CORBA-alapú elosztott alkalmazások

/Java és CORBA/

CORBA-alapú elosztott alkalmazások

/Java és CORBA/

Csizmazia Balázs
☺

2004

Eddig megjelent kiadások:
Első kiadás, 1998 január
Második, bővített kiadás, 1998 augusztus
Harmadik, elektronikus kiadás, 2004 január

Az Olvasót ezúton bíztatom arra, hogy menjenek el önkéntes véradásra és adjanak vért! Évente 1-3 alkalommal, alkalmanként egy-egy órát rá lehet szánni. Én is ezt teszem. — Jóval kevesebb a kizáró ok, mint azt az ember elsőre – főleg önmagáról – gondolná. **cs.b.**

Harmadik kiadás

ISBN ily en ninc s ☺

Kiadó: Csizmazia Balázs
Nyomtatta és kötötte maga az Olvasó, ha ezt megteszi.

Tartalomjegyzék

Ezt a könyvet szüleimnek és Anatukámnak dedikálom ☺

Szüleimnek és Anatukámnak ... ☺

Előszó

Az Olvasó valószínűleg már hallott az Internetről és a rajta működő hálózati infrastruktúra néhány eleméről, mint például az elektronikus levelezésről, vagy a WWW-ről.

Ebben a könyvben az Internet infrastruktúrájának alapvető komponenseit fogjuk megismerni, elsősorban a hálózati és elosztott alkalmazásokat készítő programozó szemszögéből.
A téma a heterogén számítógépekből álló hálózatok feletti elosztott objektumok kezelésének lehetőségét ismerteti a CORBA elosztott objektummodellje alapján.

A könyvünkben megcélzott olvasókör

A könyvet elsősorban azoknak ajánljuk, akik meg akarják ismerni a CORBA és a Java együttes alkalmazását.

Könyvünk a Java 1.2-es változatának Java IDL - CORBA kapcsolatát és ezen változat segédprogramjainak használatát feltételezi. Amennyiben az Olvasó más Java-változatot használ vagy nem a Sun Java Fejlesztői Készletét (JDK) használja, úgy néhány segédprogram neve más lehet. Legvalószínűbb az, hogy az IDL-ről Java-ra fordító program neve más már (az 1.4-es JDK-val szállított ilyen program neve `idlj`). Egyébként a példaprogramok és a CORBA architektúrájának bemutatása továbbra is használhatóak.

Csizmazia Balázs

1. Fejezet

A Java IDL és alkalmazásai

A távoli metódushívás kapcsán már megismerhettünk egy objektum-orientált elosztott infrastruktúrát megteremtő eszközt, amellyel a különböző alkalmazások objektumainak lehetőségük van egymás metódusainak meghívására akkor is, ha azok nem ugyanazon a számítógépen vannak. A távoli metódushívásban részt vevő kliens alkalmazásoknak szerezniük kell az elérni kívánt objektumra vonatkozóan egy referenciát, és ismerniük kell a távoli objektum elérhető metódusait, az objektum interfészét. A távoli metódushívás "csak" Java nyelven írt objektumok kapcsolatát képes megteremteni, mivel az implementációjában kihasznál néhány Java-specifikus eszközt (legjellemzőbb példaként az objektumok szerializálásának lehetőségét említhetjük, másodsorban pedig a Java interfész mechanizmusát is ötletesen felhasználja).

A nagy számítógépes hálózati rendszerek egy fontos jellemzője ezzel szemben a heterogenitás: egy hálózaton gyakran több különféle architektúrájú számítógépet találhatunk, különféle operációs rendszerekkel felszerelve, és még ennél is nagyobb változatosság jellemzi a rendszeren futó alkalmazásokat (gondoljuk csak meg, hogy a nagyvállalatok céljaik elérését segítő különféle jól bejáratott technológiákkal rendelkeznek, amiket nem akarnak meggondolatlanul egy új, még kiforratlan technológiával lecserélni, ezért arra még nagyon sokat kell várni, amíg egy Java vagy Java-szerű programozási környezetnek egyáltalán lehetősége lenne a korábbi kiforrt és jól működő technológiák teljes körű "kiszorítására"). Ez a heterogenitás például a rendszer adminisztrációjánál komoly nehézségeket is okozhat, de tekinthetjük ezt úgy is, hogy ezzel biztosítható annak a lehetősége, hogy minden célra a célnak megfelelő legjobb megoldást alkalmazzuk (ehhez persze szükség van a különféle komponensek működésének összehangolására, összekapcsolására). Ezt a problémát felismerve alakították meg a vezető szoftvergyártó cégek az Object Management Group (OMG) csoportot azzal a céllal, hogy kidolgozza a heterogén elosztott objektum-orientált rendszerekkel kapcsolatos alapvető fontosságú szabványokat. Ma az OMG talán legfontosabb szabványosítási területe az ún. OMA architektúra (objektumkezelési architektúra, angol nevén Object Management Architecture) kidolgozása: a heterogén környezetben lévő elosztott objektumok és alkalmazás-

komponensek rendszerbe illesztését lehetővé tevő technológiák kidolgozása.

Az OMA architektúra definiál egyrészt egy objektummodellt: eszerint az objektumok szolgáltatásait a kliensek (ezek általában más objektumok) csak az objektumok jól definiált interfészmetódusain keresztül vehetik igénybe. Az OMG definiált egy objektuminterfész leíró nyelvet, amit az objektumok kliensei által elérhető metódusok, illetve azok paraméterezésének leírására kell használni (ez az ún. OMG IDL nyelv). Az OMA architektúra emellett definiál egy objektum-referenciamodellt, amely az objektumok kapcsolatát írja le.

Az OMA architektúra referenciamodellje a következő lényeges komponensek specifikációit tartalmazza:

ORB (Object Request Broker, szabadon fordítva: a metódushívás közvetítője): a rendszer elosztott objektumait összekötő szoftverbusz (az elnevezést a számítógép hardver komponenseit összekötő busz néven emlegetett komponenstől kölcsönöztük). Ennek feladata az elosztott objektumok, illetve a rájuk vonatkozó osztott metódushívások megszervezése. Az OMA architektúra ORB komponensét (ezzel együtt az OMG IDL nyelv specifikációját) az ún. CORBA specifikációban specifikálják: ez és ennek Java vonatkozásai képezik majd ennek a fejezetnek a tárgyát.

Általános objektumszolgáltatások (Common Object Services): az elosztott alkalmazások fejlesztéséhez felhasznált alapvető szolgáltatások. Ezek az infrastruktúrát tovább bővítő, de az infrastruktúrának nem szükségképpen részét képező elemek (gondoljunk például a digitális telefonvonalak bevezetésével az előfizetők rendelkezésére álló számos, a digitális központok által nyújtott szolgáltatásokra: az objektumszolgáltatások hasonló szerepet töltenek be a CORBA infrastruktúrájára épülve)[1]. Például ezek közé a szolgáltatások közé tartozik a fejezetben bemutatásra kerülő névszolgáltatás, amely lehetővé teszi a rendszer további objektumainak és komponenseinek a feltérképezését. Egy másik érdekes példa a tranzakciószolgáltatás: ezzel lehetővé válik tetszőleges CORBA objektumok tranzakciókban való részvétele, lényegében úgy, hogy az objektum a tranzakcióban való részvételi képességét egy ilyen célú tranzakciókezelési osztálytól örökli. Ekkor az alkalmazás készítőjének nem a kétfázisú megegyezési protokoll valamely hosszadalmasan implementálható részletének az implementációjára kell koncentrálnia (ui. az alapprotokoll implementációja könnyen örökölhető egy előregyártott tranzakciókezelő osztálytól), hanem arra koncentrálhat, hogy az illető objektumnak mit is kell tennie a kétfázisú megegyezési protokoll egyes fázisaiban.

Alkalmazásterület-függő objektumszolgáltatások (Common Facilities): az objektumszolgáltatásokhoz hasonlóan az elosztott alkalmazások fejlesztését segítő szolgáltatások, de ezek közé már nem az általános infrastruktúrát biztosító eszközök

[1] Szerepük alapján nevezhetnénk őket akár közhasznú objektumszolgáltatásoknak is.

tartoznak, hanem néhány alkalmazásfüggő szolgáltatás. Például ide tartozik egy OpenDoc-szabvány alapú összetett dokumentum szolgáltatás, a DDCF (osztott dokumentum komponens szolgáltatások[2], angol nevén Distributed Document Component Facility): ez specifikálja a komponens dokumentumokból összetett dokumentumok kialakításának módját és eszközeit. A dokumentum-alapú modell azt sugallja, hogy a DDCF a bonyolultabb alkalmazáskomponensek felépítésének szintjén is egy objektum-orientált modellt támogat a hagyományos procedurális modellel szemben: e metafora alapja az, hogy a felhasználó cselekedetei egy jól meghatározott tevékenység elvégzésére irányulnak, nem pedig különféle alkalmazások futtatása áll a középpontban (ha például egy szövegszerkesztő alkalmazás lehetőséget nyújt képi információk dokumentumokba ágyazására, akkor egy beágyazott kép grafikus szerkesztéséhez nem szükséges átmenni egy rajzprogramba, hanem megtehetjük ezt közvetlenül a szövegszerkesztő programon keresztül is; a középpontban a képernyőn megjelenített dokumentum van, és a felhasználó elől rejtve marad, hogy éppen egy szövegszedő, vagy helyesírásellenőrző, vagy éppen egy rajzprogrammal dolgozik). Mivel a DDCF összetett dokumentumai és a komponens dokumentumok is mind CORBA objektumok, ezért egy dokumentum különféle részei egy hálózaton elosztva tárolhatók, és az egyes dokumentum alkotóelemek a CORBA eszközeivel szabadon együttműködhetnek valamilyen cél érdekében (a CORBA eszközeit a fejezet hátralévő részében még részletesen megismerjük, és bízom benne, hogy akkor még jobban megérthetjük az ezen együttműködés megszervezésére rendelkezésünkre álló mechanizmusokat).

Alkalmazásszolgáltatások : konkrét felhasználói alkalmazások számára szükséges szolgáltatások, esetleg egy konkrét alkalmazáshoz szükséges objektumok is tartozhatnak közéjük.

A fejezet további részeiben áttekintjük az OMG CORBA (Közös ORB, Common Object Request Broker Architecture) specifikáció számunkra fontosabb lehetőségeit, majd megnézzük, hogy hogyan írhatunk Java nyelven olyan CORBA objektumokat, amelyek más CORBA objektumok metódusait meghívhatják, illetve a metódusaikat más CORBA objektumok is meghívhatják (függetlenül attól, hogy a kliensek milyen programnyelven lettek implementálva). Megjegyezzük, hogy a CORBA rövidítés közös ORB jelentése azt tükrözi, hogy a szabvány az OMG-nek benyújtott két szabványtervezet összefésülésével jött létre.

A Java programozási rendszernek a CORBA-alapú objektumok létrehozását és kezelését támogató részét Java IDL-nek nevezik. Innen származik ennek a fejezetnek a címe is.

[2] A Java technológia területén a DDCF talán legközelebbi "rokona" a JavaBeans Java komponens modell.

2. Fejezet

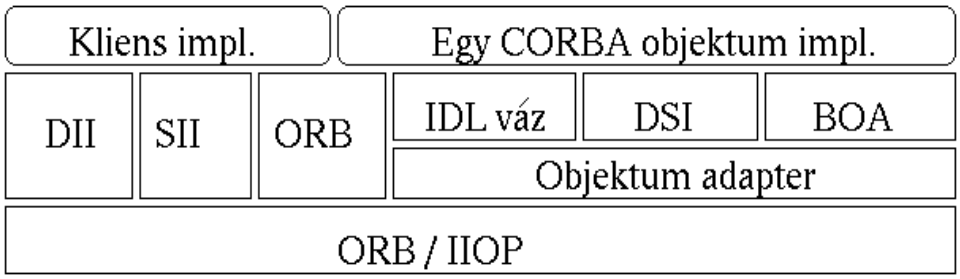
A CORBA specifikáció elemei

A CORBA szabvány definiál egy olyan kommunikációs mechanizmust, amely lehetővé teszi hálózaton elosztott objektumoknak egymás metódusainak hívását, egymás szolgáltatásainak igénybevételét. A CORBA e célok elérése érdekében definiálja az OMA ORB komponensét, jellemzőit és programozói interfészeit.

A CORBA specifikáció első szélesebb körben elterjedt változata az 1.1-es változat volt, amelynek lényeges hiányossága volt a különböző szoftvergyártók különböző ORB szoftvereinek az inkompatibilitása. A CORBA 2.0 specifikáció ezen a téren hozott lényeges újítást, az ORB szoftverek együttműködési protokolljának szabványosításával.

A CORBA 2.0 szabvány főbb elemei a következők (a szabvány ezeket a komponenseket és ezek interfészeit specifikálja):

- ORB
- OMG IDL interfészleíró nyelv
- Az OMG IDL interfészleíró nyelvnek más programozási nyelvekre történő leképezése (maga az OMG IDL egy programozási nyelvektől független szabvány)
- Interfészgyűjtemény
- Implementációgyűjtemény
- Kliens- és szervercsonkok szerkezete (mind a dinamikus, mind a statikus metódushívási modellben)
- Objektumadapterek
- ORB szoftverek együttműködési protokolljai (a GIOP általánosan használható, ESIOP DCE-alapú, valamint az IIOP nevű TCP/IP-alapú együttműködési protokollok tartoznak ide)



A fenti ábraán szerepel néhány olyan hárombetűs rövidítés, amelyről eddig nem volt szó. Most megadjuk ezeknek a rövidítéseknek a jelentését, majd a későbbi pontokban részletesebben is megismerhetjük a szerepüket.

DII : Dinamikus metódushívási interfész.

SII : Statikus metódushívási interfész.

DSI : Dinamikus szerveroldali-csonk interfész.

BOA : Alapvető szolgáltatásokat nyújtó objektumadapter.

A következő pontokban részletesebben is áttekintjük a fent megnevezett komponensek szerepét, elsősorban a CORBA-alapú alkalmazások készítőinek szemszögéből. Megismerhetjük, hogy a különféle CORBA komponensek hogyan vesznek részt egy elosztott objektumokra alapozott alkalmazás megvalósításában.

2.1. Egy CORBA ORB szerepe

Az ORB feladatát már láthattuk, és a szoftverbusz szerepéhez hasonlítottuk: a metódushívási kérelmeket eljuttatja a célobjektumhoz, a visszatérési értéket pedig visszajuttatja a metódushívás helyére. Az ORB mindezt a lehető legtranszparensebb módon kell végezze, ami azt jelenti, hogy az alkalmazások elől a lehető legjobban el kell takarnia azt a tényt, hogy a metódushívás esetleg különböző folyamatok, vagy különböző számítógépek között történik. Az ORB ezt a transzparenciát általában a következő implementációs részletek eltakarásával biztosítja:

- Egy metódushívást végző kliens elől eltakarja az célobjektum tényleges elhelyezkedését és implementációját. Egy metódust hívó kliens nem tudja, hogy az illető objektum melyik számítógépen, melyik folyamatban, milyen programozási nyelven lett elkészítve.
- A metódushívást végrehajtó kliensnek nem kell tudnia az elérni kívánt objektum állapotáról, vagyis arról, hogy az objektumimplementáció fut-e (aktív-e), vagy éppen perzisztens módon ki van tárolva a háttértárra (például egy mágneslemez

fájlrendszerére). Az ORB szoftvernek lehetősége van a hivatkozott nem aktív objektumok igény szerinti aktiválására (aktiváláskor egy objektum belső állapota a háttértáron tárolt információk alapján lesz inicializálva).

- A kliens nem ismeri az ORB által a metódushívás közvetítésére használt módszereket. Az egyazon folyamaton belüli, és azonos programozási nyelven implementált objektumok metódusainak hívása általában az objektumok implementációjára használt programozási nyelv helyi metódushívási mechanizmusaival történik, nem objektum-orientált nyelvek esetén pedig helyi eljáráshívási mechanizmusokkal. A különböző folyamatok közötti metódushívás történhet akár a TCP/IP protokollcsalád valamely transzportprotokolljával, akár valamilyen más TCP/IP-alapú vagy nem TCP/IP-alapú kommunikációs protokollal.

Az ORB feladata a távoli (és helyi) objektumreferenciák kezelése. Amikor létrehoznak egy CORBA objektumot, akkor az objektumot létrehozó folyamat ORB komponense létrehoz egy világszerte érvényes egyedi objektumazonosítót (ennek neve IOR, ORB-k közötti érvényű referencia). Az alkalmazások ezeket az IOR azonosítókat továbbadhatják más alkalmazásoknak. Az IOR azonosítók az alkalmazások közt átadhatók a metódusok paramétereiben, illetve visszatérési értékeiben; átadhatók egy speciális CORBA névszolgáltatón keresztül; ezenkívül lehetőség van az IOR referenciák szöveges formára alakítására, illetve a visszafelé konverzióra is, ami lehetővé teszi e referenciák szöveges formában történő továbbítását is.

Egy CORBA ORB biztosít egy alapvető, a Java RMI-nél megismert registryhez hasonló névszolgáltatót, amin keresztül az alkalmazások hozzájuthatnak a működésükhöz szükséges IOR referenciákhoz.

Megjegyezzük, hogy a Java RMI rendszerhez hasonlóan itt sem beszélhetünk a metódusok között kitüntetett konstruktor metódusokról. Helyettük itt is ún. objektumgyártó objektumokat kell alkalmazni (de ezek is CORBA objektumok), amelyek egy-egy metódusuk végrehajtásakor létrehoznak egy új CORBA objektumot.

2.2. Az OMG IDL interfészleíró nyelv

Említettük a CORBA szabvány elemei között az OMG IDL nyelvet (interfészleíró nyelv, angolul Interface Definition Language), mint a CORBA objektumok interfészének specifikálására használható általános eszközt. Egy CORBA objektum interfészének specifikációja tartalmazza mind az objektumok távolról elérhető metódusainak, mind az esetleges adattagjainak a specifikációját az egyes elemek típusára vonatkozó információkkal együtt (hasonlítható például egy Java interfész nyelvi elemhez). Az OMG IDL egy deklaratív nyelv, amivel csak az objektumok interfészének leírására van lehetőség, nincs lehetőség vele önmagában objektumimplementációk elkészítésére (nincs lehetőség például értékadó utasítások, szekvenciák, elágazások vagy ciklusok írására). Az OMG

IDL nyelv fontos jellemzője a programozási nyelvektől való függetlensége, ami az IDL szerkezeteknek a különféle programozási nyelvekre való leképezési szabványaival együtt kiválóan támogatja az alkalmazások egymáshoz illesztését heterogén számítógépes környezetekben is.

Az OMG IDL specifikáció főbb elemei a következők:

Modulok Ezek szerepe elsősorban az IDL névtartomány hierarchikus kiterjesztésének támogatása miatt fontos. Egy adott nevű modulban definiált azonosítókra más modulokból a modul nevével minősítve hivatkozhatunk (a minősített és a minősítő azonosítókat két darab kettőspont `::` karakterrel válasszuk el egymástól).

Konstansok Segítségükkel állandó értékeket rögzíthetünk az IDL specifikációs fájlban.

Interfészek Az IDL specifikációnak ezen részei definiálják az objektumok távolról hívható metódusait, attribútumait (ez utóbbiak tulajdonképpen az osztályok adattagjai, amelyekhez értéküket lekérdező és beállító műveletek lesznek generálva). Az interfészekben definiált metódusok kiválthatnak különféle kivételeket is; az IDL specifikációban használt kivételek a Java kivételkezelésénél megismert kivételekhez hasonlóan használhatók hibás, illetve rendkívüli helyzetek jelzésére. Az OMG IDL lehetőséget nyújt interfészek közötti öröklési relációk leírására is, támogatja a többszörös öröklődést (Java nyelv interfészeinél megismerthez hasonló szintaxissal).

Adattípusok Az IDL specifikációnak ezen részei specifikálják az interfészekben levő metódusok paramétereinek típusát, visszatérési értékük típusát, illetve az általuk kiváltható kivételek jellemzőit. Az OMG IDL adattípusait rövidesen részletesebben is át fogjuk tekinteni.

Műveletek Az IDL specifikációnak ezen részei definiálják az egyes IDL interfészelemek egy-egy metódusát, annak az ún. szignatúráját: azt, hogy mi az illető metódus neve, milyen típusú paraméterei vannak, milyen a visszatérési értéke, milyen kivételeket vált ki. Az egyes műveletek paramétereihez definiálva van az illető paraméterek átadási módja is. A CORBA három paraméterátadási módot definiál:

- Klienstől a szerver felé történő paraméterátadás. Ez az `in` szócskával van jelölve az IDL interfész specifikációban. Ez lényegében a Java nyelven megismert egyetlen ún. érték szerinti paraméterátadási mód, amit a CORBA objektumok átadása esetében egy CORBA objektumreferencia átadásával kell megvalósítani (nem pedig egy, az illető objektum belső állapotának lemásolásával kapható új objektum létrehozásával).
- Szervertől a kliens felé történő paraméterátadás. Ez az `out` szócskával van jelölve az IDL interfész specifikációjában.

- Mindkét irány egyszerre (a híváskor a klienstől a szerver felé, majd visszatéréskor a szervertől a kliens felé). Ez az `inout` szócskával van jelölve az IDL specifikációban.

Egy CORBA IDL specifikációs fájl szerkezete a következő:

```
module <modulazonosító>;

{
  <adattípusok specifikációi>;
  <konstansok deklarációi>;
  <kivételek specifikációja>;

// Egysoros megjegyzéseket ilyen módon írhatunk a programjainkba
// Ez egy második megjegyzéssor

  interface <interfésznév> : <szülőinterfésznév1>, <szülőinterfésznév2>,...
  {
    <adattípusok specifikációi>;
    <konstansok deklarációi>;
    <kivételek specifikációi>;
    <attribútumok deklarációi>;

    <visszatérési érték típusa><művelet neve> (<paraméterek>)
        [raises <kiváltható kivételek azonosítóinak listája>]
        [context ( <szövegkonstans> { , <szövegkonstans> } ) ] ;

    ...
  };

  ...

  interface Előre_deklarált_név; // ez a FORWARD elődeklaráció szintaxisa

  ...
  // itt már hivatkozhatunk az Előre_deklarált_név interfészre
  ...

  interface Előre_deklarált_név { ... }; // itt van a tényleges deklaráció

  ...

};
```

Megjegyezzük, hogy egy interfészen, illetve modulon belül a komponensek sorrendjére vonatkozóan nincs más lényeges szintaktikai megkötés, csak a szemantikus egymásra hivatkozás szabályait kell betartani. Szükséges lehet az azonosítók Pascal nyelvben megismertekhez hasonló elődeklarációja (FORWARD deklaráció), egyszerűen az előre deklarálni kívánt elem nevének a leírásával (ez a definíció lehetővé teszi az elődeklarált azonosító paraméterlistákban történő felhasználását, de mást nem).

