, aki garantálja a meglévő objektumok újrafelhasználhatóságát, a könyvtár bővítését új osztályokkal és könyvtárakkal a szervezetből és kívülről egyaránt.

Bár nem kérdéses, hogy a teljes életcikluson végigfutó egységes szemlélet a leghatékonyabb, hiszen ez tudja csak garantálni, hogy az egyik lépés outputja optimális inputja legyen a következőnek, mégis bizonyos esetekben indokolt lehet a módszerek és technikák kevert használata. Ilyen szituáció lehet például a jelenleg még magas fejlesztési költségek vagy az új módszerek fokozatos bevezetésének igénye. Annak eldöntése azonban, hogy a fejlesztés teljes életciklusában OO szemléletet valósítunk-e meg a projekt vezető feladata és felelőssége.

Biztos vagyok azonban abban, hogy *a hibrid megoldások alkalmazása átmeneti állapot*, ami megszűnik, ha a rendelkezésre álló, fejlesztést támogató szoftverek hozzáférhetővé válnak a kisebb szervezetek vagy fejlesztő-teamek számára is, és ha a fejlesztők magukévá teszik az évszázad utolsó évtizedének előremutató lehetőségét, az objektumorientált szemléletet.

Irodalom

[1] G.**Booch**: *Objec-Oriented Design*-Benjamin-Cummings, Redwood City, Cal. 1990, 1991.
[2] G.**Booch**-I.**Jacobson**-J.**Rumbaugh:** *The Unified Modeling Language Version 1.0* -Rational Software Corporation, Santa Clara, Cal. 1997.
[3] P.**Coad**-E.**Yourdon**: *Object-Oriented Analysis* - Prentice-Hall, Englewood Cliffs, N.J., Yourdon Press, New York 1990.
[4] W.**Lorensen**: *Object-Oriented Design* - CED SE Guidelines, General Electric Co., 1986
[5] J.**Martin**-J.**Odell**: *Object-Oriented Analysis & Design*-Prentice-Hall, Englewood, N.J., 1992
[6] B.**Meyer**: *Object-Oriented Software Construction* - Prentice-Hall, Englewood, N.J., 1988
[7] *The PTECH Solution* - Ptech Inc. Cambridge Mass. USA. 1994.
[8] M.**Raffai:** *Objektorientierte Konzeption im Software-Entwicklungsprocess*-Kempten, No.1992
[9] M.**Raffai:** *Információrendszer-tervezés* - NOVADAT Kiadó Győr, 1996.
[10] M.**Raffai:** *Információmenedzsment, fejlesztési módszertanok*-NOVADAT Kiadó Győr, 1996.
[11] M.**Raffai:** *OOA&D Objektumorientált elemzés/tervezés - Módszertani anyag és esettanulmány* - Kézirat 1997.
[12] J.**Rumbaugh**-M.**Blaha**-W.**Premerlani**-F.**Eddy**-W.**Lorensen**: *Object-Oriented Modeling and Design* - Prentice Hall International Inc. 1991.
[13] G.**Wilkie**: *Object-Oriented Software Engineering* - Addison-Wesley, 1993
[14] M.**Raffai**: *Az informatika fél évszázada* - Springer Hungarica, 1997.

OBJEKTUM-ORIENTÁLT STRATÉGIÁK

Boér Tamás, tamas.boer@ac.com
Andersen Consulting

1. Bevezető

Az objektum-orientált technológia önmagában nem a stratégiát jelenti. Viszont támogatja - helyesebben: támogatnia *kell* - egy szervezet, vállalat, cég stratégiáját, éspedig három különböző szinten:

- üzleti,
- szervezeti, és
- információ-technológiai

szinten. Mindhárom szint támogatása szükséges, de azt is tudni kell, hogy mindhárom szint egyidejű támogatása konfliktusokhoz is vezethet.

A következőkben az objektum-orientált technológiáról, mint az üzleti stratégiát támogató eszközről, valamint az objektum-orientált rendszerek kifejlesztésére irányuló projektek sikerét nagy mértékben befolyásoló, stratégiai fontosságú elemekről lesz szó.

2. Vállalati stratégia szintek

Egy vállalat stratégiai célkitűzéseinek a meghatározásával megbízott csapatnak nagyon jól kell értenie az üzleti helyzetet, a tevékenységeket és az összes, kölcsönhatásban levő változót, amelyet kézben szeretnének tartani.

Az üzleti célkitűzések kulcsfontosságúak az informatikai fejlesztésért felelős vezető számára, mivel az üzleti és informatikai célkitűzések összhangban kell legyenek. Amit fontos tudni az az, hogy ha az üzleti folyamatok változnak, mennyi időt fog igénybe venni a folyamatokat támogató számítástechnikai háttér megváltoztatása, hogy az a megváltozott folyamatokat is támogassa.

Az objektum-orientált technológiára való átállás feltételezi az innováció, az újítások támogatását. A változást egy pozitív irányba mozgató erőnek kell tekinteni és a szervezeten belül az üzleti folyamatok csiszolása, tökéletesítése egy állandó célkitűzéssé kell váljon. Az innovációnak a szervezeten belüli minden formában történő támogatása meg kell történjen ahhoz, hogy sikeresen át tudjunk állni az objektum-orientált technológiára.

Az információtechnológiai stratégiák között - a stratégia "érettségének" függvényében - három szintet különböztetünk meg.

I. szint. Az információtechnológiai stratégiák általában a hardver és szoftver beszerzésénél érvényesülő elvekkel kezdődnek és csekély figyelmet szentelnek az informatikai, számítástechnikai szakembergárda kialakítására és képzésére, valamint a jövőbeni változtatások iránti igényekre.

II. szint. Ez a szint már azt jelenti, hogy a szervezeten belül törekvés van a központosított számítástechnikai csoport létrehozására. Ez a centralizáció oda vezet, hogy "top-down", azaz felülről irányított tervezés folyik és hogy jobban odafigyelnek arra, hogy az informatikai háttér támogassa a teljes üzleti stratégiát és tevékenységet.

III. szint. Ez a szint már egy érett információ-technológiai stratégiát feltételez, amely úgy használja a technológiát, mint egy eszközt, amely az üzletnek stratégiai előnyöket nyújt. Ezen a szinten a számítástechnikai stratégiát és irányvonalat a hosszútávú tervek uralják. Ahhoz, hogy az objektum-orientált technológiát minél nagyobb sikerrel alkalmazhassuk egy vállalaton belül, ezt a stratégiai szintet el kell érni.

3. Az objektum-orientált technológiára való átállás akadályai

Egy objektum-orientált technológiára való átállás előtt három fontos akadályozó tényezőt kell figyelembe venni:

- a már létező számítástechnikai rendszereket,
- a létező fejlesztési kultúrát és
- a tanulási görbét.

A jelen rendszerek elsősorban információforrást jelentenek az új rendszerek számára és nem kell őket feltétlenül lecserélni. Ezért a stratégia a meglévő rendszerek "wrapper"-ekbe foglalását is magába kell foglalja. A régi és új rendszerek közötti interfészek kritikus fontosságúak és sok törődést igényelnek ahhoz, hogy megbízhatóan működjenek.

Egy megrögzött, megszokott kultúra megváltoztatása egyike a legnehezebb feladatoknak. Ha a jelen kultúrát a megváltozott körülmények miatt megváltoztatni kényszerülünk, akkor a kultúrát képviselő emberek részéről egy erős ellenállásba ütközhetünk. Egy nagyon alapos elemzés szükséges annak felméréséhez, hogy a jelen kultúra milyen mértékben és milyen sebességgel tudja magáévá tenni a szükséges változásokat.

A harmadik, szinte mindig létező akadálya az objektum-orientált technológiára való áttérésnek a tanulási görbe. Objektum-orientált technikákkal sokkal intuitívebben ki lehet fejezni a különböző rendszerekre adott megoldásokat, modelleket. Az újfajta gondolkodásnak az elsajátítása jelentős időt emészthet fel, főleg azok részéről, akik hosszú időn keresztül egy nagyon szűk, korlátozott fejlesztési környezetben dolgoztak, pl. COBOL.

4. Objektum-orientált fejlesztési stratégiák

Az objektum-orientált technológia egy *hozzáállás* valamint egy *technika* szoftver rendszerek fejlesztésére, ezért sokkal inkább a stratégia *végrehajtásával* asszociálható, mint önmagával a stratégiával. A szoftver-fejlesztési stratégia egyszerűen kijelenthetné, hogy minden fejlesztés objektum-orientált módon történjen, de ez éppen a lényeget hagyná figyelmen kívül.

Egy stratégia lényege, hogy egy adott úton, jól átgondolt elvek és szabályok szerint kell haladni, amely út hosszútávú és az átlagnál merészebb célkitűzéseket tartalmaz. A stratégiák fentről jönnek - a felső vezetés, a fejlesztési igazgató szintjéről. Fontos viszont, hogy a stratégia kialakításához, meghatározásához minden szinten járuljanak hozzá, azokat megértsék, elfogadják és támogassák.

Ha a stratégiák általános betartandó irányvonalakat szabnak meg, akkor önmagukban is általánosak kell legyenek. Az alábbiakban egy nagyon általános feladatmeghatározást, célkitűzést adunk meg, ami egy kissé furcsának tűnhet ugyan, de kiinduló alapja lehetne minden szoftver-fejlesztéssel foglalkozó vezető célkitűzéseinek:

Fejlesszünk ki a felhasználók jelen és jövőbeni igényeit kielégítő kiváló minőségű rendszereket, gyorsan, olcsón és könnyebb ellenőrizhetőséggel. [1]

Figyelem: ebben a mondatban szó sincs objektum-orientált technológiáról! Nem is lehet róla szó, mert a cél nem önmagában az objektum-orientált technológia, hanem a stratégia végrehajtása. Ezért bármely meghozott döntésnek támogatnia kell ezeket az általános célkitűzéseket.

A következő lépés pedig az előbb megfogalmazott általános célkitűzés valóra váltása objektum-orientált elemek segítségével.

5. A siker kulcsfontosságú tényezői

Egy sikeres objektum-orientált stratégia kulcsfontosságú tényezői a következők [1]:

- koncentrálás a végtermékre
- iteratív és inkrementális fejlesztés
- osztályok rendszerezése
- multidiszciplináris fejlesztői csapatok

5.1. Koncentrálás a végtermékre

A szoftverfejlesztés végcélja termékek létrehozása. A végtermékre való koncentrálás azt jelenti, hogy a fejlesztési tevékenység kitűzött eredményét állandóan szem előtt tartjuk. Eredmények alatt a felhasználó számára lényeges dolgokat értjük, azokat, amikkel a felhasználó már tud kezdeni valamit. Fontos lehet ugyan tudni, hogy az elemzés már befejeződött, de a rendszer felhasználóját ez a legkevésbé sem érdekli.

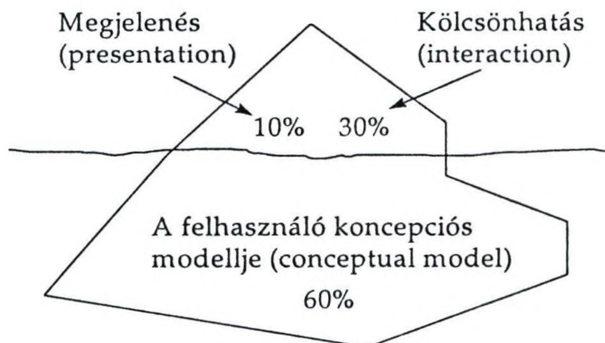
A felhasználók bevonása. A végtermékre való koncentrálás azt feltételezi, hogy a fejlesztés nagyon szoros kapcsolatban áll a felhasználókkal. A felhasználók bevonásával egyidejűleg a változások iránti igények is megnőnek. A fejlesztői csapatnak el kell ismernie, hogy a felhasználók környezetében történő változások változásokat jelentenek a rendszerre is. A felhasználók bevonása ahhoz vezet, hogy a fejlesztők is jobban megértik az üzleti változásokat és azt, hogy a kifejlesztendő alkalmazások hogyan járulnak hozzá az üzleti folyamatok megváltozásához.

Különböző elvárások. Az üzleti vezetők közötti félreértéseknek az elsődleges forrása a fejlesztésben résztvevők különböző elvárásai. Mert például, míg a felhasználó elvárása az, hogy a rendszer időben készüljön el és jól működjön, addig az üzleti vezető egyik legnagyobb gondja, hogy a rendszer mennyibe fog kerülni, a projekt menedzseré, hogy időben, az előirányzott költségeken belül és meglepetések nélkül készüljön el, a technikai szakértőé, hogy a fejlesztés legyen egy technikai kihívás és az értékesítés képviselőé, hogy legyen kész már tegnapra, ha lehet. Ezeket az elvárásokat a stratégiai célkitűzések fényében gondosan mérlegelni kell.

Elvárások teljesítése. Az elvárások mérlegelése veszélyes terület. Ami igazán fontos, az az, hogy *mire van ténylegesen szükségük* a felhasználóknak, semmint, hogy mit szeretnének. Van-e jövőképük, amit meg szeretnének valósítani, de nem képesek kifejezni vagy nem tudják, hogy a jelen technológiával azt meg is lehetne valósítani? Van-e egy olyan nyelvezet, aminek a segítségével ezt dokumentálni lehet? Egy kevésbé ismert előnye az objektum-orientált technológiának a közös nyelvezet, amelyet a projekt teljes életciklusa során nyújtani tud.

Ha az ügyfeleink az üzleti vezetők, akik megrendelték a rendszert, akkor egy jogos elvárás az, hogy a projekt indulása előtt a rendszer által jelentős üzleti előnyöket tudjunk előrevetíteni. Ezeket az előrevetített előnyöket a rendszer fejlesztése és átadása során legyünk képesek követni és az élesbe való átállás után a valóban megvalósított előnyöket képesek legyünk összehasonlítani az előrevetített előnyökkel.

Az objektum-orientált felfogásra való átállás egyik gyakran hangoztatott előnye, hogy a fejlesztésnek már egy viszonylag korai fázisa során tudunk mutatni valamit a rendszer felhasználóinak. Való igaz, hogy a grafikus felhasználói felület jelentős löketet adott az objektum-orientált gondolkodás fejlődéséhez, de ne felejtsük el, hogy a grafikus felhasználói felület elemei, amelyek által a rendszer "megjelenik" a felhasználónak (presentation) és amelyek megadják a rendszerrel való kölcsönhatás lehetőségeit (interaction), mindössze 40%-kal járulnak hozzá a rendszer használhatóságához (usability). A másik 60% az a felhasználó "koncepciós modellje" (conceptual model) a GUI alatt meghúzódó objektumokról, azok tulajdonságairól és viselkedéseiről (behaviour). Gyakran ez a 60% az, ami eldönti, hogy egy rendszer mennyire használható és mennyire fogadják el vagy milyen mértékben kelt félreértést, gyanakvást és bizalmatlanságot (1. ábra.):



1. Ábra. A használhatósági "jéghegy"

*Tesztelés.*A legfontosabb eredménye egy fejlesztésnek egy használható és letesztelt rendszer. A tesztelés az objektum-orientáltsággal nem tűnik el, sőt: még nehezebb és bonyolultabb is lehet, mint egy tradicionális, procedurális rendszer esetén. Egyedi komponensek tesztelése viszonylag egyértelmű, de az objektum-orientált rendszerek alapvetően *dinamikus* rendszerek, az objektumok létrejönnek, egymás között üzeneteken keresztül kommunikálnak, majd megszűnnek. A felhasználó által kiváltott minden egyes esemény egy üzenet-sorozatot hoz létre. Ez az üzenet-sorozat az, ami a rendszer egyes elemeit összetartja. Érthető, hogy az üzenet-sorozatok minden egyes lehetőségét letesztelni nem kis feladat.

5.2. Iteratív és inkrementális fejlesztés

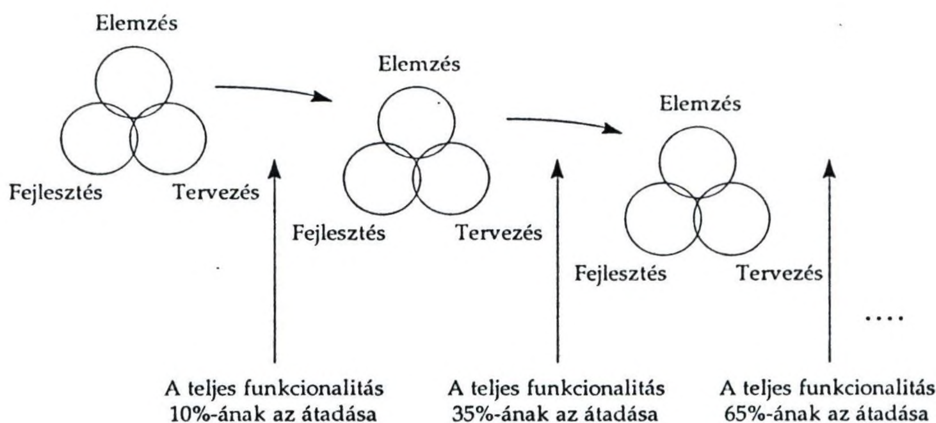
Az objektum-orientált fejlesztésnél nagyon szoros kommunikációra van szükség az elemzők, tervezők és programozók között. A tradicionális fejlesztési folyamatok (vízesés modell) ezt a szoros kommunikációt meglehetősen gátolják, ezért van szükség a szoftverfejlesztés spirál-modelljére (2. ábra.):

2. Ábra. A szoftver-fejlesztés spirál-modellje

A modell minden egyes inkrementumánál egyértelmű célkitűzéseket és alternatív megoldás-modelleket használunk. A fejlesztés során különböző technikákat és eljárásokat lehet alkalmazni [2]. A kifejlesztett verziónak és a rendszer jövőbeni lehetőségeinek az értékelése során döntés születik, hogy történjen még egy iteráció vagy álljunk meg a fejlesztéssel. Mivel a spirálnak minden egyes "fordulata" csak kis lépésekben növeli a költségeket, ezek a kicsi, rögzített költségvetésű inkrementumok hatásosak lehetnek a költségek kézben tartásához.

Egy dolog, ami a spirál-modellből nem derül ki, az az, hogy minden egyes inkrementum a végső rendszernek egy funkcionális verziója kell legyen. Funkcionális verziónak a rendszernek egy

megfelelően tervezett, részfunkcionalitással tökéletesen működő részét nevezzük, amely rész tartalmazhatja a teljes funkcionalitás akárhány százalékát (3. ábra):



3. Ábra. Rész-funkcionalitást tartalmazó verziók fejlesztése

Fontos megemlíteni, hogy ha a rendszer teljes körű, tervezett funkcionalitásának kifejlesztésére nem jut pénz, a rész-funkcionalitással átadott rendszer még mindig jelentős hasznot hozhat a felhasználóknak. A rendszer teljes funkcionalitásának mondjuk 85%-a működőképes és hasznot hozó rendszer lehet, míg ha a 100%-ot átfogó funkcionalitás mindössze 85%-ban működik, akkor félő, hogy az egyszerűen semmire nem használható.

5.3. Komponensek újrafelhasználása az osztályok rendszerezése által

Az osztályok rendszerezése abból a célból, hogy elősegítsük a komponensek újrafelhasználását, stratégiai fontosságú eleme egy objektum-orientált fejlesztésnek. Az osztályokat csoportokba kell rendszerezni, ahol egy csoport egy üzleti vagy egy infrastruktúra területet jelöl.

Az üzleti csoportok olyan funkciókat tartalmaznak, mint számlázás, tervezés, termelés, értékesítés, stb. Mindegyik csoporthoz tartozhat egy osztálykönyvtár, amely könyvtár osztályai az üzleti tevékenységeket fogják tükrözni és ezekből az osztályokból lehet az egyedi üzleti megoldásokat kifejleszteni.

Az infrastrukturális csoportokon belül olyan speciális osztályokat találunk, amelyek egy szervezet számítástechnikai környezetét támogatják. A szoftver infrastruktúrának négy nagy komponense van:

- Felhasználói felület
- Adat-menedzsment
- Kommunikáció
- Működési környezet

A *felhasználói felület* több, mint egy külső cég által szállított osztály-könyvtár. A felhasználói felület, mint a számítástechnikai környezetet támogató komponens, azokat a standardokat és irányelveket is jelenti, amelyek meghatározzák egy cég termékeinek és rendszereinek az általános külső megjelenését (look and feel). A külső cég által szállított könyvtárak osztályai felhasználhatók közvetlenül, de a felhasználó számára magasabb elvonatkoztatási szintet nyújtó keretrendszerekbe is beépíthetők.

Az *adat-menedzsment* komponensek egy szervezeten belüli változatos tároló rendszerekhez nyújtanak hozzáférést és elérést. Egyszerű volna azt mondani, hogy objektum-orientált rendszerekhez csak objektum-orientált adatbáziskezelő rendszereket alkalmaznak. A valóság viszont azt mutatja, hogy ez távolról sincs így, mivel a még létező nagyon sok, nem objektum-orientált adatbázist is ki kell aknázni, minimális módosítással. Ehhez pedig adapter osztályokat (wrappereket) kell használni.

Hasonló elvárások vannak a *kommunikációs* komponensekkel szemben is. A különböző kommunikációs protokollokhoz különböző adapter osztályokra van szükség.

Végül, a *működési és fejlesztési környezet* osztályai befolyással vannak a rendszer infrastruktúrájára is. A megfelelő termékek kiválasztása valószínű, hogy egyike a projektmenedzser legnehezebb feladatainak.

A változatos igények kielégítéséhez a komponensek beszerzésekor egy úgynevezett "multikulturális" stratégiát tanácsos alkalmazni, ami azt jelenti, hogy amilyen mértékben csak lehet, többféle nyelvet, operációs rendszert, és hardver platformot igyekszünk egyidejűleg támogatni.

A komponensek újrafelhasználása. Az objektum-orientált technológia önmagában a legkevésbé sem garantálja a kifejlesztett komponensek újrafelhasználhatóságát [3]. Az objektum-orientált technológia sokat hangoztatott előnye, a komponensek újrafelhasználása elérhető, de csak akkor, ha az osztályok léteznek, a fejlesztők tudnak róluk, ismerik a tulajdonságaikat és a menedzserek érdekeltek a létező komponenseknek a kiaknázásában. Újrafelhasználható komponensek kifejlesztése éppúgy pénzbe kerül, mint a megvásárlásuk. Azonban tapasztalat szerint 75%-os megtakarítást lehet elérni, ha osztályokat egy üzleti területen belül használunk újra és 25%-ost a különböző üzleti területek között, ha az infrastrukturális komponenseket használjuk újra [1].

5.4. Multidiszciplináris fejlesztői csapatok

Jelentős fejlesztések manapság már soha nem történnek az egyedülálló programozó elszigeteltségében. A tevékenységek túl bonyolultak és a határidők túl agresszívek. Az elemzés, tervezés és fejlesztés fázisok közötti határvonalak az objektum-orientált technológiával nem olyan egyértelműek. Az elemzés és tervezés során több erőfeszítésre van szükség, hogy a felhasználók elvárásai ki legyenek elégítve. Ez azt jelenti, hogy egyidőben szükség van az elemző, a tervező, a programozó és az üzleti szakértő tapasztalatára és tudására. Ezért multidiszciplináris csapatok kialakítására kell törekedni, amelyeken belül az adott tudás és tapasztalat egyszerre fellelhető.

6. Összefoglaló

Ne felejtsük el, hogy a siker biztosítását a stratégia szintjén kell kezdeni. Elfeledkezhetünk egy stratégiai fontosságú összetevőről és könnyen kudarcba fullad a projekt. Vagy elég figyelmet szentelünk mindegyiknek és a siker biztosítva van. Lehet, hogy az említett stratégiai elemekhez még lehet újabbakat hozzáadni, de ne feledjük el az alapvető célkitűzést:

Fejlesszünk ki a felhasználók jelen és jövőbeni igényeit kielégítő kiváló minőségű rendszereket, gyorsan, olcsón és könnyebb ellenőrizhetőséggel.

Aztán bátran állítsuk fel a magunk változatát a sikeres objektum-orientált stratégia kulcsfontosságú komponenseire: koncentrálás a végtermékre, iteratív és inkrementális fejlesztési folyamatok, osztályok üzleti és infrastrukturális területekre való csoportosítása és multidiszciplináris fejlesztői csapatok.

Irodalomjegyzék:

1. Barry McGibbon: Managing Your Move to Object Technology, 1995, SIGS Books

2. Object Expert, 1996/1997 és 1997/1998 évfolyamok

3. Andersen Consulting "Knowledge Xchange" adatbázisok

C++ STANDARD TEMPLATE LIBRARY

Porkoláb Zoltán, gsd@ludens.elte.hu

X-Close Kft. 1043 Budapest, Lőwy Izsák u. 6.

1. Bevezetés

A szoftverfejlesztés egyik legnehezebb feladata minden bizonnyal a standard könyvtárak specifikációja és implementálása. A standard könyvtár segítségével a programozó olyan feladatokat old meg, amelyeket a könyvtár megalkotója sejtett ugyan, de pontosan soha nem láthatott előre. A legtöbb ma használt programozási nyelv rendelkezik standard könyvtárral: a C programozási nyelv standard könyvtára függvény implementációkat és a hozzájuk tartozó header-fájlokat tartalmazza. Az ADA nyelv standard package-eket specifikál, az Eiffel és a JAVA nyelvek szabványos osztályokat definiálnak.

A standard könyvtárnak teljesnek és rugalmasnak kell lennie: a programozónak, mint aki építőkockákból építkezik, ki kell tudnia rakni a leggyakrabban előforduló feladat-kombinációkat. E feladatok közé tartoznak legtöbbször az input/output műveletek, az alapvető matematikai függvények, a string-kezelés, stb...

Másrészről a könyvtár legyen minimális: ne tartalmazzon redundáns elemeket. Redundáns elemek nemcsak méretbeli növekedést okoznak (és evvel a szoftver életciklusának számos pontján - a fejlesztéstől a karbantartásig – nehezítik a könyvtár használatát), de a lehetőségek túlburjánzása arra késztetik a programozót, hogy a nehezen kiismerhető könyvtári függvények helyett maga írjon - valószínűleg gyengébb minőségű – eljárásokat.

A C++ Standard Template Library (STL) olyan szoftvercsomag, amely megfelel a fentebb leírt követelményeknek és amit alkalmazva drasztikusan csökkenthető a programozó által megírandó kód, és új módszereket lehet alkalmazni az újrafelhasználható szoftver komponensek kialakításában. A könyvtár elvét Alexander Stepanov (Silicon Graphics) dolgozta ki, az implementáció David R. Musser (Hewlett-Packard) nevéhez fűződik. Az STL 1994 óta része a C++ ANSI/ISO szabvány-javaslatnak (Draft Standard).

Az STL módszertani sajátossága, hogy – bár maga is objektum orientált komponensekből épül fel – a korábban implementált osztálykönyvtárakkal szemben az adat és a rajta elvégezhető műveletek külön komponenseket képeznek. A fő komponens típusok (Container, Iterator, General Algorithm, Function object, Adapter) egyszerű technikákkal egymáshoz csatlakoztathatóak és az $O(n^3)$ nagyságrendű megírandó kód mérete $O(n)$-re csökkenthető.

A Standard Template Library – mint szoftverkonstrukciós elv – programozási nyelvtől független. Generikus (típusparaméterezési) konstrukcióval rendelkező más nyelveken is implementálható, de érdemes kísérletet tenni egyéb programnyelvi implementálásra is.

2. A C++ template nyelvi eszköz

Ahogyan azt minden C++ programozó jól tudja: a C++ nyelv Bjarne Stroustrup nevéhez kötődik. Stroustrup 1979-ben kezdett azon gondolkodni, hogyan lehetne a C nyelvet az általa jól ismert Simula67 objektum orientált eszközeivel kibővíteni. Munkája a nyolcvanas évek elején „C with classes", majd C++ nyelvként vált közismertté és rohamosan elterjedté. A C++ class fogalma olyan általános típuskonstrukció, amely megvalósítja az adat és a rajta végezhető műveletek egységét, az osztály interfészének és implementációjának szétválasztását, az öröklést és más, az objektum orientált paradigmának megfelelő nyelvi elemeket.

A nyelv első változatai nem tartalmaztak lehetőséget típussal paraméterezett függvények, vagy osztályok létrehozására. Az erre utaló első tervek 1986-ban jelentek meg Stroustrup ill. Koenig munkáiban [1]. Az új nyelvi elem bevezetését sürgette, hogy addigra bebizonyosodott; a típussal paraméterezett konstrukció nem vezethető vissza az örökléssel megvalósíthatókra. Ezt a gyakorlatban is bizonyította néhány vendor-függő implementáció bonyolultsága és nem kielégítő volta. Mára a C++ template – típus-paraméterezési – mechanizmusa kiforrott és minden jelentősebb fordítóprogramban helyesen implementált.

Egyszerűbb esetben a C++ nyelv template függvények használatát teszi lehetővé. A template függvény (2) hatékonyan és magasabb szintű nyelvi eszközzel helyettesíti a hagyományos C preprocessor „trükköket" (1), mint az alábbi példa is mutatja:

```
(1) #define max( a, b )((a)>(b) ? (a) : (b))

(2) template <class T> T max( T a, T b ) { return a > b ? a : b; }
```

A C++ template magasabb szintjét és a C makró hiányosságait jól demonstrálja, ha a maximális elem kiválasztása helyett a két elem felcserélése a feladat. A (3) megoldás nem valósítható meg (hasonlóan egyszerű) C előfordító „makróval".

```
(3) template <class T> void swap( T &a, T &b )
    {
        T c = a;   a = b;   b = c;
    }
```

A fenti példában a T formális név hatóköre a template<...> előfordulása utáni definícióra vagy deklarációra terjed ki. A fentihez hasonló template függvények fordítási időben automatikusan példányosulnak, az éppen aktuálisan alkalmazott típusra. Így ha a programunk például az alábbi kódrészletet tartalmazza (4):

```
(4) double i = 2.0, j = 3.14;
    :
    swap( i, j );
```

akkor fordítási időben létrejön („példányosul") a swap függvény kódjának egy olyan változata, ahol a két (megegyező típusú) paraméter a double típushoz tartozik

Összetettebb eset, amikor típussal paraméterezett adattípust (template class) hozunk létre. Ebben az esetben a típus paraméter szerepelhet a member-függvényekben, a a konstruktor ill. destruktor megvalósításában ill. az osztály állapotai (adat-memberek) definiálásában. Az alábbi példa a tetszőleges típusokból álló „rendezett pár" osztályt valósítja meg (5):

```
(5) template <class T, class S>
    struct pair
    {
          T first;
          S second;
          pair( const T& x, const S& y ) : first(x), second(y) { }
    };
```

A template osztályokat a konkrét típust megadva példányosítjuk, pl. (6):

```
(6) pair< int, double> x;
    pair< char*, char*> y;
```

Természetesen a típus paraméter számos új – az eddigieknél bonyolultabb – nyelvi szabály bevezetését tette szükségessé a C++ nyelvben. Gondoljunk csak a template függvények túlterhelésére (overloading), az öröklődésre, vagy a template osztályok közötti friend kapcsolatra. Nem csoda, ha a népszerű C++ levelezési listák és news-group-ok kérdései jórészt a template konstrukciókra vonatkoznak.

3. Az STL alapelvei

Körülbelül abban az időben, amikor Bjarne Stroustrup elkezdett foglalkozni a C és a Simula67 keresztezésével, egy másik szakember – Alexander Stepanov – az újrafelhasználható szoftver komponensek megvalósításának teljesen új módján kezdett dolgozni. Abban az időben Stepanov az ADA nyelvben próbálta megvalósítani elképzeléseit, de igazi sikert csak a C++ megjelenésével érhetett el. Amennyire a C++ Stroustrup nevével vált eggyé, úgy kapcsolható Stepanov neve a Standart Template Library (STL)-hez.

Ahhoz, hogy az STL alapkoncepcióját megértsük, képzeljük el azt a teret, amely egy adott programozási nyelvben megírható (alap)feladatokat jellemzi. Ez a tér háromdimenziós. Az egyik dimenzió a nyelv alaptípusait jellemzi (pl. int, long, double, char *...). A második dimenzióval jelképezzük az alaptípusokból konstruált adatszerkezeteket (pl. vektor, lista, verem...). Nyilvánvaló, hogy a két „dimenzió" ortogonális: az int típusú elemeket tartalmazó lista és a char* verem teljesen függetlenek egymástól. Végül vegyük fel hozzájuk harmadik dimenziónak az előző párokon végrehajtható elemi algoritmusokat, mint egy adott tulajdonságú elem keresése (find), rendezés (sort), összefésülés (merge).

Egy „hagyományos" programozási nyelvben – mint amilyen, pl. a C nyelv – K*L*M nagyságrendű alapfeladat képzelhető el, ahol K, L és M az előbb említett dimenziók számossága. Az egyes algoritmusokat ugyanis külön kell megírni minden egyes adatszerkezetre és adattípusra. Ennek az $O(n^3)$ nagyságrendű kódnak a megírása igen komoly terhet jelentene a standard könyvtár megvalósítójára, és valószínűleg (használhatatlanul) bonyolult könyvtárat eredményezne.

Amennyiben a nyelv tartalmaz típus paraméterezési lehetőséget – pl. Eiffel, C++ – úgy ezt a nagyságrendet csökkenteni lehet L*M-re, ami $O(n^2)$. Az igazán nagy nyereség azonban abból származik, ha sikerül olyan (template) adatszerkezeteket definiálni, amelyen tetszőleges algoritmus tud dolgozni, ill. olyan algoritmusokat, amelyek közömbösek az egyes adatszerkezetek eltérő részleteire. Ekkor minden egyes adatszerkezete és algoritmust csak egyszer kell megvalósítani (L+M, O(n) nagyságrend), és ezek kölcsönösen képesek párban együttműködni. Ezen az elven alapul az STL.

Ahogyan azt egy C++ nyelvű könyvtártól elvárhatjuk, az STL külön gondot fordít a hatékonyságra. Az általános STL algoritmusok és adatszerkezetek használata nem jelent jelentős hatékonyságcsökkenést ahhoz képest, mintha az egyes algoritmus+adatszerkezet párokat direkt módon „drótoztuk" volna össze.

4. Az STL szerkezete

Az alábbi egyszerű példa (7) megmutatja [2], hogy milyen szorosan próbálja az STL követni a „hagyományos" C programozási stílust. A példában két adatterület tartamát kell összefésülni.

```
(7) /* C/C++ megoldás */
    int  a[2000];
    int  b[3000];

    int *c = (int) malloc(sizeof(int)*(2000+3000));
    merge( a, a+2000, b, b+3000, c );


    /* STL megoldás */
    vector<int> a;
    list<int>   b;

    int c = allocate( a.size(), b.size());
    merge( a.begin(), a.end(), b.begin(), b.end(),
                row_storage_ierator<int*, int>, c);
```

Az STL szerkezetileg hat fő komponenstípusra épül: konténerek (container), általános algoritmusok (generic algorithm), iterátorok (iterator), függvény-objektumok (function objects), illesztők (adaptor) és memória-kezelők (allocator).

4.1. Konténerek

A konténerek feladata, hogy egy adott (parametrizált) típuson egy adatszerkezetet valósítson meg. A kontérekete kétfelé lehet osztani; a szekvenciális konténerek (sequence containers) lineáris adatstruktúrákat valósítanak meg, mint a

`T a[n]` rögzített hosszúságú tömb, bármely eleme közvetlenül elérhető

`vector<T>` változó hosszúságú tömb, bármely eleme közvetlenül elérhető, a legutolsó elem inzertálható ill. törölhető.
`deque<T>` hasonló a vektorhoz, de mindkét végén inzertálható ill. törölhető.
`list<T>` lineáris idejű elem-elérés, de bárhova inzertálhatok, illetve bárhonnan törölhetek.

A konténerek másik csoportja a rendezett asszociatív (sorted associative) konténerek, melyek mérete változhat és kulcs alapján történő lekérdezésre szolgálnak.

`set<Key>` egyedi kulcs, csak a kulcsot tároljuk.
`multiset<Key>` mint a set<...>, de a kulcs egyedisége nem követelmény
`map<Key, T>` egyedi kulcs, de nemcsak a kulcsot tároljuk.
`map<Key, T>` mint a map<...>, de a kulcs egyedisége nem követelmény

4.2. Általános algoritmusok

Az általános algoritmusok (pl. find, sort, merge...) az adatok reprezentációjától független eljárásokként kerülnek implementálásra. Az algoritmusok általános szabályokat adnak az adatok elérésére, és garantálják, hogy ha az adatszerkezet megfelel az algoritmusban előírt szabályoknak, akkor az algoritmus hatékony, azaz teljesítménye nem tér el lényegesen az adat reprezentációjának ismeretében megírt algoritmusok teljesítményétől.

4.3. Iterátorok

Az általános algoritmusokat és az adatszerkezeteket az iterátorok kapcsolják össze. Az iterátor feladata, hogy az adatszerkezet éppen aktuális elemét prezentálja az algoritmus számára. Mintául a C pointer fogalma szolgált (és valóban: egy pointer egyben iterátor is lehet).

Input iterátor az a p iterátor, amelyre definiáltak az alábbi műveletek:
`++p` a következő elem elérése (megcímzése).
`x = *p` a kijelölt elem értékének kiolvasása.
`p == q` iterátorok egyenlőségének vizsgálata.
`p != q` iterátorok nem-egyenlőségének vizsgálata.
Output iterátor esetén az érték kiolvasása, helyet írása biztosított (x = *p helyett, *p = x).
Forward iterátor rendelkezik az input és output iterátor művleteivel együttesen.
Bidirectional iterátor, amely hátrafelé is tud (egyesével) mozogni, azaz --p értelmezett.
Random access iterátor, amely nemcsak egyesével tud lépkedni, azaz értelmezett:

```
p + n     p - n
p += n    p -= n
i = p - q
p < s     p <= s ...
```

Stream iterátorok: istream_iterator, ostream_iterator

Az alábbi példa (8) bemutatja, hogyan írhatunk egy általános algoritmust az iterátorok segítségével [3]. Az accumulate(first, beyond, init) összegzi az elemeket egy adatszerkezeten. Megírásához input iterátort használunk. A main() rutin vector<int> és list<double> szerkezetekre teszteli a függvényt.

```
(8)   #include <vector.h>
      #include <algo.h>

      template <class InputIterator, class T>
      T accumulate( InputIterator first, InputIterator beyond, T init)
      {
           while ( first != beyond )
           {
                init = init + *first;
                ++ first;
           }
      }
      int main()
      {
           int x[5] = { 2, 3, 5, 7, 11};
           vector<int>     v( &x[0], &x[5]);

           int sum1 = accumulate(v.begin(), v.end(), 0);

           duble y[5] = { 2.0, 3.0, 5.0, 7.0, 11.0};
           list<double> l( &y[0], &y[5]);

           double sum2 = accumulate(l.begin(), l.end(), 0.0);

           return 0;
      }
```

4.4. Függvény objektumok

Az előző példa több helyen is megtörte az általánosságot. Egyrészt feltételezte, hogy az alanti adatszerkezeten és típuson értelmezett a + művelet (init + *first), másrészt, az accumulate()-ot külön-külön meg kellene írnunk minden egyes bináris műveletre. A függvény objektumok olyan osztályok példányai, amelyek egyes operációkat rejtenek el (mint például az összeadás). Segítségükkel az előző példa sokkal általánosabban is megírható (9).

```
(9)   #include <vector.h>
      #include <algo.h>

      template <class T>
      class times : public binary_function<T, T, T>
      {
           public:
           T operator() ( T x, T y) const {return x * y;}
      }
      template <class InputIterator, class T, class Bin>
      T accumulate( InputIterator first,
                    InputIterator beyond,
                    T init,
                    Bin binary_op)
```

```
{
    while ( first != beyond )
    {
        init = binary_op( init, *first);
        ++ first;
    }
}
int main()
{
    duble y[5] = { 2.0, 3.0, 5.0, 7.0, 11.0};
    list<double> l( &y[0], &y[5]);

    double p = accumulate(l.begin(), l.end(), 1.0, times<double>);

    return 0;
}
```

4.5. Illesztők és memória-kezelés

Az illesztők használatával komponensek interfészét változtathatjuk meg. Ennek célja lehet, hogy egy algoritmust, amely az eredeti komponensre nem volt értelmezett, alkalmazhassunk. Másik eset, amikor egy algoritmus viselkedését szeretnénk megváltoztatni, például, az első adott tulajdonságú elem helyett az utolsót szeretnénk megtalálni.

A memória-kezelők – amelyek közül gyakran csak a default allokátort használjuk – enkapszulálják az alacsony szintű memóriakezelést, tartalmazzák a tárolási stratégiát, pointereket, objektum-méreteket, a memória lefoglalásokat és felszabadításokat.

5. Összegzés

A Standard Template Library az újrafelhasználható kód írásának egy új, programozási nyelv független, komponens-alapú megközelítése. Jelenleg az STL-nek csak C++ implementációja létezik, és ez része is a Draft C++ Standard-nek. Maga a módszer azonban implementálható más programozási nyelvekben is, amennyiben az garantál bizonyos szolgáltatásokat, mint pl. a típussal paraméterezhető osztályok ill. függvények. Komoly eredménnyel kecsegtet az STL implementációs kísérlete az egyre népszerűbb JAVA nyelven, annak valamely kiterjesztését (pl. Pizza) használva. Ebben az esetben az STL – mint módszer – kitörhetne a C++ nyelvi elszigeteltségből és nyelvek fölött álló általános algoritmus-tárrá válhatna.

Referenciák

[1] Stroustrup, B.: *The Design and Evolution of C++*, Addison-Wesley, 1994
[2] Stepanov, A., Lee Meng: *The Standard Template Library*, Hewlett-Packard Co., 1995
[3] Musser, D., Saini, A.: STL Tutorial and Reference Guide, Addison-Wesley, 1996

Az OO szemlélet architektúrális támogatása

Vég Csaba (`vega@dragon.klte.hu`)
Kossuth Lajos Tudományegyetem
Információ Technológia Tanszék

*Az előadás a számítógépes hardware-architektúra egy egyszerű, de hatékony bővítését mutatja be, amely az objektum-orientált szemlélettel és az összetett adatok kezelésével kapcsolatos típusfeladatokat automatizálja. Az **exP** („extended pointers" — kiterjesztett mutatók) névvel rövidített architektúrális támogatás a specializált (Lisp, Smalltalk, Forth, Java...) környezetek előnyeit hagyományos (C/C++, sőt, gépi kódú) programozói környezetben is elérhetővé teszi. Az **exP** két legfontosabb előnye az összetett adatok teljes és finoman szabályozható védelme, valamint egy teljes körű megoldás az automatikus tárfelszabadítással (garbage collection/dangling pointers) kapcsolatos problémákra.*

A számítógépes **architektúrák történetében** a fejlődés két jelentős mérföldköve a *strukturált paradigma* (verem, call- és return utasítások, bázis-relatív címzés, stb.) és a *többfeladatos rendszerek* (szegmentálás, processz-váltás, virtuális memória, stb.) támogatása. A támogatás bizonyos tevékenységek automatizálását jelenti. Az ***exP*** az adatszerkezetek kezelésének összes fontosabb műveletét automatizálja, ezért a fejlődés következő lépcsőfokának egy esélyes jelöltje. Az ***exP*** automatizmusait felhasználva az adatszerkezetek kezelésének típusfeladatai a helyes megadására korlátozódnak, ezért a programozó az adatok szemantikájára összpontosíthatja a figyelmét.

Ismert, hogy az algoritmizálható feladatok egy mindenfajta támogatást nélkülöző egyszerűbb architektúrán is megoldhatók. A valóság alkalmazásai esetén azonban ez a megközelítés tévútra vezet. Támogatás hiányában az alkalmazás igényelte paradigmák típusfeladatait szimulálnunk kellene, de:

- a **szimuláció** *több erőforrást igényel* (pl. lassú, de az erőforrás jelentheti a fejlesztés erőforrásait is) a szimulálandó köztes lépések miatt,
- bonyolult és *nehézkes*, ezért a programozók általában megkísérlik azt megkerülni,
- *nem biztonságos*, mivel egy rosszindulatú felhasználó vagy egy hiba megsértheti a szimuláció szabályait, valamint
- *nehezen bővíthető*, mivel egyes függetlenül elkészített részek csak egy felületen keresztül, vagy egyáltalán nem kapcsolhatók a szimuláció világához.

Megfigyelhető, hogy az **objektum-orientált szemlélet** szorosan összekapcsolódik az összetett adatok kezelésével. Egyrészt, az objektum-orientált környezetek működtetéséhez elengedhetetlenek bizonyos, adatstruktúrák kezelésével kapcsolatos módszerek, például az adatokhoz kód kapcsolása (virtuális-metódus tábla), az objektum lezárása (destruktorok), szinkronizáció, stb. Ugyanakkor az objektum-orientált szemlélet, illetve a bezárás segítségével bonyolultabb adatszerkezetek használata is lehetővé vált, mivel a kezeléshez szükséges kód egy egyszerű felület mögé rejthető.

Az OO technikák elterjedése tehát egyben az összetett adatszerkezetek használatának robbanását is eredményezte. Ugyanakkor az OO szemlélet következetes alkalmazása olyan módszereket is igényel, amelyek a jelenlegi, hagyományos architektúrán csak gazdaságtalanul, azaz gyakorlati értelemben nem szimulálhatók. A két legfontosabb, nehezen szimulálható követelmény a következő:

1. Az objektumokra és azok minden részletére biztosítani kell az *egységes elérés és* a *védelem* lehetőségét: az objektumoknak egyetlen „univerzumban" kell elhelyezkedniük, ugyanakkor védhetőknek is kell lenniük, hogy az adatterületeken csak az arra jogosult felhasználók hajthassanak végre műveleteket. A jelenlegi architektúrákon nem oldható meg egyszerre a területek finoman szabályozható védelme és a globális elérhetőség. A szimuláció egyik lehetséges módja, ha a globális elérhetőség érdekében az objektumok egyetlen virtuális memóriában helyezkednek el és a védelmet a korrekten működő alkalmazás helyettesíti. Másik kézenfekvő lehetőség, ha a védelem érdekében az objektumok külön virtuális memóriában helyezkednek el. Ekkor azonban nem biztosított a globális elérés, melyet közös adatterületekkel és az arra/arról történő másolással kell helyettesítenünk.
2. Egy objektum a részterületeivel együtt egyetlen egységet alkot, akkor is, ha azok a dinamikus tárterületen helyezkednek el. Ezért bizonyos műveletek — tipikusan az objektumok törlése — igényli a *részek észlelhetőségét*: egy objektum törlése egyben minden részletének törlését is jelenti. A jelenlegi architektúrák esetén vagy az általánosabb szerkezetek használata miatt lemondunk az automatikus detekcióról és a részterületeket „kézzel", külön kód megadásával töröljük, vagy csak néhány speciális szerkezet (pl. objektumok tömbjének, mutató-térképek) használatát engedélyezzük.

Az ***exP*** beépített, natív megoldást kínál a fenti problémákra, és az OO szemlélettel kapcsolatos több kisebb követelményre. Ezért az ***exP*** segítségével megvalósítható egy, az *OO szemlélethez minden részletében illeszkedő architektúra.*

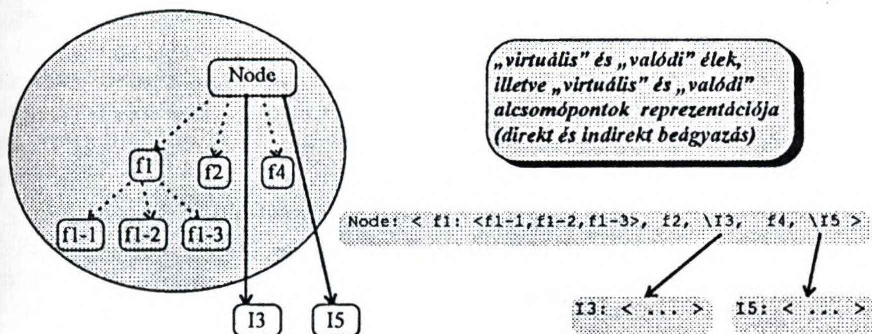
Az ***exP*** **támogatása** a következő fő területekre koncentrálódik:

- *Egyetlen virtuális memóriában* megoldja az adatok teljes és finoman szabályozható *védelmét.*
- Teljes megoldást nyújt a *„szemétgyűjtögetés"* (garbage collection) és a *„csellengő mutatók"* (dangling pointers) problémáira. Helyesen megadott tetszőleges adatszerkezet automatikusan törlődik, ha az arra vonatkozó összes hivatkozás megszűnik.
- Adathoz kód is kapcsolható (objektum-orientáltság) és ebben az esetben az *adat kódként* is hívható.
- *Kód* megadható *adatként*: ebben az esetben az első hivatkozás meghívja a kódot, hogy az előállítsa az adatot (osztott számítások).
- *Szinkronizáció*, amely az összetett adatok párhuzamos kezelésének feltétele.

Fogalmi modell

Az ***exP*** módszerei egy **fogalmi modellhez** illeszkednek, mely az összetett adatok irányított gráfként történő reprezentációján alapul. A modell a tárolt gráf éleit két

típusba sorolja. Ha az élből elérhető részgráf fix-szerkezetű, azaz a benne elhelyezkedő csomópontok száma és típusa nem változik, akkor azok a csomóponthoz rendelhetők, a csomópontot jelentő memóriaterületen tárolhatók és egy „virtuális élen" keresztül érhetők el. Változó számú vagy szerkezetű adatokat célszerű külön memóriaterületen tárolni. Ekkor a csomópontot jelentő területen „valódi élként" csak egy hivatkozást helyezünk el az alcsomópontra, így annak mérete és szerkezete is változtatható. Tehát egy csomópont *alcsomópontja* vagy a csomóponthoz rendelt terület egy (összefüggő) részterületét jelenti („virtuális él" esetén), vagy a hivatkozáson (mint „valódi élen") keresztül elérhető adatterületet.



Az *exP* architektúra esetén minden program rendelkezésére áll néhány hivatkozás, mely egy kiinduló csomópontként tekinthető. A program kizárólag az ezen hivatkozásokból elérhető részgráfot használhatja fel. Egyes programok így ugyanabban a memóriában akár teljesen független módon is működhetnek. Egy program az általa elérhető gráfnak egy *korlátozásnak* nevezett művelettel bizonyos, kevesebb jogosultságú részét képezheti és azt átadhatja valamely másik programrésznek. (Az el nem ért csomópontok tekinthetők a használat teljes korlátozásának.)

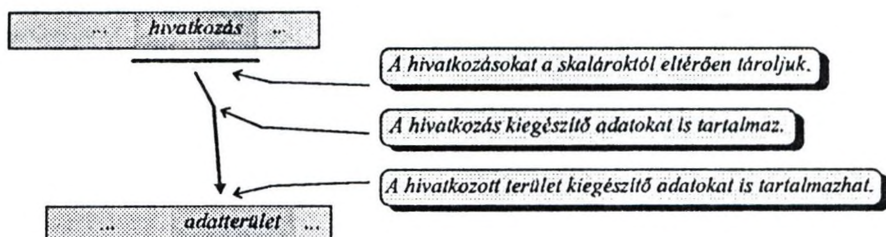
A hagyományos architektúra a („valódi") éleket mutatóként, a tömbként tekintett memória indexeiként reprezentálja. Az *exP* a szám-jellegű adatoktól eltérő módon tárolt, *hivatkozások*nak nevezett „kiterjesztett mutatókat" használ (*exP* = *extended pointers*: innen származik az elnevezés), melyek alkalmasak arra, hogy a programok csak az általuk jogosultan elérhető területeken hajtsanak végre műveleteket. Az eltérő tárolás egyben azt is lehetővé teszi, hogy az adott memóriaterületen elhelyezkedő tényleges (memóriaterületre vonatkozó, azaz nem üres) hivatkozások azonosíthatók, mely segítségével a csomópontból elérhető részgráf bizonyos (tipikusan a részgráf másolásával, illetve törlésével kapcsolatos) műveletei automatizálhatók.

Alapvető technikák

Általános értelemben az *exP* három módszer együttesét jelenti:

- a hivatkozásokat a skalároktól eltérően tároljuk („*megjelöljük*", „aláhúzzuk"),
- a *hivatkozások* a mutatott terület címén kívül *kiegészítő információkat* is tartalmaznak, valamint

- a *mutatott területen* annak használatával kapcsolatos *kiegészítő információk* is elhelyezhetők.



Védelem

A memóriában elhelyezkedő összetett adatok egy gráfot alkotnak. Ez a szemlélet a hivatkozásoknak egy, a hagyományos pointerektől eltérő reprezentációjához vezet el. Egy gráf egy csomópontja a memória egy adott területe, melyen vagy a (virtuális) alcsomópontok részterületei, vagy (valódi alcsomópontok esetén) a független területekre mutató hivatkozások helyezkednek el. Egy memóriaterület megadható a kezdőcímével, a hosszával, illetve az elérés jogaival.

Szegmens-jellegű hivatkozások

A mutatott terület hosszával, illetve a terület használatának jogaival kiegészített pointerek már nem csak a memória egy pontjára, hanem a koncepcionális szint egy adott csomópontjára is hivatkozhatnak.

Ha az architektúra tiltja az abszolút címzést és csak ilyen hivatkozások használatát engedélyezi, akkor megvalósítható az adatok finoman szabályozható védelme. Egy csomópont alcsomópontjának a kiválasztása vagy a (mutatott cím növelésével, illetve a hosszadat csökkentésével) egy részterület kiválasztását jelenti, vagy a terület egy hivatkozásán történő továbblépést.

Az ilyen típusú hivatkozások hasonlóak a szegmensekhez (pl. az Intel 80x86 szegmenseihez). Minden hivatkozás szegmensként működik, de a szegmens-táblák használatának nehézkessége és korlátozásai nélkül.

Szegmens-jellegű mutató:
Mutatott cím
Mód attribútumok:
olvasható
írható
futtatható
Terület vége

A hivatkozások védelme

Az összetett hivatkozások esetén még felmerülhetnek problémák, mivel például egy programozó adott számértéknek a hivatkozás területére történő tárolásával felülírhatja a védelmet. Az *exP* a tényleges (nem „Nil") hivatkozásokat a memóriában „aláhúzza" és azokat a skalároktól eltérően kezeli, így például tiltott egy „aláhúzott" tényleges hivatkozás területére egy skalár értéket tárolni. Az „aláhúzás" egy másik előnye, hogy így az adott memóriaterületen elhelyezkedő tényleges hivatkozások azonosíthatók, így automatizálható az összetett adatszerkezetek törlése vagy másolása. Ez a módszer feleslegessé teszi például az OO nyelvekben használt destruktorok és másoló-konstruktorok nagy százalékát.

Az aláhúzás egy egyszerű implementációja esetén a memóriát kiegészítjük egy (kilencedik) bittel, mely skalárok esetén az alapértelmezés szerinti 0 értékű. Aláhúzott hivatkozás esetén a kezdőcímhez tartozó bit alapértelmezés szerinti (0), a többi cím (bájt) esetén pedig annak komplementere (1 értékű). Így az egymás után elhelyezkedő hivatkozások kezdete is azonosítható.

Memóriaterület törlése egy ismétlődő gépi utasítással történhet, amely a magasabb memóriacímektől az alacsonyabb címekig végigellenőrzi a területet és minden, csak összetett módon törölhető aláhúzott hivatkozás esetén a hivatkozás címét egy gépi kivétellel a kezelőprogramnak adja át.

Implementációk

A védelem eddig vázolt megoldásai nem ismeretlenek a számítógépes architektúrák történetében. Az implementációk a **strukturált memória**, a *kiegészített architektúra* (tagged-architecture), *őrzött mutatók* (guarded pointers), illetve a *képesség-szerinti címzés* (capability-based addressing) címszavak alatt találhatók meg.

Már az első számítógépek idején is készítettek olyan architektúrákat, amelyeknek minden egyes gépi-szavát a szó típusára utaló egy-két bittel kiegészítették. A kiegészítő (tag-) biteket külön utasításokkal lehetett beállítani, illetve tesztelni.

A kiegészítő bitek a CPU és a rendszer-szoftver közötti szerves kapcsolódás eszközei lettek az MIT kutatásaiból kiinduló **Symbolics** nevű Lisp gép központi egységében. A mesterséges intelligencia támogatására tervezett Symbolics memóriája minden egyes 32 bites szót további két bittel egészített ki, amelyekkel például a pointerek is azonosíthatók voltak. A piac nem érdeklődött a kísérlet iránt, ezért a processzort először a Lisp-et támogató co-processzorrá dolgozták át, de ez sem vált sikeressé, így a Symbolics gépet fejlesztő cég csődbe ment.

A Smalltalk, illetve az objektum-orientált technológia sikere az **Intel** céget arra ösztönözte, hogy a technológiára épülő és egyben azt támogató architektúrát valósítson meg. Az Intel a '80-as évek elején elkezdte fejleszteni az **iAPX 432** kódnevű objektum-orientált processzort, amelyben több Smalltalk elvet is megvalósítottak. A 80286-os processzort egy olyan hídként képzelték el, amely a 8086-osok világából átvezet a tisztán az új paradigmára épülő technológiába. A 80286-ba így több Smalltalk-os koncepció is beépült, így például a szegmensek leíró táblái a Smalltalk objektum-tábláit implementálják. Az iAPX 432 kísérlet megbukott, mivel a processzor túlságosan

bonyolult és drága volt, a működési sebessége pedig csak a tizedét érte el a rivális Motorola 68000 CPU sebességének.

Az iAPX 432 utóéletét egyrészt az **Intel 960**-as RISC sorozatának **MM/MX** kódjelű speciális, „kiterjesztett architektúrájú" processzorai jelentették, másrészt az Intel és a Siemens rövid életű közös, **BiiN** nevű projektjét (1985) majd cégét (1988). A BiiN processzorokkal hibatűrő és biztonságos párhuzamos rendszert szándékoztak kifejleszteni. A két kísérleti BiiN processzor 33 bites szavakat használt. A kiegészítő bit jelezi, ha a szó pointert tartalmaz, és ekkor a szóra bizonyos műveletek nem engedélyezettek. A projekt nagy mennyiségű pénzt (kb. 300M$-t) emésztett fel, ugyanakkor bizonytalan volt a költségek megtérülése, ezért 1989 végén a teljes céget felszámolták, az eredményeket csak néhány katonai területen (repülőgépek fedélzeti számítógépében) alkalmazták.

A védett pointerek egyes kísérleti architektúrákban még ma is megjelennek. Érdemes megemlíteni az MIT **M-Machine** tömb-processzorát, amelyben a védett pointerek egy közös globális címtartomány elérésének az eszközei. Az M-Machine 64 (+1) bites pointerében 54 bit határozza meg a mutatott címet, 6 bit (logaritmikus kódolásban) az adatterület 2 hatvánnyal megadható hosszát, és további 4 bit az elérés jogosultságait. A kiegészítő bit jelzi, hogy a 64 biten érvényes pointer helyezkedik el, így az védett a felülírással szemben.

A sikertelenségek okai

A strukturált memóriák története csőd-sorozatok és látható eredmények nélkül elköltött 100M$-ok története. Mi lehetett a sikertelenségek oka?

Megfigyelhető, hogy az eddigi kísérletek mind valamely *túlspecializált* architektúrával kapcsolódtak össze. A Symbolics Lisp gép csak akadémikus körökben váltott ki érdeklődést, míg az „ipar" megmaradt a hagyományos, de hatékonyan működtethető programozási környezetek mellett. A túlspecializáltság visszahat a processzor belső *bonyolultságára* és működési sebességére is, például az iAPX 432 lassúságának egyik valószínűsíthetően lényeges tényezője, hogy az utasítások nem feltétlenül byte-határon kezdődtek és változó bit-hosszúságúak voltak, ugyanakkor nem volt ritka a viszonylag hosszú, több szónak megfelelő bit-számú utasítás sem. Harmadik okként megfogalmazhatjuk, hogy a strukturált memóriák kísérleti implementációi legfeljebb *kezdetleges szemléleti modellt* használtak, ezért a megoldások részlegesek és esetlegesek voltak. Magának az objektum-orientáltságnak az alapelvei és az alapvető követelményei sem voltak még tisztázva.

A korábbi sikertelenségek okai alapján megfogalmazhatók azok a követelmények, melyeket egy strukturált memóriára épülő architektúrának teljesítenie kell.

Az architektúra

- támogassa a *hagyományos programozói környezetek*et,
- viszonylag *egyszerű* huzalozással megvalósítható és megfelelő sebességgel működtethető legyen,
- támogatása egy *szemléleti modellre épül*ve az összetett adatok kezelésének széles spektrumát ölelje fel.

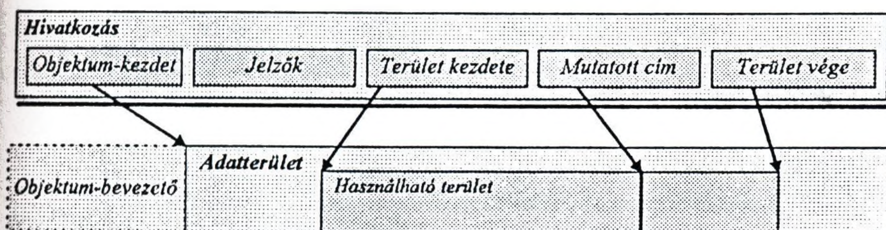
Kiegészítések

Az ***exP*** az eddigi részleges megoldások helyett teljes körű támogatást nyújt az összetett adatok kezelésére.

A szegmens-jellegű pointerek nem teljesen kompatibilisek a hagyományos programozási szemlélettel. Így például egy pointer csökkentése (p--), azaz a mutatott címnek egy tömbön belüli léptetése nem megengedett. Az ***exP*** a hivatkozást a **használható terület kezdőcímével** is kiegészíti, így a mutatott cím a terület kezdőcíme és a terület vége közötti tetszőleges címre állítható. A kezdőcím az alcsomópontok által alkotott kollekció az első elemére, a terület vége pedig az utolsó elem utáni címre mutat. A hivatkozás a kollekción belüli adott elemre vonatkozik, így a hivatkozással végigvehetjük a kollekció összes elemét.

Jól ismert módszer, hogy az adatok automatizált kezelése érdekében a memóriaterületekhez kiegészítő adatokat kapcsolunk. Ilyenek például az objektumhoz tartozó virtuális módszerek táblájának a címe (VMT). Az ***exP*** a hivatkozásokat kiegészíti a **terület tényleges kezdetének**, más néven az **objektum-kezdetnek** a címével, mely előtt a processzor által automatizáltan kezelt adatok helyezkedhetnek el (például hivatkozás-számláló, vagy a terület kezelője). A területhez tartozó adatok a a terület tényleges kezdete előtt helyezkednek el, így azonos módon kezelhetők a kiegészítő adatokkal rendelkező és nem rendelkező területek.

Az ***exP*** harmadik kiegészítése, hogy az alapvető jogosultságok mellett még **további**, az összetett adatok automatizált kezelésével kapcsolatos **jelzőbiteket** is bevezet.



Egy exP implementáció

A következőkben egy 32 bites architektúrán megvalósítható ***exP*** implementációt mutatunk be.

Általános hivatkozás esetén a logikai szerkezet a következőképpen reprezentálható: Az *Objektum-kezdet*, a *Mutatott cím* és a *Terület vége* egy-egy 4 byte-os abszolút cím. A *Jelzők* egy 2 byte-os (16 bites) bitmező. Mivel az adatterület elején általában csak egy viszonylag rövid résztерület használatának tiltása szükséges, ezért ezt egy 2 byte-os értékkel adjuk meg, így a teljes hivatkozás (a 2 hatvány) 16 byte-on tárolható. Ekkor a mutatott címnek az *Objektum-kezdet+Fejrész*re vagy az után, illetve a *Terület vége* előtti címre kell mutatni (O+H ≤ P < E).

Ha szükséges, hogy nagyobb terület magasabb címének a részterületét is képezzük (úgy, hogy csak annak használatára adunk jogosultságot), akkor az megoldható egy másik hivatkozás-típussal, melyben a terület hosszát tároljuk 2 byte-on és a Terület

kezdete egy 4 byte-os abszolút cím. (Ekkor: B ≤ P < O+L .) A huzalozás a Jelzők egy bitje alapján automatikusan észlelheti a hivatkozás típusát, illetve egy művelet utáni tárolás esetén automatikusan konvertálhat a két típus között.

Valószínűleg csak elenyésző azon esetek száma, amikor mindkét típus tartománya kevésnek bizonyul.

Rövid hivatkozás: Az esetek többségében elegendő egy speciális hivatkozás-típus, mely rövidebb területen (fele akkora területen, 8 byte-on) tárolható. Ez mindössze a kétszerese a hagyományos abszolút-pointer méretének. A rövid hivatkozás akkor alkalmazható, ha a terület viszonylag rövid (≤ 64Kbyte), nem kívánjuk az elejének használatát korlátozni, a terület kezdete egybeesik a mutatott címmel, illetve a mutatott címet nem kívánjuk a területen belül léptetni. A rövid hivatkozások például láncolások (láncolt listák, fák, stb.) esetén használható.

Az exP implementáció hivatkozás-típusai

Általános hivatkozás:

O:	***Objektum-kezdet:***	4 byte-os abszolút cím
M:	***Jelzők:***	2 byte-os (16 bites) bitmező
H:	***Terület-fejrésze:***	2 byte-os offszet
P:	***Mutatott cím:***	4 byte-os abszolút cím
E:	***Terület vége:***	4 byte-os abszolút cím

Általános (részterület) hivatkozás:

O:	***Objektum-kezdet:***	4 byte-os abszolút cím
M:	***Jelzők:***	2 byte-os (16 bites) bitmező
L:	***Terület-hossza:***	2 byte-os hossz-adat
P:	***Mutatott cím:***	4 byte-os abszolút cím
B:	***Terület-kezdete:***	4 byte-os abszolút cím

Rövid hivatkozás:

O:	***Objektum-kezdet:***	4 byte-os abszolút cím
M:	***Jelzők:***	2 byte-os (16 bites) bitmező
L:	***Terület-hossza:***	2 byte-os hossz-adat

Objektum-bevezető

Az adatterület tényleges kezdete előtt, az objektum-bevezető területén elhelyezhetők az adatterület használatára vonatkozó globális, egyedi hivatkozásoktól független információk.

Objektum-bevezető

Tárkezelés adatai	*Kezelő*	*Gyenge hivatkozás-számláló*	*Hivatkozás-számláló*	*Jelzők*	***Objektum-***

Az objektum-bevezető adatai az első Jelzők kivételével opcionálisak, azok létezésére a Jelzők egy-egy bitje utal. A **jelzők** a következők:

- ***Empty***: jelzi, hogy a terület logikailag üres, de arra még hivatkoznak, így az még nem törölhető.

- *Locked*: jelzi, hogy a terület zárolva van.
- *System-lock*: jelzi, hogy a területet a rendszer zárolta.
- *Dynamic*: jelzi, hogy a terület a dinamikus tárterületen helyezkedik el, így az objektum-bevezető elején a tárkezelés adatai helyezkednek el, illetve a tárkezelőtől lekérdezhető a terület tényleges hossza.
- *Handler*: jelzi, hogy az objektum-bevezetőben szerepel a terület kezelője
- *Up-counter*: jelzi, hogy az objektum-bevezetőben szerepel a normál hivatkozás-számláló mellett a gyenge hivatkozások számlálója is.
- *Reference-counter*: jelzi, hogy az objektum-bevezetőben szerepel (normál) hivatkozás-számláló.

Jelzők

Az *exP* implementáció a következő jelzőket használja:

- **Olvasható**: a területen elhelyezkedő értékek felhasználhatók.
- **Írható**: a területen elhelyezkedő értékek módosíthatók.
- **Futtatható**: a terület címeire a vezérlés átadható.
- **Rendszer**: a hivatkozott terület a rendszerhez tartozik. Futtatható esetben a vezérlés átadásakor a processzor rendszer-állapotba vált át.
- **Objektum-bevezető**: Az objektum-kezdet előtt van objektum-bevezető.
- **Kezelő**: Meghívható az adatterület kezelője.
- **Tulajdonos**: A terület tulajdonosa, nem csak hivatkozik arra. Ellenkező esetben a hivatkozás gyenge hivatkozásnak számít.
- **Zárolható**: a terület zárolható. A zárolt területre történő nem-rendszer hivatkozás egy kivételt eredményez, amely várakozási állapotba válthatja át a hivatkozó folyamatot a zárolás törléséig.
- **Kulcs**: a zárolt területhez van kulcsa, azt elérheti. Kivétel: ha a hivatkozás folyamata nem rendszer állapotban van és az adatot a rendszer zárolta.
- **Módosítás**: a terület módosítása előtt egy kivételt generálódik.
- **Kapu**: a terület a vezérlése alá tartozik. A terület törlése esetén a terület hivatkozásainak memóriaterületeit akkor is törli, ha azokra volt még (egyszerű) hivatkozás.
- **Távoli cím**: Távoli hivatkozás esetén a hivatkozás első felhasználásakor (például egy gépi kivétel segítségével) a hivatkozott kód végrehajtódik és azt a hivatkozást feltölti egy konkrét címmel.
- **Indirekt mód**: Indirekt (közvetett) hivatkozás. Az indirekció egy közbülső hivatkozáson keresztül történő közvetett hivatkozást jelent. Az indirekció egy hivatkozás módosítható elérése, amely más aritmetikával léptethető.

Műveletek

Hivatkozásból újabb hivatkozás képezhető a következő módokon:

- Tovább léphetünk a hivatkozott adatterület egy hivatkozásán.
- Hivatkozás a hivatkozott adatterületen belül más címre mutathat.

- Hivatkozás jellemzői korlátozhatók egy kevesebb jogosultsággal rendelkező hivatkozássá. Például képezhető egy hivatkozott terület kevesebb jogosultságú részterületére vonatkozó hivatkozás.
- Hivatkozás kiterjeszthető egy bővebb jogosultságokkal rendelkező másik hivatkozás alapján.
- Hivatkozásból képezhető egy más felhasználási módú hivatkozás.

Felhasználhatóság és hatékonyság

A három alapvető technikát jelentő keretet egy ***exP*** implementációnak úgy kell kitölteni, hogy a gráf-kezelés leggyakrabban használt műveletei automatizálhatók legyenek.

Az ***exP*** bővítések olyan műveleteket automatizálnak, melyeket bonyolult adatszerkezeteket következetes használata esetén a programoknak egyébként is tartalmazniuk kellene. Ezeket a műveleteket az ***exP*** természetesen gyorsabban és ami fontosabb, teljes körűen valósítja meg. Ugyanakkor olyan automatizmusokat is szolgáltathat, melyek teljes körű szimulációja (kóddal való megvalósítása) már gazdaságtalan lenne, ezért a gyakorlatban fel sem merül.

Az ***exP*** bővítéseivel a hagyományos processzorok is kiegészíthetők. Az ***exP*** legnagyobb előnye azonban az, hogy arra egy olyan processzor építhető, amely az objektum-orientált szemlélet követelményeit teljes mértékben megvalósítani képes.