2.3. Az OMG IDL nyelv elemei

Az OMG IDL számos alaptípust, és számos típuskonstrukciós módot biztosít a metódusok paramétereinek, visszatérési értékeinek jellemzésére. A CORBA specifikáció meghatározza az alaptípusok lehetséges értékeit, értéktartományát, ábrázolási pontosságát, stb. ...

IDL alaptípusok: Az OMG IDL alaptípusai a következők:

objektumreferencia

`long` (`signed` vagy `unsigned`): 32 bit széles egész típus (előjeles, illetve előjelnélküli).

`long long` (`signed` vagy `unsigned`) : 64 bites egész típus (előjeles, illetve előjelnélküli).

`short` (`signed` vagy `unsigned`) : 16 bit széles egész típus (előjeles, illetve előjelnélküli).

`float`, `double`, `long double`: az IEEE 754-1985 szabvány szerinti lebegőpontos típusok.

`char`: 8 bit széles karakter (létezik egy wchar típusazonosító is, 8 bitnél szélesebb karakterekhez).

`boolean`: logikai típus.

`octet`: 8 bit széles adategység. Átvitelekor nem lesz konvertálva különböző architektúrájú számítógépek között sem.

`enum`: felsorolási típus. A típusnév mellett bevezeti a típusértékek halmazát is, amelyeket vesszővel elválasztva kell felsorolnunk. Példa:

```
enum Napok { hétfő, kedd, szerda, csütörtök, péntek, szombat, vasárnap};
```

`string<n>`: karakterlánc típus, ahol az `<n>` paraméter a karakterlánc maximális hosszát adja meg, illetve nincs maximális korlát, ha ezt a szintaktikus egységet elhagyjuk.

`wstring<n>`: karakterlánc típus, `wchar` típusú komponensekből felépítve.

`fixed<n,m>`: fixpontos aritmetikai típus. Értékei olyan legfeljebb 31 jegyű számok lehetnek (ezt kell megadni a típusspecifikáció első `<n>` paraméterében), amelyeknek a második `<m>` paraméterben megadott számú számjegye van a tizedesvessző után. Például a `fixed<3,1>` típusú objektumok a 0.0-99.9 intervallumba eső számokat 1 tizedes pontossággal tárolhatják.

`Any`: az ilyen típusú objektumok bármilyen más IDL adattípusú értéket tartalmazhatnak, akár elemit, akár felhasználó által definiált, vagy összetett adattípust.

IDL konstansok: Az IDL konstansok definícióit a `const` kulcsszóval kell bevezetni. Ezután írjuk a konstans típusát, majd a konstans érték nevét. Ezután egy egyenlőség jelet követve írhatjuk a konstans értékét egy konstans kifejezés formájában. A konstans kifejezésekbe írhatók konstans literálok, valamint a következő unáris műveleti jelek: `-`, `+`, `~` (azaz az előjelek, valamint a bitenkénti komplementerképzés operátora), illetve a kétoperandusú műveleti jelek: `|` (bitenkénti logikai VAGY), `&` (bitenkénti logikai ÉS), `^`, `>>`, `<<`, `+`, `-`, `*`, `/`, `%` (maradékképzés). A kifejezéseket tetszőlegesen zárójelezhetjük is.

Példa:

```
const long harminc = 30 ;
```

A konstans kifejezések képzésekor a következő literálokat használhatjuk:

`egész literál` : nem nullával kezdődő számjegyek sorozata tízes számrendszerben értendő; nullával kezdődő számjegyek sorozata oktális (nyolcas) számrendszerben, míg a `0x` vagy `0X` karakterekkel kezdődő számjegy-sorozat hexadecimális (tizenhatos) számrendszerben értendő.

`karakteres literál` : aposztrófok közötti, egyetlen karakter hosszú konstans. A `\ooo` módon az `ooo` oktális, míg `\xhh` a `hh` hexadecimális konstanssal megadott kódú karaktert azonosítja.
A következő speciális karakterek vannak még - a "szokásos" jelentéssel - definiálva: `\n, \t, \v, \b, \r, \f, \a, \\, \?, \', \"`.

`karakterlánc literál` : karakterek sorozata (ld. az előbbi `karakteres literál` elemnél) idézőjelek között. Egymást követő karakterlánc literálok egyetlen karakterlánc literállá lesznek összefűzve (azaz az "AB" "C" egymás után írva és "ABC" ugyanazt reprezentálják). Nem tartalmazhatja a `\0` karaktert.

`lebegőpontos literál` : egy egészrészt követ egy tizedespont, azt egy tizedes rész, majd egy "e" vagy "E" karakter, és ezt követi egy előjeles egész kitevő kifejezés. Vagy az egészrész, vagy a tizedes rész hiányozhat (mindkettő nem), és vagy a tizedespont, vagy az "e" (ill. "E") és az azt követő kitevő hiányozhat (de mindkettő nem).

Az OMG IDL típuskonstrukciós eszközei: Az OMG IDL típuskonstrukciós eszközei a következők:

Az IDL tömb típusai egy- vagy többdimenziós tömbök lehetnek. A tömbök egyes dimenzióinak mérete fordítási időben előre rögzített, az adattípus deklarációjakor az indexhatárokat rögzítő pozitív konstans értékeket (vagy kifejezéseket) szögletes zárójelekben kell megadni. Példa:

```
typedef long NagyNagyVektorVektor[7][6][5][4];
```

A rekord típus konstrukciója a C/C++ nyelv struktúráihoz hasonlít. Egy IDL struktúra szerkezete a következő:

```
struct azonosító {
  típus_specifikáció1 adattag_nevek1;
  .. // típus_specifikáció2 adattag_nevek2;
  .. // stb.
}
```

A fenti példában az `adattag_nevek` részek azonosítók (ezek akár tömb típusok is lehetnek) vesszővel elválasztott sorozatát tartalmazza.

Az unió típus hasonlít a többi programozási nyelvekben is megismert unió típushoz (lásd például a C nyelv union típusát, vagy a Pascal nyelv variáns rekordjait). A CORBA union szerkezetének pontos ismertetését a fejezet hátralévő részeiben majd láthatjuk.

```
union azonosító switch (diszkrimináns_típusa) {
  case konstans_kifejezés: ...
  case konstans_kifejezés: típus_specifikáció adattag_nevek;
  .. // stb.
  case konstans_kifejezés: típus_specifikáció adattag_nevek;
  default: típus_specifikáció adattag_nevek;
}
```

A diszkrimináns típusa egész, karakteres, logikai, vagy felsorolási típus lehet. Egy-egy unióághoz tetszőleges számú `case` címkét írhatunk. `default` ág legfeljebb egy lehet, és csak akkor, ha a megadott konstans kifejezések nem fedik le a megadott diszkrimináns típus értékhalmazát.

A szekvencia típuskonstrukció előre rögzített típusú elemeknek egy dinamikusan változtatható hosszúságú sorozatát hozza létre. A létrehozott szekvencia maximális elemszámát[1], illetve elemeinek a típusát egy `sequence` kulcsszó mögé tett `<` és `>` relációs jelek közé írhatjuk. Például a `sequence<Object>` definíció CORBA `Object` osztályba tartozó elemek referenciáinak egy nem korlátozott hosszúságú sorozatát definiálja, míg a `sequence<Object,42>` az említett sorozat hosszát negyvenkettőben maximálja.

Öröklődés leírása: Az OMG IDL fontos jellemzője az osztályok (pontosabban az interfészek) közötti öröklési kapcsolatok leírásának a lehetősége (az OMG IDL ettől igazán objektum-orientált). Az IDL specifikációs fájlok szerkezetét szemléltető,

[1] Az elemszám egy pozitív egész értékű konstans érték lehet.

korábban már bemutatott példában láthattuk az interfészek definíciójának szintaxisát: az újonnan definiált interfésznév után egy kettőspontot követően vesszőkkel elválasztva soroljuk fel azoknak az interfészeknek a neveit, amelyektől az újonnan definiált interfész örököl. Megjegyezzük, hogy az OMG IDL összes interfésze örököl a `CORBA` modul `Object` nevű interfészétől (ennek tehát hasonló funkciója van, mint a Java nyelv osztályainál a `java.lang.Object` osztálynak, az ettől való öröklést nem kell explicite jelölni az interfész definíciókban).

Egy interfész specifikációjában a szülőinterfészek elemeit a nevükön érhetjük el, hacsak nem definiáltuk át őket. Ilyenkor a már említett `::` operátorral hivatkozhatunk a szülőkben definiált azonos nevű és szerkezetű azonosítókra (típusok neveire, konstansok neveire vagy kivételek neveire). Nem szabad két, azonos metódusnevet vagy adattagnevet tartalmazó interfésztől egyszerre örökölni.

Kommunikációs szemantika megadása: Az OMG IDL lehetővé teszi az objektum metódusok kommunikációs szemantikájának rögzítését is. A CORBA metódusok hívása alapvetően szinkron módon történik: a metódushívást végző program várakozik addig, amíg a hívott metódus futása befejeződik, és az esetleges kommunikációs hibákat az ORB szoftver kivételek formájában visszajelzi az alkalmazásnak. Jelenleg a CORBA 2.0 szabvány IDL nyelve egyetlen ilyen szemantikamódosító elemet tartalmaz, amire a metódusok a legfeljebb egyszeri próbálkozás szemantikájával lesznek meghívva (ha a hálózati üzenet valahol elveszne, akkor a metódus nem lesz meghívva). Természetesen az ilyen - az IDL fájlban `oneway` módosítóval deklarált - metódusok visszatérési értékére nem hivatkozhatunk, mivel ilyen körülmények között nem minden esetben van ennek értelme (az így hívott metódusok nem dobhatnak vissza kivételeket sem). E két lehetőség kombinálására is lehetőség van a CORBA dinamikus hívási modelljében, aminek az IDL fájlban való deklarációja nem lehetséges. Általában az `oneway` metódusoknak csak `in` módosítóval ellátott bemenő paraméterei lehetnek.

Adattagok: A CORBA objektumok adattagjait (a CORBA terminológiában: attribútumait) elláthatjuk a `readonly` módosítóval. Ezekhez az adattagokhoz csak az értéküket lekérdező metódusok lesznek generálva a programnyelvi leképezéseknél. Az adattagok deklarálása IDL nyelven az `attribute` kulcsszó mögött történik. Tekintsük erre az alábbi példát:

```
attribute long egy_attributum;
```

Kivételek: A CORBA definiál egy kivételkezelési mechanizmust is, amit a későbbiekben fogunk részletesebben ismertetni. Egyelőre csak annyit érdemes erről megjegyezni, hogy a Java nyelvhez hasonlóan az IDL nyelven lehetőségünk van kivételek definiálására, és az egyes metódusok definíciójában a metódus által kiváltható

kivételeket a paramétereket követő `raises` kulcsszó után kell megadni (zárójelek között felsorolni) a kiváltható kivételek neveit. Egy kivételdefiníció szerkezete a következő:

```
exception azonosító {
  típus_specifikáció1 adattag_nevek1;
  .. // típus_specifikáció2 adattag_nevek2;
  .. // stb.
}
```

A kivételek belső állapotát leíró mezőket a rekord típus specifikációjánál megismertekhez hasonlóan itt is meg kell adnunk. A rekord típussal ellentétben a kivételeknek nem kell, hogy állapotváltozói legyenek. Tekintsük erre a következő példát:

```
exception azonosító2 { }
```

Ha egy CORBA objektum valamely metódusának végrehajtása során kivételt vált ki, akkor a kimenő (`out` és `inout`) paraméterek értéke definiálatlan (ugyanez igaz a metódusok visszatérési értékére is, azaz ez is definiálatlan). A hívás helyén a kivétel kiváltásának a ténye az az információ, amit ilyenkor az alkalmazás felhasználhat.

Hívási kontextus: A CORBA alkalmazások rendelkeznek egy ún. hívási kontextussal, ami különféle nevekhez rendelt értékek halmazából áll[2] (mind a nevek, mind pedig a hozzájuk rendelt értékek szöveges formátumú adatok, a nevekre vonatkozó megkötés, hogy alfanumerikus, pont, valamint aláhúzás karakterekből állhatnak, és első karakterük betű kell legyen). A CORBA alkalmazásoknak megvannak a maguk eszközei ezeknek az elemeknek a módosítására (erről a dinamikus metódushívási interfész ismertetésekor még részletesebben olvashatunk). A CORBA metódusok által kiváltható kivételek specifikációját követő `context` záradékban a hívási kontextus azon elemeinek a nevét kell megadni, amelyeket - értékükkel együtt - át akarunk adni a hívott objektumot tároló szervernek az illető metódus végrehajtásakor; az ORB szoftver a metódus hívásakor gondoskodik ezeknek az adatoknak az objektumot implementáló szerverhez történő eljuttatásáról.

Egyéb megjegyzések: A fentiekben bemutatott lehetőségeken túl a következőket érdemes megjegyeznünk:

- Az IDL fájl feldolgozása során keresztülmegy egy makro preprocesszoron; a preprocesszornak szóló direktívák felépítése hasonlít a C++ nyelvnél megismertekhez (egy # jellel kezdődnek az őket tartalmazó sorok).

[2]Ezt az eszközt szerepét tekintve hasonlíthatjuk például az operációs rendszerek környezeti változóihoz, vagy hasonlíthatjuk őket akár a Java nyelv környezeti jellemzőihez is.

- Megjegyzéseket írhatunk egyrészt /* és */ szimbólumok közé (az említett szimbólumok közti megjegyzés tetszőleges hosszúságú lehet, és az ilyen megjegyzések nem ágyazhatók egymásba), illetve egysoros - a sor végéig tartó - megjegyzéseket a // jelekkel kezdhetjük.
- Az azonosítókkal kapcsolatban annyit érdemes megemlíteni, hogy azok alfanumerikus, valamint aláhúzás (azaz _) karakterek tetszőleges hosszúságú sorozatai lehetnek. Az első karakter betű kell legyen; az azonosítók összes karaktere szignifikáns (azaz nem az első hat vagy nyolc karaktere határozza meg egyértelműen az azonosítót, mint azt néhány programnyelvben esetleg láthattuk); a kis és a nagybetűk összehasonlításkor azonosnak tekintendők.

A fejezet hátralévő részében több megjegyzésekkel ellátott IDL fájlt láthatunk, így az OMG IDL nyelv egyéb részleteivel - mint például a konstansok, adattípusok specifikációjával - most nem foglalkozunk.

Megjegyezzük, hogy a Java RMI távoli metódushívási rendszer is használ egy interfészleíró nyelvet: magának a Java nyelvnek az interfészeit használhatjuk a távoli objektumok által nyújtott metódusok specifikálására (emlékezzünk rá, hogy a távoli objektumok interfészeit a `java.rmi.Remote` interfészből kell származtatni).

2.4. Az OMG IDL leképezése programozási nyelvekre

Az OMG IDL nyelv programozási nyelvektől függetlenül van definiálva, de a programjainkban levő CORBA objektumokat mindig valamilyen procedurális vagy objektumorientált programozási nyelven készíthetjük el (hiszen az OMG IDL egy deklaratív nyelv). Látható, hogy szükség van a specifikációs és az implementációs eszköz (azaz az OMG IDL és az illető interfész implementálására használt programozási nyelv) közötti kapcsolat megteremtésére. Ezt a kapcsolatot az IDL nyelvnek a különféle programozási nyelvekre történő leképezését tartalmazó szabványok teremtik meg. A nyelvi leképezések a következő területeket szabványosítják:

- Az OMG IDL adattípusainak a leképezését az illető programozási nyelv szabványosított adattípusaira.
- A kliens- és a szervercsonkok szerkezetét.
- A CORBA objektumok implementációjának számos más részletét (az illető nyelvre vonatkoztatva).
- Az ORB szolgáltatásainak elérését biztosító ún. CORBA pszeudo-objektumok[3] leképezési módját az illető nyelvre, illetve programozási környezetre (ide tartozik például a CORBA alkalmazások inicializálásának a módja).

[3] A pszeudo-objektum elnevezés arra utal, hogy ezek is objektumok, de esetenként nem CORBA objektumok abban az értelemben, hogy referenciáik nem adhatók át egyes CORBA metódushívások paraméterében. Ezek az objektumok nincsenek képviselve az interfészgyűjteményben sem (ld. később).

Mára már számos programozási nyelvhez elkészítették és szabványosították az OMG IDL nyelv leképezési szabályait. A CORBA 2.0 specifikáció tartalmazza többek között az OMG IDL nyelvnek a C, C++, Smalltalk programozási nyelvekre leképezésének a módját, míg például a Java nyelvre való leképezést egy önálló szabvány rögzíti (egyebek közül a COBOL és az Ada95 nyelvi leképezéseket érdemes megemlíteni).

2.5. Az interfészgyűjtemény szerepe

A CORBA interfészgyűjtemény a CORBA interfészeknek a futás során elérhető adatbázisából áll. Az osztott CORBA metódushívások végrehajtása során az ORB ennek az adatbázisnak a segítségével tudhatja meg azt, hogy a meghívott metódus milyen típusú paramétereket vár (ezekre az információkra a dinamikus CORBA metódushívások során van igazán szükség, akkor ugyanis a program a futása során döntheti el, hogy melyik metódust akarja meghívni; a dinamikus CORBA metódushívásokkal az alkalmazások akár olyan metódusokat is meghívhatnak, amelyek a lefordításuk pillanatában még nem is voltak definiálva).

Maga az interfészgyűjtemény CORBA objektumok összessége. Metódusokat ad a rendszerben élő interfészek komponenseinek a "felfedezésére". Egy alkalmazás például megteheti azt, hogy az interfészgyűjtemény metódusaival végignézi a rendszerben levő összes CORBA moduldefiníciót, majd ha az alkalmazás megtalálta a számára szükséges moduldefiníciót, akkor az ezt reprezentáló objektum metódusaival végignézheti a benne definiált interfészek, konstansok, és más IDL elemek szerkezetét.

Az interfészgyűjtemény tartalmát reprezentáló - szintén CORBA - objektumok általában a következő osztályok példányai: `ModuleDef`, `InterfaceDef`, `OperationDef`, `ParameterDef`, `AttributeDef`, `ConstantDef`, `ExceptionDef`, `TypeDef` (rendre aszerint, hogy milyen IDL konstrukciót reprezentálnak), valamint a `Repository` osztályt is ide kell sorolni, amely az interfészgyűjteményt alkotó objektumok hierarchiájának gyökerét képező objektum osztályát írja le. Megjegyezzük, hogy az általunk használt Java-alapú ORB szoftver nem biztosítja ennek az eszköznek a szolgáltatásait.

A CORBA objektummodelljének része az, hogy bármelyik CORBA objektum a `get_interface()` metódusának meghívására képes visszaadni egy, a saját szerkezetét és felépítését leíró CORBA objektumra hivatkozó referenciát, amin keresztül az alkalmazás feltérképezheti az illető objektum által nyújtott szolgáltatások interfészeit. Ez a leíró objektum az interfészgyűjteményben található. Vannak metódusok ezen interfészgyűjtemény hierarchikus bejárására, és az egyes komponensek elemzésére (pl. milyen interfészeket implementál, kiktől miket örököl, stb.).

2.6. Az implementációgyűjtemény szerepe

Az implementációgyűjtemény a rendszerben levő CORBA objektumokról tartalmaz információkat (például azt, hogy az illető objektumok hogyan érhetők el, hol vannak tárolva, illetve hogyan lehet őket aktivizálni, ha valamikor szükség lesz rájuk). Minden elkészült CORBA objektumimplementáció jellemzőit bejegyzik ebbe az implementációgyűjteménybe. A CORBA objektumok klienseit, vagyis a CORBA objektumok metódusait hívó programokat nem kell az implementációgyűjteménybe bejegyezni, hacsak maguk nem definiálnak - szerver feladatokat is ellátva - új CORBA objektumokat.

Megjegyezzük, hogy az általunk használt Java-alapú ORB szoftverben nincs implementációgyűjtemény, így ezzel a továbbiakban nem foglalkozunk.

2.7. A kliens- és a szervercsonkok

A kliens- és a szervercsonkok szerepe hasonlít a Java távoli metódushívásról szóló részben megismert kliens- és szervercsonkok szerepéhez: a klienscsonk egy távoli objektum olyan helyi reprezentánsa, amely a metódushívási kéréseket továbbítja az általa reprezentált távoli objektum felé, míg a szervercsonk a távolról érkező metódushívási kéréseket a tényleges objektumimplementációk felé továbbítják. Ezeket a csonkokat a CORBA környezetben generáltathatjuk - a Java RMI-nél megismertekhez hasonlóan - egy segédprogrammal egy-egy IDL interfész specifikáció alapján; erre alapul a CORBA ún. statikus metódushívási modellje (a statikusság itt abban áll, hogy az interfész metódusai már a csonkok generálásakor ismertek kell legyenek). Emellett a CORBA támogat egy dinamikus hívási modellt is, ahol a rendszer az interfész-specifikus kliens- és szervercsonkok helyett egy-egy "általános célú" csonkimplementációt használ (az "általános célú" csonkimplementációk képesek a metódusok olyan módú végrehajtására, hogy a hívott metódusok paraméterezését a rendszer például az interfészgyűjtemény alapján állítja össze). A dinamikus modellt a CORBA két fontosabb interfész definiálásával támogatja: a DII (dinamikus metódushívási interfész), valamint a DSI (dinamikus szerveroldali csonk-interfész) szabványos komponensekkel.

2.8. A CORBA objektumadapterek szerepe

A szerverobjektumok az objektumadapteren keresztül tartják a kapcsolatot az ORB-vel. Az objektumadaptereket azok az interfészek alkotják, amelyekkel a szerveroldali alkalmazások elérhetik az ORB szolgáltatásait. A CORBA specifikáció egyetlen ilyen objektumadaptert definiál, az ún. BOA adaptert (a BOA a Basic Object Adapter kezdőbetűiből áll össze, jelentése alapvető szolgáltatásokat nyújtó objektumadapter). A BOA lehetőséget nyújt a következő feladatok megoldására:

- Objektumreferenciák generálása CORBA objektumokhoz, valamint objektumrefe-

renciák értelmezése.

- Objektumimplementációk bejegyzése az implementációgyűjteménybe.
- Objektumimplementációk aktiválása (szükség esetén az objektumot - vagy legalábbis annak kezdőállapotát - perzisztens tárolóról betöltve), illetve objektumimplementációk deaktiválása.
- Egy metódushívás kezdeményezőjének igazolása az objektumimplementáció felé (itt egy biztonsági célú azonosítási mechanizmusról van szó).

Megjegyezzük, hogy a BOA operációs rendszertől függő eszközökkel tartja a kapcsolatot az objektumimplementációkkal. Ezek az interfészek igaz, hogy OMG IDL nyelven vannak specifikálva, de a BOA teljes hordozhatóságát komoly mértékben akadályozza a működéséhez alapvető fontosságú implementációgyűjteményben tárolt információk rendszerfüggő ábrázolása.

Természetesen a BOA adapteren kívül léteznek egyéb objektumadapterek is. Például több adatbáziskezelő definiál egy adatbázis alapú objektumadaptert, amely adatbázis-objektumokból (például táblák, nézettáblák, sorok, mezők) programnyelvi, illetve CORBA objektumokat "csinál" (az illető adatbázis-objektumok ezután CORBA objektumokként is elérhetők lesznek).

A CORBA 2.0 specifikáció alapján a BOA objektumadapterek négy alapvető implementáció-, illetve objektumaktiválási módot támogatnak (egy implementáció aktiválásán azt értik, amikor el kell indítani egy objektumot - vagy egy adott objektumcsoportot - kezelő szerverprogramot, míg egy objektum aktiválásán azt, amikor egy már futó szerverprogramban kell egy új CORBA objektumot létrehozni, és mások számára azt elérhetővé tenni):

Több objektumnak közös szerver alapú stratégia esetén több CORBA objektum egy közös CORBA szerverprogramon belül helyezkedik el. A szerver implementációt tartalmazó programot a BOA akkor indítja[4], amikor egy kliens alkalmazás valamelyik általa implementált objektum valamelyik metódusát meghívja. Az implementáció egy `impl_is_ready()` metódussal jelezheti a BOA adapternek, hogy inicializálta magát, és készen áll a metódushívási kérelmek kiszolgálására (az objektum aktiválását kezdeményező metódushívás végrehajtását a BOA csak ekkor kezdeményezi). A szerverprogram egészen addig aktív marad, amíg meg nem hívja a `deactivate_impl()` BOA metódust (ha valaki ezt követően újra el akarja érni az objektumot, akkor azt a BOA adatpernek újra aktiválnia kell).

Megjegyezzük, hogy mielőtt egy adott objektum egy metódusa végrehajtódna, egy objektumaktiválási metódus is meg lesz hívva, ami az illető objektum belső állapotát

[4]Megjegyezzük, hogy ez az indítás operációs rendszertől is függő eszközök felhasználásával történhet; a CORBA specifikáció csak annyit követel meg, hogy a BOA biztosítson eszközöket ezen feladatok megoldására.

inicializálhatja. Az illető objektum egészen addig aktív marad (azaz kész a metódushívási kérések kiszolgálására), amíg az őt tartalmazó szerverprogram fut, vagy meg nem hívja az illető objektum `deactivate_obj()` metódusát.

Objektumonkénti önálló szerver alapú stratégia esetén egy szerverprogram egy CORBA objektum implementációját tartalmazza. Egy új szerver aktiválása az illető - addig még nem használt - objektum valamely metódusának meghívásához van kötve. Miután az illető objektumot kezelő szerver inicializálta magát, a BOA `obj_is_read()` metódusának meghívásával jelezheti, hogy készen áll a kliensek metódushívási kérelmének feldolgozására, majd egészen addig készen kell állnia a metódushívási kérelmek fogadására és kiszolgálására, amíg a `deactivate_obj()` metódusát meg nem hívja.

Metódushívásonkénti önálló szerver alapú stratégia esetén egy-egy metódus hívásakor lesz az illető metódushívást kiszolgáló szerverprogram elindítva; egy-egy ilyen szerver csak egy metódushívás kiszolgálására van felkészítve, és utána befejeződik a futása.

Perzisztens szerver alapú stratégia esetén a kliens alkalmazások feltételezhetik az illető objektum állandó elérhetőségét (azt nem a BOA objektumadapter indítja, hanem például valamilyen operációs rendszer mechanizmus - például POSIX-szerű operációs rendszereken indíthatják őket akár az `init` vagy a `cron` démonok); ha egy szerver mégsem lenne elérhető vagy elindítható olyankor, amikor szükség van rá, akkor egy kivétel lesz generálva. Természetesen az objektumimplementációk elérhetőségéről itt is értesíteni kell a BOA objektumadaptert - erre az `impl_is_ready()` metódust használhatjuk.

Megjegyezzük, hogy a könyv példaprogramjainál használt ORB implementáció - a JavaSoft JOE ORB szoftvere - jelenleg nem implementálja az összes szükséges BOA szolgáltatást, ezért az ebben a pontban megnevezett metódusok helyett más nevű metódusokat fogunk meghívni. A helyzet persze előbb-utóbb valószínűleg változni fog. A másik ORB szoftverek közül a Visibroker az `osagent` nevű segédprogrammal biztosítja ezen szolgáltatások nagy részét, az OrbixWEB pedig a `putit` segédprogramjával biztosítja a CORBA szerverprogramok regisztrálását, és az Orbix démon folyamat biztosítja ezek igény szerinti indítását.

2.9. A CORBA objektumok ősosztálya: CORBA::Object

A CORBA Object nevű osztálya az összes CORBA objektum őse, így ennek az osztálynak a metódusaival minden CORBA objektum rendelkezik - ezért érdemes őket röviden áttekinteni. A `CORBA::Object` interfész IDL specifikációját a következő lista tartalmazza.

```
module CORBA {
  interface Object {
    ImplementationDef get_implementation();
    InterfaceDef get_interface();
    boolean is_nil();
    Object duplicate();
    void release();
    boolean is_a(in string a_típus_logikai_azonosítója);
    boolean non_existent();
    boolean is_equivalent(in Object másik_objektum_referenciája);
    unsigned long hash(in unsigned long maximum);

    Status create_request(in Context ctx, in Identifier művelet,
           in NVList paraméterlista, inout NamedValue visszatérési_érték,
           out Request kérés, in Flags kérés_jellemzők);
  }
}
```

A `get_interface()` metódus visszaadja az illető objektum (vagyis amelyiknek ezt a metódusát meghívták) típusát reprezentáló CORBA objektum egy referenciáját (a visszaadott objektum általában az interfészgyűjteményben található).
A `get_implementation()` metódus visszatérési értékként egy referenciát ad vissza egy implementációgyűjteménybeli objektumra. A visszaadott referencia az implementációgyűjtemény azon objektumára vonatkozik, amely annak az objektumnak az implementációjáról tartalmaz további információkat, amelyre vonatkozóan ezt a metódust meghívták.
A `duplicate()` metódus segítségével létrehozhatunk egy másolatot erről a CORBA objektumreferenciáról, míg a már nem használt objektumreferenciákat a `release()` metódussal szüntethetjük meg (erre akkor van szükség, ha a használt programozási nyelvi nem biztosít automatikus szemétgyűjtést).

Azok a CORBA objektumreferenciák, amelyeknek az értéke `OBJECT_NIL`, nem hivatkoznak semmilyen CORBA objektumra. Ezt az esetet ellenőrizhetjük az `is_nil()` metódussal, amely annak megfelelően ad vissza egy logikai igaz vagy hamis értéket, hogy az objektumreferencia értéke `OBJECT_NIL`-e (ez a Java `null` referenciájának CORBA megfelelője).

Az `is_a()` metódussal ellenőrizhetjük egy objektumról, hogy az illető objektum implementálja-e az `is_a()` hívás paraméterében megadott nevű IDL interfészt (e tényt logikai igaz visszatérési értékkel jelzi).

A `non_existent()` metódussal ellenőrizhetjük, hogy az illető objektum még él-e (vagyis az őt kezelő ORB szoftver tudja-e még hozzá a kliensek metódushívási kérését továbbítani). Ha az illető objektum még elérhető, akkor egy logikai hamis értéket ad vissza, egyébként pedig logikai igaz értékkel jelzi az illető objektum elérhetetlenségét (ahelyett, hogy egy `CORBA::OBJECT_NOT_EXIST` kivételt váltana ki).

A `hash()` metódussal egy olyan objektumreferencia-azonosító alapján számolt hash-függvény értéket kérdezhetünk le, amelynek az értéke nem változik az objektumreferencia

élettartama alatt. A várt visszatérési érték lehetséges intervallumának felső korlátját kell a metódus paraméterében megadni (az alsó határ értéke nulla).

Az `is_equivalent()` metódussal két CORBA objektumreferencia ekvivalenciájáról győződhetünk meg. Az ekvivalencia definíciója szerint két megegyező objektumreferencia ekvivalens. Két különböző objektumreferencia, amely ugyanarra az objektumimplementációra hivatkozik, szintén ekvivalens. Ennek a metódusnak a segítségével lehet elkerülni, hogy egy programon belül különböző objektumreferenciákkal hivatkozzunk azonos CORBA objektumokra.

A `create_request()` metódus szerepével később, a dinamikus metódushívási interfész ismertetésekor foglalkozunk.

Megjegyezzük Java nyelvi vonatkozásként, hogy a `CORBA::Object` osztály fent említett metódusainak a Java nyelvű megfelelői egy aláhúzásjellel kezdődnek; erről a Java IDL implementációnkkal szállított osztályreferencia-kézikönyv megfelelő oldalán olvashatunk bővebben.

2.10. ORB szoftverek együttműködésének megszervezése

A CORBA szabvány 2.0 változatának megjelenése előtt a piacon kapható ORB szoftverek együttműködése komoly problémákat okozott a megfelelő szabvány hiánya miatt. A különféle gyártók ORB szoftverei nem tudták egymás objektumainak metódusait meghívni, mivel az ORB szoftverek közötti kommunikációt minden gyártó másként oldotta meg. A CORBA szabvány 2.0 változatától kezdődően definiáltak két ORB együttműködési protokollt: az általános együttműködési protokollt, valamint a DCE-alapú együttműködési protokollt. Az általános ORB együttműködési protokoll (ez az ún. GIOP protokoll, az angol General Interoperability Protocol kifejezés alapján) az ORB szoftverek összeköttetés-alapú protokollok feletti együttműködési szabályait definiálja (lényegében egyszerű adatcsomag formátumokat definiál a metódushívási kérések továbbítására). A DCE-alapú együttműködési protokoll pedig az OSF/DCE osztott infrastruktúráját, távoli eljáráshívási mechanizmusát használja a CORBA ORB szoftverek együttműködésének megszervezésére (erre a protokollra a DCE/ESIOP néven szoktak hivatkozni). Az ORB implementációk általában az általános együttműködési protokollnak a TCP/IP összeköttetés-alapú transzport protokoll feletti implementációját támogatják, az ún. IIOP protokollt (ez az Internet-alapú ORB együttműködési protokoll). Ez az, amit minden CORBA 2.0-konform ORB szoftvernek ismernie kell. A DCE/ESIOP együttműködési protokoll opcionális, azaz az ORB szoftverekkel ezt nem kell szállítani ahhoz, hogy CORBA 2.0-konformnak nevezhessék őket (és a Java környezetben használható ORB implementációk ezt nem is szokták támogatni - igaz az IIOP mellett ez kicsit feleslegesnek is tűnhet). Az együttműködési protokollok mellett fontos az is, hogy az objektumreferenciák formája is szabványosítva legyen. A

CORBA definiál egy IOR (angol nevén Inter-ORB Reference) nevű objektumreferencia formátumot. Ez az objektumreferenciák ORB szoftverek közötti átadásának formátuma.

2.11. Az IIOP protokoll

Az OMG a CORBA specifikáció GIOP komponensében megtervezett egy nagyon egyszerű ORB együttműködési protokollt, ami inkább egy olyan absztrakt protokollnak tekinthető, amely majd valahogyan le lesz képezve a számítógépes hálózatokban használt transzport protokollok valamelyikére (például a TCP/IP protokollcsalád TCP transzport protokolljára leképezett változat az IIOP protokoll). Mivel a GIOP protokoll elég egyszerű követelményeket támaszt az alatta levő transzport réteggel szemben (lényegében összeköttetés-alapú, megbízható, bájtfolyam jellegű adatátvitelt vár), ezért elég széles azoknak a transzport protokolloknak a köre, amelyekre a GIOP leképezhető. Mi most röviden - a teljesség igénye nélkül - áttekintjük az IIOP protokoll néhány fontosabb jellemzőjét, amelynek segítségével jobban megérthetjük, a CORBA ORB szoftverek együttműködésének lehetőségeit és problémáit (a protokoll specifikációjának ismeretére a CORBA alkalmazások készítőinek nincs szükségük, mivel a CORBA 2.0 specifikáció alapján elkészített ORB szoftverek az IIOP protokoll értelmezésére és használatára már jól fel vannak készítve).

Fontos kérdés az ORB szoftverek együttműködésének megszervezésekor, hogy milyen módon lehet az elérhető CORBA objektumokat megcímezni, milyenek is legyenek (mit tartalmazzanak) a rájuk hivatkozó objektum referenciák (megjegyezzük, hogy az alkalmazói programok készítői felé az objektumreferenciák átlátszatlan konstrukciókként jelennek meg, tehát nem férnek hozzá a belsejében tárolt információkhoz). A CORBA specifikáció egy objektumreferencia általános felépítését a következőképpen definiálja (természetesen ezt is IDL nyelven):

```
module IOP {
  typedef unsigned long ProfileId;
  const ProfileId TAG_INTERNET_IOP = 0;
  const ProfileId TAG_MULTIPLE_COMPONENTS = 1;

  struct TaggedProfile {
    ProfileId tag;
    sequence<octet> profile_data;
   };

  struct IOR {  // Ez itt az IOR referencia specifikációja
    string type_id;
    sequence<TaggedProfile> profiles;
  };

  // Egyes elérhető ORB szolgáltatások leírására használt
  // MultiComponentProfile adatszerkezetek specifikációját kihagytuk innen.
```

```
};
```

Az objektumreferenciák szerkezetét az IOR struktúra definiálja. Ennek a `type_id` komponense tárolja az illető objektum pontos típusának a megnevezését (itt lényegében az interfészgyűjteménybeli azonosításra alkalmas névről van szó), valamint a `profile_data` komponensben tárolja az illető objektum azonosítására alkalmas referenciákat (ennek típusát fenti láthatjuk: `TaggedProfile`). Az objektum referenciák együttműködési protokollonként egy `TaggedProfile` adatszerkezetnek egy-egy elemét tartalmazzák (ui. ennyiféleképpen kell, és lehet az illető objektumot megcímezni, elérni, hiszen az objektum elérhetőségét bármelyik együttműködési protokollal lehetővé kell tenni). Egy ilyen elem - mint fent látható - két komponenst tartalmaz: egy tag nevű azonosítót, valamint egy `profile_data` komponenst, amelynek a belső szerkezete itt nincs rögzítve (egyszerűen egy oktettsorozat, azaz bájtsorozat). Az előbbi komponens tárolja, hogy az illető `TaggedProfile` komponens `profile_data` eleme mely együttműködési protokoll számára tárolja az objektumazonosításhoz szükséges információkat. Az IIOP protokollon keresztül elérhető objektumreferenciák esetében a `tag` komponens a fent definiált `TAG_INTERNET_IOP` konstans értéket tartalmazza, és a `profile_data` komponens tényleges belső szerkezetét a következő IDL specifikáció tartalmazza (az ezt leíró `IIOP::ProfileBody` egyes komponenseinek ismertetését a lista után tesszük meg).

```
module IIOP {
  struct Version { // Az aktuális (CORBA 2.0 spec.) IIOP protokolljának
    char major;    // verziószámának "fő" azonosítója: 1
    char minor;    // "al" azonosítója: 0
   };

  struct ProfileBody {
    Version iiop_version;
    string host;
    unsigned short port;
    sequence<octet> object_key;
  };
};
```

Az IIOP referenciák az illető objektum elérhetőségének a következő jellemzőit rögzítik:

`iiop_version` : az illető objektum kezeléséért felelős ORB szerverszoftver által használt IIOP protokoll verziószáma. A szerkezetét lásd feljebb, az `IIOP::Version` IDL specifikációjánál; megjegyezzük, hogy a CORBA 2.0 specifikációban specifikált IIOP protokoll változat az 1.0. Továbbá megjegyezzük, hogy ezen IIOP változat azonosító a későbbiek során jelezheti magának a címnek az értelmezésére vonatkozó szabályokat is (hiszen idővel ez is változhat).

`host` : az illető objektum kezeléséért felelős ORB szerverszoftvert futtató számítógép Internet-címe.

`port` : az illető objektum - vagy az őt kezelő ORB szoftver - eléréséhez használandó TCP-port azonosítója.

`object_key` : az illető objektum egyértelmű azonosítására alkalmas kulcs (belső szerkezete számunkra nem érdekes, és nincs is rá igazán szabvány, mivel az ezt létrehozó ORB szoftveren kívül másnak ezt úgysem kell értelmeznie - vagyis ezt az azonosítót az ORB szoftver generálja, és adhatja tovább a többi ORB szoftvernek).

A GIOP - és ezzel együtt az IIOP - protokoll alapú kommunikáció során elküldött üzenetek formátumát szintén a CORBA 2.0 specifikáció rögzíti (az üzenetekben használt adatábrázolási módok pontos rögzítésével). A GIOP által definiált üzenetek a következő hét csoport valamelyikébe tartoznak:

- Kérés egy objektum egy metódusának meghívására (GIOP/IIOP kódja: 0).
- A metódushívással kapcsolatos válaszadatok (GIOP/IIOP kódja: 1).
- Metódushívási kérés semmissé tétele (visszavonása) (GIOP/IIOP kódja: 2).
- A kliens lekérdezheti a szervertől egy adott IOR által azonosított objektum állapotát (vagyis például a szerver képes-e az illető objektumra vonatkozó kérések kiszolgálására, illetve ha az illető objektum elérésére más címre kell a kéréseket küldeni, akkor hol van az a bizonyos más cím) (GIOP/IIOP kódja: 3).
- Az előző pontban (3 GIOP/IIOP kódú) megnevezett kérésekre küldött válaszüzenetek (GIOP/IIOP kódja: 4).
- A szerver jelezheti kliensének, hogy le akarja zárni a klienssel felépített összeköttetést, így az ne várjon további üzenetekre (GIOP/IIOP kódja: 5). A szerver ezt a műveletet akkor használja, ha rendszererőforrások hiányában nem tud új összeköttetést létrehozni, ezért kénytelen lezárni egy még élő összeköttetést.
- A partner által megfogalmazott GIOP/IIOP üzenet formailag hibás (GIOP/IIOP kódja: 6).

A GIOP/IIOP üzenetek fejlécének szerkezetét a következő IDL specifikáció rögzíti (a CORBA szabványban maguknak az üzeneteknek a formátuma is specifikálva van, de ebben a könyvben ezzel nem foglalkozunk, mivel nem kapcsolódik a könyv témájához, ami a hálózati alkalmazások készítésének alkalmazói programozói szemszögéből használható fontosabb eszközeinek és azok hátterének az ismertetése):

```
module GIOP {
  enum MsgType { ... }; // kihagytuk, de itt a fenti GIOP/IIOP típuskódok
                        // vannak definiálva, egy-egy névvel ellátva.
  struct MessageHeader {
    char magic[4];
```

```
    Version GIOP_version;
    boolean byte_order;
    octet message_type;
    unsigned long message_size;
  };
};
```

A fent definiált `GIOP::MessageHeader` struktúra komponensei a következők:

`magic` : a GIOP szabvány alapján összeállított üzenetek azonosítója (értéke a GIOP karakterek sorozata).

`GIOP_Version` : az üzenet összeállítására használt GIOP szabvány verziószáma (jelenleg ez 1.0).

`byte_order` : az üzenet felépítésére használt bájtsorrendet specifikálja. FALSE érték "felsővég", TRUE érték "alsóvég" bájtsorrendet jelzi.

`message_type` : a GIOP üzenet típuskódja (a fenti hét típuskód valamelyike).

`message_size` : a fejlécet követő üzenet törzsének hossza oktettekben (8 bites egységekben).

Az IIOP protokoll felhasználói valószínűleg nem kizárólag a CORBA ORB szoftverek lesznek. Sokan arra számítanak, hogy a WWW-n elérhető - elsősorban a szerver feladatokat ellátó - erőforrások végső formájukban az IIOP protokollba lesznek "becsomagolva" (mármint IIOP protokollal lesznek elérhetőek), és ezzel együtt az IIOP protokoll hasonló mértékben elterjedhet, mint a WWW mai alapvető infrastruktúráját biztosító HTTP protokoll (e két protokoll alapfeladatai között sok hasonlóságot fedezhetünk fel[5], és látható, hogy az IIOP egy olyan általános protokoll, amely el tudja látni a HTTP protokoll alapú kommunikáció során felmerült kliens/szerver igényeket is). Az IIOP-alapú WWW környezetre már ma is gyakran az objektum-alapú WWW fogalommal hivatkoznak.

Végül ismertetjük az IIOP protokollal elérhető erőforrások elérési helyének leírására gyakran használt URL sémát. Ez a séma nincs RFC dokumentumokban szabványosítva, elterjedtsége indokolja azt, hogy mi is megismerjük.

```
iiop:1.0//<számítógép neve>:<port>/<objektumazonosító_kulcs>
```

A sémaazonosító az `iiop` szöveg, amit az IIOP protokoll verziószáma követ (esetünkben ez `1.0`). Az URL másik három komponensei rendre a következők:

- Az elérni kívánt CORBA objektumot tároló számítógép neve.
- Az elérni kívánt CORBA objektumot kezelő szerver (objektumadapter) TCP-portjának az azonosítója.