Algráf automatikus törlése

Ha egy csomópontra csak egyszer hivatkozunk, akkor nem szükséges hivatkozás-számlálás. Hivatkozás-számláló használatával megvalósítható egy irányított körútmentes gráf (DAG) automatikus törlése. Az ismert algoritmussal a területre vonatkozó újabb hivatkozás képzése növeli, egy hivatkozás-érték megszűnése pedig csökkenti a számlálót. Ha a számláló a 0 értékre csökken, akkor az ***exP*** egy kivételt generál, mellyel a terület felszabadítható.

Az ***exP*** körút (visszahivatkozás) esetén a következő megoldási módokat kínálja:

— *Statikus*, vagy *strukturális körút*nak nevezzük az algráfban az alsóbb szintről történő visszahivatkozást, ha a visszahivatkozó csomópont soha nem kerülhet a visszahivatkozott csomópont fölé. Erre általában egy fa olyan csomópontja esetén van szükség, amely egy „korábbi" csomópontra hivatkozik vissza. Ekkor a visszahivatkozott csomópontnak a hivatkozás-számláló mellett egy gyenge hivatkozás-számlálóval is kell rendelkezni, a visszahivatkozást pedig „nem tulajdonos" módon kell korlátozni, amelyet a gyenge hivatkozás-számláló fog nyilvántartani.

— *Dinamikus körút*nak nevezzük, ha a körút csomópontjainak elérési sorrendje változhat. Ezek a csomópontok logikailag egy közös szinten elhelyezkedő elemek

láncát reprezentálják (például egy- vagy kétirányú ciklikusan láncolt lista esetén). Ebben az esetben a közös szint forrásául szolgáló csomóponton el kell helyezni a lánc aktuális kezdetére mutató hivatkozásra egy kapu hivatkozást, amely törlése kikényszeríti a lánc első elemének (adatterületének) és így a teljes láncnak a törlését.

Egy kapu-hivatkozásnak vagy az összes nem gyenge hivatkozásnak a törlése a terület törlését eredményezheti. Maga a terület azonban nem szabadul fel, ha arra még vannak élő hivatkozások. Az *exP* az Üres jelzővel jelzi, hogy a terület logikailag már nem létezik, az él már nem mutat létező csomópontra.

A csomópont törlése egy vagy több kivételen keresztül jelentheti a terület tényleges hosszának beállítását (ha az a dinamikus tárterületen helyezkedik el), az adatterülethez kapcsolt kezelő hívását (ha van), majd (ha az nem növelte meg a hivatkozás-számlálót) a területen elhelyezkedő hivatkozások törlését, melyet a terület felszabadítása követhet, ha a terület a dinamikus tárterületen helyezkedik el.

Lehetőségek

Az *exP* architektúra hatékonyságát a következő néhány példa szemlélteti.

Az *exP* architektúrával *egységes virtuális tár* valósítható meg, ha minden programrész csak azokat a memóriaterületeket éri el, amelyekre rendelkezik hivatkozással. A hivatkozások alapkészlete meghatározza a gráf adott részgráfját, az elérhető/módosítható/futtatható területeket. Leegyszerűsödik a programrészek közötti kommunikáció: egy folyamat által termelt adatszerkezet (akár készítés közben) átadható egy másik folyamatnak. Például egy adatokat termelő processz az adatokat egy láncolt listában tárolja és adja át a fogyasztó processz számára. Az utolsó elemet zárolja, így a fogyasztó automatikusan várakozás-állapotba vált, ha azt eléri. Újabb elem hozzáfűzése esetén a termelő a következő elemre lép, mellyel a zárolást is törli, így a fogyasztó az előző elemet feldolgozhatja. Ha az elemek rendelkeznek hivatkozás-számlálóval, akkor a fogyasztó következő elemre történő lépése esetén a már feldolgozott elem automatikusan törlődik. Ha a termelő később újra fel kívánja dolgozni az adatokat, akkor elegendő annak első elemére hivatkoznia (így megnöveli a hivatkozás-számlálót, ezért a lánc nem törlődik).

Az *exP* architektúra az *összetett adatok kezelését* a szám-jellegű skalárok kezeléséhez hasonlóvá *egyszerűsíti*. Egy skalár `a = b` értékadás esetén az `a`-ban tárolt korábbi érték törlődik, és annak helyére `b` értéke kerül. Az *exP* `p = q` mutatók közötti értékadása esetén ugyanez történik, bár lehet, hogy `p` és `q` egy-egy láncolt lista, vagy fa. Ekkor `p` korábbi értéke törlődik, ami a `p` által mutatott adatszerkezet hivatkozás-számlálójának csökkenését, annak `0` értéke esetén az adathoz kapcsolt záró függvény (virtuális destruktor) hívását, a terület törlését és a használt dinamikus tárterület felszabadítását is eredményezheti. Ha a `q` többször hivatkozott struktúra, akkor a törlés előtt megnövekszik annak (megfelelő) hivatkozás-számlálója, így például ha a `p` egy fa, akkor átléphetünk annak egy részfájára úgy, hogy a fa többi része automatikusan törlődik, ha arra más nem hivatkozott.

A dinamikus tárterületen elhelyezkedő, többször hivatkozott adatszerkezetre a legegyszerűbb példa egy egyszerű karaktersorozat. Mivel a kiterjesztett mutatók ismerik a mutatott terület hosszát, ezért az *exP* architektúrákba gyakorlatilag be van építve a

sztringek kezelése. A sztringek hossza lekérdezhető, annak egy rész-sztringje egy függvénynek módosításra átadható (`substr(x,1,2)=...`). A dinamikus tárterületen elhelyezkedő, már nem hivatkozott sztringek területe pedig automatikusan felszabadítódhat.

Az ***exP*** esetén *adatszerkezet (objektum) is hívható függvényként*. A kiterjesztett mutatók meg tudják különböztetni, hogy a mutatott terület adatokat vagy végrehajtható kódot tartalmaz. Ha a hívás címe nem végrehajtható-, hanem adatterület és ahhoz kezelő van kapcsolva, akkor közvetve a kezelő hívódik meg, az adatterülettel, mint speciális paraméterrel (objektumot azonosító változóval (`*this`)). Egy függvény így az aktuális attribútum-értékeit tárolni képes objektummá cserélhető úgy, hogy a hívó programban még a gépi kód szintjén sem vehető észre a változás.

A *távoli hivatkozás* lehetőséget ad az adatszerkezetek késleltetett előállítására. A hivatkozás-érték első felhasználása egy kivételen keresztül a mutatott függvényt hívja meg, amely a hivatkozást a megfelelő, például háttértárolóról betöltött adatszerkezetre állítja. Az adatszerkezetre történő egyszerű hivatkozás így automatikusan, gépi szinten sem észrevehető módon rendel a mutatóhoz konkrét adatszerkezetet. Például egy feldolgozóprogramnak egy távoli hivatkozás átadható egy láncolt lista első elemének címeként. A hivatkozás-érték első felhasználása egy kivételen keresztül előállítja az első elemet és a következő elem címeként szintén egy távoli címet állít be. A következő elemre történő lépés újra kivált egy kivételt, mely előállítja a következő elemet. Hivatkozás-számlálás esetén a már feldolgozott elem automatikusan törlődhet, ha arra nem történt több hivatkozás. A technika természetesen kombinálható az adatterület (objektum) hívásával, így távoli hivatkozásként megadható egy kezelővel rendelkező adatterület is.

Az *indirekció* jelzése esetén a hivatkozás-érték felhasználása előtt a processzor végrehajt még egy indirekciós lépést. A technikával gépi szinten is azonos módon kezelhetők az egyszeres és a kétszeres (indirekt) hivatkozások. Ha a processzor támogat úgynevezett navigációs lépéseket, akkor a megfelelő módon megírt kereső alprogramok azonos módon kezelhetik az érték-jellegű (egyszeres), illetve a módosítható jellegű (kétszeres) hivatkozásokat. Hagyományos eszközökkel ez természetesen két alprogramot igényel, pl. C-nyelven: `for(p=L; p; p=p->N)`, illetve `for(p=&L; *p; p=&(*p)->N)`

A *módosítás-igényének jelzése* például copy-on-write jellegű adatszerkezetek automatizált kezelését teszi lehetővé. Vegyünk például egy olyan hivatkozás-számlálós objektumot amely együttesen egy leíró és egy kezelő, azaz önmaga metaobjektuma is. A leíró megadása után az első felhasználás a megfelelő módon beállítja a kezelőt. Hivatkozás-értékadással az objektumra más helyen is hivatkozhatunk. Ha ekkor beállítjuk a módosítás-jelzőt, akkor a leíró értékei ugyanúgy felhasználhatók. Módosítás esetén azonban (egy kivétel segítségével) megvizsgálható, hogy más is hivatkozik-e erre az objektumra és ebben az esetben az objektumról egy másolat készíthető (mely esetén már töröljük a módosítás-jelzőt), így a leíró korábban beállított értékei egy másik objektum készítésekor is felhasználhatók.

Irodalomjegyzék

[1] *Donald E. Knuth:* The art of computer programming. Volume 1. Fundamental Algorithms (2nd ed.). *Addison-Wesley, 1981.*

[2] *Silberschatz - Galvin:* Operating Systems Concepts (4th ed.). *Addison-Wesley, 1994.*

[3] *Richard Jones - Rafael Lins:* Garbage Collection. Algorithms for Automatic Dynamic Memory Management. *John Wiley & Sons, 1996.*

[4] *David S. Wise:* Design for a Multiprocessing Heap with On-board Reference Counting. in: Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science #201. *Springer-Verlag, 1985.*

[5] *Paul Helman - Robert Verof - Frank M. Carrano*: Intermediate Problem Solving and Data Structures. Walls and Mirrors. (2nd ed.). *The Benjaming/Cummings Publishing Company, Inc. 1991.*

[6] *Jeff Alger*: Secrets of the C++ Masters. *Academic Press, 1995.*

[7] *James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen:* Object-Oriented Modeling and Design. *Prentice-Hall, 1991.*

[8] Symbolics Technical Summary. *<http://www.lavielle.com/~joswig/symbolic-computing.html>*

[9] The Online Symbolics Museum. *<http://www.brightware.com/~rwk/symbolics/>*

[10] BiiN. *<http://nhse.npac.syr.edu/hpccsurvey/orgs/biin/biin.html>*

[11] i960 Processor. *<http://www.intel.com/design/i960/>*

[12] The M-Machine Project *<http://www.ai.mit.edu/projects/cva/cva_m_machine.html>*

TERVEZÉS, IMPLMENTÁLÁS ÉS RENDSZERFELÜGYLET SOFTWARE THROUGH PICTURES CASE ESZKÖZZEL ÉS FORTÉ-VAL

Frigó József, frigo@inf.bme.hu
Kelen András, 100263.634@compuserve.com
Szomolányi Márton triad@mail.datanet.hu
Triad Számítástechnikai és Szolgáltató Kft.

0. Bevezetés

Az előadás két együttműködő, integrált rendszer, az StP és a FORTÉ szolgáltatásait, használatát mutatja be.

Mindkét eszköz nagyméretű rendszerek megvalósításához való, amelyeken egyidejüleg sok elemző-tervező és fejlesztő dolgozik, így különös jelentőséget kap a biztonság, az együttműködés támogatása, a verziókezelés, a működő rendszer üzembiztonsága, stb.

Az előadás azt kívánja igazolni, hogyan lehet a vállalati modelltől az üzemelő rendszer felügyeletéig végigkövetni egy rendszer életciklusát.

Az előadás élő bemutató, a két rendszer egyes fontos jellegzetességeinek kiemelésével.

IQSOFT

Elektronikus áruház az Interneten

Egyed László - Nick János, IQSOFT

Napjainkban bármerre is járunk, az Interneten biztosan találkozunk a gomba módra szaporodó elektronikus üzletek, áruházak, bevásárló központok valamelyikével. Ma ez az Internet világ egyik legdinamikusabban fejlődő területe. Az üzleteket megvalósító alkalmazások azonban még korántsem használják ki a rendelkezésre álló lehetőségeket.

Amikor az ember ma vásárol, a termék kiválasztásának folyamata, mondhatjuk nyugodtan, a feje tetején áll. Ugyanis nem azt vizsgálom, hogy milyen áru lenne számomra a legjobb, hanem az ismert, vagy elérhető kereskedők kínálatából próbálom kiválasztani az ebben a lényegesen szűkebb körben a számomra megfelelőt. Ennek a talpra állítása érdekében készült el az a koncepció, amelynek lényege, hogy – a széles elérhetőség és nyilvánosság érdekében – az Interneten olyan gyűjtő adatbázisokat kellene létrehozni, amelyekben egy-egy piaci szegmens szereplőinek egy hangsúlyos (távlatilag döntő) része található meg, és az általuk kínált teljes árukészletből választhatom ki a számomra legmegfelelőbbet, miközben folyamatosan kiegészítő információkat is kérhetek a rendszertől. A kereskedő csak a végén kerül képbe, amikor megtaláltam a számomra szükséges árut (vagy szolgáltatást, stb.), akkor kérek egy listát arról, hol is találhatom meg azt, amit választottam. Ebből a kereskedőlistából kereshetem meg az adott árura vonatkozó, számomra legelőnyösebb ajánlatot, akár az ár, akár az egyéb járulékos szolgáltatások, garancia, földrajzi közelség, stb. alapján.
A választás egy fa struktúra mentén történik, egészen addig, amíg eljutunk egy egységes tulajdonságrendszerrel rendelkező árucsoporthoz. Ekkor a további szűkítés a különböző tulajdonság-paraméterek megadásával történik.

Két fontos további szempont, hogy egyrészt a rendszer soha ne adjon nulla eredményt, ezért mindig csak egy létező választék alapján adhatók meg a paraméterek, s ezek mindig megfelelnek a korábbi szűkítések szabta korlátoknak is. Másrészt, el kell kerülni a végtelen listák letöltődését, ezért a rendszer folyamatosan jelzi a találatok számát, illetve azt, hogy esetleg további szűkítésre van szükség. Ezért, ha valaki jól használja a rendszert (s ebben a rendszer sok segítséget ad), akkor mindig áttekinthető listát kap a számára megfelelő árukról.

Természetesen a kereskedők számára továbbra is biztosítani kell az egyedi megjelenési lehetőséget, amikor a vásárló csak az adott kereskedő termékeiből tud választani.

Az ismertetett elképzeléseket megvalósító elektronikus áruház igen bonyolult és összetett alkalmazás, amely komoly tervezési és megvalósítási megfontolásokat igényel. Tekintettel kell lenni arra a tényre is, hogy egy erősen piac-orientált környezetben működő alkalmazás esetén az igények gyorsan változnak és az esetleges változtatásokat rövid idő alatt kell elvégezni.

Az objektumorientált technológia az az eszköz, amely biztosíthatja számunkra a hatékony és gyors alkalmazásfejlesztést, valamint az igények változására való gyors reagálást. A technológia támogatja a fejlesztési életciklus minden fázisát az elemzés és tervezéstől, a programozáson keresztül az adatbáziskezelésig.
A bonyolult kapcsolatok hatékony kezelésére a hagyományos relációs adatbáziskezelők helyett célszerűbb objektumorientált adatbáziskezelő alkalmazása, valamint a lekérdezések gyors végrehajtása érdekében az alkalmazási logika C++ nyelven történő programozása. Az objektumok közvetlenül (transzformáció nélkül) az adatbázisban tárolódnak, ami megkönnyíti és felgyorsítja a programozók munkáját.

SzIBilla

Az MNB Statisztikai-célú Információ Bázisa

Bokor Judit, Magyar Nemzeti Bank, Statisztikai Fõosztály
Németh Miklós, IQSOFT Rt., Objektumtechnológiai Osztály

Összefoglaló

A Magyar Nemzeti Bank megrendelésére a bank Statisztikai Fõosztálya és az IQSOFT RT. szakemberei közös fejlesztésbe kezdtek egy jegybanki elemzési-, statisztikai-célú információs adatbázis és rendszer kialakítása céljából. A teljes feladat elvégzése több éven át fog tartani, és várhatóan három vagy négy projekt keretében készül el. Az elõadás a feladat célkitûzéseirõl, a tervezés, a fejlesztés és a mûködtetés technológiájáról, valamint az elsõ projekt tapasztalatairól, eredményeirõl számol be. Az elõadók a projekt résztvevõi, akik hitelesen tudják tolmácsolni a projekt tapasztalatait.

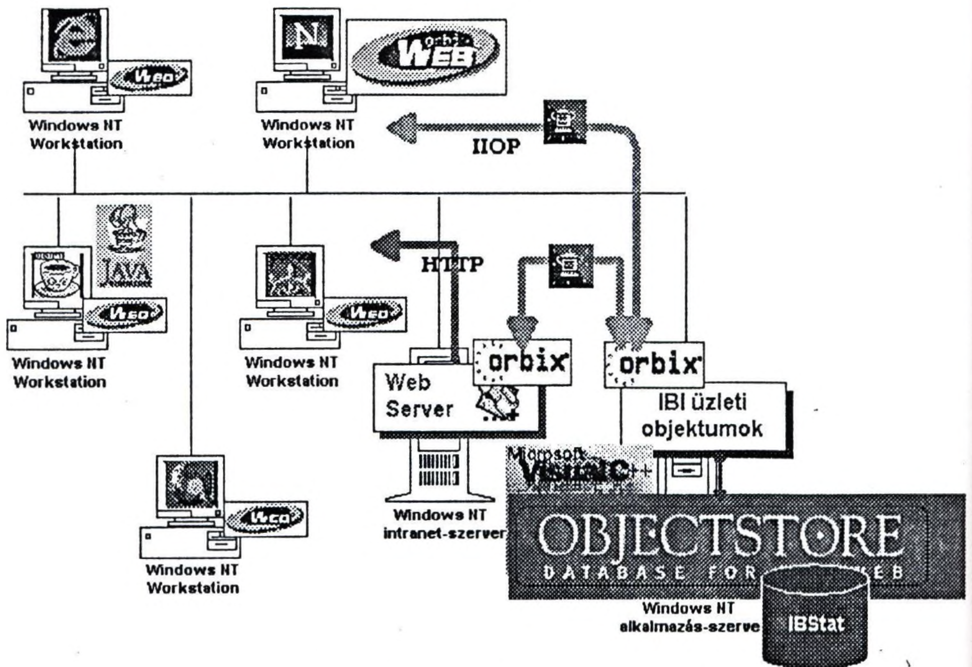
A rendszer fõ célkitûzése egy olyan adatbázis megtervezése és a mûködtetéséhez szükséges olyan szoftverinfrastruktúra megalkotása, amely hosszú távon alkalmas adatok metainformációkkal való leírására, kategorizálására, tárolására, lekérdezésére és jogosultságkezelésére anélkül, hogy rendszeresen módosítani kellene az adatbázis szerkezetét és a hozzá tartozó kezelõszoftvert. Ezzel a megközelítéssel várhatóan hoszabb lesz a felmérési, elemzési, implementációs és rendszerbevezetési munka, de hosszú távon stabilabb rendszer keletkezik, ami egy statisztikai rendszer esetén fontos szempont.

Mivel a rendszert hosszú távra tervezték, a fejlesztési technológiát is úgy alakították ki, hogy az a lehetõ legtovább legyen korszerûnek tekinthetõ. Az MNB-ben az informatikai valamint rendszerintegrálási infrastuktúráért felelõs vezetõk nem tartották szerencsésnek olyan fejlesztési technológia alkalmazását, ami egy-két éven belül elavul. Így, egy olyan Jáva, CORBA alapú intranet technológia mellett döntöttek, amihez egy objektumadatbázis társul.

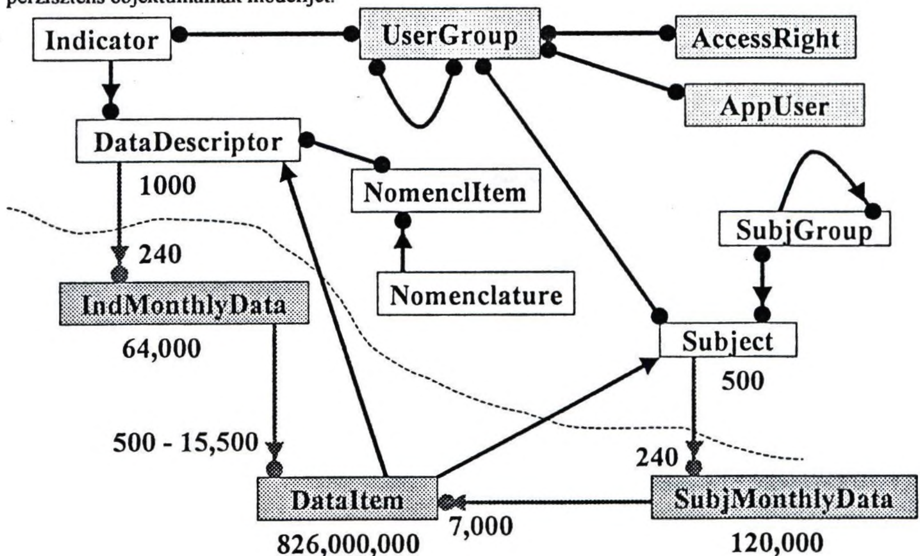
A teljes rendszer alrendszerei:

- Metainformációs alrendszer
- Adattárolási alrendszer
- Lekérdezési alrendszer
- Jogosultságkezelõ alrendszer

Az alábbi diagram a rendszer mûködési architektúráját mutatja be.



Az alábbi diagram (nagyon) vázlatosan mutatja be egy metaadatbázisvezérelt statisztikai adatbázis perzisztens objektumainak modelljét.



A fenti diagram szándékosan egyszerűsített, és nem tartalmaz számos, egy teljes statisztikai adatbázis számára nélkülözhetetlen részletet (mutatóbontások, mutatóbontásváltozatok, nómenklatúraváltozatok, stb.).

A tervezési technológia a következőket tartalmazta:

- Object Modeling Technique (OMT) mint tervezési módszertan
- PLATINUM Paradigm Plus 3.5 objektumorientált tervezési módszertanokat támogató CASE eszköz.

A fejlesztési környezet:

- Jáva 1.1 – Symantec Visual Café
- CORBA – IONA Orbix/C++ és OrbixWeb (Jáva ORB)
- C++ -Microsoft Visual C++
- Objektumorientált adatbáziskezelõ – Object Design Inc. ObjectStore 5.0

Az elõadáson szóba kerülnek a relációs adatbáziskezelõk és az objektum adatbáziskezelõk összehasonlításának kérdései is, különös tekintettel egy statisztikai adatbázisra.

Korszerű webtechnológiák

Németh Miklós, IQSOFT Objektumtechnológiai Osztály

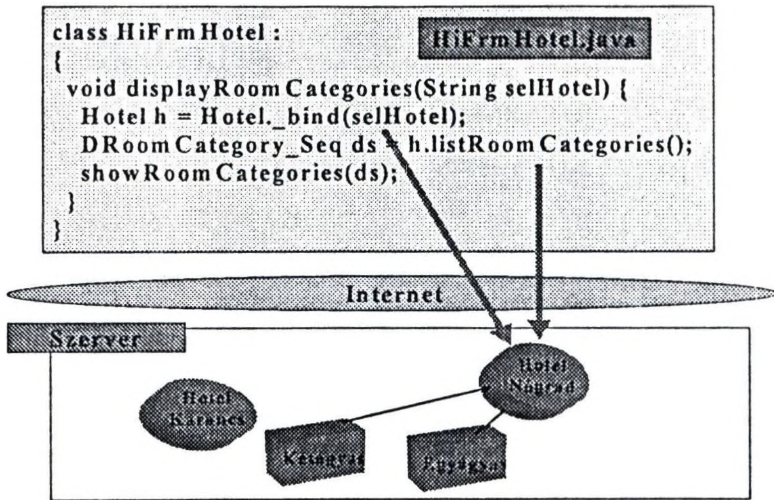
Összefoglaló

Az előadás ismerteti a legegyszerűbb webalkalmazások készítésének technikáit, néhány népszerű fejlesztőeszközt és a jelenlegi ismereteink szerinti legkorszerrűbb technológiákat.

A HTML formokra alapuláo webalkalmazások a HTML korlátozottsága és állpotnélküliségе miatt nem (igazán) alkalmas komplex vállalati információs rendszerek készítésére. Íme egy Cüben írt CGI program, ami kiírja a CGI környezet változóit:

```
// cms.c
#include <stdlib.h>
#include <stdio.h>
#define SHOWENV(ev) {char* e;e = getenv(ev);\
if (e) printf(ev "=%s<BR>",e);}
void main()
{
printf("Content-type: text/html\n\n"
"<html>\n <head>\n"
"<title>Konferenciakezelő Rendszer</title>\n"
"</head>\n <body>\n");
SHOWENV("REQUEST_METHOD")
SHOWENV("PATH_INFO")
SHOWENV("SCRIPT_NAME")
SHOWENV("QUERY_STRING")
{
      char* e = getenv("CONTENT_LENGTH");
      if (e) {
            int len; char buf[1024];
            printf("CONTENT_LENGTH=%s<BR>",e);
            len = fread(buf,1,atoi(e),stdin);
            buf[len] = 0;
            printf("[%s]<BR>",buf);
      }
}
printf("</body>\n </html>\n");
}
```

Az alábbi ábra egy Jáva és CORBA alapú rendszer részletét mutatja be.



Jelentősebb webfejlesztést támogató eszközök:

- Microsoft Visual InterDev és FrontPage
- Oracle Web Application Server
- Object Design Inc, ObjectForms
- Symantec Visual Café Jáva 1.0 és 1.1 fejlesztőkörnyezet
- Borland Jbuilder Jáva 1.1 fejlesztőkörnyezet

ObjectStore ODBMS Internet/Intranet alkalmazásokban

Nick János, IQSOFT

Az Internet robbanásszerű terjedésének köszönhetően használata fokozatosan a mindennapi életünk részét képezi. Az Internet szolgáltatásokat biztosító alkalmazások igen nagyméretűek és összetettek. A felhasználók száma egyik napról a másikra ugrásszerűen megnövekedhet, valamint a gyorsan változó követelményekre és új igényekre az alkalmazás fejlesztőinek rövid időn belül reagálni kell, ami nagy kihívás a számítástechnikai cégek számára. Az objektumorientált technológia, a komponensalapú fejlesztés az az eszköz, amelyek segítségével a cégek meg tudnak felelni az előzőekben ismertetett kihívásoknak.

Az ObjectStore objektumorientált adatbáziskezelő, az Object Design Inc. (ODI) piacvezető terméke megfelelő hátteret biztosít az Internet alkalmazások számára, lehetővé téve nagytömegű adat tárolását és nagyszámú felhasználó kiszolgálását. Működő alkalmazások sokasága bizonyítja, hogy az objektumorientált adatbáziskezelők megbízhatóak, feladatkritikus alkalmazások esetében is sikerrel alkalmazhatóak. Rendelkeznek mindazokkal a szolgáltatásokkal, amelyeket a relációs adatbáziskezelők esetében megszoktunk.

Az ObjectStore nagyszerű teljesítményét egyedülálló „virtuális memóriakezelési" architektúrájának és az objektumorientált programozási környezetek (C++, Jáva, ActiveX) natív támogatásának köszönheti. Az objektumok változtatás nélkül, natív nyelvi formájukban érhetők el és tárolhatók, támogatva az objektumorientált paradigmákat, ami nagymértékben megkönnyíti és meggyorsítja a programozók munkáját.

Az ObjectStore termékcsaládban megtalálhatók az ingyenesen letölthető személyi használatra szánt verziótól (PSE) kezdődően az igen nagyteljesítményű elosztott kliens-szerver architektúrájú ODBMS-ig a fejlesztők minden igényét kielégítő megoldások. Ez az első olyan adatbáziskezelő, amelyik minden termékében biztosítja a Jáva objektumok tárolását. Lehetővé teszi a különböző nyelveken (Jáva, C++) és platformokon létrehozott objektumok teljesen heterogén környezetből való elérését. Számos nagy, szoftverfejlesztéssel foglalkozó cég (Sun, Netscape, Microsoft, Borland, Symantec) alkalmazza termékeiben az ODI Jáva adatbázis technológiáját.

A megszokott fejlesztői környezeteken (MS Visual C++, Java Workshop, stb.) túlmenően számos vizuális eszköz segíti a programozási munkát (BluePrint, Inspector, Web Wizard), az alkalmazás és az adatbázis hatékon hangolását (Inspector, Performance Expert).

A fejlesztési munkát támogatják olyan kész komponensek (Object Managers), amelyek az Internet alkalmazások során leggyakrabban felmerülő adattípusok (Text, Image, Audio, Video, Spatial, Time Series, PDF, SGML, HTML, Java) tárolását, kezelését vagy gyakran ismétlődő feladatok (Personalization) implementációját valósítják meg.

A termékcsalád részét képezik különböző integrációs eszközök, amelyek Internet környezeten elengedhetelenek (ObjectForms, Active Toolkit for ASP), illetve a relációs világ felé biztosítják a kapcsolatot (Dbconnect, OpenAccess). Természetesen a dinamikusan fejlődő CORBA technológia legjelentősebb implementációival (Orbix, VisiBroker, NEO, ObjetBroker) is biztosított az integráció.

KOMPONENS ALAPÚ FEJLESZTÉS UNIFACE 7 OPEN 4GL FEJLESZTŐ ESZKÖZZEL

Szalontai Zoltán, szzoltan@unisoftware.hu
Unisoftware Kft.

Mindig jelennek meg új fogalmak, irányzatok, lehetőségek az információs technológiákban. Ezek nem magukért jönnek létre, hanem azért, hogy a végfelhasználó igényeit minél hatékonyabban ki lehessen elégíteni.

A Uniface kezdeti változata „objektum alapú" volt. Egy erős központi adatmodell volt a rendszer alapja, erre épülve működött a gyors alkalmazásfejlesztő komponens.

A Uniface 5 verziótól kezdve a „kliens-szerver" felépítés volt a fő irányvonal, az akkor divatos irányzatoknak megfelelően.

A többszintű kliens-szerver felépítés kezdetei, az elosztott telepítés első lehetőségei , globális, vállalati szintű fejlesztés és telepítés jellemezte a Uniface 6 verziót. Ezt az időszakot a 4GL-ek második generációjának megjelenése jellemezte, a Uniface ezek egyik vezető terméke volt.

Igen gyakran használt új fogalom manapság a „komponens alapú". A pontos definíció megadása helyett itt most az előadás a Uniface 7 oldaláról közelíti meg a kérdést.

1. Komponens alapú fejlesztés

Már az alkalmazások kifejlesztése során is érdemes újrafelhasználható komponenseket használni. A Uniface 7 három szinten kínál ilyen lehetőséget:
- adatmodell,
- komponens minták,
- logikai modell.

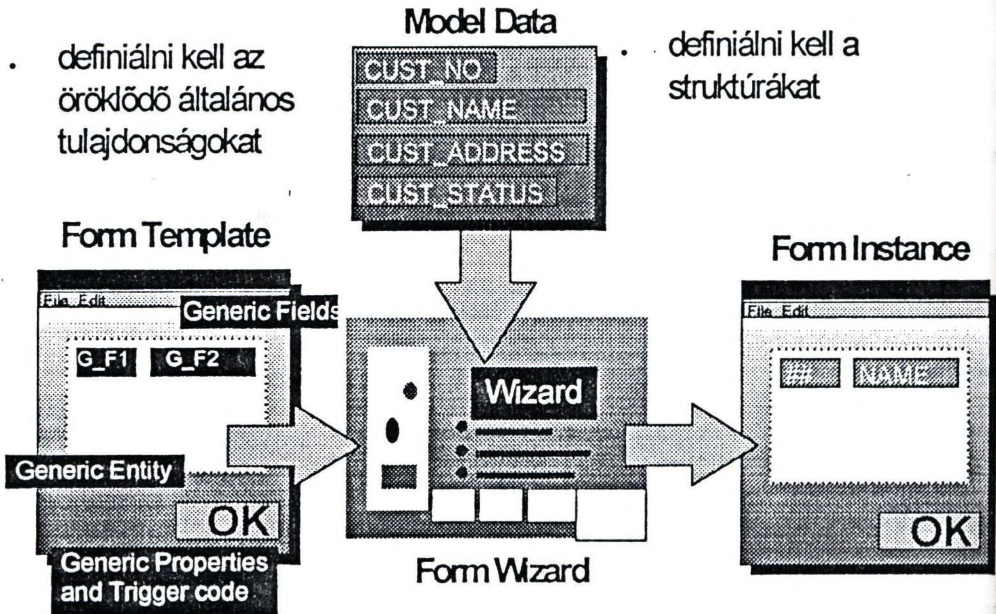
1.1. Adatmodell

Az adatmodell elsődleges feladata, hogy az alkalmazás által használt adatokat, a relációkat logikai szinten definiálja. A Uniface adatbázis-független, ezért az adatmodell sem kötődik egyik adatbázis-kezelőhöz sem. A logikai szint ezen felül azért lényeges, mert a Uniface olyan definíciós lehetőségeket is kínál itt, melyeket később az alkalmazások fejlesztésekor többször újra felhasználhatunk. Ilyenek lehetnek az entitás szintű triggerek, lista-definíciók, globális procedúrák. A definíciókat változtatva a változtatásokat öröklik a rájuk épülő formok. Ez az objektum alapú szemlélet a Uniface legelső verzióiban egy évtizeddel ezelőtt is megvolt.

1.2. Komponens minták (Template)

A Uniface 7 új lehetősége, hogy minden komponenséhez (form, report, service) template használható. Ennek nyilvánvaló előnyei:

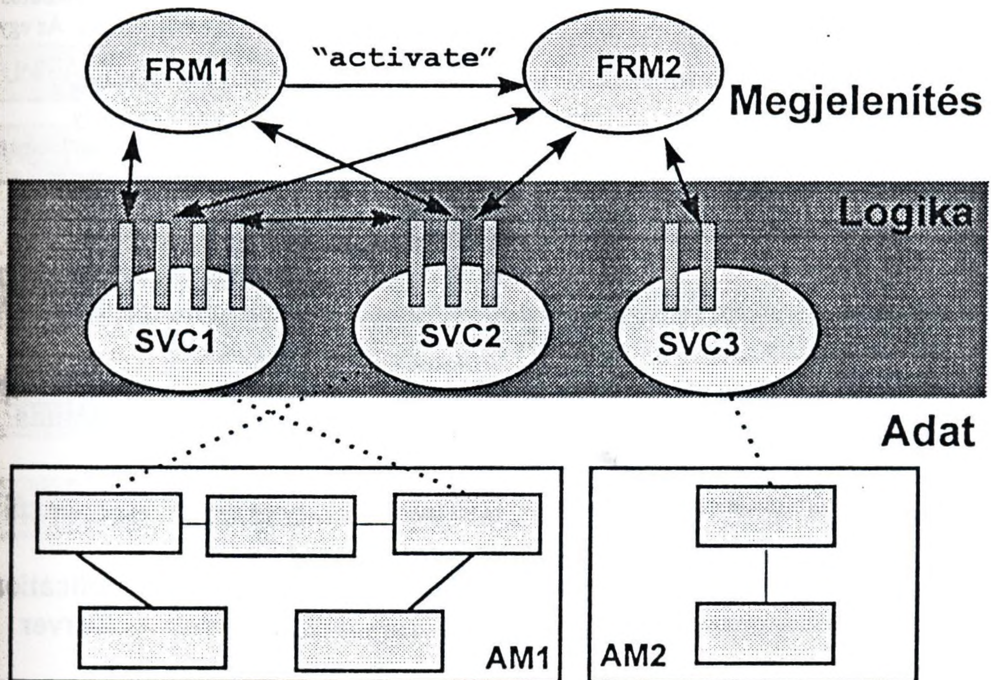
- A fejlesztés felgyorsul, hiszen például a formok programozói egy „félkész" formból (ez a template) indulhatnak ki.
- Konzisztens képet mutat egy alkalmazás akkor is, ha sok fejlesztő, különböző időben végzi a fejlesztést.
- A definíciót megváltoztatva a változást az összes form örökli, ezáltal a rendszer könnyen karbanartható.