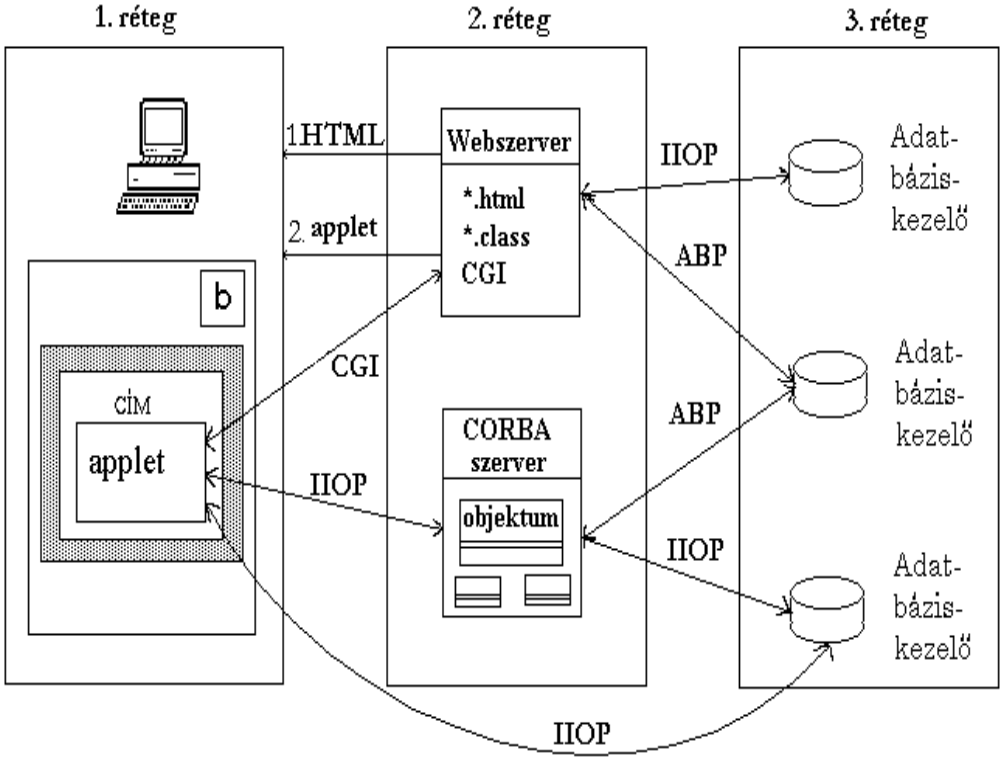
[5] Az IIOP sok tekintetben hasonlít a HTTP-hez, de míg a HTTP-t elsősorban tartalomletöltésre, addig az IIOP-t inkább szolgáltatások elérésére szokták használni.

- Az elérni kívánt CORBA objektum egyértelmű azonosítására alkalmas azonosító kulcs.

2.12. Az objektum-alapú WWW elemei

Az IIOP protokoll áttekintését tartalmazó részben már említettük az objektum-alapú WWW kifejezést, mint a korábbi HTTP protokoll alapú WWW továbbfejlesztésének egy lehetséges alternatíváját. Ebben a modellben a webről letöltött alkalmazások a hagyományos CGI-programok mellett bizonyos szolgáltatásokat nyújtó CORBA objektumokat is elérhetnek feladatuk elvégzése érdekében. Az ehhez szükséges ORB környezet beépíthető akár a kliens webböngészőjébe, de szükség esetén - például Java appletek esetén - az egész ORB letölthető futásidőben a Java szokásos osztályletöltési mechanizmusaival például az illető applet letöltési helyéről.

Az alábbi ábra szemlélteti az objektum-alapú WWW alkalmazásaiban tipikusan előforduló kliens/szerver interakciókat a megvalósításukhoz felhasznált protokollokkal együtt.



A fenti ábrán jól látható módon feltüntettük a háromrétegű alkalmazásmodell egyes rétegeit.

Az 1. réteg a kliensoldali felhasználói felületet foglalja magába - az ábra ennek egyik leggyakoribb megtestesítőjét, egy böngészőprogramot szemléltet. A webböngésző pro-

gramok a kliensoldali felhasználói felületek megvalósítására legalább két utat nyújtanak: az egyik megoldás alapjának a HTML űrlapokat tekinthetjük, míg a másik megoldás alapját a Java appletek képezik (itt a felhasználói felület a Java `java.awt` csomagjának lehetőségeire épül). A HTML űrlapokon alapuló megoldások előnye a felhasználói felület elkészítésének egyszerűsége és hordozhatósága (a programozónak nem kell alacsony szintű, grafikus felhasználói felületek készítésére használható eszközökkel bajlódnia, a hordozhatóságnak az alapja pedig az, hogy a HTML űrlapok mára már gyakorlatilag bármilyen webböngésző környezetben használhatók), hátrányuk viszont az, hogy a webböngésző lehetőségei komoly korlátokat szabnak a felhasználói felülettel szemben (az ilyen megoldásoknál sok olyan felhasználói felület alkotóelemről le kell mondani, amelyet más grafikus környezetekben mind a felhasználók, mind pedig a programozók már megszoktak és megkedveltek). A Java appleteken alapuló megoldás ezzel szemben sokkal összetettebb, elkészítéséhez sokkal több idő és ismeret szükséges a programozók részéről, de itt gyakorlatilag alig vannak a felhasználói felület szerkezetével szemben támasztott korlátok (vagy legalábbis lényegesen több jól bevált és megszokott felhasználói felület alkotóelem áll a programozó rendelkezésére[6]). A Java appletes megoldás komoly követelményeket támaszt a kliensen futó programkörnyezettel szemben abban a tekintetben, hogy a webböngésző programot itt ki kell egészíteni egy Java appletek végrehajtására képes Java virtuális géppel is, ami viszont lényegesen költségesebb az egyszerű webböngésző programok erőforrásigényeihez viszonyítva.

A második réteget a különféle alkalmazói logikák alkotják, a webszervereken keresztül elérhető különféle CGI-programok és az üzleti munkafolyamatokat modellező CORBA objektumok.

A harmadik réteg legjellemzőbb képviselői a relációs és objektum-orientált adatbáziskezelő programok, amely a világ adatainak óriási mennyiségét tárolják (ezek az adatbázisok vállalatok adatait mára már több évtizedre visszamenőleg tartalmazzák).

Az ábrán jelöltük az egyes rétegek közötti tipikus interakciókat. Első lépésként általában az történik, hogy a felhasználó elindítja a böngészőprogramot, és egy HTTP szerverről letölti azt a HTML lapot, amelyre szüksége van. Ez a HTML lap tartalmazhat HTML űrlapokat, illetve tartalmazhat hivatkozásokat Java appletekre, amelyeket a böngésző szintén letölt a HTTP szerverről, és a böngésző végül elindítja őket.

Az első és a második réteg közötti kapcsolat HTML űrlapok alkalmazása esetén elég egysíkú: a webszerveren található CGI-programokon keresztül történik (HTTP protokoll felett). Java appletek azonban nemcsak a CGI-programokat érhetik el, hanem a böngészőbe épített vagy az applettel együtt letöltött ORB segítségével elérhetik az applet

[6] Megjegyezzük, hogy a Java appletekben elérhető `java.awt` csomagot sok kritika éri egyfelől azért, mert sokak szerint még így is kevés felhasználói felület alkotóelemet biztosít, és a meglevő felhasználói felület alkotóelemek sincsenek igazán esztétikusan megtervezve. Az értékeléskor azonban figyelembe kell venni, hogy az `awt` eszközeivel már lehetőség van akár újabb alkotóelemek megírására is (bár egy összetettebb felhasználói felület alkotóelem elkészítése sok időt és programozói tudást igényel), és maguk a Java nyelv fejlesztői is dolgoznak egy ilyen osztálykönyvtáron, a JFC-n (ismertebb nevén Swing eszközkészleten), amely úgy javítja ki a `java.awt` csomag legkritizáltabb részeit, hogy közben nem veszít annak hordozhatóságából.

letöltési helyén elérhető összes CORBA objektumot is (az IIOP protokollal).

A második és a harmadik réteg közötti kapcsolat gyakorlatilag bármilyen protokollal megszervezhető, akár az IIOP protokollal, akár valamilyen szabványos vagy gyártófüggő adatbázis elérési protokollal[7] (ilyenek például a JDBC, ODBC, SQL*Net, stb. protokollok).

Külön felhívjuk a figyelmet arra - az ábrán is szemléltetett - tényre, hogy a Java appletek az ORB szoftverek segítségével a világ bármelyik CORBA objektumát elérhetik, ha a hálózati tűzfalak és a Java környezet biztonsági mechanizmusai ezt nem akadályozzák meg. Mivel a piac vezető adatbáziskezelőgyártói közül egyre többen döntenek úgy, hogy közvetlenül az IIOP protokoll segítségével is lehetővé teszik az adatbázisukban tárolt adatok elérését[8], ezért a piaci elemzők az IIOP protokoll és a CORBA rohamos elterjedésével számolnak (ami egyébként már ma is igen elterjedt, bár még inkább vállalati intranet hálózatokban jellemző).

2.13. A CORBA objektumszolgáltatásai

Az objektumszolgáltatások olyan általános célú szolgáltatások, amelyekre a CORBA-alapú alkalmazások készítőinek nap mint nap szükségük lehet. Ezek a szolgáltatások megkönnyítik a programozó munkáját azzal, hogy nem kell állandóan újraimplementálni az infrastruktúrának ezeket a részeit. Az objektumszolgáltatásokkal - a névszolgáltatás kivételével - nem foglalkozunk részletesebben. Itt csak egy felsorolás erejéig összefoglaljuk az eddig szabványosított objektumszolgáltatások neveit és az egyes szolgáltatások feladatát.

Névszolgáltatás : lehetővé teszi, hogy neveket rendeljünk CORBA objektumok referenciáihoz, és lehetővé teszi CORBA objektumok számára a nevekhez rendelt CORBA objektumreferenciák visszakeresését. A névszolgáltatás támogatja egy hierarchikus névstruktúra kialakítását, és lehetővé teszi különféle szabványos névszolgáltatási eszközöknek a névhierarchiába való integrálását is.

Perzisztencia : lehetővé teszi CORBA objektumok élettartamának függetlenné tételét az őket tároló szerver, illetve az őket elérő kliens alkalmazások élettartamától. Ehhez olyan interfészeket definiál, amelyeken keresztül egy objektum eltárolhatja az állapotát fájlokba vagy adatbázisokba (akár relációs adatbázisokba, akár objektumorientált adatbázisokba).

Eseményszolgáltatás : lehetővé teszi ún. hírcsatornák (eseménycsarornák) létrehozását, és biztosítja CORBA objektumok számára a hírcsatornára hírek (ún.

[7] Az ábrán ezt jelöltük ABP-vel.

[8] Megjegyezzük, hogy eddig is sok gyártó lehetővé tette adatbázisainak CGI- programokon keresztül történő elérését, de a megoldások gyártófüggő mivolta miatt ezek kevésbé terjedhettek el, és a CGI-programok közismerten gyenge hatékonysága szintén komoly gátat szabott elterjedésüknek.

események) küldésének a lehetőségét, valamint lehetővé teszi a hírcsatornákra küldött események fogadását is. A CORBA objektumok szabadon csatlakozhatnak a hírcsatornákra, ha tudják a csatorna nevét. Egy csatornának tetszőleges számú eseményküldő és eseményfogadó CORBA objektuma lehet, azaz nincsenek erre vonatkozó elvi korlátok. A csatornákon a kommunikáció lehet eseménytermelő-orientált vagy igény szerinti attól függően, hogy egy esemény küldését az eseményt küldő vagy az eseményt fogadó objektumnak kell kezdeményeznie.

Életciklus szolgáltatás : olyan műveleteket definiál, melyeket egy CORBA objektum készítőjének implementálnia kell ahhoz, hogy az objektumok könnyen átköltöztethetők legyenek a hálózat egyik komponenséről a másikra. Ez a szolgáltatás nemcsak objektumköltöztetést segítő metódusokból áll, hanem biztosít egy konstruktor jellegű mechanizmust, valamint biztosítja objektumok megszűntetésének a lehetőségét is.

Tranzakciókezelési szolgáltatás : lehetővé teszi, hogy tranzakciókat alakítsunk ki CORBA objektumok metódusainak hívásából. A tranzakció véglegesítésekor vagy visszavonásakor egy elosztott kétfázisú megegyezési protokoll szerint képes elvégezni a tranzakció befejezésekor felmerülő koordinálási teendőket.

Szinkronizációs szolgáltatás : párhuzamosan működő elosztott tranzakciók számára egy elosztott szinkronizációs mechanizmust biztosít (persze nemcsak tranzakciókba szervezett műveletek használhatják ezt a szolgáltatást).

Externalizációs szolgáltatás : lehetővé teszi CORBA objektumok belső állapotának kimentését, és visszaolvasását valamilyen adatcsatorna jellegű tárolóról.

Objektumkapcsolat szolgáltatás : lehetővé teszi tetszőleges objektumok közötti asszociációk kialakítását, például hivatkozási épség ellenőrzésének a biztosítására.

Objektumlekérdezési szolgáltatás : lehetővé teszi ún. lekérdezések definiálását és kiértékelését. A lekérdezések megfogalmazására egy SQL-szerű nyelv használható. Egy lekérdezés általában egy vagy több objektumot ad vissza eredményül.

Felhasználási jogosultság ellenőrzését támogató szolgáltatás : lehetővé teszi a rendszerben levő CORBA objektumok felhasználásának anyagi kompenzálásának megszervezését.

Objektumjellemzők kezelését végző szolgáltatás : lehetővé teszi a CORBA objektumokhoz névvel ellátott értékek hozzárendelését és lekérdezését.

Időszolgáltatás : lehetővé teszi egy elosztott CORBA környezetben a gépek óráinak a szinkronizálását. Biztosítja más CORBA objektumok számára, hogy bizonyos idő elteltével például egy metódusuk meghívásán keresztül jelzi az idő múlását. Visszahívás kérhető egy előre megadott időpontra, vagy megadott időközönként.

Objektumreferencia-kereskedési szolgáltatás : lehetővé teszi bizonyos szolgáltatásokat nyújtó objektumok megkeresését. Az objektum-alapú WWW-n ezek a ma is gyakran használt webkereső-programok (pl. AltaVista, Yahoo) megfelelői. A CORBA objektumként implementált szolgáltatók bejegyezhetik, hogy milyen szolgáltatásokat nyújtanak, míg a kliensek kikereshetnek egy számukra megfelelő szolgáltatót. Például a bankok bejegyezhetik lakossági folyószámla szolgáltatásukat támogató CORBA objektumaikat olyan információkkal együtt, hogy milyen kamatokat adnak a számlán levő pénzre, a számlát nyitni szándékozó ügyfél pedig kikerestetheti az igényeinek legjobban megfelelő folyószámla szolgáltatást biztosító bankot (például a legmagasabb kamat alapján).

Biztonsági szolgáltatás : a felhasználója felé teljesen átlátszó módon biztosítja a CORBA metódushívások során cserélt adatok titkosítását, a metódushívást kezdeményező kliens és a szerver azonosítását és igazolását például nyilvános kulcsú titkosítási algoritmusokkal.

2.14. A Current objektum

A metódushívások során az IIOP protokoll lehetővé teszi, hogy az ORB az aktuális programszál futási környezetéről bizonyos információkat átvigyen a hívott metódushoz. Az ilyen információkat az ORB a `CORBA::Current` interfészből származtatott objektumokban tárolja, és lehetőséget biztosít arra, hogy az alkalmazások az ezekben az objektumokban tárolt információkat lekérdezzék és módosítsák (mind a hívás helyén, mind pedig a meghívott metódusban).

```
module CORBA {
  interface Current { // az ősinterfész üres
  };
};
```

Magának az ORB-nek a működéséhez általában nincs szükség ilyen programszál-specifikus információkra, de a CORBA néhány objektumszolgáltatásához ez nagyon hasznos segédeszköz lehet. Például a CORBA tranzakciókezelési objektumszolgáltatása egy ilyen objektumban tárolja azt, hogy az éppen aktív programszál melyik tranzakcióban vesz részt, így ez az információ a metódushívásokkal együtt automatikusan továbbítva lesz: a metódusoknak nem kell egy újabb paramétert definiálni ahhoz, hogy megkaphassák ezt az információt[9].

Ezzel kapcsolatban az egyes CORBA objektumszolgáltatások specifikációjában találhatunk részletesebb információkat.

[9]Emlékezzünk rá, hogy a távoli metódushívásnál ismertetett tranzakciókezelő példaprogramban a tranzakciószerűen működő metódusoknak fel kellett venni egy olyan paramétert, amelyben annak a tranzakciónak az azonosítóját kellett átadni, amely tranzakció keretében az illető metódust végre akarjuk hajtani. A CORBA így ezt a gondot is képes levenni a programozó válláról.

2.15. A CORBA névszolgáltatása

Az OMA architektúra ismertetésekor már említettük az objektumszolgáltatások komponensét, és a névszolgáltató szerepét. Ez lényegében egy OMG IDL nyelven specifikált hierarchikus - így a hálózaton akár hierarchiaszintenként is jól elosztható - névszolgáltató, amely lehetővé teszi szöveges nevekhez CORBA objektumok (objektumreferenciák) asszociálását. A nevek hierarchiája hasonlít a hierarchikus fájlrendszerek struktúrájához: egy ún. kontextusban nevekhez asszociált CORBA objektumreferenciákat, illetve további - a kontextushierarchiában alárendelt - kontextusok referenciáit tárolják (itt a kontextus lényegében a hierarchikus fájlrendszerben megismert könyvtár/katalógus (angol nevén directory) fogalmának a megfelelője).

A névszolgáltatás lehetőséget nyújt egy többkomponensből álló név értelmezésére egy megadott kontextusban (ezzel az alkalmazások visszakaphatják az illető névhez asszociált CORBA objektum referenciáját), illetve egy megadott kontextusba egy újabb névhez egy CORBA objektumreferencia hozzárendelésére. A fájlrendszer analógiára támaszkodva azt mondhatjuk, hogy a CORBA névszolgáltató esetében nincsenek "abszolút" nevek, a nevek értelmezése mindig egy programban megadott kontextushoz viszonyítva kezdődik. Maguk a nevek egy vagy több komponensből állnak: az utolsó komponens kivételével az egyes komponensek egy-egy alárendelt kontextust neveznek meg, és csak az utolsó komponens azonosít egy CORBA objektumreferenciát. A CORBA névszolgáltató specifikációja a nevek jelölésére a következő konvenciót használja:

```
< első komponens ; második komponens ; ... ; utolsó komponens >
```

Az előbbi példában a pontosvessző karakterek szolgáltak a név egyes komponenseinek elválasztására, de ez nem jelenti azt, hogy a neveknek ténylegesen így kellene kinézniük: a CORBA névszolgáltatóval kapcsolatot tartó szoftver saját konvenciókat alakíthat ki arra vonatkozóan, hogy milyen szintaktikai szabályok szerint kell a neveket összeállítani, a névszolgáltató tényleges elérését úgyis a névszolgáltatót reprezentáló objektum OMG IDL nyelven specifikált interfészén keresztül végzi (aminél, mint azt majd látni fogjuk, nincs szükség ilyen elválasztó szimbólumra, mivel az egyes névkomponenseket külön objektumokban kell majd megadni – a Java leképezés szerint egy Java vektor elemeiben felsorolva). Maguk a névkomponensek is két részből állnak: egy azonosító és egy minősítő részből. E két rész szerepével kapcsolatban a CORBA névszolgáltatás nem támaszt semmiféle követelményeket, maga a CORBA névszolgáltatás nem foglalkozik a nevek szerkezetének az értelmezésével, de általánosságban - ismét a fájlrendszer analógiára támaszkodva - azt mondhatjuk, hogy a nevek minősítő része a legtöbb alkalmazásban hasonló szerepet tölthet be, mint a fájlnevek esetében a kiterjesztések (amit például az MS-DOS operációs rendszerben a fájlnévben levő pont karakter után írhatunk). Minősítő azonosítók lehetnek például a `c_nyelven_írt_forráskód`, `java_forráskód`, `futtatható_tárgyprogram`, de használhatunk tetszőleges alkalmazás-specifikus minősítőket is (a CORBA ezzel szemben sem

támaszt követelményeket).

A CORBA-alapú kliens/szerver alkalmazások elindulásuk után az ORB szoftveren keresztül hozzájuthatnak a helyi (vagy valamelyik távoli) számítógépen futó CORBA névszolgáltató referenciájához, és a továbbiakban ezen keresztül újabb és újabb, nevükkel azonosított objektumok referenciáihoz juthatnak hozzá.

Megjegyezzük, hogy a CORBA névszolgáltatását úgy tervezték, hogy könnyen be lehessen ágyazni ebbe a szolgáltatásba más névszolgáltatókat is. Például elkészíthetünk egy olyan Internet DNS névszolgáltatót, amely a CORBA névszolgáltató interfészen keresztül szolgáltatja a számítógépeket valamilyen szempontból reprezentáló Internetcím objektumokat.

A CORBA névszolgáltatását, annak interfészeit a `CosNaming` modul[10] definiálja. Ennek szerkezete - OMG IDL nyelven - a következő (a teljes IDL fájlt példaértékéért tettük ide, mivel jól láthatók benne a fejezetben már említett OMG IDL nyelvi konstrukciók nagy része; a modul metódusainak rövid leírását szintén a forráskódban levő megjegyzésekben láthatjuk):

```
module CosNaming
{
  typedef string Istring; // ez majd UNICODE vagy más szöveg is lehet ...
  struct NameComponent {  // egy név komponens nevét leíró rekord szerkezet
    Istring id;    // a név azonosító része
    Istring kind;  // a név minősítő része
  };

  typedef sequence <NameComponent> Name; // egy név a fenti komponensek
                                         // nem korlátozott sorozata

  enum BindingType {nobject, ncontext}; // az illető névhez tárolt objektum
                                        // kontextus-e vagy sem?

  struct Binding {              // ez kapcsolja egy adott
    Name         binding_name;  // objektumnévhez azt, hogy az egy
    BindingType  binding_type;  // kontextus vagy felhasználói objektum-e
  };                            // ezt nevezik az illető objektum típusának

  typedef sequence <Binding> BindingList; // neveket és típusukat tárolja

  interface BindingIterator; // előre definiáljuk a nevet, mivel már a
                             // specifikációja előtt hivatkozni akarunk rá
                             // ebben az IDL fájlban (elődeklaráció)

  interface NamingContext {// ez a tulajdonképpeni objektumkatalógus
                           // ebbe jegyezhetők be a (név,objektum)
                           // kapcsolatok, illetve ebből is kérdezhetők le
```

[10] Megjegyezzük, hogy a Java leképezési szabvány és elnevezési konvenciói szerint ezen szabványos CORBA modul az `org.omg.CosNaming` Java csomagba lesz leképezve.

```
  enum NotFoundReason { missing_node, not_context, not_object};
     // a fenti felsorolási típus tartalmazza egy sikertelen lekérdezés
     // sikertelenségének lehetséges okait:
     // missing_node : nincs meg a keresett nevű objektum
     // not_context : a keresést egy kontextusobjektumban kell végezni,
     //               de mi nem ezt adtunk meg paraméterként
     // not_object : a névhez nem objektum asszociációt talált

  exception NotFound {  // lekérdezés sikertelenségét jelző kivétel
    NotFoundReason why; // a sikertelenség oka
    Name rest_of_name;  // a névnek a már nem értelmezett hátulsó része
  };
  exception CannotProceed { // a névhez tartozó objektum keresését a
    NamingContext cxt;      // rendszer nem képes folytatni, de a program
    Name rest_of_name;      // ha akarja, akkor megpróbálhatja folytatni
  };                        // az itt visszaadott kontextusnál, illetve
                            // itt visszaadott névtől kezdve

  exception InvalidName{};   // szabálytalan a megadott név (pl. 0
                             // karakter hosszú)
  exception AlreadyBound {}; // a megadott névhez már rendeltek valamit
  exception NotEmpty{};      // a megszüntetni kívánt kontextus nem üres

  // itt következnek a kontextusokon végezhető műveletek, amelyeknek a
  // leírását későbbre hagyjuk (túl hosszú lenne programkódba illesztve)
  void bind(in Name n, in Object obj)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
  void rebind(in Name n, in Object obj)
    raises(NotFound, CannotProceed, InvalidName);
  void bind_context(in Name n, in NamingContext nc)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
  void rebind_context(in Name n, in NamingContext nc)
    raises(NotFound, CannotProceed, InvalidName);
  Object resolve(in Name n)
    raises(NotFound, CannotProceed, InvalidName);
  void unbind(in Name n)
    raises(NotFound, CannotProceed, InvalidName);
  NamingContext new_context();
  NamingContext bind_new_context(in Name n)
    raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
  void destroy()
    raises(NotEmpty);
  void list (in unsigned long how_many, out BindingList bl,
    out BindingIterator bi);
};

interface BindingIterator { // nevek listáját reprezentálja
                            // lehetőséget ad arra, hogy végigmenjünk
                            // a listában lévő neveken

  boolean next_one(out Binding b);
```

```
    boolean next_n(in unsigned long how_many, out BindingList bl);
    void destroy();
  };
};
```

A `NamingContext` interfész műveleteit (legalábbis azok paraméterezését) a fenti IDL specifikációs fájlban már láthattuk. A `bind()` művelettel jegyezhetünk be egy adott néven egy adott objektumreferenciát a névszolgáltató adatbázisába (azaz kontextusba); a metódus első paraméterében kell megadni a bejegyzendő nevet, a másodikban pedig a bejegyzendő CORBA objektumreferenciát. A `rebind()` művelet hasonló feladatot lát el; a kettő között a különbség az, hogy a `bind()` egy `AlreadyBound` kivételt vált ki, ha a bejegyzendő néven már van egy objektum a nevek adatbázisában, míg a `rebind()` ekkor módosítja a nevek adatbázisát. A `bind_context()` metódus működése szintén hasonlít a `bind()` működésére: egy CORBA objektum helyett egy új kontextust jegyez be a nevek adatbázisába - ezzel létrehozva egy alárendelt kontextust -, és kivételt generál, ha a megadott néven már van bejegyzés. A `rebind_context()` metódus egy kontextust jegyez be a nevek adatbázisába, és nem generál kivételt már létező bejegyzés esetén, hanem az adatbázis tartalmát módosítja. A `bind_new_context()` metódus létrehoz egy új - üres - kontextust, és bejegyzi a paraméterében megadott néven abba a kontextusba, amelyre vonatkozóan az illető műveletet meghívták. A `new_context()` művelet létrehoz egy új üres kontextust, de nem jegyzi be azt a nevek adatbázisába (ezt szükség esetén megtehetjük a `bind_context()` művelettel).

A nevek adatbázisában - pontosabban egy adott kontextusból - a `resolv()` művelettel kereshetünk meg egy adott nevű objektumot. A paraméterben a keresett nevű objektumot kell megadni, a megtalált objektumot a hívó a visszatérési értékben kapja vissza, amit a megfelelő osztályba kell konvertálnia (ui. a metódus `CORBA::Object` osztálybeli objektumként adja vissza).

A `list()` művelettel lehetőség van végigmenni egy kontextus összes objektumának nevén (ez egyfajta katalógus listázási művelet). A metódus első paraméterében kell megadni, hogy a kontextusból hány névre vagyunk kíváncsiak; a metódus visszatérésekor a második paraméterében visszaad egy `BindingList` objektumot, amely tartalmazza (legfeljebb) a kívánt számú objektumnevet, és a harmadik - `BindingIterator` osztálybeli - paraméterében visszaadja a maradék neveket tartalmazó listát, vagy `nil` objektum referenciát[11], ha nincs több eleme a kontextusnak. A visszaadott `BindingIterator` objektum tartalmát a kliens a `next_n()`, illetve a `next_one()` műveletekkel járhatja be: a `next_one()` metódus visszaadja az egyetlen paraméterében a maradék lista következő elemét, illetve logikai hamis értékkel tér vissza, ha nem maradt több elem. A `next_n()` művelet pedig a második paraméterében visszaadja az első paraméterében megadott számú objektumnak a nevét (ha van annyi), és hamis értéket ad vissza, ha nincs elem. Egy végignézett, vagy feleslegessé vált `BindingIterator` objektumot a `destroy()` metódusával szüntethetünk meg.

[11] Ezt a `CORBA::Object` interfész `is_nil()` metódusával ellenőrizhetjük.

A nevek adatbázisából az `unbind()` metódussal törölhetünk egy megadott nevű objektumot (a törölni kívánt objektum nevét a metódus paraméterében kell megadni).

Egy üres kontextus feleslegessé válása esetén a `destroy()` metódusával szüntethető meg. Csak üres kontextust lehet megszüntetni, vagyis egy kontextus megszüntetése előtt a kontextus tartalmát törölni kell.

Az előbb bemutatott műveletek sikertelen végrehajtás esetén kivételt váltanak ki. A kiváltható kivételek és lehetséges okaik a következők:

`NotFound` : Jelzi, hogy a megadott név nincs az adatbázisban.

`CannotProceed` : Jelzi, hogy az implementáció a névhez tartozó objektum keresését nem tudta befejezni valami miatt. A visszaadott kivétel objektum tartalmazza azt a kontextust, ahol a keresés megszakadt, valamint a keresett név maradékát. E kivétel például olyankor lesz generálva, ha a keresést egy olyan távoli számítógépen levő kontextusnál kellene folytatni, amely időlegesen - például egy hálózati hiba miatt - leszakadt a hálózatról.

`InvalidName` : Jelzi, hogy a paraméterben megadott név szabálytalan. A nulla karakter hosszú nevek általában nem megengedettek, de a névszolgáltató-implementációk más korlátozásokat is támaszthatnak a nevekkel szemben.

`AlreadyBound` : a kontextusban a megadott névhez már van objektum tárolva.

2.16. Java IDL, a Java RMI és a "melyiket a három közül" problémája

Olvasónkban - más programozókhoz hasonlóan - jogosan merül fel a kérdés, hogy vajon szükség van-e a Java RMI eszköz mellett a Java IDL-re (vagy éppen fordítva), hiszen mindegyik eszköz objektumok hálózati elosztását támogatja. Ugyanez a kérdés felmerült már sokakban, és a Java technológia fejlesztői a kétkedések eloszlatása végett azt nyilatkozták, hogy nem áll szándékukban egyik technológiától sem megválni, mivel e két technológia nem ugyanazt a fejlesztői kört célozza meg.

A Java RMI objektummodellje jól illeszkedik a Java eredeti objektummodelljéhez, és használatához még egy új - nevezetesen az OMG IDL - nyelvet sem kell a szoftverfejlesztőnek megtanulnia, mivel itt a Java nyelv interfészeit használva kell a távoli objektumok távolról hívható metódusait specifikálni. A Java RMI is ellát bizonyos ORB feladatokat a távoli metódushívások közvetítésével, de ez a feladata is sokkal egyszerűbb annál, amit egy CORBA "kaliberű" ORB szoftvernek ismernie kell: a Java RMI szoftvernek csak a Java objektumok közötti közvetítői szerepre kell koncentrálnia, míg a CORBA ORB szoftvereknek emellett számos más dologgal is kell foglalkozniuk (gondoljunk csak a CORBA által megkövetelt közös hálózati adatábrázolási mód miatt szükséges adatkonverziókra, vagy a CORBA dinamikus lehetőségeire).

Megjegyezzük, hogy a szoftvergyártás nagyjai ezt a problémát úgy akarják megoldani, hogy a CORBA kommunikációs protokollját, az IIOP-t, ki akarják bővíteni úgy, hogy az képes legyen a Java RMI számára szükséges infrastruktúra biztosítására, és a CORBA használatának egyszerűbbé tétele érdekében már több megoldást is kidolgoztak, amelyek közül a legjobban talán a Visigenic Caffeine terméke sikerült. Ez a szoftver lehetővé teszi távoli objektumok Java-alapú implementációját, és biztosítja az IIOP kliens- és szerverccsonkok automatikus generálását az objektumimplementációkat tartalmazó Java bájtkód fájlok alapján (vagyis az OMG IDL ebben a környezetben csak egy alternatívaként merül fel).

A CORBA vagy RMI dilemma másik megoldásaként a JavaBeans komponensmodell kínálkozik. Eszerint a megoldás szerint az alkalmazásokat nem RMI vagy CORBA objektumok összességéből kell felépítenünk, hanem Java komponensekből: mindegy, hogy egy objektum RMI vagy CORBA módon érhető el, mivel ha azt "becsomagoljuk" egy Java komponensbe, akkor az illető objektum interfésze az egységesnek tekintett komponensarchitektúra keretében lesz hozzáférhető. Megjegyezzük, hogy ezzel a módszerrel a problémát egy szinttel feljebb tolták, mivel a komponensarchitektúrák közt egy hasonló verseny figyelhető meg, mint azt az elosztott objektumokon alapuló architektúráknál is láthatjuk (gondoljunk csak a CORBA, DCOM, RMI hármasra). A komponensarchitektúrának is megvan a maga hármasa (vagy akár azt is mondhatjuk, hogy többese). Ezen a piacon jelen van a CORBA alap legkézenfekvőbb komponensarchitektúrája az OpenDoc (CORBA berkekben gyakran hivatkoznak az OpenDoc-alapú ún. DDCF szabványra). A piacon van továbbá a Microsoft OLE/ActiveX technológiája, aminek a Java-alapú környezetekben való elterjedésének a feltételei még nem adottak (vagy legalábbis a technológia helyzete ma még - e sorok szerzője szerint - nem látszik rózsásnak). A harmadik - és a Java programozók számára legkézenfekvőbben alkalmazható - komponensarchitektúra a már említett JavaBeans. Folynak kísérletek abban az irányban is, hogy a CORBA/OpenDoc "bekebelezze" az ActiveX technológiát, illetve integrálja a JavaBeans architektúrát, de e mérkőzés kimenetele még nem egyértelmű (de ez már egy másik könyv története).

Fontos azonban azt is látnunk, hogy a JavaBeans önmagában csak egy komponensmodell, mely lehetővé teheti alkalmazások kényelmes összeszerelését akár valamilyen grafikus alkalmazásépítő eszköz segítségével is. A JavaBeans önmagában nem biztosít egy elosztott infrastruktúrát, mint amilyent például a CORBA biztosít (bár készíthető olyan Java komponens - egy Java bean - amely Java komponens interfészt biztosít valamilyen kommunikációs infrastruktúra elérésére). A legtöbb szoftvergyártó e két modellt valószínűleg úgy fogja ötvözni, hogy az elkészített szoftverkomponensek olyan Java komponensek (bean-ek) lesznek, amelyeket könnyen összeilleszthetünk más komponensekkel grafikus komponensösszeépítő eszközök segítségével (pl. a Bean Development Kit-tel), de az illető Java komponensek a CORBA infrastruktúrán keresztül fognak érintkezni a háttérben található vállalati adatbázisokkal.

Képzeljünk el például egy pénzügyi szolgáltató cég által elkészített WWW-alapú vásárlást lehetővé tevő Java komponenst. Egy ilyen komponenssel szemben a most ismertetett szempontjaink alapján két alapvető elvárást támaszthatunk:

- egyrészt azt, hogy egy pénzügyi tranzakciók lebonyolításában járatlan programozó is könnyen beépíthesse alkalmazásaiba, feltehesse weblapjaira (egyszerűen egy Java bean-ként)
- másrészt pedig azt, hogy szükség esetén képes felvenni a kapcsolatot a vásárlások pénzügyeit intéző bank számítógépével, ami a gyakorlatban azt jelentheti, hogy ismeri vagy legalábbis képes megszerezni a bank által biztosított szolgáltatásokat megvalósító CORBA objektumok referenciáit, valamint képes azokkal CORBA metódushívásokkal kommunikálni.

Visszatérve a címben említett problémára még annyit érdemes hozzátenni, hogy a Java RMI kiválóan megfelel akkor, ha tisztán Java nyelven írt alkalmazások kapcsolatát megteremtő olcsó ORB szoftverre van szükségünk (ami lényegében csak egy CORBA szabványban rögzítettekhez hasonló (!) távoli metódushívási eszközt biztosít). Ha a Java nyelven kívül más nyelven készített hálózati objektumokat is el akarunk érni, vagy a CORBA magasabb szintű szolgáltatásaira is szükségünk van (például a tranzakciókezelésre, amit még kevés ORB gyártó implementált) akkor a Java RMI kiesik a választható alternatívák közül, és marad a CORBA[12].

[12]Hacsak nem mi magunk újraimplementáljuk ezeket a szolgáltatásokat a Java RMI felett.

3. Fejezet

CORBA-alapú programozás Java környezetben

A fejezet hátralévő részében megnézzük, hogy a CORBA egyes komponensei (az IDL nyelv a kliens és a szerver oldalán, az ORB pszeudo-objektumok) hogyan lesznek leképezve a Java nyelvre, azaz hogyan készíthetünk Java nyelven CORBA objektumokat, és hogyan hívhatjuk meg - akár más nyelven elkészített - CORBA objektumok metódusait Javából. Áttekintjük mind a statikus, mind pedig a dinamikus hívási modell biztosításáért felelős Java-elemeket, a hangsúlyt elsősorban a kliensoldali felületre helyezve (persze a szerver oldali felületet is bemutatjuk, és ismetelten felhívjuk a figyelmet arra, hogy a kliens és a szerver szerepek viszonylagosak, és csak egy-egy bizonyos kapcsolat tekintetében nevezhetjük az egyik résztvevőt kliensnek, a másik résztvevőt pedig szervernek).

A piacon számos ORB szoftver található, amelyek - elsősorban a szabványnak a technológia iránti igényhez képest való késése miatt - kicsit különböznek a megvalósított Java nyelvi leképezésben. Igaz ugyan, hogy a szabványosítási folyamat még e sorok írásakor sem fejeződött be, de a szabvány már eléggé ki van dolgozva ahhoz, hogy a Java nyelvi leképezésnek ne valamelyik gyártó által használt dialektusát ismertessük, hanem magát a véglegesként várható szabványt. A piacon kapható ORB szoftverek között látható ilyen jellegű különbségek idővel várhatóan elsimulnak majd[1], amint az egyes gyártók haladnak a szabvány végleges formájának támogatása irányában.

Ma - e sorok írásakor - a piacon kapható, Java alkalmazások írásához használható ORB szoftverek közül az alábbiakat érdemes megemlíteni:

Visibroker : a Visigenic terméke, amit a szabványt követve fejlesztik (bár e sorok írásakor egyes lehetőségeiben a szabvány "előtt" áll). Azt érdemes megemlíteni róla,

[1] Ha ez nem történne meg, akkor valószínűleg elbúcsúzhatnánk a Java programok bájtkód szintű hordozhatóságától, vagy minden Java alkalmazással együtt kellene szállítani azt az ORB szoftvert, amelyhez illesztették.

hogy a Netscape WEB-böngésző programba is beépítették ezt az ORB szoftvert, így emiatt is várható a komolyabb elterjedése.

OrbixWEB : az IONA terméke, amit a szabványt szorosan követve fejlesztenek.

JOE : a JavaSoft ORB szoftvere, így várhatóan minden Java virtuális gépnek része lesz ez a termék (a neve a Java Object Environment angol szavakból származik, ami magyarul nagyjából annyit jelent, hogy Java-alapú objektumkörnyezet).

CORBAplus/Java : az Expersoft terméke; e sorok írásakor még néhány elem hiányzik belőle, de e hiányosságai várhatóan hamarosan megszűnnek.

A legtöbb ORB szoftver támogatja a CORBA névszolgáltatást is, a többi CORBA objektumszolgáltatás implementációja még nem tekinthető az ORB szoftverekkel általánosan szállított komponensnek.

Itt említünk meg egy másik eszközt is, ami igaz, hogy nem követi a CORBA szabványt (a leírásokból kitűnik, hogy a szerzőnek ez nem is volt igazán szándéka), de egy ahhoz sok tekintetben hasonló eszköz: a HORB. Ez egy - oktatási és kutatási célokra - szabadon elérhető, ORB feladatokat ellátó szoftverimplementáció. Mivel a teljes szoftver forráskódjával együtt elérhető, ezért a kutatási célokra jól módosítható Java ORB implementációt keresőknek valószínűleg érdemes megtekinteniük a szoftver készítője, Hirano Satoshi által írt WEB-lapot a `http://ring.etl.go.jp/openlab/horb/` címen (ez az URL cím e sorok írásakor biztosan élt, várhatóan a továbbiakban is élni fog, de probléma esetén a WEB-kereső programok segítségével biztosan megtalálhatjuk a projekt esetleges új helyének a címét).

3.1. Az OMG IDL leképezése Java nyelvre

Az IDL fájlokban szereplő azonosítók, nevek általában változtatás nélkül lesznek a megfelelő Java nyelvi objektumok neveként felhasználva. Amennyiben a generált Java kódban valamilyen névütközés fordulna elő, akkor a Java névre történő leképezés során előállt név elé egy aláhúzás karakter kerül. Egyes nevek fenn vannak tartva a Java kódot előállító IDL fordító számára (például a legtöbb IDL objektumhoz generálva lesz egy - később bemutatásra kerülő - ún. holder osztály, amelynek a nevét a rendszer úgy képezi, hogy az illető objektum (vagy osztály) Java nevéhez hozzáfűzi a `Holder` szócskát; a `Holder` mellett vannak `Helper`, illetve `Package` szócskára végződő automatikusan generált nevek). Amennyiben egy felhasználó által definiált IDL adattípus nevéből kapott Java azonosítónak a neve ütközhetne egy másik, az előbbi módon automatikusan generált, Java azonosítóval, akkor az IDL fájlban az esetleges ütközéseket figyelmen kívül hagyva megválasztott azonosító Java képe az említett aláhúzás karakterrel kezdődik.

A Java kulcsszavai sem használhatók azonosítóként, így ezek elé is ki lesz rakva szükség esetén egy aláhúzás karakter (persze ha az illető Java kulcsszó szabálytalan

IDL azonosító, akkor nincs probléma, mert azt az IDL-ről Java nyelvre fordító programnak úgysem kell tudnia lefordítani, hanem elég egy erre vonatkozó hibaüzenetet adnia a programozónak).

A CORBA modulok Java csomagokra lesznek leképezve; egy IDL modulból generált Java csomag neve megegyezik az IDL fájlban megjelölt CORBA modul nevével. Tekintsük például a következő IDL moduldefiníciót:

```
module Próbamodul {
  ...
};
```

A fenti modulból generált Java forrásfájl a

```
package Próbamodul;
...
```

sorral kezdődik, és ezt követik az IDL modul törzsében definiált elemek Java megfelelői.

Az egymásba ágyazott OMG IDL modulok egymásba ágyazott Java csomagokra lesznek lefordítva.

```
module m1 {
  module m2 {
    ...
  };
};
```

Például a fenti konstrukcióból a következő Java csomagot kapjuk:

```
package m1.m2;
...
```

A CORBA interfészek egy azonos nevű Java interfészre lesznek leképezve. Erre tekintsük a következő példát:

```
interface MenthetőDolgok { ... };
```

A fenti interfészből generált Java konstrukció a következő:

```
public interface MenthetőDolgok extends org.omg.CORBA.Object { ... };
```

Megjegyezzük, hogy ezekhez generálva lesznek egy-egy ún. holder és helper segédosztályok is, amelyek szerepét a paraméterátadási módok leképezéséről szóló résznél fogunk ismertetni.

Az interfészek adattagjaihoz egy-egy értéklekérdező, illetve egy értékbeállító metódus lesz generálva: a CORBA objektumok kliensei e metódusok segítségével érhetik el az

objektum megfelelő adattagjának értékét. E két metódus neve megegyezik, de szignatúrájuk eltérő: a lekérdező metódus a lekérdezett értéket visszatérési értékeként adja vissza, míg a beállítandó értéket az azt beállító metódus a paraméterében várja. Az IDL specifikációban `readonly` módosítóval ellátott adattagokhoz - más nyelvi leképezésekhez hasonlóan - csak lekérdező metódusok lesznek generálva.

Egy IDL interfészből generált Java interfész természetes módon örököl az illető IDL interfész szülő interfészeiből generált Java interfészektől.

Az IDL fájlban levő konstansok közül azok, amelyek nem IDL interfészeken belül vannak deklarálva olyan `public` módosítóval ellátott Java interfészekre lesznek leképezve, amelyeknek egyetlen `value` nevű, `static` és `final` módosítóval ellátott osztályváltozója van, amely a definiált konstans értéket tárolja. Tekintsük a következő példát:

```
const long EgészÉrték= 32765;
const boolean Igaz = TRUE;
const char Rbetű = 'R' ;
const unsigned short Kicsi = 4333;
const double PI = 3.14159265;
```

Ebből a következő interfészek lesznek létrehozva:

```
public interface EgészÉrték {
  public final static int value = (int)32765;
}

public interface Igaz {
  public final static boolean value = true;
}

public interface Rbetű {
  public final static char value = 'R';
}

public interface Kicsi {
  public final static short value = (short)4333;
}

public interface PI {
  public final static double value = (double)3.14159265;
}
```

A Java nyelvi leképezés rögzíti azt is, hogy hogyan használhatjuk ezeket a konstansokat: `EgészÉrték.value` módon hivatkozhatunk rájuk a programból.

Az IDL interfészekben definiált konstansok az interfész nevével megegyező nevű Java interfészben lesznek létrehozva (ld. az interfészek leképezéséről szóló 3.1. pontot is). A Java interfészben a konstanst reprezentáló adattag neve megegyezik a konstans

IDL nyelvű definíciójában megadott névvel, de az illető adattag `public static final` módosítóval van ellátva.