A form template általános entitásokat, mezőket tartalmaz. Ezekhez tulajdonságokat, triggereket definiálhatunk. A template-eken alapoló formokat a Form Wizard hozza létre: ez rendeli össze az általános entitásokat és mezőket az adatmodellben definiáltakkal.

1.3. Logikai (Business Process) modell

Egy jól struktúrált alkalmazáson belül a jól elkülöníthető funkciókat eddig is háttérben futó, láthatatlan formokkal oldották meg. Ezeket pedig több helyről, megfelelőlen felparaméterezve hívták meg. Az ilyen megoldásokat a Uniface 7 új támogatást nyújt. Az alkalmazói folyamat logikáját (business logic) a „szolgáltatások" (services) valósítják meg. A szolgáltatásokon különböző műveleteket definiálhatunk. Az alkalmazások, formok ezeken a műveleteken keresztül érik el a szolgáltatásokat.



Ilyen módon teljesen szétválik a megjelenítési, a logikai és az adatelérési szint. (Az adatelérési szint leválasztása már az előző verzióban is lehetséges volt.)

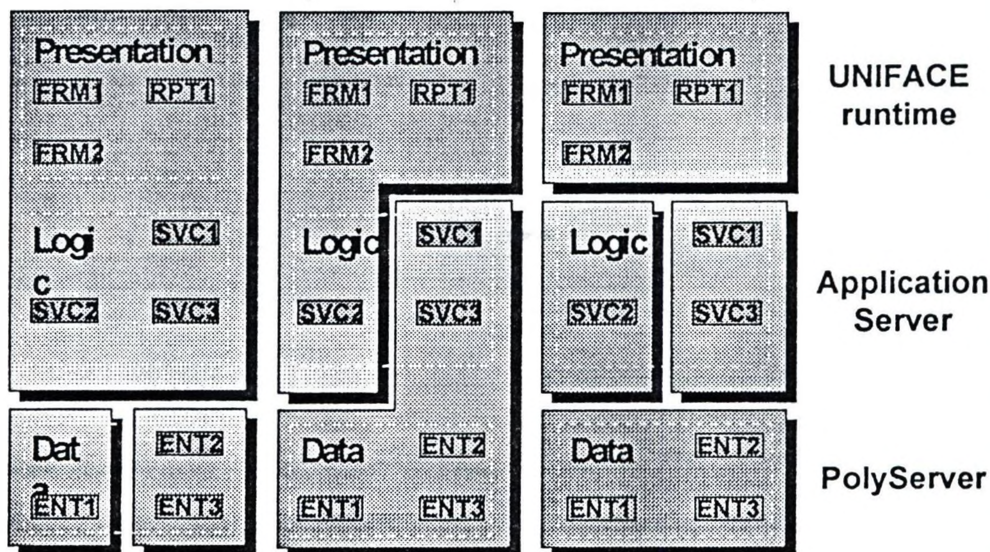
A szolgáltatások együtt kezelik az adatokat és a rajtuk végezhető műveleteket, a tényleges adatelérést elrejtik a megjelenítési szint elől. A szolgáltatások belső megváltoztatása láthatatlan a megjelenítési szint számára.

2. Komponens alapú telepítés, partícionálás

A Uniface 7 új komponenseit nem kizárólag a fejlesztés és a karbantartás hatékonyabb támogatásáért hozták létre. Legalább ilyen fontos szempont volt a komponens alapú partícionálás. Az adatelérési szint leválasztására, partícionálására az előző Uniface verziók is képesek voltak: a PolyServer komponens hajtotta végre a szerver oldalon az adatbázis közvetlen elérését. Az előző Uniface változatok külső middleware eszközökhöz kapcsolódva voltak képesek az alkalmazás logikájának partícionálására.

Telepítés során a Uniface 7 már saját eszközeivel biztosítja a többrétegű kliens-szerver felépítést. A logika számításigényes részeit egy erős szerverre áthelyezve az alkalmazás működése jelentősen felgyorsítható, több szerver esetén a szerverek teljesítménye kiegyensúlyozható. Az egyes szolgáltatások szülség esetén szinkron vagy aszinkron módon is aktivizálhatók.

A partícionálás eredményeként olyan „vékony" kliens hozható létre, amely kizárólag a megjelenítéssel foglalkozik. Ezt jól kihasználja a Uniface 7 új eleme a Web Enabler, amely lehetőséget ad arra, hogy web böngészővel is elérhetünk Uniface alkalmazásokat. Web-en keresztül így bármilyen bonyolult alkalmazást elérhető és hatékonyan futtatható valamelyik alkalmazás szerveren.

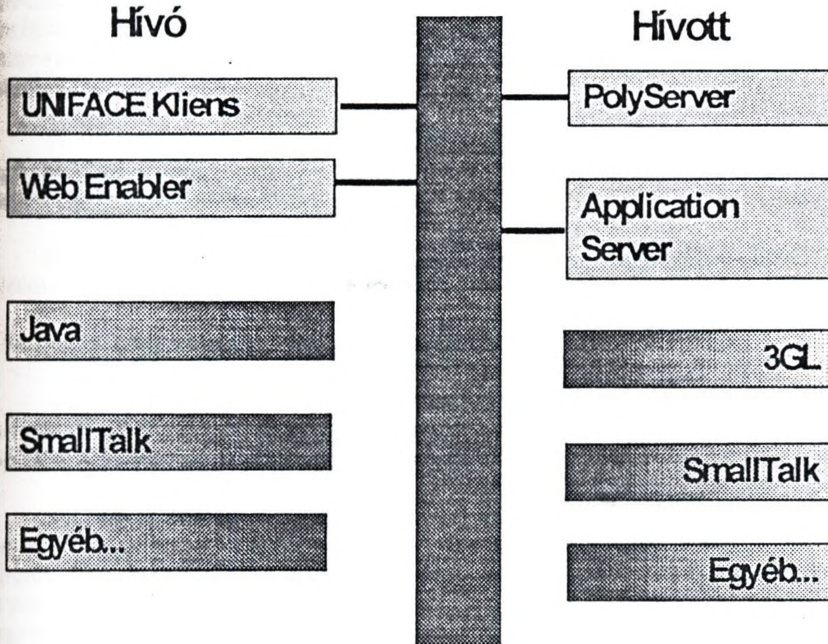


A komponensek által támogatott környezet is kibővült. A PolyServer már most, az Alkalmazás szerver pedig a jövő évben lesz IBM Mainframe környezetben elérhető.

3. Komponens alapú együttműködés

A Uniface 7 saját eszközeivel képes az alkalmazások logikájának partícionálására. Az előző verziók külső middleware eszközöket hívtak ehhez segítségül.

A Uniface ez évben várható legújabb 7.2-es változata megfordítja a helyzetet: maga a Uniface biztosít más külső eszközök, alkalmazások számára többszintű partícionálási lehetőségét. Az új komponens neve: Object Request Broker.



Külső alkalmazások így képesek lesznek Uniface szolgáltatások elérésére: közkedvelt GUI eszközök is lehetőséget kapnak nagy bonyolultságú, megbízható rendszerek elérésére, akár Mainframe tranzakciók futtatására is.

Az ORB a másik oldalon képes beintegrálni minden szabványos, más eszközökben írt szolgáltatást. Így jól bevált, régóta működő rendszereket lehet Uniface oldalról elérni.

Végezetül az is lehetséges, hogy egy már meglévő, külső eszközzel írt szolgáltatáshoz valamely GUI eszköz az ORB-n keresztül fér hozzá. Itt az ORB a partícionálási lehetőséget felajánlva közvetítő szerepet játszik.

IBM COMPONENT BROKER

Nyikes Tamás, tnyikes@vnet.ibm.com
IBM Magyarország

1. Bevezetés

Körülbelül huszonöt év telt el azóta, hogy az objektum-technológia megjelent. Ezalatt az idő alatt a fejlesztés elérte azt az érettségi fokot, hogy ma már objektum-technológiát használva fejlesztik az új számítástechnikai alkalmazások nagy részét. Másrészről viszont még napjainkban sem találunk igazi nagy rendszereket, melyek teljesen objektum-technológiára épülnének. A legtöbb ember számára az objektum technológia egy új nyelv, egy új gondolkodásmód elsajátítását jelenti hátat fodítva a régi hagyományos rendszereknek.

Ha valaki objektum-orientált fejlesztésekbe kezd a kliens oldalon SmallTalk, C++, Java vagy más OO programozási nyelvet használva, rémálommá válhat számára az új alkalmazások meglévő, hagyományos rendszerekhez illesztése.

Érdemes figyelmet szentelni néhány oknak, mielőtt továbblépnénk, mely továbblépés például az IBM Component Broker bevezetése lehet az Önök vállalatánál.

2. Szabványos interfészek az újrafelhasználás biztosításához

Az objektum rendszerek bevezetésének egyik központi kérdése, hogy jól definiált interfészek mögé különböző megvalósításokat helyezhetünk. A különböző forrásokból származó objektumok újrafelhasználása széles körben elterjedt szabvány csomagok elfogadását feltételezi.

Az Object Management Group (OMG) definiálta ezeket a szabványokat átfogó objektum menedzsment architektúrájában (OMA). Az IBM Component Broker az első nagyléptékű infrastruktúra, mely az OMG szabványokon alapul és újrafelhasználható komponenseket támogat. Fontos az is, hogy ez a technológia - az IBM révén - bárki számára könnyen hozzáférhető, megvásárolható. Az IBM Component Broker lelke pedig a Component Broker Connector, röviden CBConnector, mely biztosítja az objektum technológia és a hagyományos számítástechnika kapcsolódását egymáshoz.

3. Szabványos interfészek a szétosztás biztosításához

Az osztott objektum rendszerek alapkérdése, hogy az egyik rendszerben lévő objektumok hogyan kommunikáljanak a másik rendszerben lévő objektumokkal a hálózaton keresztül. Egy tipikus nagyvállalati környezetben az objektum-rendszereket különböző szállítók biztosítják, ezek különböző platformokon futnak és természetesen más működési irányelveknek felelnek meg.

Az OMG definiálta a CORBA specifikációban, hogy különböző platformokon található objektumok milyen módon kommunikálhatnak egymással. Az IBM Component Broker CORBA szabványon alapszik és az osztott objektumok kommunikációjához GIOP és IIOP protokollokat használ.

4. Szabványos interfészek a menedzsment biztosításához

Az osztott objektum rendszerek bevezetésének alapja, hogy biztonságos elérést biztosítsunk, megfelelő felhasználói igényeket ki tudjunk elégíteni és a meglévő rendszerekhez hozzá tudjunk férni. A szakma egyre közelebb kerül ilyen jellegű menedzselési igények szabványos kezelésének kidolgozásához. Az IBM CBConnector az első többplatformos, nagy rendszerek esetében is használható infrastruktúra, mely osztott objektum alkalmazások menedzselésére képes a több évtizedes tudás felhalmozásának eredményeképpen.

5. Szoros integráció az újrahasznosíthatóság biztosításához

Az osztott objektum-rendszerek megvalósításának fontos eleme a meglévő rendszerekkel való szabványos módon történő integráció. Sok meglévő nagyvállalati rendszer esetén a meglévő rendszerek újrahasznosítása objektum-orientált módon nem valósítható meg hagyományos módszerekkel. Az IBM CBConnector technológia lehetőséget ad a hagyományos rendszerekhez való kapcsolódásra.

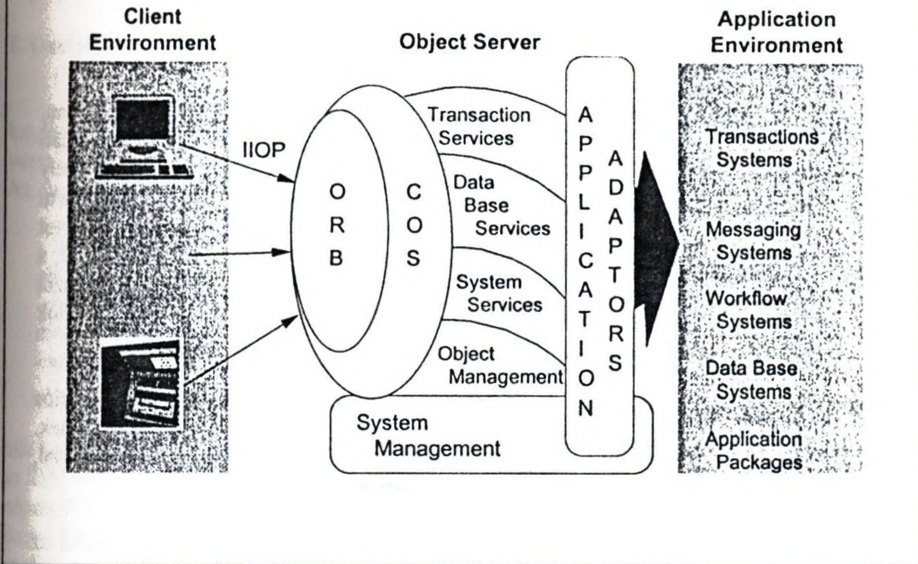
6. Az IBM CBConnector infarstruktúra felépítése

Az osztott objektum rendszerek felépítéséhez a megalapozott objektum tervezést kell kiforrott rendszertervezési gyakorlattal ötvöznünk. Egy üzleti modell objektum-orientált tervezése az objektum viselkedések és kapcsolatrendszerek felülről lefelé történő elemzésével történhet. A rendszer tervezése pedig objektum komponensek növekvő rétegzésével érhető el, tehát alulról felfelé haladva. A CBConnector mindkét irányú megközelítést támogatja.
Az infrastruktúra felépítését három részre bonthatjuk:

- végrehajtás
- fejlesztés
- felügyelet

7. Osztott objektum alapú rendszerek futtatása az IBM Component Broker segítségével

Miután az üzleti modellt elkészítettük, mint objektumok halmazát, futtatási környezetben nagy számú felhasználó éri el a szolgáltatásokat.A CBConnector a Component Broker futtatási környezete. A CBConnector világban felhasználók ezrei érhetik el az üzleti objektumokat alkalmazási környezetek százainak osztott rendszerében.
Minden üzleti objektum az interfészei által van definiálva, melyek az OMG Interface Definition Language (IDL) ben vannak megfogalmazva és a szerver memórájában Java, C++, SmallTalk vagy OO Cobol kódon futnak. A CBConnector a CORBA szolgáltatások széles skáláját valósítja meg, mint például az életciklus, azonosítás, külsővé tétel, név, biztonság, esemény, perzisztencia, egymásmellettiség és tranzakciós szolgáltatások. Alkalmazás adapterek ellenőrzik az üzleti objektumok megjelenésének tárolóit.



1. ábra CB futtatási környezet

A **kliens futtatási környezet** támogatja a Windows, OS/2 és Internet/intranet böngészőket. ActiveX és CORBA-megfelelt C++ kliens alkalmazások a CBConnector szerver objektumait minden további támogatás nélkül egyből elérhetik. Természetesen Visual Basic, Visual J++ és Visual C++ alkalmazásokba is számos platformon beépíthető a CBConnector támogatás. Mind HTML, mind Java appletek támogatottak az Internet/intranet eléréshez. A későbbiekben Smalltalk és OO Cobol támogatás is tervbe van véve. A CBConnector kliensek proxy technikát felhasználva érik el a szerver objektumokat, tehát a proxy-k semmilyen alkalmazási logikát sem valósítanak meg, csupán kéréseket továbbítanak a szerver oldal felé. A kliens kód vagy egy Web szerverről töltődik le, vagy pedig a Component Broker Toolkit segítségével építhető be egy meglévő alkalmazásba.

A kliensek tehát egy **Object Request Broker** (ORB) segítségével cserélnek üzenetet a szerverrel. Az ORB az alapelem, mely többlépcsős alkalmazási környezetekben osztott objektum kezelést képes megvalósítani. Legfőbb értéke, hogy helytől független programozási modellt biztosít. Egyrészről szeretnénk, hogy napjaink bonyolult hálózati számítástechnikai architektúrájának megfelelően ne kelljen azzal foglalkozni, hogy egy adott szolgáltatást mely kiszolgálótól kapunk meg. Másrészről nem szeretnénk, hogy az üzleti objektum elkészítésénél alacsony szintű kommunikációs protokoll problémákkal kelljen bajlódni. A CBConnector ORB-ja CORBA 2.0-nak minden fontosabb vonatkozásban megfelel, mint például a C++ kötések és az interfész tárház. Az ORB támogatja más ORB-okkal való kommunikációt a szabványos GIOP/IIOP protokollokon keresztül.

A CBConnector alapú osztott objektum rendszerek központi eleme az Objektum Szerver (**Object Server**). Ez a rész vezérli a CBConnector többi alkotóeleme közötti együttműködést. Az Objektum Szerver számos részelemmel rendelkezik melyek közül a fontosabbak a következők:

- Objektum transzakció monitor (OTM)
- Instancia menedzserek és instancia menedzsment keretrendszer
- Menedzselt objektum keretrendszer
- Objektum szolgáltatások
- Nyelvi együttműködést biztosító rész

Az *objektum tranzakció monitor* új dimenziót nyit az osztott objektum rendszerek világában, elsősorban, hogy nagyfokú skálázhatóságot biztosít. Ezt azáltal éri el, hogy segít a kliens oldali metódus kéréseket szerverek egy csoportjára leválasztani. Kötési szabályok döntik el, hogy mely szerverek kapják meg a metódusok végrehajtását. Ez növeli a rendelkezésre állást, hiszen nem elérhető szervereket kikerülhetnek a hívások. Kliens oldalon az OTM egy logikai szerver csoportot jelent, így a kliensnek nem kell mással foglalkoznia, mint az alkalmazási logikára koncentrálni, a szerver oldali bonyolult topológia el van rejtve előle. Bár az OMG CORBA specifikációja nem szól tranzakció monitor kérdésekről, az IBM elengedhetetlennek tartja ilyen TP típusú funkciók biztosítását, skálázhatóság és nagy rendelkezésre állás biztosításához.

Az *instancia menedzserek* biztosítják a CBConnector belső működését. Tranzakciós képességek, adatbázis-kezelők elérése, perzisztencia és biztonsági kérdések melyeket kezelnek. Például a CBConnector első változata DB2 instancia menedzsert tartalmaz CICS támogatással, melyeket később további instancia menedzserek követnek majd.

A *menedzselt objektum keretrendszerek* biztosítják, hogy egy adott alkalmazáson belül az alkalmazás logikát (üzleti objektumok) szétválaszthassuk a felhasználói felülettől. Az üzleti objektumok nem kezelnek perzisztens állapotokat. Ezt a feladatot az adatobjektumokra bízzák, akik az adatbázis-kezelő, vagy tranzakció monitor rendszerrel közösen végzik el az adatok szinkronizációját. A keretrendszer részeként adott alosztály az üzleti objektumból menedzselt objektumot készít, mely kapcsolódhat aztán számos felülethez, platformtól függetlenül. Ez a technológia a fejlesztő elől rejtett, az objektumok belső kommunikációját biztosítja.

Az objektum-orientált rendszerek esetében is szükség van azonosításra, az elérés ellenőrzésére, létrehozásra, konzisztencia biztosításra, stb. Ezeket a szolgáltatásokat hívjuk *objektum szolgáltatásoknak*. A CBConnector számos CORBA 2.0 szintű objektum szolgáltatást valósít meg, melyek közül a fontosabbak a következők:

- név szolgáltatás
- biztonság
- életciklus
- esemény szolgáltatás
- külsővé tevési szolgáltatás
- azonosítási szolgáltatás
- lekérdezési szolgáltatás
- objektum tranzakciós szolgáltatás
- egyidejű elérési szolgáltatás

Nézzük végül a *nyelvi együttműködés*t biztosító részt! A CBCOnnector objektum modellje a CORBA 2.0-án alapszik. A CBConnector számos futtatési könyvtárat tartalmaz, melyek egy adott processzen belül lehetővé tesznek nyelvek közötti hívásokat. Az objektum modell a Java irányba fejlődik, a fő fejlesztési irány a C++ és Java objektumok közötti átjárhatóság. Az első verzióban a

szerver oldali üzleti objektumok C++ alapúak lesznek, hogy nagyobb sebességet, érettséget, és meglévő rendszerekhez való jobb kapcsolódást biztosítsák. Később természetesen Java alapú objektumok is megjelennek majd.

8. Osztott objektum alapú megoldások fejlesztése az IBM Component Broker segítségével

Egy jól megtervezett objektum rendszer elkészítése az üzleti folyamatok vizsgálatával kezdődik, majd az üzleti modell megalkotásával folytatódik. A CBToolkit lehetőséget biztosít vezető objektum modellező eszközök (Rational Rose, Select OMT, stb.) modelljeinek befogadására. Az importálás után a CBToolkit a következő előnyöket nyújtja:

- a nagyvállalati üzleti modell felfogható mint az üzleti objektumok és alkalmazások gyűjteménye, melyet a Component Broker futtatási környezete kezel.
- az üzleti modell meglévő rendszereket is magába kell, hogy foglaljon, nem csak új objektum megvalósításokat.
- az üzleti modell számos alternatív nézetet nyújt a végfelhasználóknak

Modellezés analízis után a következő fontos fejlesztési lépés a szerkesztés-fordítás-hibakeresés (Edit, Compile, Debug, ECD) fázisa. Általában elmondható, hogy a CB Toolkit elemei az IBM VisualAge családjának tagjaival kiválóan együttműködnek. Azonban lehetőség van más népszerű ECD környezetek használatára is, mint például az MS Visual C++, vagy az MS Visual Basic. Tehát, az IBM VisualAge C++, vagy VisualAge Java környezetben kifejleszthetjük a végfelhasználói felületet. A kliens oldali JavaBeans komonensek a CORBA IIOP kommunikációs buszon keresztül érik el a szerver oldali üzleti objektumokat.

2. ábra CB Toolkit

Az objektumépítő (Object Builder) tartalmaz olyan részeket, melyek képesek együttműködni akár a Microsoft ActiveX komponensekkel is, így összekötve a két világot. Az objektumépítő fő funkciója a szerver oldali objektumok elkészítésének támogatása. A már megszokott VisualAge-es vizuális építést és részekből történő fejlesztést használja. Mivel azonban a CBConnector nagyban keretrendszerekre épül, így egy további módszert a keretrendszerek testreszabásával és teljessé tételével való programozást is megvalósítja. Az objektumépítő, a fentebb említett, és más objektum-orientált analízis és tervező eszközökből is képes inputot befogadni.

A CBConnector alkalmazások üzleti objektumok kliens nézeteinek gyűjteményei. A CBConnector programozási modellje a következő objektum kategóriákat különbözteti meg:

- alkalmazási objektumok (állapotok melyek a kliensek és a szerver által kezelt üzleti objektumok egymásra hatásának folyamatát örökítik meg)
- üzleti objektumok (a nagyvállalati modell alapvető külső objektumai)
- összetett üzleti objektumok (magasabb szintű külső objektumok, melyek dinamikusan jönnek létre)
- állapot objektumok (perzisztens állapot, melyre az alap üzleti objektumok épülnek a futtatás során)

A CBConnector mind programozói könyvtárakat, mind segédeszközöket is tartalmaz csoportos fejlesztéshez. Ezek az eszközök egy osztott tárházon alapulnak, mely hatékony felügyeleti protokollok segítségével a teljes fejlesztői rendszer integritását megőrzi, a programozók hatékonyságát pedig növeli. Az IBM VisualAge TeamConnection biztos alapot jelent nagy csoportos programozási környezetekben. A verziókövetés, hibakezelés és a konfiguráció csak néhány példa számos funkciójára.

9. Osztott objektum rendszerek felügyelete az IBM Component Broker segítségével

Nem egyszerű feladat robosztus alkalmazást fejleszteni még egy kiforrott alkalmazásfejlesztési környezetben sem. Ennél azonban sokszorta nehezebb egy alkalmazást egy hatalmas, éles hálózatban folyamatos módban működtetni. Ez rávilágít egy kicsit a rendszerfelügyelet fontosságára, és óriási kihívásaira. CBConnector System Management (CBConnector/SM) az IBM húsz éves saját intranetes- és megszámlálhatatlan ügyfélhelyzet tapasztalataira épülő megoldása.

Főbb aspektusok a következők:

- telepítés (a Component Broker és a verzió felügyelet lokális és központi telepítése)
- definíció (gazda gépek, szerverek, szerver csoportok, alkalmazások és azok szerverekkel való kapcsolata)
- működés (szerverek elindítása és megállítása, alkalmazások elindítása és megállítása)
- monitorozás (szerver működési állapota, szerver teljesítmény statisztikák, hiba, aktivitás és követés figyelés)

A CBConnector/SM főablaka az információ ellenőrző ablak (*Information Controller Window*), amely a menedzselt objektum hálózatot szemlélteti egy közös adatmodellen keresztül (Common Data Model). A közös adatmodell a konfigurációs adatok struktúrájának egy leírása, vagy sablonja. A közös adatmodell három részre bontható:

- a modell világ
- a telepítés világ
- a kép világ

A modell világban lévő információ a rendszer topológiáját írja le magas szinten. A teljes rendszer *menedzselt zónákra* bontható. Minden egyes zónában *konfigurációk* találhatók. Például külön éjszakai és nappali zónát hozhatunk létre más-más konfigurációval. Az *alkalmazások szerver modelleken* (szerver típusokon) lehetnek konfigurálva. Instancia menedzser *konténerek* állnak az alkalmazások rendelkezésére. A szerver modellnek *klónjai* lehetnek, melyek a szerver típusok egyedeit jelentik. *Gazdának* hívjuk a gépet magát, melyen a szerver fut. *Cellákba* szervezhetők a gazda gépek. A modell világ részét képezik mindezen objektumok és azok tulajdonságai.

A teleptés világ alacsony szinten adja meg a rendszer topológiáját. A részét képező adatok mind a rendszerfelügyelő alkalmazásban, mind annak ügynökeiben megtalálhatók. Az alkalazások leírása számos ismérv alapján történik. Ilyenek például a DDL-ek, interfész tárházak és osztályok, melyek bennük szerepelnek. A menedzselt objektumosztályok és adat objektum oszályok, melyeket használnak.

A kép világot a rendszerfelügyeleti alkalmazás építi fel a modell világ és a telepítés világ adatai alapján. Minden egyes szervert definiál a helyi ügynökben.

Talán sikerült bepillantást nyújtani az osztott objektumok kérdésének bonyolultságába, és hogy hogyan valósíthatunk meg ily rendszereket.

ELOSZTOTT OO RENDSZEREK TERVEZÉSE, IMPLEMENTÁLÁSA ÉS FELÜGYELETE

Frigó József, frigo@inf.bme.hu
Kelen András, 100263.634@compuserve.com
Szomolányi Márton triad@mail.datanet.hu
Triad Számítástechnikai és Szolgáltató Kft.

0. Bevezetés

Az előadás az elosztott környezetre szánt nagyméretű projektek elemzési - tervezési - megvalósítási módszereit mutatja be különös figyelmmel a többrétegű kliens-szerver architektúrából származó lehetőségekre és előnyökre.

Rámutat arra, hogy hogyan lehet a logikai tervezést elválasztani a fizikai megvalósítástól és bemutatja a dinamikus alkalmazásparticionálás előnyeit.

VÁLLALATI MODELLEZÉS ÉS BPR A RUMMLER-BRACHE MÓDSZERTANNAL

Huba Zoltán
Hungária Számítástechnika Kft.
Kelen András, 100263.634@compuserve.com
Szomolányi Márton triad@mail.datanet.hu
Triad Számítástechnikai és Szolgáltató Kft.

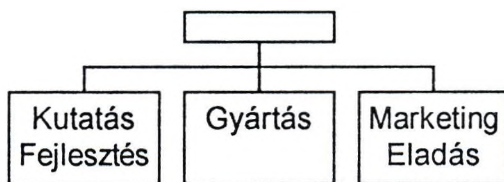
0. Bevezetés

A vállalati modellezés nem rendelkezik olyan múlttal, mint pl. a strukturált, vagy objektumorientált modellezés, ill. rendszertervezés. Ennek feltehetőleg az is az oka lehet, hogy a vállalatok átszervezésének követelménye sokkal később merült fel, mint a "rendcsinálás" igénye a szoftver rendszerek fejlesztése során. Ezen túlmenően, míg pl. a kezdeti időkben tucatjával jelentek meg az OO módszertanok, amelyekből végül mára csak néhány jelentőséggel bíró maradt meg, addig igazi vállalati modellezési módszertan máig kevés van. Ezek között szerepel a Geary A. Rummler és Alan P. Brache nevekkel fémjelzett módszertan.[1.]

Fontos megjegyezni, hogy a módszertant nem elvontan vállalati modellezés céljából fejlesztették ki, hanem a vállalat tevékenységének javításán dolgoztak és ebből alakult ki a modellezés módszertana.

1. A vállalat különböző nézetei

1.1 A hagyományos (horizontális) nézet



Ennek a hagyományos nézetnek az a veszélye, hogy létrejön az ú.n. "siló hatás", amely nem más mint az egyes részlegek vezetői csak a saját egységük vertikális áttekintésére képesek és minden egyéb funkció számukra zavaró tényezőt jelent.

1.2 A rendszer (horizontális) nézet

Az ábra (terjedelmi okok miatt a további ábrákat a konferencián fogjuk kiosztani) három további alkotóelemet tartalmaz: a vevőt, a terméket és az ügymenetet (workflow). Ezzel az ábrázolással lehetőség nyílik azt megmutatni, hogy ténylegesen hogyan folyik a munka, amely általában a funkciókorlátokon átnyúló folyamatokban (process) valósul meg. Ugyancsak láthatóvá válik a belső vevő-beszállító kapcsolatrendszer, mellyel a tényleges termékek, szolgáltatások előállítása történik.

A szerzők tapasztalata szerint a teljesítmény növelésének legjobb lehetősége éppen a funkciók közötti kapcsolatokban rejlik. Jó példa erre az új termék ötletekkel előálló marketing csoport a kutatás-fejlesztés részére, az új termék átadása a kutatás-fejlesztési részlegtől a gyártás számára, vagy a számlázási információk átadása a pénzügy részére.

1.3 A szervezet mint alkalmazkodó (adaptív) rendszer

A módszertan a vállalat (szervezet) működését a környezethez (elsősorban a piachoz) való alkalmazkodás alapján modellezi. A szervezet, mint feldolgozó egység az egyes erőforrásokat, bemeneteket (tőke, nyersanyag, technológia, emberi erőforrások) termékekké és szolgáltatásokká dolgozza fel, amelyeket a fogadó számára (piac) juttat el. A pénzügyi eredmény osztalék és tőkenövekmény formájában jelentkezik. A rendszert a saját belső feltételei szabályozzák, de a legfontosabb hajtóerő a piaci visszajelzés. A versenytársak hasonló erőforrásokkal dolgoznak és ugyanarra a piacra termelnek. Ezt a teljes üzleti képet a társadalmi, gazdasági és politikai környezetbe helyezi el.

A szervezeten belül funkciókat (alrendszerek) találunk, amelyek az egyes bemeneteket specifikus termékekké, szolgáltatásokká alakítják. Végül a szervezet rendelkezik a felügyelő mechanizmussal (vállalat vezetés), amely értelmezi a különböző visszajelzéseket és meghozza a szükséges döntéseket.

2. A szervezet teljesítményének három rétege

2.1 A szervezeti réteg

Ez a vállalat ú.n. makró rétege. Itt jelenik meg a szervezet struktúrája és ennek a váznak az egyes elemei vannak kapcsolatban a piaccal és a szervezet funkcióival. Ennek a rétegnek a fontosabb változói, amelyek a teljesítményt befolyásolják, a stratégia, a makró szintű szervezet céljai és mértékei, a szervezet felépítése és az erőforrások biztosítása.

2.2 A folyamat (process) réteg

A szervezeti ábra funkcionális határait átlépve láthatóvá válik az a mód, ahogy egyes feladatok végrehajtása történik (ügymenet - workflow). A szervezet tevékenysége ezernyi kapcsolattal valósul meg az egyes folyamatok között. Ilyenek pl. a termelési, a marketing, az eladási, a számlázási folyamatok. A szervezet működése nem lehet jobb, mint amilyen az egyes folyamatokban megvalósul. Ezek azok a változók, amelyeket a vállalat vezetőinek úgy kell kialakítani, hogy kielégítsék a vásárlók igényeit, hatékonyan működjenek, és hogy a célokat a követelményeknek megfelelően alakítsák ki.

2.3 A munkavégző réteg

Az egyes folyamatok megvalósításban emberek tevékenykednek. Azok a teljesítmény változók, amelyek erre a rétegre jellemzők a következők: felvételi és előmeneteli rendszer, munkakörök kialakítása, felelősségi körök, visszacsatolás, javadalmazás és jutalmazás, kiképzés.

3. *A kilenc teljesítmény változó*

A szervezet teljesítménye (hogyan felel meg a vevők igényeinek) a 2. pontban részletezett három réteg céljaitól, szerkezetétől és a vezetői tevékenységtől függ. A teljesítmény változóinak egyik dimenzióját a három réteg, a másikat pedig a célok, a szerkezet és a vezetői tevékenység adja.

	Célok	Tervezés	Vezetés
Szervezeti réteg	Szervezeti célok	Szervezet tervezés	Szervezet vezetés
Folyamat réteg	Folyamat célok	Folyamat tervezés	Folyamat vezetés
Munkavégző réteg	Munka célok	Munka tervezés	Munka vezetés

A szervezeti és folyamat rétegben a vezetés az alábbi feladatokat tartalmazza: Célok kezelése, teljesítmény kezelés, erőforrás kezelés, kapcsolódások kezelése. Ezen utóbbinak az a feladata, hogy a köztes események kezelhetők legyenek.

A munkavégző rétegben a vezetés az alábbi feladatokat tartalmazza: teljesítmény követelmények, feladatvégzés segítés, következmények kezelése, visszacsatolás, készségek és tudás kezelése, egyéni képességek kezelése.

Az alábbi táblázat azt foglalja össze, hogy a kilenc teljesítmény változó esetében milyen kérdéseket kell megválaszolni a folyamat során.

	Célok	Terv	Vezetés
Szervezeti réteg	**Szervezeti célok** • Kialakult-e a szervezet stratégiája? • A stratégia megfelel-e a külső veszélyeknek és lehetőségeknek? • Meg lettek-e határozva a stratégia alapján a szervezet megkívánt eredményei és teljesítménye	**Szervezet tervezés** • Minden szükséges funkciót figyelembe vettünk? • Minden funkcióra szükség van? • Megfelelők-e a funkciók közötti jelenlegi ki- és bemenetek • A formális szervezeti struktúra segíti-e a stratégiát és a rendszer hatékonyságát?	**Szervezet vezetés** • Ki lettek-e jelölve a megfelelő funkció célok? • Mérhető-e a teljesítmény? • Az erőforrásokat megfelelően osztották-e el? • A funkciók közötti kapcsolatok kezelve vannak-e?