Tekintsük a következő példát:

```
interface IFACE {
  const long EgészÉrték= 32765;

  ... // további elemek

}
```

Ebből a következő Java interfész lesz létrehozva:

```
public interface IFACE {
  public static final int EgészÉrték = (int)32765;

  ... // további elemek képei

}
```

Az OMG IDL elemi adattípusok a következő táblázat szerint lesznek Java adattípusokra leképezve:

Az OMG IDL típusa	A generált Java objektum típusa
boolean	boolean
char, wchar	char
octet	byte
string, wstring	java.lang.String
short, unsigned short	short
long, unsigned long	int
long long, unsigned long long	long
float	float
double	double

Megjegyzések:

- Az OMG IDL logikai `boolean` típus TRUE és FALSE (logikai igaz, illetve hamis) konstansai a Java nyelv `true`, illetve `false` konstansaira lesznek leképezve.
- Az OMG IDL karakteres `char` típusa a nyolcbites ISO 8859-1 (Latin-1) karakterkészlet elemeit tárolhatja, míg a Java nyelv ilyen típusa a 16 bit hosszú UNICODE karakterkészlet elemeit tárolhatja. Amennyiben egy nem Latin-1-es karaktert akarnánk egy CORBA metódus paraméterében átadni, akkor az ORB egy `CORBA::DATA_CONVERSION` kivételt generál.
- Az `octet` IDL típus egy nyolcbites értéket tárolhat, így érthető, hogy a képe a Java `byte` típusba tartozik.
- Az IDL fixpontos `fixed`, valamint a hosszú lebegőpontos `long double` típusának még nincs szabványos Java nyelvi megfelelője. A tervek szerint az előbbinek a

`java.math.BigDecimal` osztály lesz a Java képe, míg az utóbbira vonatkozóan még csak találgatnak, hogy egy új Java alaptípust vezessenek-e be erre a célra, vagy pedig egy új `java.math.BigFloat` nevű osztályt.

- Az OMG IDL `string` típusával kapcsolatban meg kell említeni ugyanazt, amit a karakteres típussal kapcsolatban már említettünk, vagyis a 8 bites CORBA Latin-1 és a 16 bites Java UNICODE karakterkészletek különbözőségéből eredő konverziós problémákat az ORB egy `CORBA::DATA_CONVERSION` kivétellel jelzi. Az IDL fájlban specifikált hosszra vonatkozó korlátozások megszegése esetén pedig az ORB egy `CORBA::MARSHAL` kivételt generál.

- Az IDL `wstring` adattípusa képes a Java UNICODE karaktereinek ábrázolására, ezzel kapcsolatban a korlátozások megszegésével kapcsolatos `CORBA::MARSHAL` kivételt kell megemlíteni, amit az ORB akkor vált ki, ha egy CORBA metódus paraméterében hosszabb Java karakterláncot akarunk átadni, mint amennyi ott meg van engedve[2].

- A Java `null` referencia nem használható például nulla hosszú karakterláncok, vagy éppen üres tömbök ábrázolására (ezekre a célokra használhatunk egy nulla karakter hosszú karakterláncot, vagy egy üres tömböt ...).

- A Java nem biztosít előjel nélküli (`unsigned`) adattípusokat, ezért az IDL metódushívások során a nagy előjel nélküli értékek előjeles értékekként (hibásan) lesznek kezelve (általában negatív számként).

A CORBA felsorolási típusok egy `final` módosítóval ellátott, `int` típusú konstansokat tartalmazó `final` módosítóval ellátott Java osztályra lesznek leképezve. A konstansok neve megegyezik a felsorolási típus lehetséges értékhalmazának elemeivel (egy aláhúzás karaktert elé téve), és egy nullától kezdődő és egyesével növekvő érték lesz e konstansokhoz rendelve. A Java osztály neve megegyezik a felsorolási típus OMG IDL definíciós fájljában megadott nevével. Tekintsük a következő példát:

```
enum Napok { hétfő, kedd, szerda, csütörtök, péntek, szombat, vasárnap};
```

Ebből a következő Java konstrukció lesz generálva:

```
public final class Napok {
  public static final int _hétfő = 0;
  public static final Napok hétfő = new Napok(_hétfő);
  public static final int _kedd = 1;
  public static final Napok kedd = new Napok(_kedd);
  public static final int _szerda = 2;
  public static final Napok szerda = new Napok(_szerda);
```

[2]Megjegyezzük, hogy e sorok írásakor a JOE ORB szoftver nem támogatja a `wchar`, illetve a `wstring` adattípusokat.

```
  public static final int _csütörtök = 3;
  public static final Napok csütörtök = new Napok(_csütörtök);
  public static final int _péntek = 4;
  public static final Napok péntek = new Napok(_péntek);
  public static final int _szombat = 5;
  public static final Napok szombat = new Napok(_szombat);
  public static final int _vasárnap = 6;
  public static finel Napok vasárnap = new Napok(_vasárnap);

  public int value() {
    ... // visszaadja a Napok objektum értékét egésszé konvertálva
  }

  public static Napok from_int(int value) { // létrehoz egy Napok típusú
    ...                                     // értéket egy adott egész
  }                                         // típusú érték alapján

  private Napok(int) {
    ... // konstruktor
  }
}
```

A fenti példán keresztül láthatjuk, hogy generálva lesz egy `value()` nevű metódus is, ami a felsorolási típusunk típusértékeinek megfelelő egész típusú értéket adja vissza, illetve definiálva lesz egy ellenkező irányú konverziót megvalósító `from_int()` nevű metódus is (ez a paraméterében megadott egész értéknek megfelelő felsorolási típus típusértéket adja vissza).

Az OMG IDL rekord típusai egy `final` módosítóval ellátott Java osztályra lesznek leképezve (a Java osztály neve megegyezik az IDL nyelven definiált rekord típus nevével). E Java osztály a rekord típus komponenseit (mezőit) egy-egy példányváltozóban tárolja, és biztosít metódusokat ezeknek az értékeknek a beállítására és lekérdezésére, valamint biztosít egy paraméter nélküli konstruktort is a megfelelő mezők későbbi inicializálásához.

Tekintsük példaként a következő IDL struktúra definícióját:

```
struct Személy {
  string Neve;
  string Címe;
};
```

Ebből a következő szerkezetű Java fájl lesz generálva:

```
final public class Személy {
  // a rekordkomponensekből a következő példányváltozók lettek:
  public String Neve;
  public String Címe;
  // A két alapkonstruktor :
  public Személy() {};
  public Személy(String Neve, String Címe) {
```

```
    ... // itt this.Neve = Neve; vagy hasonló jellegű értékadások lehetnek
    ... // bár inkább másolni kell az értékeket, mintsem a referenciát
    ... // megőrizni (lásd a java.lang.Object osztály clone() metódusát)
  }
}
```

Az OMG IDL unió típusai egy olyan Java osztályra lesznek leképezve, amelynek a neve megegyezik az illető unió típus IDL definíciós fájlban megadott nevével, továbbá a következő jellemzőkkel rendelkezik:

- van egy paraméter nélküli konstruktora
- van egy `discriminator()` nevű metódusa, ami megadja, hogy az illető objektum éppen milyen típusú értéket tartalmaz (ez mögött egy ugyanilyen nevű adattag áll, ami azonban a külvilág számára nem látható, és tárolja, hogy hányadik címkéhez tartozik az unió típusú objektum által tárolt aktuális érték)
- van egy-egy lekérdező metódusa minden lehetséges tartalmazott típusra (és címkére) vonatkozóan[3]
- van egy-egy módosító metódusa minden lehetséges tartalmazott típusra (és címkére) vonatkozóan.

Megjegyezzük, hogy az alapértelmezés komponensre (ez a `default` címke) vonatkozóan is szükség van egy-egy fent említett metódusra.

A tartalomlekérdező metódusok `CORBA::BAD_OPERATION` kivételt generálnak, ha rosszul használják őket (azaz ha az uniónak nem azon komponensének a lekérdező metódusát használják, amelyik komponenst legutoljára beállították).

Megjegyezzük, hogy ha két címke is ugyanarra az unióbeli komponensre vonatkozna, akkor egy olyan metódus is generálva lesz, amely paraméterében várja annak megjelölését, hogy melyik címkéhez is tartozik a beállított érték (lásd az alábbi példában a `helyes()` metódus harmadik, két paraméterrel ellátott definícióját).

Példaként tekintsük a következő IDL specifikációs fájlt:

```
union KisUnió switch (Napok) { // az unió neve tetszőleges, lehet ez is
  case hétfő:     long méret;  // első ág
  case kedd:      short idő;   // második ág
  case szerda:
  case csütörtök:
  case péntek:   boolean helyes; // harmadik ág, három címkéhez
  default:        long évjárat;  // "egyebek" ág (ha nem a fentiek)
};
```

Itt pedig az ebből generált Java kód lényegesebb részletei következnek:

[3]Megjegyezzük, hogy az unió egy ága több címkéhez is tartozhat.

```
final public class KisUnió {
  // néhány adattagban tárolja az unió lehetséges értékeit, de
  // ez nem látható kívülről, így mi is csak ezt írjuk, hogy:
  private ... ; // pl. itt lehet az, hogy: "private long méret;"
  private ...diszkriminátor... ; // tárolja, hogy az unió melyik ága
                                 // tartalmazza az aktuális értéket
  // itt következik a konstruktora
  public KisUnió() {
    // ez általában üres, nem történik semmiféle inicializálás
  }
  public Napok discriminator() {
    // visszaadja, hogy az ileltő objektum az uniónak épp melyik
    // ágához tartozó értéket tárolja
  }
  // itt következnek az egyes ágakon tárolt értékeket kezelő metódusok
  // méret
  public int méret() {
    // visszaadja a "méret" komponens értékét
  }
  public void méret(int value) {
    // beállítja az unió "méret" komponensét, és a diszkriminátorban is
  }
  // idő
  public short idő() {
    // visszaadja az "idő" unió komponens értékét
  }
  public void idő(short value) {
    // beállítja az unió "idő" komponensét, és a diszkriminátor értékét
  }
  // helyes
  public boolean helyes() {
    // visszaadja a "helyes" unió komponens értékét
  }
  public void helyes(boolean value) {
    // beállítja az unió "helyes" komponensét (a diszkriminátort "szerda"
    // konstans - felsorolási típusérték - értékre aktualizálja)
  }
  public void helyes(Napok discriminator, boolean value) {
    // beállítja az unió "helyes" komponensét (a diszkriminátort az első
    // paraméterében megadott - felsorolási típusérték - értékére állítja)
  }
  // évjárat
  public int évjárat() { ... }
  public void évjárat(int value) { ... }
}
```

Megjegyezzük, hogy "default" ágat az IDL specifikációban is csak akkor írhatunk, ha az explicite megnevezett címkék nem fedik le a teljes típusértékhalmazt. Ha nem írunk "default" ágat, de szükség lenne rá, akkor a rendszer automatikusan létrehoz egy `default()` (vagy ha ilyen azonosító már van, akkor `_default()`) nevű ágat, amelyet meghívva az unió diszkriminánsa egy olyan értékre lesz beállítva, amely egyik ágnak sem

fordul elő a címkéjében (lényegében egy érvénytelen értékre).

Továbbá megemlítjük, hogy a generált Java fájl a jelenlegi - hivatalos szabványt most még általában nem követő - OMG IDL-ről Java nyelvre fordító programokkal előállított Java kód szerkezete (ezzel együtt a metódusok paramétereinek a típusa is) eltérhet a fent bemutatottaktól. Azt tanácsoljuk, hogy a pontos szintaxisnak nézzünk utána az ORB szoftverünkkel szállított dokumentációkban (vagy nézzük meg az ORB szoftverünkkel szállított IDL-ről Java nyelvre fordító program által generált kódot).

Az OMG IDL szekvencia és tömb típusai közvetlenül a Java nyelv tömb típusára lesznek leképezve (azaz ahol egy metódus vagy adattag a programban egy IDL tömb vagy szekvencia típusú adatot vár, ott a Java leképezés egy Java tömb formájában fog megvalósulni). Mivel a Java tömbjeinek a mérete fordításkor még nem rögzített, de a program futása során egy létrehozott tömb mérete nem változtatható, ezért a generált Java osztály ellenőrzi azt, hogy egy IDL műveletnek átadott Java tömb megfelel-e az IDL metódus paramétereire vonatkozó korlátozásoknak (például azt, hogy egy Java tömb mérete nem nagyobb annál, mint ami az illető metódus specifikációjában meg van adva; ilyenkor az ORB egy `CORBA::MARSHAL` kivételt generál).

Tekintsük a következő példát, egy IDL szekvencia definiálását:

```
typedef sequence <Elem_típus_neve> Konstruált_típus_neve;
attribute Konstruált_típus_neve Adattag_neve;
```

Ebből a következő Java kód lesz generálva:

```
public Elem_típus_neve[] Adattag_neve;
```

A typedef definíciók leképezése a következőképpen van megoldva.

A Java nyelvnek nincs `typedef` konstrukciója. A `typedef` kulcsszóval definiált új típusnevek minden előfordulásuknál helyettesítve lesznek a definíciójukban megadott típuskonstrukcióval.

Megjegyezzük, hogy az összes `typedef` konstrukcióhoz lesz generálva egy-egy helper osztály, holder osztályok pedig csak a szekvencia és a tömb `typedef`-típusdefiníciókhoz lesznek generálva (ezekről az osztályokról a későbbiekben lesz szó - lásd a 3.4. és 3.2. pontokban, most elégedjünk meg ezzel a ténnyel).

Ezzel kapcsolatban tekintsük a következő példát:

```
struct Valami {
  string a;
  string b;
};

typedef Valami EgyRekord;
```

Az ebből generált Java konstrukciók a következők:

```
final public class Valami {
  ..
  .. // a rekord típusnak megfelelő Java konstrukciók
  ..
}

public class EgyRekordHelper {
  ..
  .. // a rekord típushoz tartozó helper osztály
  ..
}
```

Az OMG IDL nyelven specifikált kivételek a Java nyelv kivételeire lesznek leképezve. Egy IDL nyelven specifikált kivétel egy olyan - vele megegyező nevet viselő - Java osztályra lesz leképezve, amely példányváltozóként tartalmazza a kivétel adattagjait, illetve rendelkezik néhány - jól definiált - konstruktorral.

A CORBA alapvetően kétféle kivételtípust definiál: a CORBA kommunikációs rendszer működése során felmerülő problémákat jelző kivételeket (ezeket nevezik CORBA rendszerkivételeknek), valamint a felhasználó által definiált kivételeket. A Java nyelvi leképezésben az előbbiek a `java.lang.RuntimeException`, míg az utóbbiak a `java.lang.Exception` osztálytól lesznek - igaz közvetetten - származtatva.

A felhasználó által definiált kivételeket reprezentáló osztályok Java képei az `org.omg.CORBA.UserException` Java könyvtári osztálytól lesznek származtatva. Mivel a Java nyelv nem teszi lehetővé osztályoknak az interfészekbe történő beágyazását, ezért az esetlegesen így definiált osztályok (vagy akár a kivételek) egy speciális célú csomagba lesznek generálva, ahol a csomag nevét az IDL típusnév mögé illesztett `Package` szócskából kaphatjuk meg[4].

Tekintsük erre a következő példát:

```
exception kivétel1 { string ok; };
```

Az ebből generált Java kód szerkezete a következő:

```
final public class kivétel1 extends org.omg.CORBA.UserException {
  public String ok;
  public kivétel1 () { ... };
  public kivétel1 (String r) { ... };
}
```

Tekintsünk egy másik felhasználó által definiált kivételosztályt, ahol a kivétel egy IDL interfészbe van beágyazva:

[4]Megjegyezzük, hogy hasonló a helyzet az IDL interfészekbe ágyazott IDL interfészekkel (és más, beágyazott Java osztályokra leképezett IDL elemekkel) is, ezek is ilyen módon lesznek Java nyelvre leképezve (mármint úgy, hogy a tartalmazó interfész neve mögé írt `Package` szócskával kaphatjuk meg a beágyazott interfészt tartalmazó Java csomag nevét).

```
module Példa {
  interface Valami {
    exception kivétel2 {};
  };
};
```

Az ebből generált Java kód szerkezete a következő:

```
package Példa.ValamiPackage;  // szokás szerint

final public class kivétel2 extends org.omg.CORBA.UserException {
  public kivétel2 () { ... } ;
}
```

A CORBA rendszerkivételek Java képei az `org.omg.CORBA.SystemException` Java osztálytól lesznek származtatva. Ezeknek a kivételosztályoknak vannak olyan metódusaik, amelyekkel hozzáférhetünk a hiba okát jelző fő illetve "al" hibakódokhoz, és a hiba szöveges leírásához. Az osztályok neve megegyezik a CORBA IDL specifikációjukban definiált nevükkel, és mind az `org.omg.CORBA` Java csomagban találhatók (már eddig is láthattuk, hogy a CORBA interfészek Java nyelvre történő leképezésével kapott osztályok ebben a csomagban, vagy ennek a csomagnak egy - a csomagok hierarchiájában - alárendelt csomagjában találhatóak).

A következő programlista bemutatja e kivételek ősosztályát, és ezek szerkezetét. A `CompletionStatus` osztály egy IDL nyelven definiált felsorolási típus képe, és az utoljára végrehajtott művelet eredményességéről tárol információkat. A felsorolási típus lehetséges értékei és ezek jelentése a következő:

`COMPLETED_YES` : az illető művelet végrehajtása sikeresen befejeződött.

`COMPLETED_NO` : az illető művelet végrehajtása sikertelenül zárult.

`COMPLETED_MAYBE` : nincs pontos információ a végrehajtás eredményességéről.

```
package org.omg.CORBA;

public final class CompletionStatus {
  // A kivételt kiváltó művelet végrehajtásának eredménye
  public static final int _COMPLETED_YES = 0,
                          _COMPLETED_NO = 1,
                          _COMPLETED_MAYBE = 2;
  public static final CompletionStatus COMPLETED_YES =
    new CompletionStatus(_COMPLETED_YES);
  public static final CompletionStatus COMPLETED_NO =
    new CompletionStatus(_COMPLETED_NO);
  public static final CompletionStatus COMPLETED_MAYBE =
    new CompletionStatus(_COMPLETED_MAYBE);
  public static final CompletionStatus from_int(int) { ... }
  private CompletionStatus(int) { ... }
```

```
}

abstract public class SystemException
                      extends java.lang.RuntimeException {
  // az adattagok tárolják a hiba okát azonosító kódokat
  public int minor;
  public CompletionStatus completed; // eredményességről ...
  // az egyetlen konstruktor művelet
  protected SystemException(String ok, int minor,
                            CompletionStatus status) {
    super(ok); // a szülőosztály konstruktora tárolja el a hiba okát
    this.minor = minor;    // a hiba okát jelző egész típusú hibakód
    this.status = status;  // a befejezett művelet állapotáról információ
  }
}

final public class UNKNOWN // ez a CORBA::UNKNOWN IDL kivétel Java képe
                   extends org.omg.CORBA.SystemException {
  public UNKNOWN() { ... }
  public UNKNOWN(int minor, CompletionStatus completed) {
    ...
  }
  public UNKNOWN(String ok) { ... }
  public UNKNOWN(String ok, int minor,
                 CompletionStatus completed) { ... }
}
```

Megjegyezzük, hogy az `org.omg.CORBA.SystemException` kivételt reprezentáló osztálynak a bekövetkezett hiba (illetve kivétel) okáról tárolt információihoz a programozó a megfelelő kivételosztály `minor`, illetve `completed` adattagjain keresztül férhet hozzá. Ezek nyilvános adattagok, vagyis bárki belelát a szerkezetükbe. Továbbá megjegyezzük, hogy a CORBA IDL mindegyik másik kivételének a szerkezete az előbb bemutatott UNKNOWN osztályhoz hasonló konstrukcióra lesz leképezve (a paraméter nélküli konstruktorral létrehozott kivétel objektum nem ad vissza információt a kivétel kiváltásának okáról, a második konstruktor a numerikus hibakódok beállítását teszi lehetővé, a harmadik konstruktorban a hiba okának szöveges leírását adhatjuk meg, míg a negyedik konstruktort használjuk akkor, ha a hiba okának mind a szöveges, mind pedig a numerikus értékekkel kódolt okára vonatkozó információt el akarjuk tárolni a létrehozott kivétel objektumban). Fontos tudni, hogy bármilyen IDL művelet kiválthat CORBA rendszerkivételeket (például oka lehet ennek az, ha az elérni kívánt objektumot tároló szerver időlegesen vagy véglegesen leszakad a hálózatról, ezért az alkalmazások készítőinek erre fel kell készülniük - hasonlóan a Java RMI rendszer `java.rmi.RemoteException` kivételek kezeléséhez - ahogy azt a fenti névszolgáltató példáján keresztül is láthattuk - ezeket a kivételeket nem kell az IDL specifikációban feltüntetni).

Az alábbi táblázatban összefoglaljuk a CORBA rendszerkivételek kiváltásának tipikus okait. A hozzájuk tartozó `minor` kódérték nincs szabványosítva, így jelentése függhet

attól, hogy milyen ORB szoftvert használunk (részletekért lásd a használt ORB szoftver leírását)).

`CORBA::BAD_PARAM` kivétel : `null` paramétert adtunk egy Java IDL metódusnak, vagy egy helper osztály típuskényszerítési `narrow()` metódusa sikertelen.

`CORBA::COMM_FAILURE` kivétel : A CORBA objektumreferenciában tárolt számítógép megadott (TCP-)portja nem érhető el, vagy nem sikerült a hálózati adatküldés. Valószínűleg a kommunikációs partner lebontotta az összeköttetést, vagy összeomlott.

`CORBA::DATA_CONVERSION` kivétel : nem sikerült egy objektumreferencia előállítása a szöveges reprezentációjából.

`CORBA::INTERNAL` kivétel : az ORB szoftver felélesztése, elindítása sikertelen, vagy olyan IIOP-üzenetet kapott, amelyet nem tud értelmezni.

`CORBA::INV_OBJREF` kivétel : A megadott IOR nem tartalmaz érdemleges információt az elérni kívánt CORBA objektum helyéről.

`CORBA::MARSHAL` kivétel : A kapott CORBA objektumreferencia nem értelmezhető, vagy egy nem támogatott IDL adattípust próbálunk beolvasni a hálózati összeköttetésről, vagy próbálunk arra írni (ez előfordulhat például olyankor, ha az érkezett vagy átvinni kívánt karakter nem ISO Latin-1-es, azaz nem esik a 0-255 intervallumba).

`CORBA::NO_IMPLEMENT` kivétel : a művelet implementációja nem elérhető.

`CORBA::OBJ_ADAPTER` kivétel : nincs meg a hivatkozott objektumadapter (a kliens vagy szerver oldalon).

`CORBA::OBJ_NOT_EXIST` kivétel : a keresett objektum nem létezik, vagy a talált szerver nem az, amelyik az IOR-t létrehozta.

`CORBA::UNKNOWN` kivétel : a hívott metódus olyan kivételt váltott ki, amelyre az alkalmazás nem számított.

A CORBA Any típusa is - természetesen - leképezhető bizonyos Java nyelvi konstrukciókra, de mivel a Java nyelv önmagában nem biztosít egy ezzel egyenértékű lehetőséget, ezért e leképezés egy kicsit bonyolultabb az eddigiekben megismert konstrukcióknál (gondoljuk csak meg, hogy a Java nyelv szabványos `java.lang.Object` osztálya nem őse minden Java adattípusnak, így például a nyelv beépített elemi adattípusainak - mint az `int`, `long`, `bool`, `char` stb. típusoknak - sem, ezért önmagában ez nem jelent megoldást a CORBA `Any` típusának az ábrázolására).