	Folyamat célok	Folyamat tervezés	Folyamat vezetés
Folyamat réteg	• A legfontosabb folyamatok céljai kapcsolódnak-e vevői/szervezeti követelményekhez?	• Ez a leghatékonyabb folyamat a folyamat célok eléréséhez?	• A megfelelő részcélok meg lettek-e határozva? • A folyamat teljesítménye figyelemmel van-e kisérve? • Elegendő erőforrást biztosítottak-e a folyamat számára? • A folyamat egyes lépcsői közötti kapcsolatokat figyelemmel kísérik-e?
	Munka célok	**Munka tervezés**	**Munka vezetés**
Munkavégző réteg	• A munkák eredményei és előírásai kapcsolódnak-e a folyamat követelményekhez (amelyek végső soron a vevői és szervezeti követelményeket tükrözik)?	• A folyamat követelmények tükröződnek-e az egyes munkaköri leírásokban? • A munkafázisok logikus sorrendben vannak-e? • Kidolgozták-e a megfelelő üzletpolitikát és eljárásokat? • A munkakörnyezet ergonómikus-e?	• A végrehajtók megértették-e a célokat (azokat az eredményeket, amelyeket teljesíteniük kell az adott cél érdekében)? • A végrehajtók rendelkeznek-e elegendő erőforrással, egyértelmű jelzésekkel, prioritásokkal és logikus munkakörrel? • A végrehajtók részesülnek-e elismerésben, ha elérik a kitűzött célokat? • A végrehajtókat tájékoztatják-e arról, hogy megfeleltek-e a célkitűzéseknek? • Rendelkeznek-e a végrehajtók a szükséges tudással a cél eléréséhez? • Abban az esetben, ha a fenti öt kérdésre mind igennel válaszoltak, rendelkezik a végrehajtó a fizikai, szellemi és gazdasági kapacitással a munka céljának eléréséhez?

4. Objektumtechnológia a módszertanban

A Rummler-Brache módszertan fogantatásától objektum orientált. Tekintettel arra, hogy a modell csak akkor teljes, ha tartalmazza a felhasználási eseteket, forgatókönyveket és alapvető objektumokat, valamint a vállalati modell kialakítása során célszerű a később megvalósítandó rendszer elemzését- tervezését könnyitendő kapcsolatot teremteni a tervező eszközökkel, ezért maga a módszertan ezeket az elemeket is tartalmazza.

5. A módszertan használata

A módszertan használatához általában eszközökre van szükség. Az eszköznek részben lehetőséget kell adni az általában nagyszámú elemzők együttműködésére a megfelelő védelemmel, másrészt a bonyolult összefüggések ábrázoláshoz és egy egyes nézetek közötti navigáláshoz közös tervezési adatbázisra (repozitori) van szükség.

5.1 A Software through Pictures (StP) Enterprise Analyst eszköze

Az eszköz a módszertan megvalósítását tűzte ki célul, de azon túl számos kiegészítő szerkesztőt és a munka megkönnyítésére böngészőket, a testreszabáshoz kifinomult parancs (script) nyelvet tartalmaz.

6. Irodalom

[1.] Improving Performance (2. kiadás), G.A. Rummler - A.P. Brache, Josse-Bass Publishers, 1995

[2.] Enterprise Analyst User Manual, Aonix, 1996

NAGY OO RENDSZEREK KOOPERATÍV TERVEZÉSE

Frigó József, frigo@inf.bme.hu
Kelen András, 100263.634@compuserve.com
Szomolányi Márton triad@mail.datanet.hu
Triad Számítástechnikai és Szolgáltató Kft.

0. Bevezetés

A feladatok felosztási lehetőségeiről, particionálási módszereiről és az ezt támogató eszközökről szól az elődás.

Az előadás a nagy projektek nagyszámú résztvevőinek együttműködésének problematikáját vizsgálja.

OSZTÁLYKÖNYVTÁRRA ALAPOZOTT ALKALMAZÁSFEJLESZTÉS A Megatrend Kft. KERETÉBEN

Gajdics György, gajdics@megatrend.hu, Megatrend Kft.

1997

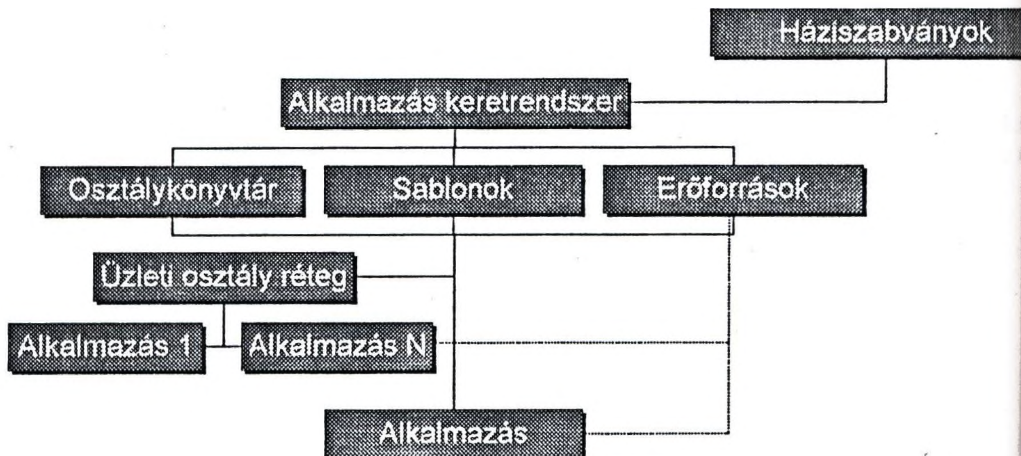
1. Kihívás

- Igény egy Windows alapú információs rendszerre, amely:
  - Könnyen bővíthető
  - Egységes felhasználói interfésszel és arculattal rendelkezik
  - Paraméterezhető, skálázható
  - Felhasználó által testre igazítható
  - Könnyen tanulható

2. Válasz a kihívásra

- Infosys® v2 Vállalati Információs Rendszer
  - Objektumorientált felépítés
    - Előnyök
      - Egységes a felhasználói interfész és arculat
      - Könnyen tanulható alkalmazás
      - Megbízható, robusztus alkalmazás
      - Fejlesztési idő csökkenés
      - Paraméterezhető, skálázható
      - Felhasználó által testre igazítható
      - Az általa meghatározott házi szabványok utólag is könnyen olvashatóvá, bővíthetővé teszik a kódot
    - Hátrányok
      - Nagyobb méretű kezdő kód és alkalmazás
      - Lassabb programvégrehajtás (mint oop nélkül)
      - Komplexitás

3. Alkalmazásfejlesztés



- Nincs szükség alkalmazásfejlesztés közben
  - Hibakezelő (exeption handling)
  - Tranzakciókezelő (transaction handling)
  - Adatellenőrző (data validation)
  - Stb. rutinok fejlesztésére
  - Valamint arculattervezésre

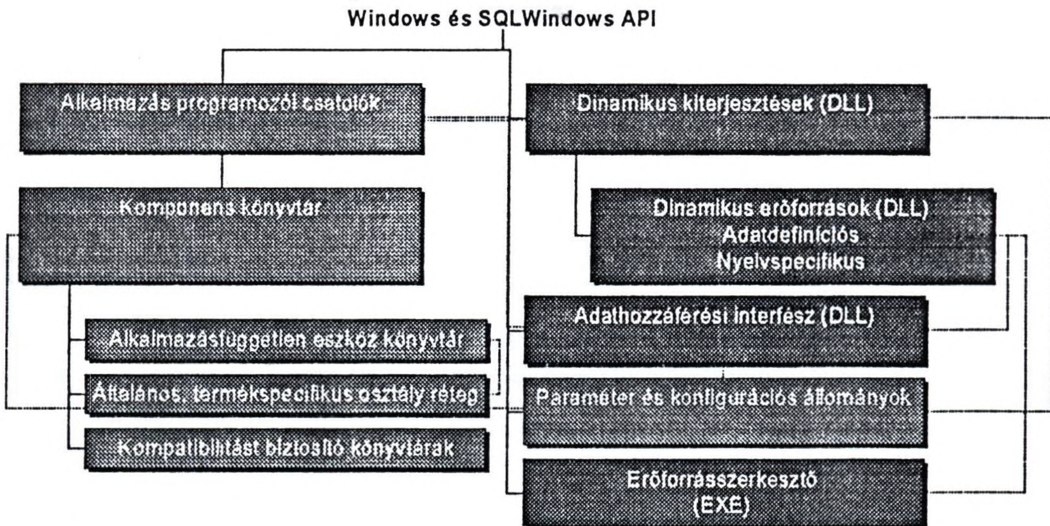
4. Alkalmazásfejlesztő keretrendszer

- Támogatás
  - Prototípusok
  - Egyszerű karbantartó programok
  - Nagy bonyolultságú interaktív alkalmazások
  - Batch feldolgozás
  - Programozás nélküli, de az alkalmazásba szorosan integrálódó riportok készítése
  - Nemzeti nyelv támogatás
  - Komplex jogosultság kezelés
  - Stb.

5. Az alkalmazáskeret felépítése

- Könyvtár csoportosítás
  - SQLWindows (Centura) könyvtárak
    - Komponens könyvtár
    - Csatoló könyvtárak
    - Tools jellegű osztályréteg
    - Általános termékspecifikus, üzleti osztály réteg
    - Kompatibilitást biztosító könyvtárak
  - Dinamikus library-k (DLL)
    - Adtahozzáférési csatoló (DAI)
    - Dinamikus kiterjesztő könyvtár (DXL)
    - Adatdefiníciós erőforrás könyvtár (DAT)
    - Nyelvspecifikus erőforrás könyvtárak (HUN,ENG,.)
  - Kisegítő, inicializáló és konfiguráló file-ok

- Felépülési hierarchia



6. Miért fejlődik, változik vagy bővül egy osztálykönyvtár

- A fejlesztőeszköz korlátoz
- A fejlesztőeszköz új lehetőségeket ad
- Új funkciókra van szükség (általános bővítés)
- Teljesítménynövelés
- Hibajavítás

7. Adathozzáférési interfész

- Alapvető szolgálatatások
  - Kivételkezelés (Exeption Handling)
  - Tranzakciókezelés (Transaction Handling)
  - Nyomkövetés (Tracing)
  - SQL parancs értelmezés és konverzió (Parsing)
  - Automatizmusok
  - Tools jellegű szolgáltatások

- Felépítés
  - SAL (SQLWindows Application Language) szint
    - rendszerkonstansok
    - rendszerváltozók
    - az alkalmazást a DLL-lel összekapcsoló interfész függvények
  - DLL szint
  - Paraméter és elemző file-ok szintje

8. Az osztálykönyvtár főbb jellemzői

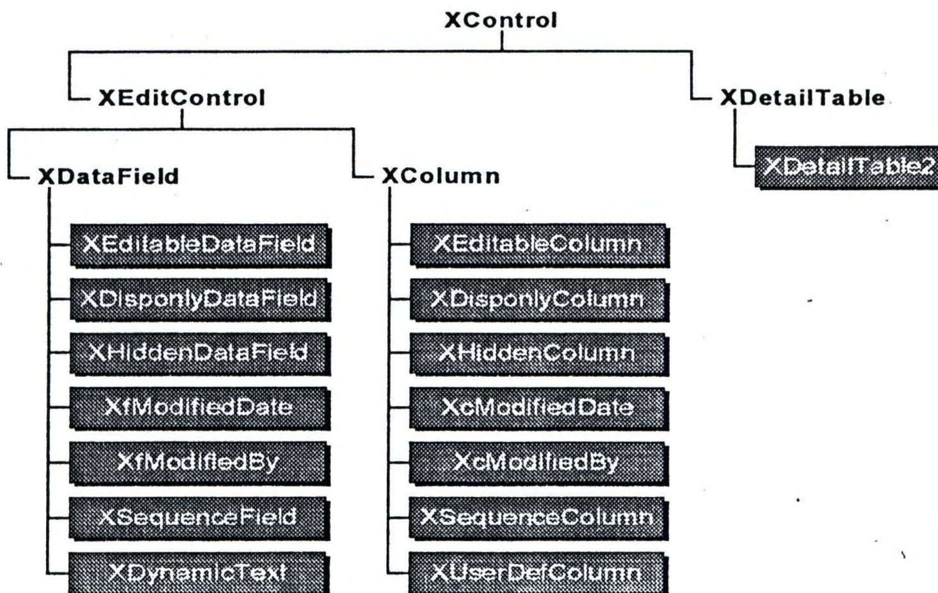
- Alapvetően MDI típusú alkalmazások támogatása
- Funkcióazonosítókon keresztül dinamikusan összehangolt szabványos menü-, toolbar- és funkció-kezelés
- Dinamikus, erőforrásból betöltődő vezérlő és kódválasztó objektumok
- Egy tranzakción belüli fej/tétel/altétel kezelés
- Tulajdonság, állapot és viselkedés flag-ek kezelése
- Szükség szerint felüldefiniálható osztályrétegek
- Kölönböző adatbázismotrorok használata külön programozás vagy programelágazás nélkül
- Felhasználó által elérhető ad-hoc kereső és rendezőfeltételek külön programozás nélkül
- Dinamikus, programozás nélkül (akár a felhasználó által is) készíthető és utólag szorosan az alkalmazásba integrálódó riportok és lekérdező táblák
- Háromszintű jogosultságkezelés
  - Belépési

- Menüszintű
- Funkciószintű

- Nemzeti nyelvek támogatása

9. Alap osztályhierarchia

10. Tools jellegű osztályréteg


XControl
XEditControl
XDetailTable
XDetailTable2
XDataField
XColumn
XEditableDataField
XDisponlyDataField
XHiddenDataField
XfModifiedDate
XfModifiedBy
XSequenceField
XDynamicText
XEditableColumn
XDisponlyColumn
XHiddenColumn
XcModifiedDate
XcModifiedBy
XSequenceColumn
XUserDefColumn

11. Információk

- A Class Library-val kapcsolatos bővebb információval az érdeklődőknek az alábbi címen állok rendelkezésre.

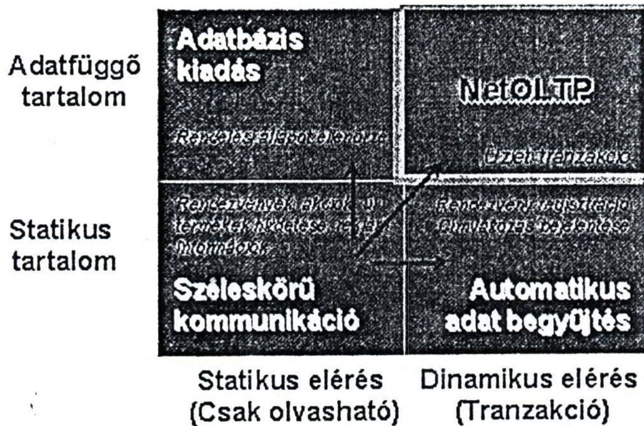
- Gajdics György vezető programozó
  - Megatrend Kft.
    - 1082 Budapest, Üllői út 52/b.
    - Tel.: 1/333-7929
    - Fax: 1/333-7316
    - E-mail: gajdics@megatrend.hu

ELOSZTOTT OO TECHNOLÓGIÁK TÁMOGATÁSA TRANZAKCIÓ SZERVEREKKEL

Sebestyén Zsolt (Sebestyén.Zsolt@axis.hu), Takáts Tamás (Takats.Tamas@axis.hu)
Axis Számítástechnikai Kft.

1. Statikus és dinamikus Internet lapok

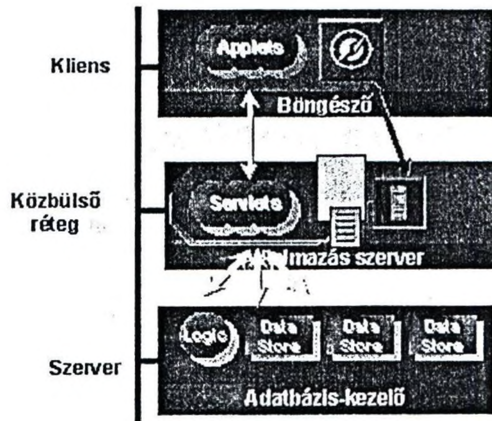
Az Intranet/Extranet/Internet felhasználásokat osztályozhatjuk aszerint, hogy a web lapok tartalma, illetve a bekért információ statikus vagy dinamikus-e. A négyféle lehetőség az 1. ábrán látható. Az első három esetben a nem teljesen megbízható hagyományos Internetes protokollok kielégítő eredménnyel használhatók. Az ábrán NetOLTP-vel jelölt esetben azonban gyökeresen más a helyzet: az üzleti folyamatok tökéletes megbízhatóságot követelnek. Nyilvánvaló ugyanis, hogy a feladott áruházi rendelés elfogadása után az ellenérték banki átutalásának is meg kell történnie, nem várhatjuk a felhasználótól, hogy egy esetleges ***Socket error*** hibaüzenet után újra próbálkozzon a bank elérésével. A megoldás a kliens/szerver rendszerekből jól ismert *tranzakció* lehet: a folyamat vagy teljes egészében végrehajtódik, vagy visszaáll a kiinduló helyzet. Vannak már olyan eszközök, amelyek Web környezetben támogatják a tranzakció-végrehajtást. A következőkben azokat a programokat tekintjük át, amelyeknek az objektum-orientált programozás szempontjából is jelentősége van.



1. Négyféle Internet alkalmazás

2. A többrétegű architektúra

Amikor a hagyományos kliens/szerver architektúra túlhaladásával megjelentek a heterogén osztott rendszerek, a modell bővülésével megjelent a *köztes réteg*, és így kialakult a három- vagy többrétegű architektúra. Ezt mutatja a 2. ábra. A szerver oldalon tároljuk az adatokat, a kliens oldalon végezzük a grafikus megjelenítést, az alkalmazás-logika pedig megoszlik a három réteg között. A megosztásnak nincsenek szigorú határai. Ha az alkalmazás-logika nagy része a kliens oldalon fut, kövér kliensről beszélünk. Ha a kliens oldal leginkább csak a megjelenítéssel foglalkozik, sovány, vagy vékony kliens architektúrával van dolgunk. Az Internet esetében ez utóbbi helyzet a kívánatos. A kövér kliens modell esetén a futtatandó Web alkalmazást ugyanis első használat előtt le kell tölteni, és telepíteni kell. A sovány kliens lényegesen flexibilisebb. Ekkor a kliens számítógépen futó böngésző legfeljebb egy egyszerű kisalkalmazást tölt le a használat előtt.



2. A három rétegű architektúra

3. A komponens fogalma

Az OO szakemberei számára jól ismert az *objektum* fogalma. A *komponens* egy olyan objektum, mely speciális tulajdonságokkal rendelkezik:

- egy másik alkalmazásba, az úgynevezett konténerbe kerül beépítésre
- többszöri felhasználásra készült

A komponens mérete a legegyszerűbb osztálytól kezdve a komplett üzleti alkalmazásig terjedhet. A komponens modellre készültek (készülnek) szabványok, ezek közül a legelterjedtebbek:

- **ActiveX.** A Microsoft által támogatott komponens szabvány. A COM (Common Object Model) szabványon alapul, kiegészített szabvány a DCOM. Szinte minden Windows alapú fejlesztőeszköz képes ActiveX komponensek előállítására (pl. Visual Basic, Visual C++, Visual Java, Delphi, PowerBuilder, Power++). Rendkívüli elterjedtségét az a tény magyarázza, hogy a Windows alatti OLE (ill. OCX) objektumok könnyen ActiveX komponensekké alakíthatóak.
- **CORBA.** Az Object Management Group által készített ajánlás. Hasonló célokra készült, mint az ActiveX, annál valamivel bonyolultabb, de rugalmasabb szabvány. Nem korlátozódik a Windows felületekre, pl. UNIX alatt is szép számban található. Corba objektumokat készíthetünk C++, Java, SmallTalk és Ada programnyelveken. Windowsos elterjedése a közeli jövőben várható, pl. a PowerBuilder következő változata már támogatni fogja. Az osztott objektumok kommunikációja az IIOP (Internet Inter-ORB Protocol) ajánlás szerint történik.
- **JavaBeans.** A JavaSoft ajánlása. Sokkal újabb, mint az előző kettő, emiatt még kevésbé elterjedt. Java nyelven fejleszthető, bármely Java platformon. Az osztott objektumok kommunikációja az RMI (Java Remote Method Invocation) ajánlás szerint történik.

Megjegyezzük, hogy komponensek létrehozása nemcsak a fenti szabványokkal lehetséges. A nem szabványos komponensek támogatásáról később még szó lesz.

4. A tranzakció szerver feladatai

A vékony kliens alkalmazása esetén az üzleti logika teljes egészében a köztes rétegben valósul meg. A szükséges algoritmusok végrehajtásához önálló objektumokat, az előző fejezetben leírt komponenseket készíthetünk. A közbülső rétegben futó program azonban nem egyszerű komponens-működtető, hanem jelentős hatásnövelő tényező. Az egyes komponensek és adatbázis-szerverek megkeresésével, a kommunikáció lebonyolításával, a tranzakciók megszervezésével olyan hasznos segédtevékenységet végez, mint a serpák: az ő idegenvezetésük, csomaghordásuk nélkül nem létezhet hatékony Himalája expedíció. Az alábbiakban áttekintjük az elvégzendő tevékenységeket.

4.1. A komponensek tranzakcióba szervezése

A komponensek számára ismeretlen fogalom a tranzakció. A szerver úgy valósítja meg a tranzakciót, hogy ehhez a komponensek működését nem kell megváltoztatni. A tranzakció szerver kapcsolatban áll a kliens oldalon futó böngészővel, az adatbázis-kezelőkkel, illetve egyéb alkalmazás szerverekkel, így a teljes folyamat kézbentartásával lehetséges a tranzakciók indítása, nyugtázása, illetve hiba esetén a visszagörgetés.

4.2. Az egyfelhasználós komponensek működtetése többfelhasználós környezetben

A tranzakció szerver a rendszer-erőforrások kezelésével, az adatok megosztásának támogatásával lehetővé teszi, hogy a komponensek úgy futhassanak multitaszkos környezetben, hogy a programozóknak nem kell e bonyolult tevékenységeket beépíteniük a komponensekbe.

4.3. Az alkalmazás szerver számítógép erőforrásainak kezelése

A hagyományos kliens/szerver rendszerek esetén előzetes vizsgálatok, számítások végezhetők a rendszer terhelésére vonatkozóan. Az Internetes környezet azonban – az általános hozzáférhetőség miatt – lehetetlenné teszi a terhelés jóslását. A hatékony működés érdekében a tranzakció szervernek készen kell állnia a rendszer erőforrások jobb kihasználására. Biztosítania kell az igény szerinti memória-kiosztást, a többfonalas futtatás lehetőségét, és szükség esetén több CPU bevonására is sor kerülhet.

4.4. Az adatbázis-kezelő kapcsolatok szervezése

A tranzakció szerver a végrehajtási idő csökkentése érdekében folyamatos kapcsolatot tart az adatbázis-kezelővel. A kapcsolat kiépítése ugyanis időigényes feladat, ezért nem lehet felhasználónként dinamikusan kapcsolatot nyitni. Ehelyett folyamatosan nyitva kell tartani néhány kapcsolatot, és a sok felhasználót a megnyitott vonalakon kell multiplexelni.

4.5. A tranzakció idejére tartós kapcsolat kiépítése a klienssel

Az Interneten megszokott HTTP protokoll egyik sajátossága, hogy csak az üzenetváltás idejéig él: a kívánt lap megérkezése után automatikusan lebomlik. Tranzakciófeldolgozásra ez a módszer nem alkalmas, ezért a kezdeti, rendszerint HTTP alapú kapcsolatfelvétel után át kell térni valamilyen tartós kapcsolatot biztosító protokollra.

A tranzakció szerveren létrejött komponens-példány és a kliens közötti kommunikáció az úgynevezett proxy objektum segítségével történik. Minden komponenshez tartozik egy proxy objektum, mely az objektum kliens oldali megjelenítésén kívül a kommunikációt végzi. Az osztott objektum szabványok tartalmaznak kommunikációs protokollt is (pl. IIOP, DCOM), egyéb esetekben a tranzakció szerver által támogatott protokollok jöhetnek szóba (pl. TDS: Tabular Data Stream). Az egyes alkalmazások esetén megkívánt adatbiztonsági követelményeknek megfelelően választható valamilyen titkosító eljárás is (pl. SSL).

5. Forgalomban levő tranzakció szerverek

Vizsgálódásaink során két olyan eszközt találtunk, mely megfelel a tranzakció szerverrel szemben megfogalmazott elvárásoknak. Ezek a *Microsoft Transaction Server* és a *Sybase Jaguar Component Transaction Server*. Lényegében mindkettő a fent leírtak szerint működik. A köztük levő eltérések leginkább abból fakadnak, hogy a Microsoft – mint bármely más területen – igyekszik mindenre saját megoldást adni, míg a Sybase – a nyíltság jegyében – sokféle szabványnak és adottságnak igyekszik megfelelni. Ennek megfelelően:

A Microsoft Transaction Server:

- kizárólag ActiveX komponensek beépítését teszi lehetővé
- a tranzakció szerver és a kliens proxy között a DCOM-ot valósítja meg
- csak Windows NT-n működik

A Sybase Jaguar CTS:

- támogatja az ActiveX komponensek beépítését
- nem standard komponensek (C++, Java, PowerBuilder 5) használata is lehetséges
- a szabványos komponens protokollok (DCOM, IIOP) mellett saját protokoll is használható (TDS)
- többféle tranzakció-szervezés: implicit tranzakció (egy adatbázis-kezelő esetén), kétfázisú commit Microsoft DTC (Distributed Transaction Coordinator) vagy JTS (Java Transaction Service) használatával
- a következő verzióban lesz CORBA támogatás is
- jelenleg Windows NT-n és SUN Solarison fut

Megjegyezzük, hogy egyéb termékek is léteznek a tranzakció szerverek körében. Példaként említhetjük az *Oracle Web Application Servert*, mely szintén tranzakciósítja az Internetes alkalmazásokat. Programozási interfésze azonban nem komponens alapú, így az OO témakörébe nem illeszthető bele. Említést teszünk még az *IBM Component Broker* rendszerről, mely CORBA komponenseket kezel. Az összehasonlításból azért maradt ki, mert lapzártáig nem sikerült elegendő információt beszerezni róla. (Reméljük, az előadást már kiegészíthetjük ezzel is.)

OBJECT ORIENTED TECHNIQUES IMPLEMENTING DISTRIBUTED CORBA OBJECTS IN JAVA

Zoltán Porkoláb, gsd@ludens.elte.hu

Department of General Computer Science, University Eötvös Loránd Budapest
1088 Budapest, Múzeum krt.6-8, Hungary

1 Introduction

Computer technology has always some mayor "buzzwords", here we would like to mention only a few of today's keywords: Object oriented programming, distributed computing, and the Internet. However they are in different areas of information technology, there is something interestingly common in all three topics: they started rather probe and true technologies than one based on well-elaborated scientific theory. Let be honest: in all the above technologies practice preceded theory.

The most characteristic features of object oriented programming has already appeared in Simula67: classes as a kind of type-construction with the separation of interface, both methods and accessible (now we say: public) members. Individual objects (as the instances of a class) with well-defined inner state, and a self-identification possibility (this). Inheritance of course has also roots in Simula67. This has happened well before Bertrand Meyer, Grady Booch or James Rumbaugh has published their books and even the term "object oriented" would become well known. The theory of distributed computing has longer history, but was connected mainly to the database technology. New everyday techniques – DCE, RPC and others – have no theoretical background or description.[1] The Internet was (is) really something self-developed. In recent years there are theoretical approach for proving network protocols using temporary logic etc. But it has even now a minor influence to the practice.

When we are arriving at the point, when these keywords – object oriented programming, distributed computing and the Internet – meets, it is worth to look at computer technology whether there was already a practical approach in the industry for these. We can find such technology. CORBA now is industrial standard technology on distributed object oriented programming.

2 Structure of CORBA

CORBA (Common Object Request Broker Architecture) is a product of a consortium – called Object Management Group, OMG – which includes over 700 companies (originally formed 13 only) from all of different areas of computer technology, software vendors, developers and users. OMG

do not produce software they are rather creating standards allowing interoperability and portability of distributed object oriented applications. The power of OMG comes from the fact that all the great software manufacturers (except one) belonging to the group. The only one exception is Microsoft, which has his own object broker technology called DCOM (Distributed Component Object Model).[2] OMG's view at the distributed object oriented applications is OMA – Object Management Architecture. In OMA there are system oriented components such Object Request Brokers and Object Services and application oriented components such Application Objects and Common Facilities.

Of these the foundation is the object Request Broker or ORB. ORB manages all the communication between the other components. It allows objects interact in heterogeneous, distributed environment, independent of the platforms on which these objects reside and technologies – including operating systems and programming languages – used to implement them. In performing its task it relies on Object Services which are responsible for general object management such as creating objects etc. Common Facilities and Application Objects are the components closest to the end user, and in their functions they invoke services of the system components.

The central component of CORBA is the Object Request Broker (ORB). It is responsible for creating and managing connections between objects including all the network communication between different address spaces (machines). Over the ORB objects are acting like they would be in the same address space: they are accessing each other's public members, calling public methods, passing parameters and receiving return values. Client objects communicate calling ORB core via IDL stub or through the Dynamic Invocation Interface. IDL is Interface Definition Language – a declarative language to specify CORBA classes. As IDL is pure declarative it doesn't bother the implementation of the class but the interface. IDL is object oriented, so it includes class definitions (IDL calls it as "interface"), attributes and method declarations, also inheritance and exception handling. As IDL is about only the interface of classes, it declares only to public (accessible) attributes and methods. IDL is language independent, so it has its own data types, with the possibility of defining different aggregations and other type constructions.

The IDL compiler is a tool what maps IDL language definitions to a certain programming language such as JAVA or C++. In the case of JAVA it maps interface definitions to a set of JAVA interface files, and class definitions. The IDL compiler made sources could be compiled without modification by the JAVA compiler. What the developer should write his own is the server side implementation of server classes, and the client code.

3 Structure of CORBA-based JAVA programs

As mentioned earlier, the IDL description is language independent, so theoretically all CORBA-based program has the same structure. In practice, programmer can use the advantages of the programming language when implements client or server side. In particular JAVA language makes the programmer of client side has a choice: writing a JAVA application or an applet.

3.2 Clients as JAVA applications

JAVA applications on client-side at first glance acting like ordinary client-server solutions. The JAVA Virtual Machine – implemented as interpreter or just-in-time (JIT) compiler runs the client code at client side machine. The client code is communicating with the ORB installed on the local computer in order to convey a request for an operation invocation to the server, which then sends results via the ORB back to the client. In this scenario client JAVA code and client-side part of ORB should be installed prior.

This scenario has some major disadvantages. Maintenance cost at client side can grow as newer and newer versions of client code should install on to distant machines. Also the right setting up the client-side ORB not an easy task, needs a professional.

3.2 Clients as JAVA applets

A JAVA applet can also be a CORBA client. JAVA applets are stored on (server-side) host machines and downloaded to client in runtime. In this scenario, a Web browser – installed previously on local machine – downloads and runs the JAVA client code. New generation of browsers also include client part ORB code. Netscape 4.x uses Visigenic's Visibroker ORB JAVA classes, and can act as a complete CORBA client. As Web browsers now is almost a necessary part of workstations, the maintenance problems of the first scenario has not been arisen.

JAVA security model only allows applets to open network connection to the host from which have been downloaded. This restriction is in conflict with the CORBA model, which allows clients to invoke operations on objects regardless of their physical location. Other problem arises when firewall is used in the network. Firewalls restrict communication between hosts, allowing only well-specified types of traffic, such e-mail and Web related ones. These problems could be worked around with different implementation specific ways. Certainly, here the final solution has not been achieved yet.[3]

4 An example application

In this chapter we show the basic techniques of CORBA programming with the help of an example. In the example we create a group of objects in type Vehicle, Bus and Truck and a client application using the objects. Vehicle, Bus and Truck are arranged in object oriented inheritance hierarchy: a Bus and a Truck is also a Vehicle, so Vehicle is a base class and Bus and Truck the derived classes. The objects of the different classes are located on different hosts, as the client application does it too. Client uses the remote objects as they were in the local machine.

This scenario is not far from the real life. Different classes can represent different services (so called Business Object) ,from different providers, e.g. the local Police administers general information on vehicles (class Vehicle), local transport company provides information on buses, so does an other

company on trucks. All the companies have their own databases and the classes are using this local information to implement the appropriate "Business Objects".

4.1 IDL definition of the problem

First we should define the IDL description of the project. Here we declare only the interface of the classes – public data members and methods – and the (inheritance) connection between the classes. As IDL is pure declarative we doesn't bother implementation details.

```
// veh_exam.idl
module veh_exam
{
    interface vehicle
    {
        readonly attribute string plate;
        readonly attribute string owner;
        string print();
    };
    interface bus : vehicle
    {
        attribute short places;
    };
    interface truck : vehicle
    {
        attribute short weight;
    };
};
```

We use the IDL compiler to create JAVA source files: interface and class definitions. Those files can be compiled using the JAVA compiler, without any modification. Here we should mention that IDL compiler applies a number of transformations on the interface description. The most important transformation is to change the public data members – "attributes" – to (a pair of) public functions. Since the client and server are not in the same address space there is no possibility directly access each other's data. The two IDL generated methods are used to set and get field values in the distant address space. As IDL is describing only the interface of classes – such methods are generated only for public data members. In fact, IDL goes further: one can distinguish read-only attributes, i.e. "object constant" values. This kind of attribute cannot be written, only serves data for output. For such attributes only the "get" function is generated.

4.2 Implementing server classes

In the next step we implement server side classes. We use JAVA interfaces and – ORB specific – JAVA code generated by the IDL. We also write the real implementation code of server classes. Until this point we were largely language and ORB vendor independent, here we should use some vendor specific code. In the example code bellow we used Iona's OrbixWeb 2.0 version ORB to implement Vehicle, Bus and Truck classes.

```
package veh_exam;
import IE.Iona.Orbix2._CORBA;
import IE.Iona.Orbix2.CORBA.SystemException;

/* Implementing vehicle, Creates a new Vehicle
```

```
 */
public class VehicleSrv
{
    public static void main(String args[]) {
    // Here is mainly vendor and/or implementation dependent code
    // handles the connection with the client and communicates with the ORB.
    // The code includes creating a new VehicleImpl object.
    }
}
/* Implementation of Vehicle class The object state:(plate, owner).
 *   Constructor.  The get_ methods for public data members
 *   (There is no set_ method: all data were readonly)
 *   Implementation of public print method
 */
class VehicleImpl implements _vehicleOperations
{
   public VehicleImpl(String r, String t) {    // Constructor
       plate = r;
       owner = t;
   }
   public String get_plate() { // The get_ method for attribute plate
       return plate;
   }
   public String get_owner() { // The get_ method for attribute owner
       return owner;
   }
   public String print() {      // Public print method
       return "Generic vehicle, licence plate = "+plate+", owner = "+owner;
   }
   // class state
   protected String plate;
   protected String owner;
}
```

As we see above, after a necessary code of server-side house-holding (what we were only marked in the code), we writes a true object oriented code, with constructor, encapsulated data fields representing the class state, and public methods as interface. The only exceptional: there are no public data members, we use the get_ and set_ public methods to set the class values.