Az `Any` adattípus Java leképezése lényegében egy olyan unió típus, amelynek része az összes elemi Java adattípus, valamint maga az `Object` osztály is. Az implementáció

biztosít metódusokat tetszőleges adattípusok eltárolására és kiolvasására (de egy `Any` típusú objektum tartalmát csak olyan formában olvashatjuk ki, ahogyan beraktuk - különben egy `CORBA::BAD_OPERATION` kivétel lesz kiváltva). Az `Any` típusú objektumok tartalmára vonatkozó típus információk lekérdezésére megvannak a megfelelő metódusok. A következő lista az `org.omg.CORBA.Any` specifikációjának egy részletét tartalmazza:

```
package org.omg.CORBA;

abstract public class Any {
  abstract public boolean equal(org.omg.CORBA.Any);
  // a tartalmazott objektum típusának lekérdezése/beállítása
  abstract public org.omg.CORBA.TypeCode type();
  abstract public void type(org.omg.CORBA.TypeCode);
  // elemi típusok tárolása és visszanyerése ...
  abstract public short extract_short() throws org.omg.CORBA.BAD_OPERATION;
  abstract public void insert_short(short);
  abstract public short extract_ushort() throws org.omg.CORBA.BAD_OPERATION;
  abstract public void insert_ushort(short); // IDL unsigned short
  abstract public int extract_long() throws org.omg.CORBA.BAD_OPERATION;
  abstract public void insert_long(int);
  // ... itt sok-sok hasonló szerkezetű metódust kihagytam ...
  abstract public org.omg.CORBA.Any extract_any()
           throws org.omg.CORBA.BAD_OPERATION;
  abstract public void insert_any(org.omg.CORBA.Any);
  abstract public org.omg.CORBA.Object extract_Object()
           throws org.omg.CORBA.BAD_OPERATION;
  abstract public void insert_Object(org.omg.CORBA.Object);
  // objektumok tárolása és visszanyerése ...
  // kivételt vált ki, ha az objektum a megadott típuskóddal inkonzisztens
  abstract public void insert_Object(org.omg.CORBA.Object,
           org.omg.CORBA.TypeCode) throws org.omg.CORBA.MARSHAL;
  abstract public String extract_string()
           throws org.omg.CORBA.BAD_OPERATION;
  abstract public void insert_string(String)
           throws org.omg.CORBA.MARSHAL, org.omg.CORBA.DATA_CONVERSION;
  abstract public String extract_wstring()
           throws org.omg.CORBA.BAD_OPERATION;
  abstract public void insert_wstring(String);
  abstract public TypeCode extract_TypeCode()
           throws BAD_OPERATION;
  abstract public void insert_TypeCode(TypeCode);
  // A többi, számunkra most kevésbé érdekes metódusokat kihagytuk ...
}
```

Láthatjuk, hogy az egyes IDL adattípusokhoz tartozik egy-egy metódus, amivel az olyan típusú adatokat eltárolhatjuk, illetve kinyerhetjük az `Any` típusú értékeket reprezentáló `org.omg.CORBA.Any` osztályba tartozó Java objektumokból. Például az int IDL típusú értékeket az `insert_int()` metódussal állíthatjuk be, illetve az `extract_int()` metódussal kérdezhetjük le; ezeknek az értékbeállító metódusoknak a

paraméterükben kell megadni a beállítandó értéket (a paraméterükben az IDL típus Java megfelelőjét kell írni), az értékkinyerő metódusok pedig visszatérési értékükben adják vissza az `Any` típusú objektumokból kinyert értéket.

Az `equal()` nevű metódus használható `Any` osztályba tartozó objektumok tartalmi egyenlőségének vizsgálatára (e metódus implementációja egyszerű típusok esetén az egyenlőség vizsgálatát a Java nyelvi eszközökkel megoldhatja, de objektumok esetén az illető objektumok azonos nevű metódusára bízza az egyenlőség eldöntését).

Említést érdemelnek még a `type()` metódusok is, amelyekkel az `Any` osztályba tartozó objektumok által tartalmazott érték típusát lehet lekérdezni, illetve módosítani (ez utóbbi esetben a módosítás után a korábbi értékhez nem lehet hozzáférni, az elvész[5]; ezt elsősorban a később bemutatandó CORBA `out` paraméterátadási mód támogatására tervezték). A tartalmazott adat típusára vonatkozó információkat az `org.omg.CORBA.TypeCode` osztály példányaival reprezentálhatjuk, illetve állíthatjuk be és kérdezhetjük le (lásd az előbbi metódusok szignatúráját), aminek a lényegesebb metódusait a következő listából megismerhetjük.

```
public abstract class TypeCode extends Object
{
  public abstract boolean equal(TypeCode tc) throws SystemException;
  public abstract TCKind /* int */ kind() throws SystemException;
  public abstract String id() throws BadKind, SystemException;
  public abstract String name() throws BadKind, SystemException;
  public abstract int member_count() throws BadKind, SystemException;
  public abstract String member_name(int index) throws BadKind, Bounds,
                                                SystemException;
  public abstract TypeCode member_type(int index) throws BadKind, Bounds,
                                                  SystemException;
  public abstract Any member_label(int index) throws BadKind, Bounds,
                                                  SystemException;
  public abstract TypeCode discriminator_type() throws BadKind,
                                                SystemException;
  public abstract int default_index() throws BadKind, SystemException;
  public abstract int length() throws BadKind, SystemException;
  public abstract TypeCode content_type() throws BadKind, SystemException;
}
```

A `kind()` metódus visszaad egy, az illető objektum típusának konstrukcióját reprezentáló egész értéket (ami meghatározza azt is, hogy milyen további metódusokat hívhatunk meg az előbbiek közül - ld. később). A lehetséges egész érékek a CORBA nevű IDL modulban definiált következő felsorolási típus típusértékei közül kerülhetnek ki:

```
enum TCKind { tk_null, tk_void, tk_short, tk_long, tk_ushort, tk_ulong,
  tk_float, tk_double, tk_boolean, tk_char, tk_octet, tk_any,
  tk_TypeCode, tk_Principal, tk_objref, tk_struct, tk_union, tk_enum,
  tk_string, tk_sequence, tk_array, tk_alias, tk_except } ;
```

[5]Ha ilyenkor mégis megpróbálnánk elérni, akkor egy `CORBA::BAD_OPERATION` kivétel lesz kiváltva.

Megjegyezzük, hogy a `CORBA::TCKind` felsorolási típus értékei nem típuskódokat, hanem típuskonstrukciós módokat definiálnak. A típuskódokat a `CORBA::TypeCode` interfész elemei reprezentálják. A továbbiakban a `CORBA::TypeCode` interfészt fogjuk részletesebben megismerni.

Az `id()` metódus visszaadja az illető típusnak az interfészgyűjteménybeli egyedi azonosítóját, míg a `name()` metódus az illető típus nevét adja vissza (az őt tartalmazó legszűkebb egységen - pl. modulon vagy interfészen - belül). A `member_count()` metódus struktúrákon, uniókon és felsorolási típusokon hívható, és visszaadja az illető típus komponenseinek a számát[6]; ha a komponensek nevére lennénk kíváncsiak, akkor ehhez az információhoz a `member_name()` metódussal férhetnénk hozzá (e metódus paraméterben megadva azt, hogy hányadik komponens érdekel minket; hasonlóan a `member_type()` metódushoz, amely viszont az illető komponens típusára vonatkozó információkat adja vissza). A `member_label()` metódus unió típusú objektumok esetén hívható, és visszaadja azt, hogy az illető objektumban tárolt érték az unió melyik ágához rendelt címkéhez tartozik. A `discriminator_type()` metódus szintén csak unió típusú objektumok esetén hívható, és visszaadja az unió ágainak a felcímkézéséhez használt adattípus típuskódját. A `default_index()` metódus szintén csak unió típusú objektumok esetében hívható, és visszaadja az illető unió típus `default` címkéjű ágához tartozó indexet, illetve -1 értéket ad vissza, ha az illető unió típusnak nincs ilyen ága. A `length()` metódus sorozatok, tömbök, és karakterláncok esetén hívható, és visszaadja az illető adattípus hosszára vonatkozó felső korlátot (illetve nullát, ha nincs ilyen korlát). A `content_type()` metódus ugyanígy sorozatokra, valamint tömbökre vonatkozóan hívható meg, és visszaadja az illető sorozat (vagy tömb) elemeinek a típusát (pontosabban: annak típuskódját).

Az alábbi programrészletben láthatjuk, hogy hogyan hozhatunk létre egy `void` tartalomtípusú `Any` objektumot[7].

```
import org.omg.CORBA.*;

...

TypeCode tc1 = orbRef.get_primitive_tc(TCKind.tk_void);
Any eredm = orbRef.create_any();
eredm.type(tc1);
```

3.2. A CORBA paraméterátadási módjai

Míg a Java nyelv csak az érték szerinti paraméterátadást támogatja (objektumok átadásakor az objektumreferencia kerül érték szerint átadásra), addig a CORBA a már említett háromféle paraméterátadási módot definiálja: az `in`, `out` és az `inout`

[6] Ahol ennek nincs értelme, ott ez a metódus egy `CORBA::BadKind` kivételt generál, ami az `org.omg.CORBA.TypeCodePackage` csomagban van.

[7] Az ORB szoftvert reprezentáló `orbRef` objektumról a későbbi részekben még többet olvashatunk. Most elégedjünk meg annyival, hogy így kérdezhetjük le a `void` adattípushoz tartozó típuskódot.

módokat, amelyek közül az `in` az egyetlen, amely érték szerinti paraméterátadást jelent, míg a másik két paraméterátadási módban a kliens által átadott paraméter értéke is megváltozhat.

A CORBA Java nyelvi leképezésében e két nem érték szerinti paraméterátadási módot speciális segédosztályokkal, ún. holder osztályokkal támogatják: a különféle IDL alaptípusok és a felhasználó által definiált adattípusokhoz definiálva van egy ún. holder osztály, amely az illető adattípus elemeit tartalmazza, és ennek az osztálynak az elemei lesznek szükség esetén `out` és `inout` paraméterként átadva. Egy IDL adattípushoz tartozó holder osztály nevét az illető IDL adattípus Java nyelvi képét tartalmazó osztály neve mögé írt `Holder` szócskával képezhetjük: például a `Tranzakció` nevű osztályhoz (interfészhez) tartozó holder osztály neve `TranzakcióHolder`.

Az alapvető IDL adattípusokhoz tartozó holder osztályok az `org.omg.CORBA` Java csomagban vannak definiálva. Szerkezetüket tekintve a holder osztályokról azt mondhatjuk, hogy van egy `value` nevű adattagjuk, amely a holder osztály által tartalmazott osztály objektumait tartalmazza. A holder osztályoknak van egy alapértelmezés szerinti konstruktoruk, ami a tartalmazott objektum tartalmát a Java nyelv szabályai szerint inicializálja; valamint van egy olyan konstruktoruk, amely a holder által tartalmazott osztály egy objektumát várja paramétereként, és a paraméterben kapott objektummal inicializálja a value komponenst.

Tekintsük például a `short` Java adattípus megfelelőjét jellemző holder osztály szerkezetét:

```
package org.omg.CORBA;

final public class ShortHolder {
  public short value;         // nyilvános elérésű, bárki láthatja
  public short ShortHolder(); // A "value" short típusú érték a
                              // példányosításkor "szabályosan"
                              // inicializálva lett
  public short ShortHolder(short kezdőérték) {
    value = kezdőérték;       // itt tárolja el a paraméterbeli értéket
  }
}
```

Figyeljük meg, hogy az osztály és konstruktora neve nagy kezdőbetűvel van írva.

Megjegyezzük, hogy a `value` adattag nyilvános (`public`) láthatósággal rendelkezik, így az objektumpéldányok által tartalmazott értékhez az alkalmazás könnyen hozzáférhet.

A paraméterek leképezésének áttekintésére tekintsük a következő példát!

```
interface Egyinterfész {
  long próba(in long inParam, out long outParam, inout long inoutParam);
};
```

Ebből a következő Java kódrészlet készül:

```
public interface Egyinterfész extends org.omg.CORBA.Object {
  int próba(int inParam, IntHolder outParam, IntHolder inoutParam);
}
```

Ha a programozó a fenti CORBA metódust meg akarja hívni egy Java programból, akkor azt például a következő módon teheti meg:

```
Egyinterfész eif = ... ; // létrehozunk egy megfelelő objektumot
int inParam = 23;
IntHolder outHolder = new IntHolder();
IntHolder inoutHolder = new IntHolder(41);
int eredmény = eif.próba(inParam, outHolder, inoutHolder);
... outHolder.value ... // itt használjuk fel a visszaadott értéket
... inoutHolder.value ... // itt használjuk fel a visszaadott értéket
```

Látható - és gondolom érthető is -, hogy az `inout` metódus paraméterekben a metódusnak átadandó értéket a metódushívás előtt már be kell állítani, amit a fenti példában a megfelelő konstruktor alkalmazásával végzünk el (a metódus harmadik aktuális paraméterében átadandó érték a 41, ezt állítottuk be a fenti programrészletben az `inoutHolder` nevű változóban).

Egy felhasználó által definiált - `<típus_neve>` nevű - adattípushoz generált holder osztály szerkezete a következő (néhány számunkra kevésbé érdekes metódust leszámítva):

```
final public class <típus_neve>Holder
             implements org.omg.CORBA.portable.Streamable {

  public <típus_neve> value;
  public <típus_neve>Holder() {}
  public <típus_neve>Holder(<típus_neve> kezdőérték) {}

  .. // egyebek, pl. _write(), _read()
}
```

Az egyes metódusok szerepe magától értetődő (a holder osztály által tárolt adatra a `value` adattagon keresztül hivatkozhatunk, míg a két konstruktor - szükség esetén egy adott kezdőértékkel - inicializálja a holder értékét).

Az `org.omg.CORBA.portable.Streamable` interfészben definiált `_read()` és `_write()` metódusok valósítják meg az egyes adattípusokhoz a tartalom bájtsorozatra másolását és a tartalomnak az onnan történő visszamásolását. Szerepe az `Any` típus szemantikájának megvalósításában fontos, de alkalmazói programok készítéséhez az ezzel kapcsolatos további részletek ismerete nem szükséges.

Az alábbi IDL specifikáció egy visszahívásos szerkezetet szemléltet.

```
module VisszahivoIDL
{
    interface Visszahivas
    {
```

```
        void figyelj(in string uzenet);
    };

    interface Bejelentes
    {
        string ittVagyok(in Visszahivas objRef, in string kivagyok);
    };
};
```

Egy kliens alkalmazás a szervert a `Bejelentes` IDL interfészen keresztül érheti el, megadva neki az első paraméterben egy kliensen belül levő CORBA objektumreferenciát, amely egy olyan CORBA objektumra hivatkozi, amely a `Visszahivas` IDL interfészt implementálja. A második paraméter a visszahívásos technika szemléltetése szempontjából nem érdekes, azon keresztül a kliens egy szöveget juttathat el a szerverhez. A szerver később bármikor meghívhatja a klienstől megkapott `Visszahivas` IDL interfészt implementáló objektum `figyelj()` nevű metódusát.

A fenti IDL definícióban definiált interfészeknek az alábbi két Java interfész feleltethető meg:

```
package VisszahivoIDL;    // Bejelentes interfész képe

public interface Bejelentes extends org.omg.CORBA.Object {

    String ittVagyok(VisszahivoIDL.Visszahivas objRef, String kivagyok);

}



package VisszahivoIDL;    // Visszahivas interfész képe

public interface Visszahivas extends org.omg.CORBA.Object {

    void figyelj(String uzenet);

}
```

3.3. A CORBA-alapú alkalmazások szerkezete

A CORBA-alapú alkalmazások szerkezetére vonatkozóan általában nincsenek korlátozások. Minden CORBA-alapú alkalmazásra jellemző, hogy elindulása után felveszi a kapcsolatot az ORB szoftverrel (egy speciális ún. ORB objektumon keresztül). Ezután a kliens alkalmazások ezen az ORB objektumon keresztül juthatnak a rendszerben lévő többi CORBA objektumra hivatkozó objektumreferenciákhoz (így például a CORBA névszolgáltató referenciájához is), és ezen keresztül jelenthetik az ORB szoftvernek egy új CORBA objektumot szolgáltató szerver elindulását (vagy éppen leállását).

Bár az ORB szoftvert reprezentáló objektummal (illetve osztállyal) a későbbiekben még részletesebben is foglalkozunk, itt most röviden megnézzük, hogy milyen módon inicializálhatja az alkalmazás az ORB szoftvert, és milyen módon juthat hozzá az alkalmazás egy adott nevű CORBA objektum referenciájához a CORBA névszolgáltatójából. Erre azért van most szükség, mert a következő pontokban bemutatott egy-két példát a jobb érthetőség kedvéért már egy ORB környezetbe beillesztve ismertetjük, ezért az ORB környezet részletezését nem lehet kikerülni (és az ORB szoftvert reprezentáló osztály ismertetését sem tudtuk érdemben előbbre hozni, mivel hasznos lesz a részletek megismeréséhez a statikus és a dinamikus metódushívási interfészek ismerete).

Az ORB inicializálását az `org.omg.CORBA.ORB` osztály statikus `init()` metódusával tehetjük meg. Ezt a metódust többféleképpen is felparaméterezhetjük, mi azt a változatát használjuk, amely az első paraméterében várja a főprogram argumentumait (a program indítására használt parancssorban levő argumentumokat), a második paraméterében pedig néhány ORB-függő környezeti jellemző adható meg (de - mint az egyszerűbb programjainkban látni fogjuk - e második paraméterben átadhatjuk a `null` értéket is). Az `init()` metódus visszaad egy referenciát egy újonnan létrehozott ORB objektumra, amin keresztül megkaphatjuk például a CORBA névszolgáltató[8] egy objektumreferenciáját a `resolve_initial_reference()` metódussal, a `NameService` karaktersorozatot megadva a metódus paraméterében. Ezután a névszolgáltatót a visszakapott objektumreferencián keresztül - a fejezet elején ismertetett névszolgáltató interfész Java leképezéseként kapott metódusokkal - érhetjük el.

```
ORB orbref = ORB.init(args, null); // 1. paraméter: főprogram argumentumai
org.omg.CORBA.Object ref = orb.resolve_initial_reference("NameService");

NamingContext ncref = NamingContextHelper.narrow(ref);
NameComponent nc = new NameComponent("Keresett_objektum_neve", "");
NameComponent keresett_komponens[] = {nc};
// itt ncref.resolve(keresett_komponens) visszaad egy referenciát a
// névszolgáltatóból kikeresett nevű objektumra
```

Az ORB szerveroldali inicializálásánál használt további metódusok ismertetését későbbre hagyjuk.

Megjegyezzük, hogy az alkalmazások CORBA objektumok referenciáihoz a névszolgáltató (és az annak elérésére használt `resolve_initial_reference()` metódus) mellett hozzájuthatnak más úton is: más alkalmazások által szöveges formára konvertált objektumreferenciák CORBA objektumreferenciává történő visszakonvertálásával. A referencia szöveges reprezentációját az alkalmazás akár megkaphatja a hálózaton keresztül - például TCP/UDP socketokkal vagy akár közönséges elektronikus leveleken keresztül -, vagy pedig egy fájlból, vagy akár a felhasználó is beadhatja azt. Egy CORBA objektumreferencia karakterlánc reprezentációját az ORB `object_to_string()` metódusával kaphatjuk meg, míg a fordított irányú konverzióra az ORB `string_to_object()`

[8] Ne felejtsük el, hogy a névszolgáltatást is egy CORBA objektum implementálja.

metódusát használhatjuk (lásd a részletekről az ORB szoftvert reprezentáló objektumot bemutató pontot).

3.4. A típuskényszerítés CORBA eszközei

A Javához hasonlóan a CORBA objektummodellje is támogatja a dinamikus kötést, és lehetővé teszi CORBA objektumreferenciák leszármazott-, illetve szülőosztály felé történő konvertálását az öröklési hierarchiában (és csak erre, vagyis az öröklési láncon kívül eső interfészekre történő konverzió tilos!). Erre például szükség lehet CORBA névszolgáltatók elérésekor, ugyanis ott - tekintsük például a `resolve()` metódust - egy objektum visszakeresésekor a visszakapott objektumreferencia `Object` típusú (ne feledjük, hogy itt nem a Java nyelv objektumainak ősosztályáról van szó, hanem a CORBA objektumhierarchia csúcsáról), és ahhoz, hogy az illető objektum osztályának a metódusait is meg tudjuk hívni, a referenciát a megfelelő osztályba kell konvertálni.

Tekintsük a következő példát egy interfészhierarchia definiálására:

```
interface i1 { // ez a hierarchia teteje, örököl az OMG Object osztálytól!
 ..
}
interface i2 : i1 {
  ..
}
interface i3 : i1 { // figyeljük meg, hogy ez nem örököl i2-től!
  ..
}
interface i4 : i2 {
  ..
}
```

Tegyük fel, hogy egy `i4` osztályba tartozó objektum egy referenciáját bejegyezzük a CORBA névszolgáltatójába. Ha ezt egy alkalmazás visszakeresi onnan, akkor egy `CORBA::Object` interfészt implementáló objektum referenciát kap vissza, amit konvertálhat akár az `i1`, akár az `i2`, akár az `i4` interfészeket reprezentáló specifikusabb osztályokba, de nem konvertálhat az `i3` osztályba, mivel az egy másik öröklési ágnak a része (a konverzió részleteit lásd alább). Ha egy ilyen konverzió sikertelen, akkor egy `CORBA::BAD_PARAM` kivétel lesz kiváltva.

Meglepő, de az előzőek alapján nyilvánvaló, hogy ha egy CORBA metódushívás visszatérési értéke például `CORBA::Object` típusúként van megjelölve (azaz ezt írtuk az illető metódus IDL specifikációs fájljában), de a visszaadott CORBA objektum valójában ennek egy leszármazottja, akkor ennek az objektumnak csak a `CORBA::Object` interfészben definiált metódusaira és adattagjaira hivatkozhatunk. Ez azért van így, mert az illető CORBA objektumot reprezentáló klienscsonk a hívott metódus IDL specifikációjában megadott objektumot reprezentálja, nem pedig a visszakapott objektum "dinamikus"

CORBA típusának megfelelőt (a Java környezet nem ismeri a visszaadott CORBA objektum CORBA típusát).