Now, we write the code of the Bus class. As we have seen in the IDL description, Bus is a derived class with the base class Vehicle. We well make profit on this inheritance relationship, when we implement the Bus class: we use VehicleImpl as base class of BusImpl.

```
package veh_exam;
import IE.Iona.Orbix2._CORBA;
import IE.Iona.Orbix2.CORBA.SystemException;
/* Implementing bus, Creates a new Bus
 */
```

```
public class BusSrv
{
     public static void main(String args[]) {
     // Similar "house-keeping" than at vehicle.
     }
}
/* Implementation of Bus class
 *   The object state: places (plate and owner inherited from VehicleImpl).
 *   Constructor. The get_ and set_ methods for public data members
 *   Implementation of public print method (polimorphism!)
 */
class BusImpl  extends VehicleImpl  implements _busOperations
{
   public BusImpl(String r, String t, short f) { // Constructor
        super( r, t);
        places = f;
   }
   public short get_places() {    // The get_ method for places
       return places;
   }
   public void set_places( short f) { // The set_ method for places
       places = f;
   }
   // The public print methode overwrites the Vehicle's print()
   public String print() {
        return "Bus, plate = "+plate+", owner = "+owner+", places="+places;
   }
   // class state
   private short places;
}
```

As we see, we can gain from the inheritance relationship between Vehicle and Bus. In other words: the server-side is a true object oriented program.

Important to see, that we implemented Bus class's print() method independent from the Vehicle's one. Calling print() from the client on a Bus object, this method will overwrite the base class's method and implements full polymorphism. We write similar code implementing the Truck class.

4.3 Implementing the client side

Here we implement the sample client application, which will use the server classes implemented above.

```
package veh_exam;
import IE.Iona.Orbix2._CORBA;
import IE.Iona.Orbix2.CORBA.SystemException;
import java.lang.String;

public class VehicleClient
{
  public static void main(String args[]) {
    /* Here found hosts, build connections, etc... . */
    _vehicleRef  g[] = new _vehicleRef[3];
```

```
        try {
            g[0] = vehicle._bind(":VehicleSrv", hostname1);
            g[1] = bus._bind(":BusSrv", hostname2);
            g[2] = truck._bind(":TruckSrv", hostname3);
        } catch (SystemException ee) {
            System.out.println(ee.toString());
            System.exit(1);
        }
        // The test part of client: calling print() on different classes
        try {
            for ( int i = 0;  i < g.length;  ++i )  {
                System.out.println(g[i].print());
            }
        } catch (SystemException ee) {
            System.out.println(ee.toString());
            System.exit(1);
        }
    }
}
```

Here we used _vehicleRef as a general interface for Vehicle class. The interface file _vehicleRef was generated by the IDL compiler after the IDL description. As we used this general interface to store each object references, client "does not know" the type information of the array elements g[i]. However, print() methods are polymorph, so when we call print method on different classes' objects we get different results.

5 Comparing CORBA with other distributed methods

There are lot of arguments why use CORBA rather than other methods. Here we discuss a number of them:

— Using CORBA the programmer don't need to bother to implement lower level functionality, such as network protocol, file descriptors and low level resource management. Function method calls are using argument passing without explicit needs of arranging data in network independent format. Such techniques as argument passing, marshalling function return values are given for the user of CORBA interface.

— The problem can be described in IDL in a language independent format. Programming language binding can be delayed in the late implementation phase. It is possible to implement each class in different programming language – in the one, which fits the best to the problem or (in a legacy system) which is available.

— Distributed database offers data only, and client code should know how to interpret these data. CORBA offers methods, data and its interpretation together are encapsulated in "business objects " at server side.

— CORBA is a real object-oriented solution. In contrast of other forms of distributed processing, an ORB does not simply invoke a remote function – like RPC (Remote Procedure Call) – it calls a

method on a well-defined target object. This means: data and operations on it could form objects, they are not separated as in RPC or in most not object oriented databases.

— CORBA also makes possible polymorphism: different classes can have methods on the same name, and also calling the same method on different objects belonging to the same class could be distinguished. As different objects can answer different way for the same method invocation, polymorphism naturally appears in CORBA.

— Possibly the most interesting thing is the call back possibility. In a traditional distributed system there is a server side and a client side. Client side is the initiative, it activates some server process and server serves back information (or do something). If client does not makes call server cannot serves information at all. However in CORBA objects are communicating calling each other's methods. A server also can call client's method by own. (This client method regularly implemented in a separate thread in the client, for the purpose of real concurrency.)

What does it mean in the practice? Imagine we see an applet, just downloading from a far host. It shows data from a database. Meanwhile data can change in the database – some concurrent user uses the same or different application updating the database. Now the server – who recognised that – is able to send message to all the clients (via call back), that database has modified, please reload data.

6 Comparing JAVA with C++ as implementation languages

We mentioned that CORBA is language independent: classes both server and client side could be implemented in a number of languages. Nowadays C/C++, Smalltalk, COBOL and JAVA language mappings exist. Of those, C++ is well known, portable language, existing in many platforms. Why chose JAVA rather than C++?

6.1 Language elements where JAVA has shortages

JAVA sometimes called C-plus-plus-minus-minus. This opinion reflects a few shortages of the language.

— Possibly the greatest one is the lack of parametrised types, or templates as it called in C++ ("generic" in ADA). The JAVA dynamic type checking with the instanceof() method, the Class and Object classes cannot compensate these shortage. As inheritance not always can be used instead of automated code generation of templates, programmer should de-normalise his code repeating similar patterns. (There are non-standard extensions of JAVA — like Pizza (wwwipd.ira.uka.de/~pizza) —offering advanced language features like template types and references to functions.)

— Another problem is the lack of multiple inheritance. JAVA doesn't support this language element. Many claims that multiply inheritance is a feature not necessary in object oriented language. I rather agree with Grady Booch, who said: "Multiple inheritance is like a parachute; you don't need it very often, but when you do it is essential".[4]

— Garbage collection (lack of destructor). JAVA – unlike C++ – provides automatic garbage collection. This means in one side that programmers not have to worry about memory leaks – a frequent problem in bad C++ programs. On the other side with constructor and destructor programmer can fully control the start and the end of a variable's lifetime. At the end of the

lifetime more could have to do than simple free memory; it could have to decrement counters, handling dangling pointers, etc. There is finalize() method in JAVA, but its calling time not well defined – after variable had gone out of scope.[5]
– Some "minor" shortages, like lack of friend declaration – which would enable certain "friend" classes to access non-public members – has the consequence that programmer should declare unnecessary public methods – like default constructors, which could harm encapsulation.

6.1 Language elements where JAVA has advantages

– Portability. JAVA is now the ultimate portable language. Comparing C++ 's source code portability, JAVA 's byte code has a higher level of portability. JAVA Virtual Machine can be implemented with little effort on different platforms and serves as a layer hiding lower level implementation layers.
– Applets. This is a brand new way of programming. One can create and use client-server solutions without installing software at client side at all. (Supposing a Web browser is available on client side, but today this is essential on most platform.)
– Built in, platform independent multithreading. Multithreading is essential in recent concurrent applications – not mentioned the distributed processing. JAVA language has platform independent, easy to use, language defined solution for multithreading. (In C++ one should use different libraries as there is no accepted multithreading standard library.)
– Built in security model. The language defines an acceptable security model, which separates the resources as accessible and not accessible for a running applet. This security model perhaps not the ultimate one, but most cases serves as sufficient tool.
– Very well defined exception handling. Unlike a C++ programmer, JAVA programmer can handle not only his own exceptions but also system exceptions.
– Language defined graphical tools. In practice, different vendors usually creates different user interface libraries. In JAVA, event the model and basic graphical elements is defined in the language definition.

Despite of the problems mentioned in the previous chapter, JAVA has a lot of advantages over C++ in distributed applications. JAVA is a moderate safe language. It can be compared to Pascal or Algol68. It is not as easy to express ourselves as in C or C++, but what we coded and compiled successfully is perhaps the same what we wanted to express.

References

[1] Coulouris, G. et al.: *Distributed Systems*, Addison-Wesley, 1994.
[2] Orfali R., Harkey, D.: *Client/Server Programming with JAVA and CORBA*, Wiley, 1996.
[3] Vogel, A., Duddy, K.: *JAVA Programming with CORBA*, Wiley, 1997.
[4] Booch, G.: *Object-Oriented Design*, Benjamin/Cummings, 1991.
[5] Stroustrup, B.: *The Design and Evolution of C++*, Addison-Wesley, 1994.

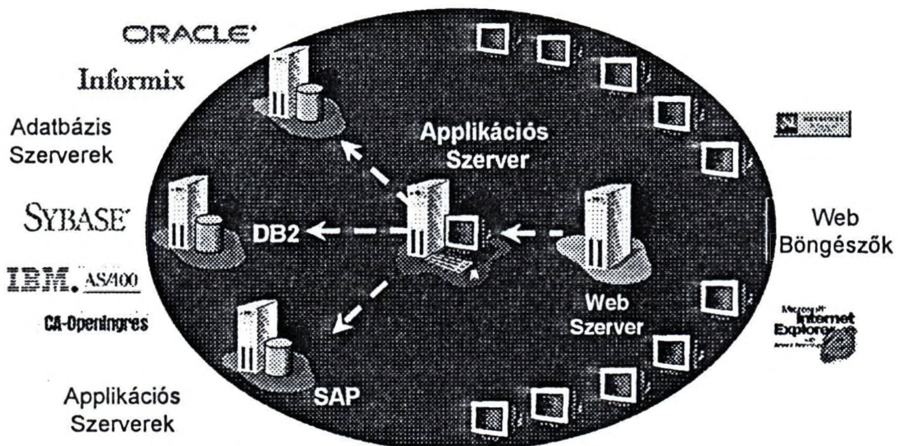
centura WEB alkalmazások fejlesztése

Firnága László, IQSOFT

A Centura cég stratégiája a folyamatos fejlesztés. A „divatot" követve a hagyományos, strukturált környezetről (3.0 verzió) lassan áttért az objektum alapú program környezetre (4.0 – 5.0 verziók). Az elmúlt két év internetes inváziója is mély nyomot hagy a Centura stratégiájában.

Mivel a siker követelménye „az árral együtt úszni" (néha az ár ellen jobb, csak fárasztó, de edzettebb leszel tőle), a Centura is bedobta ötleteit, megvalósításait, melynek segítségével az internet világ lehetőségeit is élvezhetik a Centura fejlesztők, felhasználók. Mivel most az elosztott OO a kulcsszó, ezért azokról a termékekről nem lesz szó melyek érintik az internetet de nincs közük a OO-hoz (pl. QuestWeb).

Három termékről lesz szó, ezek a: **Patriot, Web Developer és Tomahawk.** Ezek a fedőnevek olyan megoldásokat takarnak melynek segítségével a megszokott környezetünkből (már mint Centura)

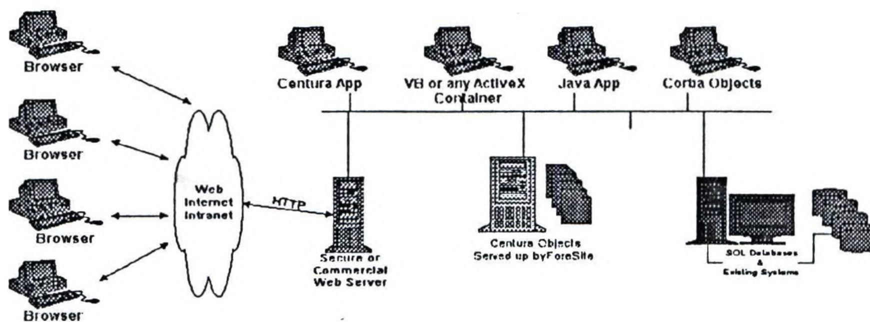


készíthetünk olyan programokat melyek változtatás nélkül futtathatóak kliens/szerveres vagy HTML üzemmódban, úgy hogy az adatbázis kapcsolat megmarad függetlenek és natívnak,

és egy tranzakció élettartalma nem korlátozódik egy HTML lapra, hanem akár a program elejétől a végéig is tarthat.

Ennek a megvalósítására a Centura a Foresite nevű aplikációs szervert használja, mely képes arra, hogy az „webes" kéréseket továbbítsa az alkalmazás felé és az onnan jövő válaszokat/ablakokat HTML formátummá gyúrja és küldi a kliens böngésző felé.

Külön csemegének szánom annak a terméknek a bemutatását (Tomahawk - CTD 2.0) melynek segítségével programjainkból JAVA kódot generálhatunk, vagy direkt módon JAVA nyelven programozhatunk.



A Centura nem ígér kevesebbet mint azt, hogy többrétegű alkalmazásokat tudunk fejleszteni. Ezáltal az üzleti szabályokat egy független szerverre, jelen esetben Centura Foresite nevű objektum szerver, tehetjük és onnan: Centurából, JAVA-ból, CORBA, DCOM protokoll segítségével férhetünk hozzá. Természetesen alkalmazásainkba ActivX és Java Beans objektumokat is integrálhatunk.

Firnága László

Workflow és az Internet

Tihanyi Péter, IQSOFT

A workflow technológia

- irodaautomatizálás
- munkafolyamatok nyomonkövetése (irodai, vállalati, üzleti)
- tetszőleges tevékenységek egymásutánja (pl: brókerházak telefonügyletei, hiteligénylés-elbírálás - bankok, kárcsetügyintézés - biztosítók, panaszügyintézés, engedélyezés - önkormányzatok, stb.)
- nem "bedrótozott" folyamatok
- a felhasználó konfigurálhatja a rendszert (számítástechnikai - de nem rendszerszervezési [!] ismeretek nélkül)
- résztvevők szintje: a rendszer "fogja a kezüket"
- szigorú és ad-hoc folyamatok
- management szint: áttekintés gombnyomásra
- eredete: workgroup, e-mail, stb.
- kapcsolat a dokumentumkezeléssel: elvileg függetlenek, gyakorlatilag azonban szorosan együttműködnek (pl. DOKTÁR-ArchiWare integrálás)

Együttműködés: Workflow és az Internet

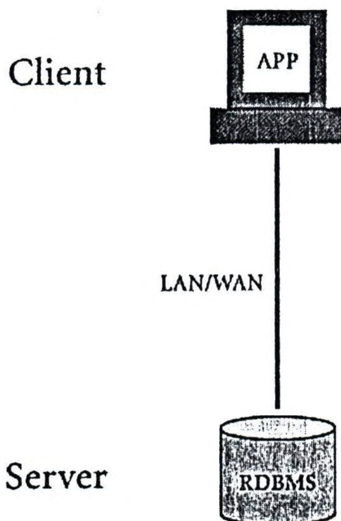
- A workflow az Internetet továbbítási mechanizmusként használja:
  - A workflow-t implementálni lehet Internet-en alapuló alkalmazásként (W*f*MC Interface 2)
  - Workflow rendszerek kommunikálhatnak más workflow rendszerekkel Interneten (extraneten) keresztül (W*f*MC Interface 4)
- A workflow szerveren eltárolt adatokhoz Web típusú hozzáférés (W*f*MC interface 5)
- A folyamat definíció meghív Internet Applet-eket
- A workflow Internet objektumokat vezérel

A KOMPONENS ALAPÚ ARCHITEKTÚRA 4GL SZEMSZÖGBŐL

Pokó István, Poko.Istvan@axis.hu
Axis Számítástechnikai Kft.

A modern 4GL RAD eszközöknek egyre több olyan tulajdonsággal kell rendelkeznie, mely megfelel azoknak a kihívásoknak, amelyeket napjaink vállalatai megkövetelnek. Ezért először egy rövid áttekintést szeretnék tenni az üzleti alkalmazások architektúrájának fejlődéséről: melyek azok a tényezők, amelyek befolyásolják a hagyományos kliens/szerver felépítés elmozdulását az osztott, komponens alapú modell irányába. Szeretném bemutatni a Sybase cég Adaptív Componet Architecture (ACA) elképzelését, melynek segítségével létrehozható a mai igényeket teljes mértékben kielégítő komplex, web alapú, robusztus alkalmazás, a szervertől a köztes rétegen át egészen a kliens oldalig terjedően.

Az 1990-es évek elején a PC-k már olyan komoly erőforrásokkal rendelkeztek, hogy a LAN hálózatokkal párosulva kialakították egy új vállalati számítástechnikai infrastruktúrát. Ez a változás egy újabb forradalmat jelentett a szoftverpiacon, hiszen a központi feldolgozásra optimalizált "mainframe" rendszerekről át kellett térni egy kifinomult, decentralizált kliens/szerver felépítésű adatbázis kezelésre. A változás természetesen befolyásolta a fejlesztő eszközöket is, hiszen ekkor jelentek meg az objektum orientált programozás, a könnyen tanulható és használható 4GL nyelvek, az előre megépített objektumosztályok, a grafikus fejlesztői felület fogalmak és szolgáltatások. A fejlesztő cégek sikeresen megépítették azokat az első kliens/szerver alkalmazásokat, amelyekben egyidőben több végfelhasználó gépén kliensként futottak a programok, és ezek hálózaton keresztül interaktív kapcsolatot tartottak egy központi adatbázis szerverrel.



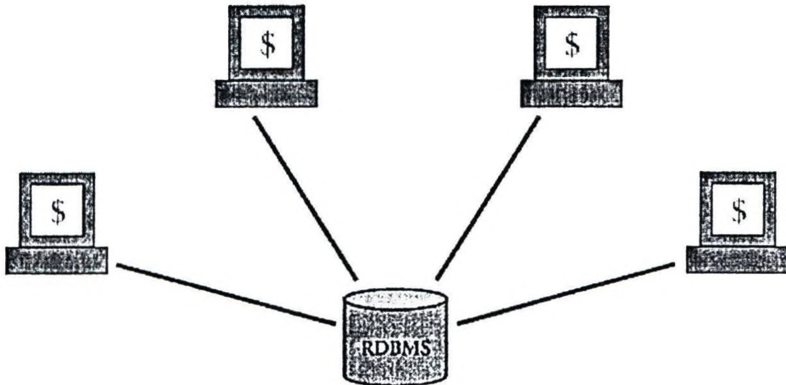
1. A kliens/szerver felépítés

A PowerBuilder fejlesztőeszköz az elsők között biztosította a fejlesztők számára, hogy gyorsan megépíthessék alkalmazásaikat az új kliens/szerver paradigma szerint. Megjelenése után két évvel a PowerBuilder de facto ipari szabvánnyá vált a professzionális csoportos alkalmazásfejlesztő környezetek között. 1994-re a PowerBuilder vezető pozíciót ért el az alkalmazásfejlesztő eszközök piacán. A siker többek közt annak köszönhető, hogy a kifejlesztett nagy teljesítményű, kifinomult alkalmazások ténylegesen, nagyvállalati szintű környezetben is kielégítették a felhasználók igényeit.

Az 1990-es évek közepére az akkor már hagyományosnak mondható kliens/szerver architektúra kliens oldalán két igen komoly probléma jelentkezett: 1.) a "kövér" kliens és 2.) az alkalmazás adminisztráció.

A "kövér" kliens alkalmazások hátránya abban mutatkozott, hogy hatékony futtatásukhoz igen komoly erőforrásokat igényeltek a kliens oldali PC-ken CPU teljesítmény, diszk terület és RAM méret tekintetében. Az erőforrás-igények az alkalmazások bonyolultságának növekedésével egyre jobban fokozódtak, melyeket csak drága hardver beszerzésekkel lehetett kielégíteni, amit ha nagyvállalati méretben több száz PC-re vetítünk, igen komoly költségnövekedést eredményezett.

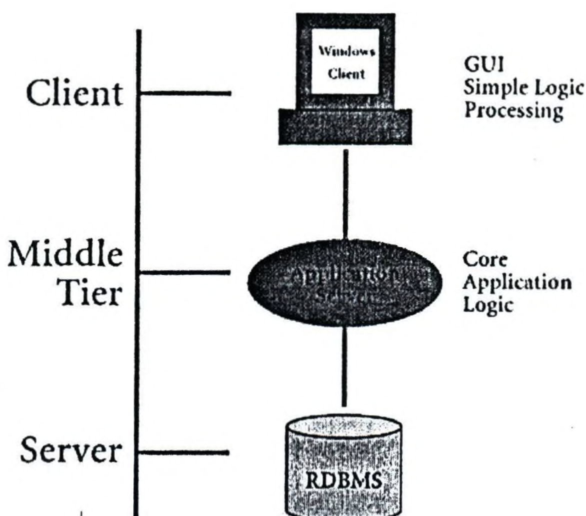
A második probléma, hogy a kliens/szerver modellben a kliens oldali adminisztráció igen nehézkes. A modell felépítéséből adódóan az alkalmazás logika minden egyes felhasználónál telepítve van, így ezek karbantartása, az új verziók telepítése igen nagy felelősséget és átfutási időt okoz, amelynek szintén költségnövelő hatása van.



2. A "kövér" kliens

1995-re egy új variációja jelent meg a hagyományos kliens/szerver modellnek, mely arra hívatott, hogy megoldja az előbbiekben felvázolt problémákat. Az új architektúrában a kliens oldalt két részre osztották: egy kliens alkalmazásra és egy alkalmazás szerverre. A kliens alkalmazás feladata a felhasználói felület megjelenítése, és néhány egyszerű logikai ellenőrzés végrehajtása. Ez a program egy alacsony költségű PC-n is futtatható, és minimális karbantartást igényel. Az alkalmazás magja - amit hívjunk üzleti logikának - az alkalmazás szerverben található. Ez a szerver egy köztes rétegben a kliens és az adatbázisszerver között található, mely által egy új, három rétegű

architektúra jön létre. Fizikailag valamilyen hálózaton keresztül érik el szolgáltatásait a kliensek, amely szükség esetén kapcsolatot teremt az adatbázisszerverrel, és a kívánt információkat visszajuttatja a kliensnek, amely azokat megjeleníti.



3. A három rétegű architektúra

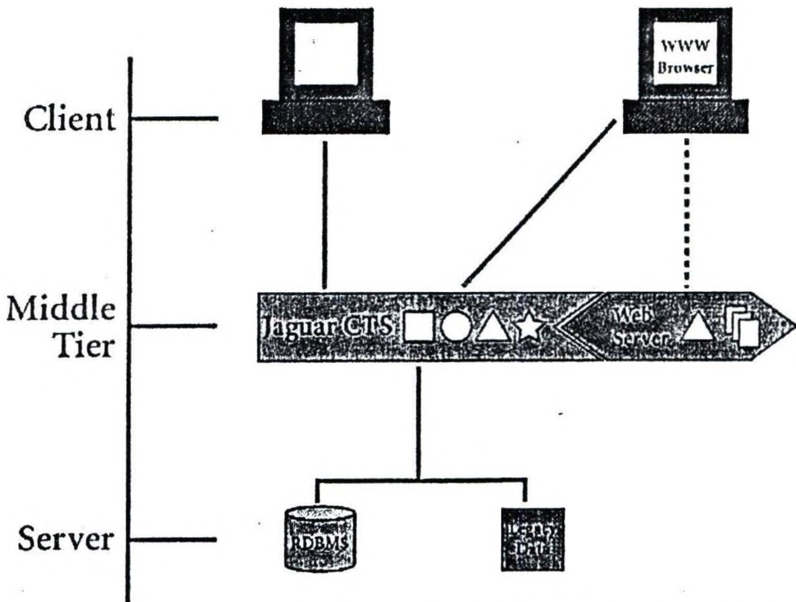
Összefoglalva: a három rétegű architektúra segítségével sikerült megoldani a hagyományos kliens/szerver felépítés problémáit, hiszen az üzleti logikát egy központi alkalmazás szerverre telepítve egyszerűbbé és olcsóbbá válik a változások követése és azonnali érvényre juttatása, míg a kliens oldalon maradó programrészek már egy sokkal kisebb erőforrású PC-n is futtathatóak.

A PowerBuilder 5.0 megjelenésével a fejlesztők először jutottak olyan lehetőséghez, hogy szabadon eldönthessék: hagyományos, vagy n-rétegű alkalmazást akarnak építeni. Alkalmazás szervereiket telepíthették UNIX vagy Windows NT gépekre, míg kliens oldalon használhattak Windows, UNIX és Macintosh operációs rendszerű gépeket is.

A három vagy többrétegű modellnek még egy nagy előnye van, mégpedig az, hogy kitűnően alkalmazható az Interneten. A Netscape Internet böngészőjének 1994-es megjelenésével egy igen dinamikusan növekvő üzleti és marketing lehetőség alakult ki a cégek számára. A létező Internet szabványokra alapozva igen könnyen megalkotható a kliens/szerver modell web alapú verziója, hiszen a web böngésző megfelel a kliensnek, a web szerver a szervernek és a http protokoll a kommunikációs csatornának. Az Internetben lévő lehetőségekre egyre több cégvezető ismert rá, és természetesen minél előbb ki akarta aknázni azokat. Az informatikai vezetőknek így meg kell küzdeniük azzal, hogy egyidőben elérést biztosítsanak a helyi hálózaton a belső munkatársaknak, akik különböző platformokat használnak az üzleti alkalmazások futtatására, valamint biztosítani kell, hogy a világon bárhonnan el tudják érni a cég web lapjait anélkül, hogy a külső felhasználók fizikailag valóban rákapcsolódnának a belső hálózatra. Az alkalmazásfejlesztők számára ez azt jelenti, hogy olyan Internetes üzleti alkalmazásokat kell fejleszteni, melynek telepítése,

karbantartása megoldható egy központi biztonsági szerver használatával (adminisztráció nélküli kliens), valamint elérhető a világon minden felhasználó számára bármilyen platformmal is rendelkezzék. Az ilyen típusú alkalmazások kifejlesztéséhez természetesen megfelelő eszközökre is szükség van, melyeket két nagy csoportra oszthatunk. Az egyikbe azok tartoznak, amelyek a klasszikus Internetes web-server/page-based alapú modellt támogatják, és megpróbálják adatbázis-elérési, állapot- és kapcsolat-felügyelő, esemény vezérelt logikai képességekkel felruházni fejlesztőeszközeiket, azonban ezek a kezdeményezések még gyerekcipőben járnak ahhoz, hogy valóban meg lehessen valósítani velük egy hathatós alkalmazásfejlesztési támogatást. A másik csoportba azok tartoznak, akik meglévő hagyományos kliens/szerver felépítésű fejlesztőeszközeiket egészítik ki webes képességekkel. Mindkét változat gyengesége a modellben található, hiszen sem a web server/page alapú, sem a hagyományos kliens/szerver architektúra nem képes kielégíteni az előzőekben felsorolt web alapú üzleti alkalmazások követelményeit.

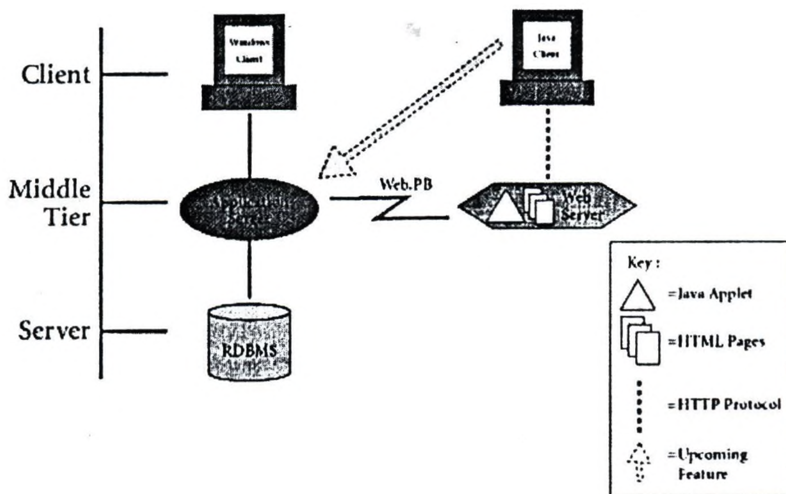
A Sybase cég által kidolgozott modell nem csupán valamiféle kiterjesztése a már létező tranzakciós vagy web page alapú alkalmazásoknak, hanem kombinálja a két terület által felkínált előnyöket. Hiszen egy modern web alapú alkalmazásnak előnyére válna egy tranzakció kezelő köztes réteg és a hagyományos adatbázis elérés, amellett, hogy megmaradna a server/browser kapcsolat. A Sybase Adaptive Component Architecture-ben egybeolvad a web szerver a köztes rétegben található alkalmazás szerverrel, és közösen alkotnak egy rugalmas tranzakciószervert. Ennek a köztes rétegű tranzakcószervernek a szolgáltatásai közé tartozik mind a közvetlen, mind az Internetes kapcsolatú felhasználók támogatása, a háttérben lévő adatbázisszerver kapcsolatok felügyelete, befogadja az üzleti logikákat tartalmazó objektumokat. A Sybase által kifejlesztett tanzakciószerver neve: Jaguar Component Transaction Server (CTS).



4. A Jaguar CTS. Magába foglalja az üzleti logikát, melyet mind a közvetlen, mind a web alapú kliens elérhet.

A web alapú alkalmazások fejlesztőinek egyik legnagyobb problémája az, hogy a hagyományos kliens oldali felhasználói felület megjelenését biztosítani kell a webes kliensek gépén is. A webes kliens egy böngészőn keresztül látja a világot, ezért az alkalmazásokat olyan darabokra kell szétszedni, hogy azt a böngésző elérhesse, letölthesse, futtathassa. Mindezeket dinamikusan kell megoldani, hiszen a webes kliensek ismeretlen időben, számban és platformon használják az alkalmazásokat. Egyes szakértők szerint a Java fog megoldást nyújtani a kliens oldali logika problémáira. A Sun Java programnyelve és végrehajtási környezete biztosítja, hogy az ebben megírt alkalmazás kód (Java applets) bármilyen platformon letölthető és virtuálisan futtatható. A Java magába foglal bizonyos biztonsági funkciókat is, hiszen ezt a nyelvet használva a fejlesztő nem tudja elérni a kliens gép erőforrásait, ezért nem tud egy "file save" vagy egy "disk format" parancsot kiadni. A Java specifikál egy Java Virtual Machine (Java VM) futtató környezetet is, melynek feladata, hogy a kliens gépeken ellenőrizze a letöltött kódot, és értelmezze azt a futtatás alatt. A Java fejlesztők e technika használatával képesek Java appleteket, Java servleteket (szerver oldali Java kód), JavaBeans-eket (Java komponens) vagy komplett Java alkalmazásokat írni.

A Java appletek/Beanek ideálisak a web alapú kliens oldali logika megvalósítására. Ezek a kliens oldali komponensek közvetlenül menedzselhetők, elérhetők és letölthetők Sybase Jaguar CTS segítségével, és így létrejöhet egy komplett web alapú alkalmazás megoldás. Egy másik hathatós megoldás lehet a PowerBuilder fejlesztő eszköz Web. PB nevezetű eszköze, melynek segítségével a PowerBuilder alkalmazás-szerverben található alkalmazás-logikát a Java kód képes elérni. A PowerBuilder 6.0 fejlesztő eszközben megtalálható egy Java proxy generátor, mely megengedi a Java klienseknek, hogy - a web server és a Web.PB használata nélkül - közvetlenül elérjék a PowerBuilder alkalmazás szervert. A PowerBuilder technika segítségével a 3GL típusú nyelv helyett egy produktív 4GL RAD környezetben fejleszthetik komponenseiket.



5. A PowerBuilder a Java kliensek számára közvetlen elérést fog biztosítani az alkalmazás szerverhez

Jelenleg a 4GL eszközöknek nemcsak a Javat kell támogatniuk, hanem a Plug-in és az ActiveX komponenseket is. A PowerBuilderben már most használhatóak ezek a komponensek, de futtatásukhoz egy PowerBuilder VM (2-5 DLL) és egy ActiveX vagy Plug-in DLL szükségeltetik a web kliensen. A PowerBuilder némi előnyt nyújt biztonsági téren, hisz az általa használt ActiveX vagy Plug-in DLL-ek biztosítják a felhasználó számára, hogy ne hajthasson végre olyan kódot, amely megsértené érdekeit. A PowerBuilder komponensek és a PowerBuilder VM természetesen elérhető a legnépszerűbb Windows és Macintosh platformokon. E technika használatával a jelenlegi PowerBuilder fejlesztők a létező kliens alkalmazásaikat minimális módosítással áttehetik web kliens felületre, míg a webes alkalmazást fejlesztők egy valódi 4GL RAD környezetben dolgozhatnak, mely köztudottan igen nagy produktivitású.

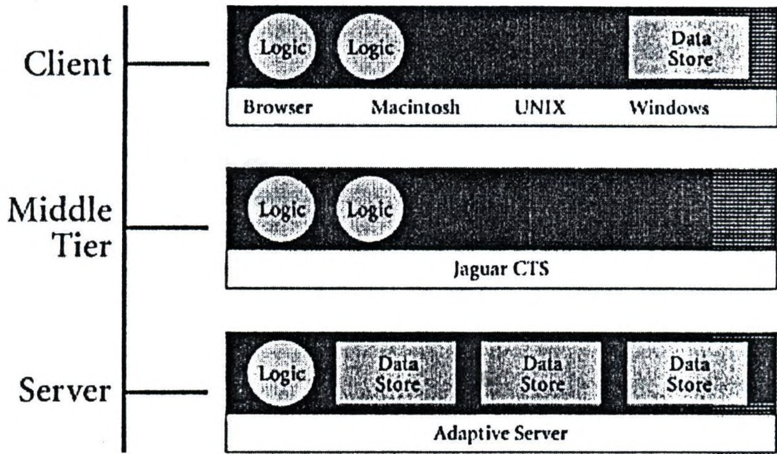
A Java mellet szólni kell arról a már meglévő két szabványról, melyek szintén szóba jönnek a komponens alapú alkalmazások fejlesztésénél. A CORBA és a COM/DCOM szabványok egymástól függetlenül lettek létrehozva az OMG konzorcium és a Microsoft által. A CORBA szabvány alapján készített objektum együttműködhet más CORBA objektumokkal, végrehajthatják azok metódusait, elérhetik azok attribútumait. A COM/DCOM szabvány alapján elkészített ActiveX komponenseknek együttműködési képessége van. Bármelyik szabvány felhasználásával fejlesztett komponensek - függetlenül attól, hogy ki és milyen eszközzel írta azokat - a felhasználás széles skáláját biztosítják a fejlesztőknek. A szabványos komponensek természetükből adódóan könnyen karbantarthatók és újra felhasználhatók, mely által a létrehozott alkalmazások is rugalmasak és kellően modulárisak lesznek.

A holnap igényeit kielégítve a PowerBuilder 6.0 biztosítja az osztott, komponens alapú alkalmazások létrehozását. A fejlesztőeszköz által képviselt architektúra megengedi a következő generációs komponensek felhasználását a fejlesztés során. A PowerBuildert használó fejlesztők mindezt a jól ismert 4GL RAD környezetben végezhetik a népszerű Powerscript nyelv és PowerBuilder objektumosztályok használatával. A hagyományos PowerBuilder komponenseken kívül a fejlesztők használhatnak a PowerBuilder által generált komponenseket, C++ komponenseket, ActiveX és CORBA komponenseket, a PowerBuilder által generált JavaBean-eket, melyek akár az alkalmazás köztes rétegében is elhelyezkedhetnek. A PowerBuilder által generált komponensek menedzselése a köztes rétegben történhet PowerBuilder alkalmazásszerverrel, Jaguar CTS szerverrel vagy Microsoft Transaction Serverrel.

Összefoglalva: a Sybase Adaptív Komponens Architektúrája figyelembe veszi azokat a jelenlegi fejlődési trendeket, amelyek megtalálhatók az osztott alkalmazások, web alapú alkalmazások és a nagy vállalati rendszerek felépítésében, és definiálni fogják azt az átfogó és rugalmas architektúrát, mely meg fogja határozni a jövőbeni üzleti alkalmazások szerkezetét. Újrafelhasználható, szabványos komponensek lesznek az alapjai azoknak az építőkockáknak, melyek a holnap üzleti alkalmazásait alkotják. Ezek a komponensek valószínűleg független gyártóktól fognak származni, egy részük házi készítésű lesz, másrészük a kereskedelemben árusított termékek lesznek. A komponensek szabványosítottak lesznek, teljesíteni fogják azokat az előírásokat, melyek megtalálhatók az ActiveX, CORBA és JavaBean specifikációkban. Megtalálhatók lesznek a kliens gépeken, a köztes réteg alkalmazás szerverében és a háttérben lévő adatbázisszerverben is, használni fogják azokat a szabványos protokollokat, melyeket a COM/DCOM és a CORBA definiál. Nagy teljesítményű tranzakciószerverek fogják menedzselni az üzleti (alkalmazás) logikai komponensek futtatását. A holnap osztott, komponens alapú

alkalmazásai heterogén környezetben fognak működni, többtípusú adatbázis szervereket fognak elérni és hálózati szolgáltatásaik a LAN-on túl ki fognak terjedni az Internet/intranet/extranet hálózatokra is.

A Sybase Adaptív Komponens Architektúrája egy olyan modell, amely az előző bekezdésekben felvázolt fejlődési irányzatot teljes mértékben kiszolgálja. Meghatározza a kliens, a köztes és szerver rétegeket, a végrehajtási szempontokat, definiál két komponens típust (logikai komponens és adattár), melyet bármely rétegben használhatunk. Tehát mindkét fajta komponenst használhatjuk az adatbázis szerverben, menedzselhetjük őket a köztes rétegben a Jaguar Component Transaction Server segítségével, a kliens oldalon pedig futtathatók mind web böngészőben, mind közvetlen adatbázis kapcsolatban lévő alkalmazásban.



5. Sybase Adaptive Component Architecture (ACA)

A PowerBuilder 6.0 4GL RAD eszköz megvalósítja azt a fejlesztési környezetet, amelynek segítségével megalkothatjuk azokat a logikai komponenseket, amelyek képesek meghajtani ezeket az alkalmazásokat. A PowerBuilder 6.0 és a Sybase Adaptive Component Architecture együttesen egy olyan nagy teljesítményű fejlesztőeszköz és alkalmazás architektúra, mely biztosítja a sikeres üzleti alkalmazások létrehozását.

OO technológiák az Oracle stratégiai fejlesztéseiben

Klotz Tamás, tklotz@hu.oracle.com
Oracle Hungary

Összegzés

Az Oracle8 evolúciós módon vezeti be az objektumokat felhasználói számára. Ez az objektum és a relációs modell nyom nélküli integrációját biztosítja. A felhasználóknak többé nem kell választaniuk a relációs modell egyszerűsége és rugalmassága, valamint az objektum modell intuitivitása között. A felhasználók a legmegfelelőbb adatmodellt használhatják alkalmazásaikhoz, és előnyt jelent számukra a produktivitási haszon, valamint a objektumok kényelmes karbantartása.

Az Oracle8 objektumok kitapossák az utat az adatbázis bővíthetőségéhez. A bővíthetőséggel az Oracle8 adatbázis platformmá válik a adatkazetták fejlesztése számára, amelyek olymódon bővítik az Oracle8 kapacitását gazdag, struktúrálatlan adatok kezeléséhez.

Bevezetés

A kompetitív piac nyomása arra ösztönzi a cégeket, hogy időben és költséghatékony módon építsék és fejlesszék alkalmazásaikat. A cégeknek egyre inkább olyan alkalmazásokat kell építeniük, amelyek szorosan igazodnak üzleti modelljeikhez és eljárásaikhoz. Például, a biztosító társaságoknak csökkenteniük kell az arra fordított időt, hogy a potenciális ügyfél alkalmasnak minősül-e a személyes vagy az üzleti biztosításra. A Wall Streeten állandó igény mutatkozik arra, hogy a pénzügyi szolgáltatást nyújtó cégek rendszeres jelleggel új pénzügyi eszközöket hozzanak létre. A gyárosoknak szerelősoraikat még rugalmasabbá kell tenniük annak érdekében, hogy gyorsan

alkalmazkodjanak a változó piaci feltételekhez, és hogy termékeik a fogyasztók ízléséhez igazodjanak. Ezek az üzleti feltélelek megkövetelik, hogy az eljárásokat támogató információs rendszerek és alkalmazások nagyon rugalmasak és könnyen átalakíthatóak legyenek. Közelebbről, az üzleti alkalmazások tervezési, fejlesztési, karbantartási és telepítési költségeit csökkenteni kell.

Ugyanakkor egyre nő az üzleti alkalmazások által kezelt adatok komplexitása és gazdagsága. A World Wide Web bevezetése lehetővé tette az olyan anyagok közlését, amelyekben szöveg kép és audio adatok vannak. Mivel a cégek a Web kereskedelmi felhasználásán dolgoznak, IT alkalmazásaiknak - a skaláris adatok kezelése mellett - kezelniük kell tudni a dokumentumok, képek, hanganyag és video formájábant megtestesülő adatokat. Ahogyan a számítógépes hardver egyre erőteljesebbé válik, az alkalmazásoknak is meg kell kísérelniük, hogy az adatokat újszerű és komplikált módon vessék össze. A cégeknek sok esetben azt is meg kell próbálniuk, hogy ezt a komplex feldolgozást többszörös csomópontokon keresztül terjesszék egy hálózaton.

Ezen tanulmánynak a célja, hogy az olvasó számára áttekintést nyújtson az Oracle8 elképzelésről, arról, hogy az Oracle8 hogyan foglalkozik ezekkel a kérdésekkel, továbbá, hogy áttekintést nyújtson az Oracle8 néhány kulcsfontosságú komponenséről.

Indokolás

Eltérés mutatkozik a relációs adatbázis rendszerek (RDBMS) által támogatott adatmodell (sorok és oszlopok) és az alkalmazások által használt adatmodell (sok beágyazási szinttel rendelkező komplexum) között. Ez az eltérés arra ösztönözte a fejlesztőket, hogy először is gondosan térképezzék rá alkalmazási adatmodelljüket az egyszerű relációs táblákra. Ezt követően az allkalmazási adatmodellt újjáépítik az alkalmazásban található relációs táblákról. Az alkalmazások között

nincs megosztott logika. Ez nem csak időrabló folyamat, de hibára is hajlamosít.

Sok alkalmazás tervezési és fejlesztési csoport vonta le azt a következtetést, hogy az alkalmazások objektum-orientált módon való fejlesztése eleget tehet ezeknek az igényeknek és kihívásoknak. A nagyvállalati számítógépes környezet részeként olyan adatbázis rendszerre van szükség, amely rendelkezik a jelenlegi relációs adatbázis rendszerek skálázhatóságával és robosztusságával amellett, hogy támogatást biztosít az objektum technológia számára.

Az ilyen adatbázisnak komplex, struktúrált adatokat kell tudnia tárolni és kezelni, valamint olyan nagy, struktúrálatlan, állomány-specifikus adatokat, mint például szöveg, kép, audio és video. Képesnek kell lennie lekérdezési funkció biztosítására az ilyen új típusú adatokra nézve. Emellett, a relációs adatbázis technológia jelenlegi felhasználói hatalmas befektetéseket eszközöltek a már meglévő alkalmazások és adatbázisok kiépítésébe. Olymódon kell támogatást biztosítani az adatbázisban található objektumok számára, hogy a régi relációs alkalmazások, sémák és adatok együtt tudjanak létezni az új objektumbázisú sémával, adatokkal és alkalmazásokkal. Az adatbázisban található objektumok számára úgy kell biztosítani a támogatást, hogy ne csorbítsák a relációs adatbázisok jelenlegi skálázhatóságát, teljesítményét és adattovábbítási kapacitását.

Az Oracle8 tervezési célkitűzései a következők:

1. Az Oracle7 típusú rendszer megerősítése egy objektum-relációs típusú modellel.

2. Olyan infrastruktúra, amely biztosítja a objektumbázisú hozzáférést az Oracle8-ban tárolt adatokhoz, és a minimumra csökkenti a fenyegető eltérést az alkalmazáshoz szükséges adatmodell és az Oracle8 által támogatott adatmodell között.

3. Kereteket biztosít az adatbázis bővíthetőség számára, hogy támogassa az adatkazetták kifejlesztését a különböző csomópontokra.

Emellett, az adatbázissal és a objektumkapcsolási gerinccel egyaránt jól együttműködő, a objektumtechnológiára alapozott tervezési, fejlesztési és vezetési eszközök teljes köre biztosítja az üzleti rendszerek új generációjának kiépítését.

Előnyök

Az Oracle8 objektumok nem tolakodó módon bővítik az Oracle7 kapacitását. A objektumok előnyei közül néhányat az alábbiakban ismertetünk.

Megőrzi a felhasználók befektetéseit

Az Oracle8 objektumok a felhasználói befektetést két módon őrzik meg: képesek együttélni a relációs adatokkal és hozzáférhetők SQL-en keresztül. Az objektum típusokat akkor lehet használni, ha a rendszer adat típus engedélyezett, vagyis egy oszlop típusa objektumtípus lehet. Ezen túlmenően, az Oracle8 úgy bővíti a nézet mechanizmust, hogy lehetővé teszi a felhasználók számára, hogy objektumokat állítsanak össze a relációs táblákról. Erre, mint Objektum Nézetre hivatkoznak. Objektum nézeteket SQL utasításon keresztül hozhatunk létre és azok módosíthatók. Ennek következtében nincs szükség a meglévő adatbázisok és alkalmazások módosítására annak érdekében, hogy kihasználják a objektumok nyújtotta előnyöket. Az objektumokat SQL utasításokkal is létre lehet hozni és el lehet érni, azzal a lekérdezési nyelvvel, amelyet a felhasználók jól ismernek.

Továbbfejleszti az Oracle7 technológiákat

Az Oracle7 olyan érett RDBMS, amely azt a teljesítményt, megbízhatóságot és rendelkezésre állást biztosítja, amelyet a felhasználók elvárnak és igényelnek. Az Oracle8 úgy bővíti az adatmodellt, hogy támogassa az objektumokat, de egyben megőrizze az Oracle7 minden technológiai sajátosságát, és így a felhasználók a létfontosságú alkalmazások során a megszokott biztonsággal használhatják a objektumokat.

Javítja a felhasználói produktivitást

Az Oracle8 objektumok úgy javítják a felhasználói produktivitást, hogy kiküszöbölik a felhasználói alkalmazások és az adatbázis közötti adattípus keveredést. Az objektumok lehetővé teszik a felhasználók számára, hogy egy olyan "intelligens" adattípust határozzanak meg, amely a leginkább alkalmas alkalmazásaikhoz, és módszerekkel lássák el az objektumtípust. Mindez kiküszöböli azt az időt, amelyre azért van a felhasználóknak szüksége, hogy az adatokat egyszerű táblázatokban normalizálják, majd újból összekomponálják azokat alkalmazásaikban. Az objektumokba zárt módszerek lehetővé teszik a felhasználók számára, hogy a módszereket megosszák alkalmazásaik körében. Mindez nem csak a megduplázást küszöböli ki, de azokat a hibákat is csökkenti, amelyek a megduplázással járnak.

Bővíti az adatmodellt

Az Oracle8 objektumokat az Oracle7 típusrendszerének természetes bővítésével valósították meg. A objektumok együtt tudnak élni a relációs adatokkal ugyanabban az adatbázisban. Vannak olyan alkalmazások, amelyek természetesebb módon illeszkednek a relációs modellhez, és vannak olyanok, amelyek az objektum modellbe simulnak jobban. Az integrált objektumokkal a felhasználók objektumokkal és relációs adatokkal egyaránt rendelkeznek ugyanabban az adatbázisban. A legmegfelelőbb adatmodellt használhatják alkalmazásaikhoz. Ezen túlmenően, a felhasználók

adataikat még mindig a relációs táblákban tárolhatják (néhány alklmazásra nézve), de azokhoz, mint objektumokhoz, más alkalmazásokhoz definiált Objektum Nézeteken keresztül férhetnek hozzá. A objektumok viszont SQL-en keresztül elérhetők, beleértve az objektum és a relációs táblák közötti kapcsolódási pontokat is.

Támogatja az ipari szabványokat

Az Oracle8 objektumok megfelelnek az ipari szabványoknak. Az SQL szabvány ipari lekérdezési nyelv a relációs adatbázis számára. Az objektumokat az ANSI X3H2 által definiált SQL-en keresztül lehet kezelni. Az objektumok kezelése érdekében az SQL-t jelenleg bővítik az ANSI X3H2 szerint. A bővítésekre általában, mint SQL3 hivatkoznak. Az Oracle az ANSI X3H2 aktív tagja és komolyan kivette részét a bizottság munkájából. Az Oracle aktív tagja az Object Management Group (OMG)-nak is.

A objektumokon túl: Bővíthetőség és adatkazetták

Vegyünk példának egy olyan egészségügyi ellátási információs rendszert, ami a páciensek adatait kezeli. Az ilyen rendszerek napjainkban egyszerű, skaláris adatokat tartalmaznak a páciens neve és kora, a diagnózis szöveges leírása, stb. tekintetében. A tipikus páciens dossziék azonban ultrahang vagy röntgen felvételeket is tartalmaznak. Bár az elmúlt években a relációs adatbázisok támogatták a nagy bináris adatok tárolását, nem biztosították a lekérdezést e nagy bináris adatok tekintetében. Ez azt jelentette, hogy a páciens dossziék olyan alapvető komponenseit, mint például az orvosi felvételek, ritkán kezelték a skaláris adatokkal, és ez a rendszerből származtatott információk rosszabb minőségéhez vezetett.
Annak érdekében, hogy ezeket a nagy, gazdag, struktúrálatlan adatokat teljes mértékben kezeljék az adatbázisban, az Oracle az adatbázisban található számos komponenshez interfészt definiál. Ezek a

komponensek lehetővé teszik a felhasználók számára, hogy definiálják saját indexelési módszerüket, optimalizációs költségüket, adminisztrációs módszereiket (import/export, betöltés), stb. Ezekkel az interfészekkel az Oracle vagy külső szállító specializált adatkazettákat fejleszthet ki e gazdag, struktúrálatlan adatok kezelése érdekében.

Kulcsfontosságú komponensek

Az Oracle8 objektumokat egy olyan egyedüli típusú rendszer támogatja, amely egyaránt kezeli a objektumt és a relációs adatokat. A rendszer kohéziós jellege egyszerűsíti az alkalmazás fejlesztést, könnyebb együttműködést alakít ki a fejlesztők és az alkalmazások között, továbbá egyszerűsíti az alkalmazások karbantartását és fejlesztését. Az Oracle8 kulcsfontosságú komponensei a következők:

Típus kezelő

A objektumok olyan felhasználó által definiált típusok (objektum típusok), amelyek egy vagy több sajátossággal, valamint nulla vagy sok módszerrel rendelkezhetnek. A objektum típusokat bárhol lehet használni, ahol rendszer típust alkalmaznak, vagyis mint oszlop típusokat a relációs táblákban. Az Oracle8 emellett támogatja a objektum táblák koncepcióját is. Ezek a objektumok táblái. Az objektum táblában található minden egyes objektum rendelkezik egy a rendszer által generált azonosítóval (OID). Az OID globálisan egyedülálló és sohasem újra felhasználható.

Az objektum típus attribútuma lehet rendszer típusú (szám, karakter, stb.), objektum típusú vagy REF típusú. A REF típus hivatkozás egy, az objektum táblán tárolt objektumra. A módszer olyan funkció, amely az objektum típus egy példányán működik. megvalósítható PL/SQL-ben vagy olyan 3GL-ben, mint például C/C++. Az egyéb típusokat a standard SQL-lel hozható létre a CREATE TYPE utasításon keresztül.

Nyelvi interfész

A objektumok definiálhatók és hozzáférhetők a SQL-en, PL/SQL-en és OCI-n keresztül. Az SQL a legalkalmasabb az adatbázis lekérdezésére, objektumok létrehozására és manipulálására. Az SQL és a PL/SQL az SQL3-mal összehangban kerül bővítésre a objektum típusok kezelése érdekében.

Nézetek

Bővítik az Oracle8 nézetekkel kapcsolatos képességeit, hogy lehetővé tegyék a felhasználók számára a relációs nézetek definiálását a objektumokon és a Objektum Nézeteket a relációs adatokon. A megerősített nézet kapacitás fenntartja a logikai és fizikális adatbázis tervek elkülönítését. Ez lehetővé teszi az alkalmazások számára, hogy transzparensen férjenek hozzá a relációs adatokhoz, mint objektumokhoz vagy a objektumokhoz, mint normalizált relációs adatokhoz.

Az Objektum Nézetek lehetővé teszik a felhasználók számára, hogy módszereket definiáljanak a bonyolult szemantikák feloldása érdekében.

Objektum cache-vezérlő

Az Oracle8 objektum cache-t biztosít a hatékony objektumkezeléshez az ügyféli oldalon. A objektumok másolatait be lehet vinni az ügyfél objektum cache-be. Az alkalmazások a memóriasebességen haladhatnak át a objektumokon és az aktualizálásokat vissza lehet vinni a tartós tárolóba a programozási interfészek bővítéseinek alkalmazásával (OCI).

A objektum cache az alábbi funkciókat kínálja az alkalmazásoknak:

- az adatbázis objektumok transzparens letérképezése "gazda" nyelvi objektumokra.
- A tartós adatok láthatatlan, hatékony memória kezelése. Az alkalmazásoknak nem kell aggódniuk a memóriának az adatbázis objektumok hozzáférése érdekében való kiosztása és felszabadítása miatt.
- transznacionális szemnantikát biztosít az ügyféli objektumok számára; vagyis az adatbázis munkaterületen a tartós objektumok változtatását propagálja.
- olyan interfészeket biztosít a komplex objektum eléréshez (pl. a kapcsolódó objektumok gráfja), amelyek csökkentik az egész hálózatot megkerülő utazásokat.
- gazdanyelvi megfelelőket biztosít az objektumok számára

Bezárt módszerek

A bezárt módszerek alapvetőek a objektumok számára. Az adatbázis szervernek kell támogatnia a szerver oldalon tárolt és végrehajtott bezárt módszereket. Az Oracle8 lehetővé teszi a felhasználók számára, hogy ezeket a módszereket a PL/SQL-ben vagy más 3GL-ben implementálják. Ezeknek a módszereknek biztonságosaknak kell lenniük és be kell tartaniuk az adatbázis minden integritási szabályát.

Nagy komplex objektumok

A objektumok azon nagy és komplex objektumok támogatására is alkalmasak, amelyek saját szemantikával rendelkeznek. Az Oracle8

támogatást biztosít a nagy komplex objektumok hatékony tárolásához. Ezek a nagy objektumok opcionálisan tárolhatók a tárolásra optimizált táblaterületeken vagy külsőleg olyan egyéb médiaeszközökön, mint például a Kodak Photo CD. A nagy komplex objektumok hatékony támogatásához biztosítania kell a nagy objektum részeinek véletlenszerű leolvasásait.

Objektum Típus Fordító

A Objektum Típus Fordító "gazda"-nyelv típusokat gnerál (pl. C típusok vagy C++ & Java osztályok) az adatbázisban található objektum típusok számára. Ezeket a generált típus definíciókat a "gazda"-nyelv alkalmazásokban használják az adatbázis objektumokhoz való transzparens hozzáférés érdekében. Emellett az objektum hozzáféréshez előzetes fordítóprogram támogatást is biztosítanak.

Következtetés

Az Oracle8 a objektumokat evolúciós módon vezeti be. A felhasználók számára teljes relációs és objektum interfészt, vagyis SQL-t, PL/SQL-t és OCI-t biztosít adataikhoz. Az Oracle8 objektumok az adatbázis bővíthetőséghez vezető utat is kitapossák, és így a felhasználók az igényekhez igazíthatják adatbázisaikat vagy a saját adatkazettájuk fejlesztésével, vagy pedig egy független eladótól származó adatkazetta installálásával.

Az Informix objektum-relációs architektúrája

Balogh Kálmán

Az RDBMS gyártók front-end eszközeikben már évek óta objektumorientált fejlesztést kínálnak. Az objektumorientáltságból fakadó előnyök azonban nem használhatók ki teljes mértékben, ennek ugyanis a kiszolgálók relációs volta korlátokat szab. Az Informix már 1996-tól rendelkezik az Illustra objektum-relációs kiszolgálóval. A kiszolgálók egységesítése, és a front-end eszközöknek az új Universal Serverhez való hangolása az előadás témája

- Az alkalmazások okozta kihívások
  - az Intranet/Internet jelentősége a felhasználók számára és a felhasználás minőségére
  - új alkalmazástípusok, OLAP
  - új adattípusok, korlátlan kiterjeszthetőség igénye (példákkal).
- A válaszok csoportosítása (alkalmazási területek, szabványok, megvalósítási technológiák)
  - objektumorientált DBMS-ek és korlátaik
  - az objektum-relációs megoldás.
- Az objektumorientáltságból fakadó előnyök
  - nyitottság, integrálhatóság, újrafölhasználási, testre szabási lehetőségek
  - specifikus osztálykönyvtárak, példák (Web, szövegkezelő, idősorozat, térképi, 2D, 3D, stb. DataBlade modulok).
  - tartalom szerinti keresés támogatása
  - komplex alkalmazások felépítése
  - hatékonyság.
- A Universal Server technológiája, mint az RDBMS-megoldások általánosítása
  - adatok, indexek és műveletek típusspecifikus kezelése
  - optimalizálás
  - tranzakciókezelés.
- Fejlesztés a nyitott szerverre - a minőség biztosításának technológiája
- Együttműködés az objektumorientált front-end eszközökkel
  - osztálykönyvtárak
  - alkalmazás partícionálás
  - OLE, ActiveX
  - Web
  - Java
  - 4GL rendszerek (Powerbuilder, Visual Basic, NewEra)
  - CASE eszközök.

ADATBÁZISTERVEZÉSI TAPASZTALATOK A MOL RT. FŐNIX PROJEKTJÉBEN

Lukovics László
Szabó József
MOL Rt. Kenőanyag Üzletág Komárom

1. Bevezetés

A MOL Rt. többi egységéhez hasonlóan a Kenőanyag Üzletágnál is - mely a hazai kenőanyag termelés központja, a valószínűleg jól ismert Carrier termékcsalád szülőhelye - változó az egyes területek informatikai támogatottsága. A kialakult rendszerek integrációja gyenge, egyes kapcsolható részek esetében is inkább a redundáns kialakítás dominál.

A legfontosabb jelenlegi rendszerek a következők:

- a DOKU rendszer, amely gyártási recepturák és minőségi előírások nyilvántartására, napi aktualitások szabályozására alkalmas;
- a TERMEL a termelés elszámolásokkal kapcsolatos nyilvántartást (anyagmozgások és anyagfelhasználások) biztosítja;
- folyamatirányítási rendszer a Komáromi Finomító Olajkeverő üzemében;
- folyamatirányítási rendszer a Komáromi Finomító Kenőzsír üzemében;
- további kisebb programok az egyes tervezési szakaszok elkészítéséhez (gördülő tervek, negyedéves, féléves, stb.), műszaknaplók vezetéséhez, anyagrendelések előkészítéséhez.

Mivel ezen rendszerek kialakítása régebben történt, ezért az akkor még jó technológiájú, de mára már elavultnak tekinthető Clipper-ben íródtak meg. Ez természetesen nem jelenti a kész rendszerek automatikus avultságát, hiszen képesek még a folyamatos működést biztosítani, de felmerültek olyan korlátok, melyek elhárítására már nem alkalmasak, illetve átalakításuk nem lenne arányban a kívánt befektetéssel.

2. Felmerült igények

Legfontosabb feladatot az egyes rendszerek integrálása, központi MOL SAP rendszerhez igazítása, közös adatbázison való működtetése jelentette, amely végrehajtásához a jelenlegi összes rendszer kódjában változtatni kellene, illetve adatszervezésüket jelentősen módosítani. Ekkor azonban fellépne a konkurens hozzáférések problémája, amire a Clipper nem nyújt kielégítő lehetőséget. Szakirodalmak tucatjai bizonyítják, hogy szóban forgó méretű rendszereket irreális elképzelés megfelelő biztonsággal kialakítani XBASE alapokon. Gondoljunk csak az indexelési problémákra. További gond lenne egy jogosultságkezelési rendszer kialakítása, amit be kellene építeni minden egyes alkalmazásba. Fenti okok alapján a jelenlegi rendszerek kódjait 30 - 80 %-ban meg kellene változtatni, illetve kialakítani egy közös adatbázist, amihez további erőforrások szükségesek.

Ennyi erőfeszítés eredményeképpen létrejöhetett volna egy integrált Clipperes rendszer, amely adatbiztonsága valószínűleg jobb lenne a jelenleginél - de gyengébb a kívánatosnál -, nagyobb rugalmasságot azonban nem biztosítana. Tovább is magában hordozná az igényt a technológiai váltásra, amit néhány éven belül újabb költségeket eredményezne, s ezzel bizonytalanná tenné a korábbi befektetések megtérülését.

Ezek alapján ésszerűbb volt egy olyan fejlesztés megkezdése, amely során technológiailag fejlett eszközökkel, relatíve gyorsan, nagy hatékonyságú rendszer alakítható ki.

3. A megvalósítás módja

A választás Oracle relációs adatbázis alapú kliens-szerver alkalmazások kialakítására esett, Centura fejlesztési környezetben. A feladat bonyolultsága megkövetelte egy tervezési módszertan használatát, amely segítségével áttekinthetőbb lett a projekt működése. Erre az OMT (Object Modelling Technique) módszertant választottuk, ami a jelenleg legfejlettebb objektum-orientált változatok közé tartozik. Szoftveres segédeszközként (CASE) a Platinum cég Paradigm Plus programcsomagját választottuk. Ennek segítségével könnyen dokumentálhatók a projekt egyes szakaszai, kézben tarthatóak a tervezési eredmények. Sok segítséget jelentett a programcsomag

részét képező fejlesztő nyelv amellyel elkészítettünk egy olyan generátort, ami segítségével a megtervezett diagramokból SQL szintaktikának megfelelő, az Oracle számára értelmezhető szkriptek készíthetők. Így a fejlesztési lépések egyik fázisát automatizálni tudtuk, ezzel jelentős mennyiségű időt takarítottunk meg.

A generátor kifejlesztésekor lépett a projekt egy olyan útra, amiről nem lehetett, és nem is akartunk hátrálni. Elkezdtünk technológiákat kifejleszteni. Alapként az IQSoft Rt. „Objektum Orientált Üzleti Megoldások" technológiáját használtuk, de rövidesen jelentősen túlléptünk azon.

Megpróbáltunk, és azt hisszük talán többé-kevésbé sikerült is megvalósítani azt a magunk elé tűzött célt, hogy olyan saját fejlesztésű rendszert készítsünk, amely megállja helyét bármely összehasonlításban a nagynevű, drága kereskedelmi rendszerekkel. Önmagunk termékét pozicionálni, saját munkánkról véleményt mondani mindig kényes feladat, leginkább majd az idő és a felhasználók döntik el, hogy mennyit ér a rendszer.

Előadásunk rövid keretein belül azokat az érdekes, talán máshol is alkalmazható technológiákat fogjuk felvillantani, melyek segítségével elértük, hogy a rendszer :

- teljesen zárt adatbázist tartalmaz
- tökéletesen konzisztens az adathalmaz
- tranzakció szintű jogosultságkezelést biztosít
- tetszőleges számú idősíkot képes egyszerre kezelni
- támogatja a teljesen dinamikus kliens oldali / applikációs szerveres kötegelt feldolgozást
- átláthatóvá, mérhetővé teszi az Üzletág és Finomító tevékenységét
- intelligens ügynökök segítségével figyelmeztet, tanácsot ad
- támogatja a dinamikus felhasználói felület váltásokat (többnyelvűségi problémák)
- eleget tesz a Minőségbiztosítási követelményeknek
- eleget tesz a Környezetirányítási követelményeknek

Ebben az írott anyagban ezek közül csak egyet ismertetünk, a virtuális idősíkokra kidolgozott módszert. Azért érdekes ez a terület, mert igazából nincs rá általánosan elfogadott módszertan. A szakirodalom ismertet néhány lehetőséget, azonban egyik se alkalmazható mindenhol.

Egyik igény volt a rendszerbe rögzítendő adatok naplózásának, változtatásának nyomon követhetősége. A Komáromi Finomítóban jelenleg is működő dokumentációs rendszer képes arra, hogy egy tetszőleges időpontra visszaállva („virtuális idő") az adatokat akkori állapotukban lehet megtekinteni. Tehát az adott napon aktuális receptúra kapható vissza az akkor fennálló tartalommal. Ennek kibővítéseként szükséges volt úgy kialakítani az adatstruktúrát, hogy nem csak a kapcsolatok, hanem a hivatkozási tartalmak is visszaálljanak az adott virtuális időre.

Egy példán keresztül megvilágítva a következő feladat megoldására volt szükség: ha a virtuális idő egy hónappal ezelőtti dátumra mutat, akkor az abban az időpontban aktuális receptúrát kell visszaadni. Egy receptúra azonban több alapanyagból áll össze, amiket szintén az adott időpillanatnak megfelelően kell mutatnia a rendszernek. Mivel az összetétel változhat, szükség van arra, hogy a virtuális időpontban fennálló kapcsolatok visszakereshetők legyenek. Azonban adatmódosításra is lehetőséget kell adni a rendszernek, s ezt oly módon kell végrehajtani, hogy a meglévő kapcsolatok ne szűnjenek meg. Tehát, ha egy alapanyag neve változik, akkor azt nem új néven kell felvenni, hanem a meglévőt nevet kell módosítani, eltárolva a régi nevet, illetve a módosítás dátumát. E miatt, ha visszakeressük egy receptúra virtuális időben aktuális verzióját, akkor még azt is biztosítani kell, hogy a tartalmazott alapanyagok kapcsolt információinak tartalma is megfeleljen a vitruális időbeli tartalomnak. Vagyis a virtuális időben lévő nevet kell visszaadni és nem az azóta módosítottat.

A probléma megoldására a tárolandó információkat két csoportra bontottuk, törzs jellegű adatokra és általános felhasználásúakra. Törzs jellegű lett minden olyan típusú adat, amely főleg hivatkozásként szerepel más adatok között és változása fennálló kapcsolatainak módosításával járna. Ilyen típusúak lettek például: anyagok, termékek, szervezeti egységek, géptípusok, folyamatok fejadatai, stb.. A többi információ inkább zárt egységet képez, esetlegesen hivatkozásokkal törsz jellegű adatokra. Az általános jellegű információkról elmondható, hogy adott verziója egy meghatározott időintervallumon belül érvényes, módosításuk esetén rendre újabb

verziói jönnek létre. Ide sorolhatók a receptúrák, minőségi előírások, vagy az egyes folyamatok (beérkezés, gyártás, kiszerelés, MEO vizsgálat, stb.) részletes adatai.

Mivel ezen csoportokba tartozó elemek működése megegyezik, ezért célszerű közös tulajdonságaikat és működési jellemzőiket kiemelni. Erre az objektum-orientált technikáknak lehetőséget is biztosítanak a generalizáció alkalmazásával, melynél egy ősből származtatva minden „gyerek" megkapja a „szülőjére" jellemző elemeket.

A törzsek időbeli tartalmának visszakereshetőségére kialakítottunk egy TÖRZS osztályt, mely hordozza az időben fennálló kapcsolatokat. A tartalom változásának rögzítésére létrehoztunk egy NAPLÓTÖRZS osztályt kapcsolódva a TÖRZS osztályhoz. Minden egyes törzs jellegű tábla tehát a TÖRZS osztályból származik és létrejön egy párja is NAPLÓ+OSZTÁLYNÉV szintaktikával, ami a NAPLÓTÖRZS osztályból származik. A korábbi kapcsolatok rögzítését szintén a naplózó osztályok végzik, hivatkozva az eredeti törzs jellegű osztályok azonosítóira. Az általános jellegű információkat hordozó osztályok tulajdonságait egy ÉRVÉNYESSÉG nevű ősbe emeltük ki, amely jellemzője, hogy tartalmának érvényessége egy intervallummal határozható meg

A generátort felkészítettük a törzsek és az általános jellegű osztályok generálási feladataira is. Tervezés során a törzs jellegű osztályokat a „Központi törzsek" alrendszerben helyezzük el. A generátor minden itt talált osztályra hivatkozást készít a TÖRZS osztályban és ellenőrzi, hogy az öröklési diagrammokra felkerült-e az ezt ábrázoló kapcsolat. Továbbá automatikusan létrehoz egy naplóosztályt is kapcsolódva a NAPLÓTÖRZS osztályhoz. Az ÉRVÉNYESSÉG-ből származó osztályok esetében ellenőrzi a kapcsolatokat és a neveket, amelyek rendben levősége esetén legenerálja a kapcsolatokhoz szükséges hivatkozásokat.

Reméljük sikerült nagyvonalakban érzékeltetni a problémát és a megoldást. Az előadáson az összes többi felsorolt technológiát is ismertetjük és lehetőség szerint bemutatjuk.

Vasúti menetrendek újraütemezése szimulációs úton
II. Folyamatorientált szimulációs ütemezés

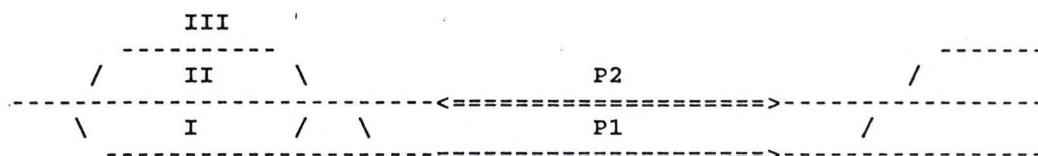
(Gáspár András, Harang BT, Budapest)

1. A feladat MÁV beruházások-felújítások szoftvertámogatása volt a MÁVTI Kft-vel megkötött korábbi szerződés alapján [3].

2. Az 1989 óta felhalmozódott tapasztalatok [2] alapján a vasúti probléma kifejtésére csakis egy olyan módszertani anyag [4] lehet alkalmas, amely egy túlegyszerűsített modellt fejt ki, de célja felvázolni a vasúti szimulációs ütemezés alapproblémáit, hivatkozási alapot adva a tényleges vasúti problémával történő módszeres összehasonlítás számára.

3. Rátérve a túlegyszerűsített szimulációs modell definiciójára, A vizsgált vasúti részhálózat egy vasútvonal lesz. A vonalnak állomásai vannak. Rajzokon a kezdőpontot a baloldalon, a végpontot pedig a jobboldalon ábrázoljuk. Az állomásokon csakis vonatközlekedésre alkalmas vágányok vannak. A két szomszédos állomás közötti úgynevezett állomásköz egy vagy több pályát tartalmaz. A pályák az állomásokon kívüli vágányok. Háromfajta üzemmódban használhatók:
   - egyirányú forgalom a végpont vagy a kezdőpont felé,
   - kétirányú forgalom.

   Az allokálandó erőforrások a vágányok és a pályák. Egy-egy állomás vágányai (I,II,III) továbbá egy-egy állomásköz ugyanazon irányba használható pályái (P1,P2) egyenrangúak:



Az input-menetrend egy-egy vonatról a következőket tartalmazza:
- az azonosításra alkalmas vonatszámot,
- a prioritási-osztályt,
- a vonatrelációt: első állomás - utolsó állomás,
- az érkezési és az indulási időpontokat egy-egy állomáson,
- a minimális tartózkodási időt egy-egy állomáson,
- a vonat minimális menetidejét két szomszéd állomás között.

A minimum idők a vonat késése esetén is betartandó, követelmény jellegű adatok. A minimum idők nem az érkezési és indulási adatok különbségei!

VONATSZÁM= 7932			PRIORITÁS=07	
állomás	érkezés [ó:p]	min. tartózkodás [p]	indulás [ó:p]	min. menetidő [p]
7	-	10	07:45	5
8	07:52	2	07:58	3
9	08:01	0	08:01	2
10	08:04	2	08:06	3
11	08:09	2	08:11	6
12	08:17	10	-	-

Bármilyen erőforrást egyszerre csak egyetlen vonat használhat. A vonatobjektumok az első állomáson generálódnak. Ezután a vonat számára vágányt allokálnak és betolják az indító vágányra, ahol a vonat kivárja az indulási időt. Az utolsó állomáson, az előírt tartózkodási idő letelte után, a vonatot kitolják a fogadó vágányról és az általa lefoglalt vágányt felszabadítják, majd a vonat terminálódik. Egyébként vágányt a vonatérkezés előtt kell allokálni és a vágány a vonatindulás után szabadul fel. A pályát az indulás előtt kell allokálni és az érkezés után szabadul fel. Így az érkezés és az indulás körül, egy bizonyos időre, egyidejűleg két erőforrás is használatban van. A modell egyszerűsítése érdekében ez az időtartam nulla hosszúságú.

A kiszolgálás rendjét a vonatprioritás határozza meg, egy-egy prioritási-osztályon belül pedig a kiszolgálási modell FIFO.

A feladat szimulálni a vonatközlekedést és regisztrálni a megvalósult, ún. "output-menetrendet". Utána külön feladat lesz listázni + analizálni + rajzokon szemléltetni + animációval szemléltetni azt.

4. Mivel a különböző vonatok eseményei egyazon pillanatban követik egymást, az időfogalom alkalmatlan, helyette más fogalom használatára kényszerülünk. Ezek a sorszámozott döntések lesznek. Így az időintervallumokat nemcsak időpontok, hanem döntésekre hivatkozó mutatók is határolhatják.

A döntéseket egyértelműen azonosítja első attributumuk a sorszám. Második attributumuk természetesen a döntési időpont. A döntések egyik részosztálya az ALLOC, amely az erőforrások allokálásának és felszabadításának leírására szolgál, másik részosztálya pedig a NOTALLOC, amely egy vonat feltartását úgy írja le, hogy a vonat számára tilos egy erőforrás allokálása, bizonyos feltételek esetén.

A döntéseket parancsként fogjuk fel újraszimulációnál illetve a megtervezett menetrend végrehajtásakor. E szemléletben egy allokálási/felszabadítási parancs szerkezete a következő lesz:

> A V vonat S állapotában
> allokálandó az R erőforrás
> az I időintervallumra !

Parancsként felfogva egy NOTALLOC-döntést, annak szerkezete a következő lesz:

> A V vonat S állapotában
> tilos allokálni az R erőforrást
> az I időintervallumban !

A további leírásban végtelen méretű központi tárat képzelünk el és nem foglalkozunk például azzal a kérdéssel, hogy a szimuláció alatt meghozott döntések adatbázisa melyik tárolóban helyezkedik el. Ha azonban a szimuláció megszakad, akkor feltételezzük, hogy az adatbázisok előzetesen kimentődtek a háttértárolóra.

A döntések további attributuma a döntés fajtája. Szemantikáját később fejtjük ki, azonban már most megadjuk a lehetséges döntésfajtákat és a sorszámok fajtától függő értékeit:

- normál döntések , [1, 999999]
- javaslatok , [1000001,1999999]
- végleges döntések, [2000001,2999999]

5. Most sorszámmal látjuk el a vonat forgalmi-szakaszait: Egytől kezdve, a vonat útvonala mentén, az első állomástól az utolsóig. Páratlan számot kapnak az állomási szakaszok, párost az állomásköziek. A nulla sorszámot kapja az a forgalmi-szakasz, amelybe a vonat a generálásakor kerül és amelyben addig marad, amíg be nem tolták a kiinduló állomásra. A legnagyobb szám ahhoz a forgalmi-szakaszhoz tartozik, amelyben a vonat terminálásáig lesz azután, hogy megérkezett a végállomására.

Egy-egy vonat közlekedése ezekután már egy alkalmas nyelvtannal írható le. Ugyanis a vonatközlekedés nem más mint az egymástkövető forgalmi-szakaszok sorozata. A terminális szimbólumokat csupa kisbetű jelöli. Ezek a vonatok elemi állapotait reprezentálják. A nem-terminális szimbólumok jele a csupa nagybetű.

```
< VONATKÖZLEKEDÉS > ::=
          < BETOLÁSI SZAKASZ >,
          list { < ÁLLOMÁSI SZAKASZ > < ÁLLOMÁSKÖZI SZAKASZ > }
          ----
          < KITOLÁSI SZAKASZ >

< BETOLÁSI SZAKASZ > ::=
        < helyből induló vonat betolás előtt >
        opt list { < betolás előtti feltartás > }
        --- ----
        < betolás előtti allokálás >
        < betolás >

< ÁLLOMÁSI SZAKASZ > ::=
        < minimális tartózkodási idejét töltő vonat, ind. előtt >
        opt list { < indulás előtti feltartás > }
        --- ----
        < indulás előtti allokálás >
        < kijáró vonat >
        < indulás utáni felszabadítás >

< ÁLLOMÁSKÖZI SZAKASZ > ::=
        < állomásközi távolságot leküzdő vonat >
        opt list { < bejárat előtti feltartás > }
        --- ----
        < bejárat előtti allokálás >
        < bejáró vonat >
        < bejárat utáni felszabadítás >

< KITOLÁSI SZAKASZ > ::=
        < minimális tartózkodási idejét töltő vonat, kit. előtt >
        < kitolás >
        < kitolás utáni felszabadítás >
        < feloszló vonat kitolás után >
```

A feltartások opcionális listái a feltartásokról szóló NOTALLOC döntéseket időrendben tartalmazzák.

A fenti nyelvtan alapján a vonat állapotát a következők jellemzik:

- a vonatszám,
- a forgalmi-szakaszra (BETOLÁSI, ÁLLOMÁSI, stb.) utaló kódszám,
- a feltartás sorszáma az opcionális listában (ha a vonat fel van tartva), vagy
- a terminális szimbólum kódszáma (ha a vonat nincs feltartva).

A fordítóprogramok elméletében jártas olvasónak a nyelvtan láttán azonnal ütemezési elképzelései támadnak. A "backtrack"-algoritmus azonban több okból se megy. Először is vegyük észre, hogy a nyelvtan csak egyetlen vonat közlekedését írja le, a vonalnak pedig nagyszámú, egymással versengő vonata van. A "backtrack"-algoritmust tehát minden vonatra külön-külön kell megvalósítani.

Nem megy azonban az a "backtrack"-algoritmus se, amelyben egy-egy kiválasztott vonatot ütemezünk végig. Ugyanis úgy ellentmondásba kerülnénk a kiszolgálás rendjével, hiszen nem létezik garancia arra, hogy a vonat egyre távolodó allokálási helyein az azonos prioritású vonatok közül a korábban igényt bejelentő vonat kapja meg a kívánt erőforrást. Az ütemezést tehát minden lépés után meg kell szakítani azért, hogy egy monitor megállapíthassa az ütemezés éppen soronkövetkező vonatát.

6. A vasúti szabályzatok gyakorlatilag kizárják a patt-helyzetek keletkezését. Szimulációs ütemezés esetén azonban a patt-elkerülő algoritmusnak kell feltartania az elemi patt-helyzetbe utolsóként belépő vonatot. Vagyis, megfelelő időben NOTALLOC-döntést kell hoznia.

7. Legyen R egy olyan erőforrás, amit két vonat is igényelt, időben átfedéssel:
   - a VA vonat a [TA1,TA2] időintervallumra,
   - a VB vonat pedig a [TB1,TB2] időintervallumra.

   Tegyük fel továbbá, hogy a kisebb prioritású VA vonat igénye korábbi, mint a nagyobb prioritású VB vonaté. Időtengelyeken szemléltetve:

   Ekkor a szimulációs ütemezés során, egy előrelátás nélkül alkalmazott prioritásos kiszolgáláskor, úgynevezett prioritási-konfliktus fog fellépni a VA és a VB vonatok között, mivel az R erőforrás a TB1 időpontban még foglalt lesz a kisebb prioritású VA vonat által. A prioritási-konfliktus a szimulációs ütemezés alapvető problémája. Természetesen elkerülhető, ha a jövő megfelelő algoritmussal előre látható. Ilyen algoritmus azonban általában nem létezik. A vasúti szimulációs ütemezésben azonban a közeljövő egy része kiszámítható, vagy igen jól becsülhető.

8. A kidolgozott algoritmusban a vonatok a SIMULA nyelv PROCESS részosztályába tartoznak és az idő folyását a SIMULATION osztály definiálja. Mi a teendő azonban prioritási-konfliktus észlelésekor? A döntések adatbázisára alapozva automatikusan fel kell tárni a fellépett prioritási-konfliktust:
   1. lépés: Meg kell határozni azt az allokációs döntést, amely közvetlen oka volt a fellépett prioritási-konfliktusnak.

2. lépés: Vissza kell lépni az időben ehhez az allokációs döntéshez azért, hogy azt NOTALLOC döntéssé módosítsuk.
3. lépés: Újra kell szimulálni a forgalmat a módosított döntéssel.

9. Hogyan léphetünk vissza az időben?

A SIMULATION osztály az idő folyásának egy irreverzibilis modelljét definiálja. Ha valaki csak a SIMULATION osztályt használja, akkor gépidőt pazarolva, a nulla időponttól kiindulva, újra kell szimulálnia a módosítandó döntésig bekövetkezett eseményeket.

A SIMULA nyelv azonban lehetőséget biztosít az időfolyás non-standard modelljének definiálására is. Például egy új szimulációs modell egy olyan multi-"backtrack"-algoritmust is megvalósíthat, amelynek szereplői az ütemezést vezérlő monitor és az önálló vonat-objektumok. Az ütemező monitor kiválasztja az ütemezésben soronkövetkező vonatot, és vezérlést ad a kiszemelt vonatnak az ütemezési lépés megtétele céljából, a SIMULA nyelv "CALL" korutin-hívásával. Egy-egy vonatobjektumon belül a hívási lánc tükrözi a nyelvtan szerint már levezetett vonatállapotot és minden vonat önállóan alkalmas egyfajta "backtrack"-algoritmus végrehajtására: azaz a monitor parancsát végrehajtva a hívási lánc szükségszerinti lebontására is képes.

A továbbiakban feltesszük, hogy a szimulációs ütemezés számára létezik egy elég hatékony szoftver módszer az időben visszalépésre, a kiválasztott döntés módosítására.

10. Az ütemezési probléma egyszerűsítése céljából a vonatok prioritásának megfelelően egy dekompoziciót hajtunk végre: Ha a vonatoknak N db prioritási szintjük van, akkor az ütemezést N fázisra bontjuk fel. Az ütemezés a legnagyobb prioritású, azaz a legfontosabb vonatokkal kezdődik, lényegében az alábbi ciklusutasítás szerint:

```
FOR I:=N STEP -1 UNTIL 1 DO
```

Az ütemezésből mindig kimaradnak azok a vonatok, amelyek prioritása kisebb a ciklusváltozó értékénél.

Azok a vonatok, amelyek prioritása megegyezik a ciklusváltozó értékével, megkisérlik végrehajtani az input-menetrendet. Ezeket a vonatokat a továbbiakban Ivonatnak nevezzük. Amikor már sikerrel fejeztünk be egy-egy ütemezési fázist, az Ivonatok forgalmi és erőforrásfoglalási adatait az output-menetrend részeként elmentjük.

Azok a vonatok, amelyek prioritása nagyobb, mint a ciklusváltozó értéke, már nem az input-menetrendet, hanem az egyik megelőző ütemezési fázisban már véglegesen elfogadott output-menetrendet hajtják végre. E vonatokat a továbbiakban Ovonatnak nevezzük. Az Ovonatok jövőbeli erőforrásigénye pontosan előrelátható, mivel ütemezésük egy korábbi fázisban már lezajlott és erőforrásfoglalásaikról pontos menetrend készült.

A vázolt prioritásos-dekompozicióval igen nagyszámú prioritási-konfliktust el lehet kerülni.

A normál döntések mindig az Ivonatokra, a végleges döntések pedig az Ovonatokra vonatkoznak. Az ütemezési fázis befejezése után az összes normál döntést végleges döntéssé kell konvertálni.

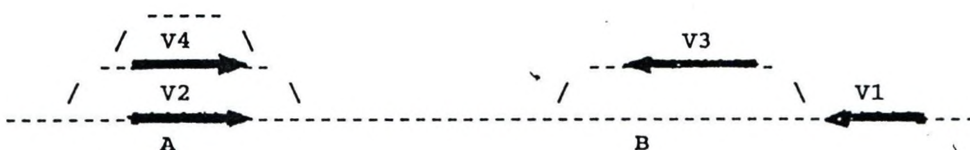
11. Most szabályokat kellene megadnunk a döntések kezelésére, a konzisztencia biztosítására. A szabályok sorszámozottak és sorszámuk szerinti növekvő sorrendben alkalmazandók. Ilyen szabályok például a következők:

R1: A D1 döntés törlése esetén törölni kell az összes olyan normál döntést, amely a D1 döntést követi és a D1 döntéstől függ.

R2: A D1 döntés törlése esetén javaslattá kell konvertálni az összes olyan a D1 döntést követő normál döntést, amely független a D1 döntéstől.

R3: Az újraszimulálás során egy javaslatot el kell fogadni, ha az megvalósítható. Az elfogadott javaslat ekkor ismét normál döntéssé alakul és a javaslat törlésre kerül.

12. Most jellegzetes példákon keresztül kellene demonstrálni a kidolgozott szabályok hatékonyságát. Az alábbi ábrán a V1 és a V2 Ivonat, a V3 és a V4 pedig Ovonat.



Tegyük fel a következő eseménysorozatot:
- A V2 Ivonat a D2 döntéssel allokálja az A és B állomás közötti pályát és elhagyja az A állomást.
- A V1 Ivonat a D1 döntéssel allokálja a B állomás utolsó szabad vágányát és megérkezik a B állomásra.
- A V2 Ivonatot állomási vágány hiányában a D22 NOTALLOC-döntéssel feltartják a B állomás bejárati jelzője előtt. A D22 döntés függ a D2 döntéstől és indoklásán keresztül függ a D1 döntéstől.
- A V1 Ivonatot állomásközi pálya hiányában a D11 NOTALLOC-döntéssel feltartják a B állomás kijárati jelzője előtt. A D11 döntés indoklásán keresztül függ a D22 döntéstől.
- a V4 Ovonat a D4 döntési pillanatban prioritási-konfliktust észlel, mert a számára szükséges pályát a V2 Ivonat foglalja.

A fenti példában a prioritási-konfliktus feloldásaként a D2 döntést kell módosítani. Így az ütemezés a D2 döntéshez lép vissza és azt az F2 NOTALLOC-döntéssel (feltartással) helyettesíti:

A V2 Ivonatnak
az A állomás egyik vágányát foglaló állapotában,
tilos allokálni az AB állomásköz egyetlen pályáját,
a [F2,D4] időintervallumban !

A helyettesítés lépései:
1. A D2 döntés törlésre kerül.
2. Az R1 szabály végrehajtása során törlésre kerül a már törölt D2 döntéstől függő D22 NOTALLOC-döntés is.
3. Az R1 szabály ismételt végrehajtása során törlésre kerül a már törölt D22 döntéstől függő D11 NOTALLOC-döntés is.

4. Az R2 szabály végrehajtása során a D1 döntés a D1* javaslattá konvertálódik.
5. Feljegyzésre kerül az F2 NOTALLOC-döntés.

Az újraszimulálás során ismét levezetődik a D1 döntés és nem lesz már akadálya annak se, hogy a V1 Ivonat továbbmenjen az A állomás felé.

13. Most a prioritási-konfliktus feloldását más szemléletben írjuk le. A prioritási-konfliktus feloldásakor visszarendeljük egy vonatkonvoj legelső vonatát. Csakhogy ezzel gyakran lehetővé válik az is, hogy a vonatkonvoj többi vonata - más permutációban - sikertelen kisérletet tegyen a hiányzó erőforrás allokálására. Ennek megelőzésére a már rendelkezésre álló adatok és egyszerű automatikus becslések útján az algoritmus egy sor további feltartási javaslatot (NOTALLOC-döntést) dolgoz ki - de megismételt szimuláció nélkül - pusztán gondolatkisérletek segítségével.

A vonatkonvoj feltartása azonban szabaddá teszi az utat az ellenirányú vonatkonvoj előtt is. Kapacitáshiány esetén ezek közlekedése is feltehetően sikertelen lenne. Az algoritmusnak tehát a gondolatkisérletek során az ellenirányú konvojra is gondolnia kell.

14. Mindeddig a vonatok egy-egy prioritási osztályán belül a FIFO kiszolgálási modellt alkalmaztuk. A vasút előírásai szerint azonban az azonos prioritású Ivonatok közül a nagyobb késésű Ivonatot kell előbb kiszolgálni. E követelmény teljesítését könnyű visszavezetni a prioritásos-kiszolgálásra egy olyan (dinamikus) Dprioritás-fogalommal, amely egyesíti magában a vonatnak az input-menetrendtől örökölt prioritását a vonat pillanatnyi késésével.

Fel kell készülni azonban e Dprioritás-definició egy súlyos hibájára is: az idő múlása megváltoztathatja a korábban már elfogadott kiszolgálási sorrendet végtelen ciklust okozva. Könnyű ugyanis olyan példát konstruálni, amely egymást feltartó vonatok versengéséről szól. Egy ilyen példában a mozgó vonat késése nem nő tovább, mialatt az álló vonat késése tovább nő, ami előbb-utóbb Dprioritási-konfliktust okoz. A vonatok ezt követő szerepcseréje végtelen ciklust eredményezhet. Éppen ezért, a forgalomtervező lehetőséget kap arra is, hogy egyes Ivonatok között határozott kiszolgálási sorrendet írjon elő, az újraszimuláció idejére. Például forgalomtervezői utasítás lehet a következő:
Kiszolgálási sorrend: Először V2, azután V4.

Más példán azt kellene demonstrálni, hogy egy állomást elhagyó vonatkonvoj esetén, az input-menetrend által előírt Ivonatindítási sorrendet az algoritmus képes helyesen felülbírálni az output-menetrend számára.

15. Sajnos a vázolt algoritmus egy túlegyszerűsített modellre épül. Oldalakon keresztül sorolhatók azok a követelmények, amelyek a vasúti célokra valóban használható modellt és algoritmust megkülönböztetik a fent ismertetettől.

Hivatkozások:

[1] A.Gáspár - Gy.Visontay - P.Csáki:
Tasking and Run Time Library Handling in SIMULA 67 - Implemented on CDC 3300.
Computer and Automation Institute,
Hungarian Academy of Sciences, 1977.

[2] A.Gáspár: Embedding SIMULATION to Clipper.
22nd Conference of ASU on "Object Oriented Modelling+Simulation".
Clermont-Ferrand, France, 1996.

[3] Gáspár A.: Vasúti menetrendek újraütemezése szimulációs úton.
I. Országos Objektum-Orientált Konferencia. 206-214. o.
Kecskemét, 1996.

[4] A.Gáspár: Fine-Scheduling of Railway-Time-Tables by Simulation.
23nd Conference of ASU on "Object Oriented Modelling+Simulation".
Stara Lesna, High Tatras, Slovakia, 1997.

Köszönettel tartozom Takács Lászlónak (MÁVTI) a vasúti feladat specifikációjáért és Csáki Péternek (SzTAKI) lektori munkájáért.

Az IBM Magyarországon

Amit az IBM-ről tudni kell...

Az IBM egy világméretű informatikai cég, amely 163 országban több mint egymilliárd ügyféllel rendelkezik. Ezen túlemően, az IBM:

- a világ legnagyobb számítástechnikai vállalalata; összbevétele 1996-ban meghaladta a 75 milliárd dollárt

- A világ legnagyobb szoftverháza; szoftvergyártásból származó bevételei 1996-ban meghaladták a 13 milliárd dollárt

- A világ legnagyobb informatikai szolgáltatója; bevételeit az utóbbi négy év során evente több, mint 20%-kal növelte

- Az erőforrás-kihelyezés (outsourcing) területén világelső

- Európában az IBM több, mint 1.000 nagyszabású rendszerintegrációs projektet tudhat a magáénak.

Mindezeken túlmenően, az IBM rendelkezik a legnagyobb adat-, audio- és video hálózattal a világon, amely az IBM Global Network nevet viseli és a cég hálózati számítástechnikai stratégiájának alapköve a világ minden pontján

Az IBM 1936-ban települt meg Magyarországon, Watson Elektromos Könyvelőgépek Kft. néven. A vállalat azóta túlélt egy világháborút, jónéhány politikai vihart és forradalmat, de e vészterhes évek alatt is megszakítás nélkül szolgálta magyarországi ügyfeleit. Mint a világon mindenhol, Magyarországon is, az IBM az ipar egyes területeire specializálódott szervezeti egységekre osztva, iparspecifikus megoldásokkal és szolgáltatásokkal áll az ügyfelek rendelkezésére. Ez a szervezeti felépítés teszi lehetővé, hogy az IBM világszínvonalú, az igényeknek legmegfelelőbb termékeket, megoldásokat és szolgáltatásokat nyújtson, széles piaci körben.
Az IBM Magyarországi Kft. ma több mint 250 fős, magasan képzett és elkötelezett szakembergárdájával és több mint 100 üzleti partnerével áll kiterjedt ügyfélköre rendelkezésére budapesti központjában, valamint győri és miskolci képviseletén.
A Székesfehérváron létesített IBM Storage Product Kft.-ben a legkorszerűbb technológiának megfelelő merevlemez gyártás folyik 1995-óta. A száz százalékban exportra történő gyártás értéke 1996-ban meghaladta a 400 millió dollárt.

NOKIA Family b/w A/4 #?? 1997/10/09 15:14 Page 1

Egy hajóban evezünk

Mi hárman, a Nokia Mobile Phones, a Nokia Telecommunications és a Nokia Display Products együtt versenyzünk a hazai telekommunikáció és számítástechnika piacán. Termékeinkkel és szolgáltatásainkkal kezdettől fogva arra törekszünk, hogy emberközelivé tegyük a legmodernebb technikát. Hiszen a verseny Önökért folyik.

NOKIA
CONNECTING PEOPLE

A Nokia Szoftvertechnológiai Laboratóriumában dolgozták ki egyebek között -- a laboratóriumot vezető Pertti Lounamaa szorgalmazta, merőben békés célú Stratégiai Szoftverkezdeményezés (Strategic Software Initiative) részeként -- az Octopus szoftverfejlesztési módszert. Az Octopus három helsinki szoftverfőmérnök, Maher Awad, Juha Kuusela és Jurgen Ziegler munkája, s főleg beágyazott valós idejű rendszerekbe való objektumorientált szoftverek létrehozására való. Ennek az 1993-tól tartó fejlesztésnek az eredményét a szerzők az Object-Oriented Technology for Real Tine Systems: A Practical Approach Using OMT and Fusion című kötetben (Prentice Hall, 1996) foglalták össze.

Az objektumorientált szemléletmódot a következőképpen foglalja össze a Nokia erről szóló Web-oldala: ez a szemlélet objektumok együtteseként látja a rendszereket; az objektumok adatokat és funkciókat foglalnak magukba, a valós világban meglevő tárgyakat modellezik, és jól meghatározott felületeken át kapcsolatot tartanak (kommunikálnak) egymással. Az objektumorientált szemlélet -- folytatja az említett Web-oldal -- csökkentheti az összetett rendszerek bonyolultságát, növelheti teljesítményüket, hibatűrésüket és kiterjeszthetőségüket, magasabb szintűvé teheti a szoftver-újrahasznosítást (osztályok, tervezési minták és keretek újrahasznosítását). Az objektumorientált szemléletmód -- olvashatjuk továbbá --világszerte de facto szabvány a szoftverfejlesztés jó néhány területén, például a felhasználói felületek és az adatalkalmazások világában.

Ami közelebbről az Octopust illeti, az az objektumorientált módszerek felhasználása a telekommunikációs és az ipari irányítás körébe vágó alkalmazásokra. Egybefoglalja a szoftverfejlesztés főbb szakaszait, és mintegy hidat ver az objektumorientált módszerek és a valós idejű rendszerek között.

Az Octopus szerint a fejlesztés folyamata a következő szakaszokra oszlik:

-- a rendszer iránti követelmények feltárásának szakasza;

-- a rendszer architektúrájának kialakítása: a nagy rendszer részrendszerekre bontása és a "rendszernövesztés" lehetőségeinek meghatározása;

-- a részrendszereknek (vagy a részrendszerek részeinek) egymással párhuzamos fejlesztése (elemzése, tervezése és implementációja) növesztés soron következő lépésének megfelelően, majd ennek a szakasznak a szükség szerinti megismétlése.

A rendszerkövetelmények felmérésének szakaszában tisztázódik, hogy mik a szoftver iránti lényeges kívánalmak, diagramok készülnek a rendszerkönyezetről, használatieset-diagramokon és adatlapokon meghatározódik a rendszer funkcionális és dinamikus viselkedése. Mindezt a rendszer és környezete közötti kölcsönhatást leíró forgatókönyvekkel is meg lehet tenni. A rendszer ebben a szakaszban még fekete doboz.

A következő szakaszban kialakítandó rendszerarchitektúra részrendszerdiagrammal írja le a rendszer szerkezetét; ezen társulásként jelennek meg a részrendszerek közötti fő érintkezési felületek. A funkcionális modell felelősségi lapokból épül fel, a dinamikus modell pedig használatieset--részrendszer kapcsolati diagramokból. A részrendszerek még fekete dobozok ebben a szakaszban.

A részrendszerek egymással párhuzamos fejlesztésének szakaszában minden részrendszeren elvégzendő a részrendszer-elemzés. Ebben a strukturális modellt a rendszer osztálydiagramja alkotja, az osztályleíró táblával kiegészítve, a funkcionális modell műveleti lapokból áll, a dinamikus modell pedig egy eseménylistából, eseménycsoport-diagramból, eseménylapból áll. A funkcionális modell és a dinamikus modell is fekete dobozként kezeli a részrendszert, a strukturális modell azonban már hivatkozik az osztályokra.

A részrendszerek tervezésekor a dinamikus modell objektumkölcsönhatási szálakból épül fel --

ezek a szálak később minősített események szálaivá alakulnak át --; ezután következik az objektumcsoportok leszármaztatása. Ezekből rendszerezett módon körvonalazódnak a folyamatok, osztályok és folyamatok közötti üzenetváltások: a strukturális modell összetevői. A funkcionális modell a tagfüggvények vázlataiból áll; a tagfüggvények bele vannak ágyazva a strukturális modell osztályvázlataiba.
A Octopus az elterjedt OMT-kre és a Fusion módszerre támaszkodik, de a valós idejű rendszerek tervezésében használatos módszereket is felöleli. Tekintetbe veszi a valós idejű rendszerekre jellemző egyedik sajátosságokat: a párhuzamos működést, a szinkronizálást, a megszakítások kezelését, a hardverinterfészeket és a végponttól végpontig számított válaszidőket.
Az Octopusról szóló említett kötet hoz két tényleges példát is az Octopusszal való fejlesztésre: az egyik egy előfizetőivonal-vizsgáló, a másik a gazdaságos utazási sebesség megállapítására való rendszer.

Oracle a világban és Magyarországon

Korszerû adatbáziskezelõk, fejlesztõeszközök, komplex alkalmazások, Webes verziók megjelentetése, a Network Computer koncepciójának kidolgozása, bevezetésének hathatós támogatása egy-egy fontos állomás az ambíciózus fejlõdést megvalósító, a felhasználók számára egyre inkább stratégiaipartner-szerepet vállaló Oracle Corporation mûködésében. A Kaliforniában alapított Oracle Corporation már több mint egy évtizede meghatározó szereplõje a világ szoftverpiacának. Jellemzõ, hogy az Oracle Corporation a világ adatbáziskezelõ piacából - versenytársait messze megelõzve - mintegy 50 százalékot tudhat a magáénak. Az Oracle mindenekelõtt relációs és más típusú adatbáziskezelõ szoftverei által vált világhírûvé és naggyá. Legutóbb megjelent, a hálózati számítástechnika széles körû elterjedését lehetõvé tevõ adatbáziskezelõje, az Oracle8 már a huszonegyedik századot idézi. Az Oracle kínálatában szerepelnek sikeres fejlesztõeszközök is. Az utóbbi idõben pedig egyre nagyobb mértékben terjednek a világban az Oracle által fejlesztett számítógépes alkalmazási rendszerek, mint például a nagy, vállalati pénzügyi-gazdálkodási vagy gyártási folyamatokat vezérlõ, illetve irodaautomatizálási szoftvertermékek. Az Oracle Corporation a világ közel száz országában rendelkezik leányvállalattal. Éves árbevétele több mint 5 milliárd dollár, ami dinamikusan tovább nõ. A cég növekedési rátája eléri az évi 30-35 százalékot. A világ legjelentõsebb szoftvercégeinek élvonalába tartozó Oracle Corporation 1993-tól leányvállalata, az Oracle Hungary révén van jelen a magyar piacon. Termékei, tevékenysége segítségével kiemelkedõ számítástechnikai eredmények születtek például az állami, kormányzati szervezeteknél, a telekommunikáció területén, a bankok-biztosítók világában. Kiterjesztette mûködését többek között a gyártó vállalatok, a közmûvek, az egészségügy számítógépalkalmazói felé is. Az Oracle Hungary folyamatosan növeli bevételeit, nagyon sikeresen fejleszti üzleti mûködését Magyarországon.

A Sun a vállalatok szolgálatában

A Sun 1982-es alapítása óta folyamatosan vált a számítástechnikai szakma meghatározó vállalatává. Mindez többek között annak is köszönhetõ, hogy mûködése során mindvégig követte a kezdetek kezdetén megfogalmazott cégfilozófiát: *"A hálózat maga a számítógép"*.

A vállalat által kínált termékek és technológiák a nyílt, rugalmas hálózati megoldásokat igénylõ vállalatok igényeit maximális mértékben képesek kielégíteni. A Sun rendszerei egyaránt otthon vannak a kereskedelmi és mûszaki típusú környezetekben, és persze az Internetes alkalmazások területén is.

A világszerte mintegy 17 000 alkalmazottat foglalkoztató cég a termékek és a szolgáltatások széles skálájával - kiegészítve mindezt a stratégiai fontosságú partnerek megoldásaival - biztosan képes megfelelni a világszerte mind kiterjedtebb ügyfélköre elvárásainak. A Sun technológiáját világszerte minden meghatározó iparág alkalmazza: az ipari termelés, a szállítás/szállítmányozás, a pénzügyi szolgáltatások (bankok, biztosítótársaságok) csakúgy, mint a távközlés, az egészségügy és a kormányzati szervek. A Sun termékskálája az elmúlt évek során jelentõsen bõvült: a munkaállomásoktól kezdve a munkacsoport-, osztályszintû- és mainframe-teljesítményû szervereken, továbbá adatközpont megoldásokon át megtalálhatók többek között a tároló rendszerek, rendszer-szoftverek és hálózatkezelési megoldások is.

A különbözõ termékek kialakítása során a Sun soha nem feledkezett meg a hálózat nyújtotta elõnyök szem elõtt tartásáról: azaz technológiái, termékei és szolgáltatásai tükrözik azt a szemléletet, mely szerint bármely szervezet sikere attól függ, hogy milyen jól képes a kulcsfontosságú információkat elosztani, tárolni és elõhívni, függetlenül attól, hogy azok a hálózat mely pontján találhatók.

A Sun 1992 õszétõl képviselteti magát Magyarországon. A budapesti irodából nemcsak a magyarországi tevékenységet vezetik, hanem ez a cég közép-európai központja is. Az 1992-96 közötti idõszakban a vállalat igen sikeres volt Magyarországon, a hazai munkaállomás és munkaállomás-szerver piac vezetõjeként tartják számon.

A vállalat értékesítési csatornapolitikájában hazánkban is a közvetett - tehát a partnereken keresztül megvalósuló - értékesítési modell a meghatározó.

A Microsoft

Az 1975-ben alakult Microsoft Corporation 11,5 milliárd dolláros forgalmával a világ legnagyobb forgalmú független szoftvercége.

A Microsoft elnök-vezérigazgatójának, Bill Gatesnek a víziója egy olyan világ, amelyben az információk mindenkinek a keze ügyében vannak. ("Information at your fingertips") Ehhez mind az asztali számítógépeken, mind pedig a szervereken megfelelő szoftverek szükségesek. A hagyományosan asztali (ügyféloldali) szoftvereket gyártó Microsoft 1993-ban a Windows NT termékvonal megteremtésével megvetette a lábát a vállalati számítástechnikában is. A Windows NT Server nemcsak fájlszerverként, hanem alkalmazásszerverként is kitűnően használható: a Windows NT Serveren futó szerveralkalmazásokat tartalmazó Microsoft BackOffice szoftvercsomagban megtalálhatók a legfontosabb funkciók, amelyekre egy vállalatnak az informatikai infrastruktúrája kialakításakor szüksége van, a relációs adatáziskezelőtől a rendszerfelügyelen keresztül a fürtözés (clustering) támogatásáig.

A Windows NT és a BackOffice forgalma 1996-ban elérte az 1 milliárd dollárt, ami önmagában is a legnagyobb szoftvercégek közé emelné a Microsoftot. A BackOffice alkalmazások közül a legfigyelemreméltóbb talán a Microsoft SQL Server, ugyanis a Transaction Processing Council TPC-C sebességtesztjei közül a tíz legjobb ár/teljesítmény viszonyút kivétel nélkül a Microsoft SQL Server nyújtotta, különböző hardvereken.

A Microsoft az Internet előretörésével megszilárdította vezető pozícióját a PC-s szoftverek piacán. A COM (Component Object Model) szoftverkomponens-technológiát internetes szempontból áramvonalasítva egy olyan objektummodellt tudott létrehozni, amely egyaránt szolgálja az Internet- és az intranetfelhasználókat, valamint a "hagyományos" (nem internetes) alkalmazások fejlesztőit és felhasználóit. Ennek a technológiának az elosztott rendszerekre való kiterjesztése a DCOM (Distributed COM). Mára elmondhatjuk, hogy a COM köré egy igazi szoftverkomponens-piac szerveződött, és ebben egyedülálló a versenytárs objektommodellekkel szemben. A DCOM infrastruktúra a Windows NT 4.0 1996-as megjelenése óta piaci termék, több nem-Microsoft platformra pedig 1997-ben jelent, illetve jelenik meg.

Nemcsak a számítástechnika, de az üzleti élet is elismeri a Microsoftot: a Microsoft stabil jövőjébe vetett hitet tükrözi az, hogy kb 160 milliárd dolláros összértékével a Microsoft az iparág legmagasabbra értékelt vállalata.

A Microsoft honlapja a **Hiba! A könyvjelző nem létezik.** címen érhető el.

EGYMÁSRA ÉPÍTVE

IQSOFT

[illegible]42 Budapest, Teleki Blanka u. 15-17
Tel. [illegible]-5949, 363-2200
Fax 220-5596
E-mail: iqsoft@iqsoft.hu
Internet cím:http: www.iqsoft.hu

HATÉKONY MEGOLDÁSOK TELJESKÖRŰ VÁLLALATI INFORMÁCIÓS RENDSZEREK KIÉPÍTÉSÉHEZ. HONOSÍTOTT ÉS SAJÁT FEJLESZTÉSŰ ALKALMAZÁSOK. KONZULTÁCIÓ, FEJLESZTÉS, BEVEZETÉS, OKTATÁS ÉS MINDEN SZAKASZRA KITERJEDŐ TÁMOGATÁS.

17038

Conference Tours Kft
Rendezvény-,
Kereskedelemszervezési
és Utazási Iroda

1055 Budapest, Kossuth tér 6-8.
✉ 1372 Budapest, P.f. 451