E konverziót segítendő az IDL-ről Java nyelvre leképező program generál minden egyes felhasználói típushoz egy ún. helper osztályt, amely ezt a konverziót segítő metódusokat is tartalmaz, és általános szerkezetüket a következő lista szemlélteti.

```
public class <típus_neve>Helper {
  public static void insert(org.omg.CORBA.Any a, <típus_neve> t) {...}
  public static <típus_neve> extract(org.omg.CORBA.Any a) {...}
  public static org.omg.CORBA.TypeCode type() {...}
  public static String id() {...}
  // !... a következő sor csak interfészekből generált helpereknél van:
  public static <típus_neve> narrow(org.omg.CORBA.Object obj)
         throws org.omg.CORBA.BAD_PARAM;
}
```

A helper osztály `insert()` és `extract()` metódusai használhatók a definiált IDL típusú objektumok CORBA `Any` típusú objektumokba történő berakására, illetve onnan történő kivételére. A `type()` metódus a típushoz tartozó - és az `Any` típus leképezésénél megismert - típuskódot adja vissza. Az `id()` metódus megfelel a `org.omg.CORBA.TypeCode` osztály azonos nevű metódusának, vagyis az interfészgyűjteménybeli azonosító lekérdezésére használható. A `narrow()` metódus tesz lehetővé objektumreferenciáknak az öröklési hierarchiában lefelé történő konvertálására.

Tekintsük a következő IDL specifikációt, és az azt követő programrészletet, amelyben egy kliens alkalmazás megszerez egy objektumreferenciát a CORBA névszolgáltatótól, és konvertálja az objektum megfelelő dinamikus típusára.

Az IDL specifikáció:

```
interface oxigén {
  long mindegy_hogy_mi_van_itt();
};
```

A programrészlet, amely egy fent definiált oxigén osztályba tartozó referenciát kinyer a névszolgáltatóból, pedig a következő:

```
ORB orbref = ORB.init(args, null); // paraméter: főprogram argumentumai
org.omg.CORBA.Object ref = orb.resolve_initial_reference("NameService");
NamingContext ncref = NamingContextHelper.narrow(ref);
NameComponent nc = new NameComponent("Oxigén-1", "");
NameComponent keresett_komponens[] = {nc};
oxigén oxigénref = oxigénHelper.narrow(ncref.resolve(keresett_komponens));
```

A fenti példa első két sorában megszerezzük az ORB szoftver CORBA objektumreferenciáját, majd ezután ennek segítségével megszerezzük a helyi CORBA névszolgáltató objektumreferenciáját (a második sorban visszakapott referencia a névszolgáltató számunkra érdekes "gyökér"-kontextusa). A harmadik sorban a visszakapott referenciát (amely eredetileg a CORBA Object osztályába tartozik) kontextusobjektumra hivatkozó

referenciává konvertáljuk (mivel a második sorban végrehajtott művelet specifikációjából tudjuk, hogy a visszakapott referencia erre hivatkozik). Ez volt a példában az első (alkalmazásokban tipikusan előforduló) típuskényszerítési helyzet.

A következő ilyen helyzet a hatodik sorban lesz, ahol a névszolgáltató által visszaadott "gyökér"-kontextusból kikeressük az `Oxigén-1` nevű CORBA objektumhoz rendelt referenciát (ezt teszi a `resolve()` nevű metódus), és a névszolgáltatótól visszakapott referenciát konvertáljuk az `oxigén` IDL típusúra, a belőle generált `oxigénHelper` Java osztály `narrow()` nevű metódusával. Ezután már akár meg is hívhatjuk az oxigén osztály metódusait a kapott objektumreferencián. Bár a konverzió szempontjából kevéssé érdekes, de megemlítjük, hogy a negyedik és az ötödik sorban egy olyan `NameComponent` osztálybeli objektumot állítottunk össze, amelyben beállítottuk a keresett objektum nevét (`Oxigén-1`-et). Erre a viszonylag összetett eljárásra azért volt szükség, mert a `CosNaming` modul `resolve()` metódusa a keresett objektumot azonosító nevet várja, ami névkomponensek sorozatából áll, és míg a negyedik sorban egy névkomponenst hoztunk létre (`Oxigén-1` a név, és üres a karakterlánc név minősítője), addig az ötödik sorban létrehoztunk egy csak ezt az egy nevet tartalmazó Java tömböt - emlékezzünk rá, hogy ez a szekvencia IDL adattípus Java nyelvi képe -, és ezt tudtuk a hatodik sorban a `resolve()` metódus paraméterében mint a keresendő objektum nevét megadni (ilyen a `resolve()` metódus paraméterezése ...).

A példa nem egy teljes program, de már láthatók rajta azok az alaplépések, amelyeket az összes Java-alapú CORBA alkalmazásnak meg kell tennie ahhoz, hogy elinduljon. A példából még hiányzik a kivételkezelés (mint azt már említettük, bármelyik IDL metódushívás kiválthat adott esetben egy CORBA rendszerkivételt), illetve hiányzik a szerveroldali objektum `oxigén` implementáció is (a létrehozott objektumreferenciát a névszolgáltatóba bejegyző programrészekkel együtt). Hamarosan ezekről is szó lesz.

3.5. Névszolgáltató-interfészek képe

Az alábbi listában a Java programok írásának segítése céljából megadjuk a CORBA névszolgáltatóval[9] kapcsolatos két lényeges interfész Java leképezéseként kapott Java interfészeket[10], majd néhány alkalmazási mintát is bemutatunk (persze ezekre a dolgokra az IDL Java nyelvi leképezési szabályainak ismeretében magunk is rájöhetnénk).

```
public interface NamingContext
       extends org.omg.CORBA.Object { // org.omg.CosNaming csomag

  public abstract void bind(NameComponent[] n, Object obj) throws NotFound,
```

[9] Az IDL specifikációt megtalálhatjuk a 2.15. fejezetben.

[10] A Java leképezés eredményeképpen létrejöttek további interfészek, holder- és helperosztályok. Ezek közül itt egy példaprogramon keresztül mutatunk be néhányat. A többivel nem foglalkozunk, mivel a mindennapi alkalmazásfejlesztési munkák során ritkán van rájuk szükség. Természetesen róluk is olvashatunk a könyv CORBA névszolgáltatását bemutató részében, illetve a Java fejlesztői rendszerünkkel szállított osztályleírásokban.

```
        CannotProceed, InvalidName, AlreadyBound;
  public abstract void bind_context(NameComponent[] n, NamingContext nc)
        throws NotFound, CannotProceed, InvalidName, AlreadyBound;
  public abstract void rebind(NameComponent[] n, Object obj) throws NotFound,
        CannotProceed, InvalidName;
  public abstract void rebind_context(NameComponent[] n, NamingContext nc)
        throws NotFound, CannotProceed, InvalidName;
  public abstract Object resolve(NameComponent[] n) throws NotFound,
        CannotProceed, InvalidName;
  public abstract void unbind(NameComponent[] n) throws NotFound,
        CannotProceed, InvalidName;
  public abstract void list(int how_many, BindingListHolder bl,
        BindingIteratorHolder bi);
  public abstract NamingContext new_context();
  public abstract NamingContext bind_new_context(NameComponent[] n)
        throws NotFound, AlreadyBound, CannotProceed, InvalidName;
  public abstract void destroy() throws NotEmpty;

}
```

A `BindingIterator` IDL interfész Java képe a következő:

```
public interface BindingIterator
       extends org.omg.CORBA.Object { // org.omg.CosNaming csomag

  public abstract boolean next_one(BindingHolder b);
  public abstract boolean next_n(int how_many,
      BindingListHolder bl);
  public abstract void destroy();

}
```

A következő programmal a `BindingIterator` interfész metódusainak használatát szemléltetjük. A program végigmegy az alapértelmezés szerint elérhető CORBA névszolgáltató gyökérkontextusán, közben kiírja a benne levő objektumnevek első komponensét (mind az objektumazonosítót, mind a típusazonosítót).

```
// IDLListaz.java
//
// Egy programhiba: mivel üreshalmazon nem false-t, hanem null-t
//  kapunk, a program 0, ill. 1-elemü nevszolgaltaton hibas!

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class IDLListaz {

  public static void listaz(NamingContext ncRef) {
```

```
    BindingListHolder blh = new BindingListHolder();
    BindingIteratorHolder bih = new BindingIteratorHolder();
    BindingHolder bh = new BindingHolder();

    ncRef.list(1, blh, bih); // Elso név lekérdezése
    Binding nevlista[] = blh.value;
    BindingIterator bi = bih.value;
    // Az első nevet kiírom
    if (nevlista.length > 0) {
      System.out.println("név: "+nevlista[0].binding_name[0].id+
             " típus:"+nevlista[0].binding_name[0].kind);
    }
    // A maradék nevek egyesével
    boolean vanmeg = bi.next_one(bh);
    while (vanmeg) {
      System.out.println("név: "+bh.value.binding_name[0].id+
             " típus:"+bh.value.binding_name[0].kind);
      vanmeg = bi.next_one(bh);
    }
    bi.destroy();
  }

  public static void main(String[] args) {

   try {
    ORB orb = ORB.init(args, null);
    org.omg.CORBA.Object ncObjRef=orb.resolve_initial_references("NameService");
    NamingContext ncRef=NamingContextHelper.narrow(ncObjRef);
    listaz(ncRef);
   } catch (Exception e) {
    e.printStackTrace();
    System.out.println("Egyéb kivétel: "+e+"/"+e.getMessage());
   }
  }

}
```

A fenti program a `CosNaming` IDL-modul specifikációja alapján listázza a gyökér névkontextusba bejegyzett neveket (azok első komponensét).

3.6. Kliens- és szervercsonkok generálása

Miután elkészítettük a távolról - CORBA eszközökkel - elérni kívánt objektumaink IDL nyelvű specifikációját, az IDL specifikációból el kell készíteni a kliens- és a szervercsonkokat tartalmazó Java osztályokat (illetve kliens- vagy szervercsonkot tartalmazó Java osztályt, annak megfelelően, hogy csak a kliens-, vagy csak a szerveroldalt akarjuk-e Java nyelven implementálni). E csonkok elkészítése a feladata az `idltojava` programnak, és emellett még elkészíti az IDL fájlban található további elemek (struktúrák, uniók, fel-

sorolási típusok, kivételek, ...) Java megfelelőit is.

Például a `macska.idl` nevű fájlban levő IDL specifikációból a következő parancs beadásával kaphatjuk meg a kliens- és szervercsonkokat implementáló Java osztályokat:

```
idltojava -fserver -fclient macska.idl
```

Ennek a parancsnak a hatására a `macska.idl` fájlon végigmegy a makrók feldolgozásáért felelős C preprocesszor, és elkészülnek mind a szerver-, mind pedig a kliensc-sonkokat tartalmazó Java osztályok. Amennyiben a gépünkön nincs egy C preprocesszor installálva (vagy annak elérési útvonala nincs benn a `PATH`-ban, akkor a `-fno-cpp` programargumentumot kiadva a C preprocesszor nem lesz elindítva, de ekkor nyilván a neki szóló IDL konstrukciók sem lesznek feldolgozva). A C preprocesszor nélküli installáláskor a következő parancs beadásával generálhatjuk a kliens- és szervercsonkokat:

```
idltojava -fserver -fclient -fno-cpp macska.idl
```

A `-fserver` paraméter utasítja az IDL fordítót, hogy generálja a szükséges szervercsonk osztályt, a `-fclient` paraméter pedig arra utasít, hogy a klienscsonkokat tartalmazó osztály is automatikusan generálódjon.

Szükség esetén használhatjuk még a `-fverbose`, valamint a `-fversion` paramétereket is; az előbbi hatására az `idltojava` fordító a futása során mutatja, hogy mit csinál (milyen meneteket hajt végre), az utóbbi hatására pedig a használt `idltojava` verziószámát írathatjuk ki.

Az `idltojava` alkalmazás - a fenti hívás után - a következő fájlokat állíthatja elő az `.idl` forrásfájlban levő elemekhez:

`_elemnévImplBase.java` : a szervercsonkot tartalmazó osztály (szerepét ld. később).

`_elemnévStub.java` : a klienscsonkot tartalmazó osztály.

`elemnév.java` : az IDL specifikációs fájlban definiált IDL elemek Java képét tartalmazza.

`elemnévHelper.java` : a CORBA-alapú típuskényszerítést támogató helper osztályt tartalmazza.

`elemnévHolder.java` : a nem érték szerint átadott paraméterek tárolását támogató holder osztályt tartalmazza.

Megemlítjük, hogy az `idltojava` program egyre több implementációja lehetőséget nyújt delegáció-alapú (ún. tie) szerveroldali csonkok generálására (ez a technika a C++ nyelvi ORB-k világában már régóta ismert, nem egy Jávában bevezetett technológiai újításról van szó). A szerveroldali elemek ismertetésekor bemutatjuk ennek a használatát is. A delegáció-alapú szervercsonkok generálását általában az `idltojava` segédprogramnak átadott `-ftie` paraméterrel kérhetjük, és a generált fájlok neve `_elemnévTie.java`, illetve `_elemnévOperations.java` lesz (ismertetésüket lásd később).

3.7. Statikus és dinamikus metódushívási interfész

Egy CORBA objektum valamelyik metódusát kétféleképpen hívhatjuk meg: statikus vagy dinamikus módon. A statikus hívás a távoli metódushívásnál megismert klienscsonkokhoz hasonló eszközökkel történik; itt is klienscsonkokról beszélhetünk, de a CORBA objektumok klienscsonkjai a már említett IIOP protokollal kommunikálnak egymással, nem pedig a Java RMI által használt (kevésbé elterjedt) kommunikációs protokollal. Dinamikus hívás esetén lehetőség van akár egy, a program fordításakor még nem is ismert interfészű objektum metódusainak a hívására is (bár ekkor a paramétereknek a fordítási időben történő típusegyeztetése nem megoldható).

A statikus metódushívás megvalósításához egy programnak két dologra van szüksége: egyrészt egy megfelelő CORBA objektumreferenciára (itt lényegében egy CORBA IOR, ORB szoftverek közötti érvényű referenciáról van szó), valamint a megfelelő CORBA objektum egy klienscsonkjára (a klienscsonk az elérni kívánt objektum interfészének az OMG IDL nyelven leírt specifikációjából generálható az `idltojava` programmal).

A dinamikus hívás bonyolultabb a statikusnál, mivel a klienscsonk feladatait ekkor a programozónak kell elvégeznie, de ez annak az ára, hogy itt nincs szükség minden elérni kívánt objektum interfészének fordítási időbeli ismeretére, illetve nincs szükség klienscsonk objektumokra sem ahhoz, hogy a különféle interfésszel rendelkező objektumok metódusait végrehajthassuk. Az "ideális" dinamikus hívás menete vázlatosan a következő:

1. Megszerzünk például egy névszolgáltatótól egy CORBA objektumreferenciát arra az objektumra vonatkozóan, amelyiknek a metódusát meg akarjuk hívni.

2. Megszerzünk egy referenciát az elérni kívánt objektum CORBA interfészét leíró `CORBA::InterfaceDef` objektumra (ez egy, a CORBA interfészgyűjteményben levő objektum). Ez alapján megismerhetjük az illető objektum által támogatott CORBA metódushívási mechanizmusokon keresztül elérhető metódusokat, valamint azok paraméterezését (a meghívni kívánt metódust például a neve alapján a `lookup_name()` metódussal találhatjuk meg, az egyes metódusok szignatúráját pedig a `describe()` metódussal kérdezhetjük le az interfészgyűjteményből).

3. A visszakapott információk alapján (vagy a meghívni kívánt metódus szignatúrájának ismerete alapján) össze kell állítani a meghívandó metódusnak átadandó paramétereket egy ún. névvel ellátott értékek listájában (ez egy CORBA pszeudo-objektum[11], amely a `CORBA::NVList` IDL interfészt implementálja). Ezt a - kezdetben még egyetlen elemet sem tartalmazó - listát az ORB szoftvert reprezentáló objektum `create_list()` metódusának meghívásával hozhatjuk létre, majd a

[11] A pszeudo-objektumok Java leképezésével kapcsolatban általánosságban annyit mondhatunk, hogy ezek nincsenek képviselve az interfészgyűjteményben, hozzájuk nem lesznek holder és helper osztályok generálva, és olyan `public abstract class` módosítóval ellátott Java osztályokra lesznek leképezve, amely osztályok nem bővítenek más osztályt.

lista objektumot például az `add_item()` metódusával tölthetjük fel az átadandó paraméterekkel (az `out` módú paramétereknek a már megismert módon kell csak helyet csinálni). A listába - mint azt majd látni fogjuk - `Any` típusú objektumokat kell felvenni (vagyis az átadandó adatok mellett azok típusát is meg kell majd jelölni).

4. Létre kell hozni egy, magát a metódushívást reprezentáló objektumot (az objektum a `CORBA::Request` interfészt implementálja, és e reprezentánsok a CORBA objektumok `_create_request()` metódusának meghívásával hozhatók létre).

5. Végre kell hajtani a metódushívást, az azt reprezentáló - és a 4. pontban létrehozott - objektum megfelelő metódusának meghívásával. Az erre a célra meghívható metódusok a következők:

   `invoke()` : szinkron metódushívás.

   `send_deferred()` : aszinkron metódushívás. A programszál futása nem várakozik a metódushívás befejeződéséig, hanem tovább folytatódik, és a program a későbbiekben a metódushívást reprezentáló objektum `poll_response()`, illetve a `get_response()` nevű metódusával kaphat információkat a végrehajtás eredményéről, visszatérési értékekről.

   `send_oneway()` : a metódushívás "legfeljebb egyszer" szemantikával lesz végrehajtva (lásd a korábban már említett `oneway` módosító szerepét).

A dinamikus hívás menetének ismertetésekor az "ideális" szócskát azért kellett kiraknunk, mert a Java IDL jelenlegi implementációja nem tartalmazza az interfészgyűjtemény elérésének támogatását, így a metódus paraméterezését más információk alapján kell megtennünk (például úgy, hogy valahonnan máshonnan már tudjuk, hogy az illető metódus milyen paramétereket vár ...).

A dinamikus metódushívás során a következő Java osztályokat használjuk fel (az előbb említett módon; megjegyezzük, hogy ezek az osztályok a megfelelő dinamikus metódushívási interfészek IDL specifikációjából lettek generálva): `NamedValue`, `NVList`, `Request`, `ORB`. Az ORB szoftver reprezentáló `CORBA::ORB` interfésszel a következő pontban foglalkozunk részletesen.

Először nézzük meg a `NamedValue` osztály szerkezetét! Az osztály egy dinamikus metódushívás egy-egy paraméterének leírására használható. Ezen osztály főbb komponensei a következők:

- A hívott metódus egy paraméterének a neve (a metódus OMG IDL nyelvű specifikációjában leírt paraméterének a nevéről van szó).

- A megnevezett paraméter értéke (emlékezzünk rá, hogy az `Any` típus az érték mellett annak típusát is hordozza).

- A megnevezett paraméter átadási módja, lehetséges értékei a `CORBA::ARG_IN`, `CORBA::ARG_OUT`, valamint a `CORBA::ARG_INOUT` nevű OMG IDL nyelven definiált értékek (ezek megfelelnek rendre az `in`, `out`, illetve az `inout` CORBA paraméterátadási módoknak).

```
package org.omg.CORBA;

public interface ARG_IN {
  int value = 1;
}

public interface ARG_OUT {
  int value = 2;
}

public interface ARG_INOUT {
  int value = 3;
}

public abstract class NamedValue extends Object {
  public abstract String name() throws SystemException;
  public abstract Any value() throws SystemException;
  public abstract int flags() throws SystemException;
}
```

Az `NVList` osztály `NamedValue` objektumok listáját reprezentálja.

```
public abstract class NVList extends Object {
  public abstract int count() throws SystemException;
  public abstract NamedValue add(int flags) throws SystemException;
  public abstract NamedValue add_item(String itemName,
                                      int flags) throws SystemException;
  public abstract NamedValue add_value(String itemName,
                                       Any val,
                                       int flags) throws SystemException;
  public abstract NamedValue item(int index) throws Bounds,
                                                    SystemException;
  public abstract void remove(int index) throws Bounds, SystemException;
}
```

A konstruktor egy üres listát hoz létre. A lista elemszámát a `count()` metódussal kérdezhetjük le. Az `add()` metódus létrehoz egy új `NamedValue` objektumot, és felfűzi a lista végére (a paraméterben a létrehozott objektumnak a paraméterátadási módot jelző részét kell megadni). Az `add_item()`, valamint az `add_value()` metódusok szintén egy új listaelemet fűznek a lista végére, míg az előbbi az új listaelemnek csak a nevét és a paraméterátadási módját várja a paramétereiben, addig az utóbbi az új listaelemhez tartozó értéket is. A lista egy adott sorszámú elemét az `item()` metódussal kérdezhetjük le, a metódus paraméterében kell megadni a kívánt elem indexét. A lista egy adott sorszámú elemét a `remove()` metódussal törölhetjük, a metódus paraméterben a törölni

kívánt elem indexét adjuk meg. A törlés után a törölt elemet követő listaelemek indexei eggyel csökkennek. Ha olyan objektumindexet adunk meg, amilyen indexszel nincs objektum a listában, akkor az illető metódusok egy `Bounds` kivételt váltanak ki (e kivétel specifikációját a `CORBA::Bounds` IDL egység tartalmazza).

A metódushívásokat a `CORBA::Request` osztály objektumai reprezentálják. Ennek az osztálynak a szerkezetét a következő listát követően ismertetjük.

```
public abstract class Request extends Object {
  public abstract Object target() throws SystemException;
  public abstract String operation() throws SystemException;
  public abstract NVList arguments() throws SystemException;
  public abstract NamedValue result() throws SystemException;
  public abstract Environment env() throws SystemException;
  public abstract ExceptionList exceptions() throws SystemException;
  public abstract ContextList contexts() throws SystemException;
  public abstract Context ctx() throws SystemException;
  public abstract void ctx(Context newCtx) throws SystemException;
  public abstract Any add_in_arg() throws SystemException;
  public abstract Any add_in_arg(String name) throws SystemException;
  public abstract Any add_inout_arg() throws SystemException;
  public abstract Any add_inout_arg(String name) throws SystemException;
  public abstract Any add_out_arg() throws SystemException;
  public abstract Any add_out_arg(String name) throws SystemException;
  public abstract void set_return_type(TypeCode tc) throws SystemException;
  public abstract Any return_value() throws SystemException;
  public abstract void invoke() throws SystemException;
  public abstract void send_oneway() throws SystemException;
  public abstract void send_deferred() throws SystemException;
  public abstract boolean poll_response() throws SystemException;
  public abstract void get_response() throws SystemException;
}
```

A `Request` osztály példányaival reprezentálhatunk metódushívásokat. A konstruktora egy üres kérésobjektumot hoz létre (bár e metódushívási kéréseket reprezentáló objektumokat általában a CORBA objektumok `_create_request()` metódusával szokták létrehozni). A `target()` metódus visszaadja annak a CORBA objektumnak a referenciáját, amelynek a metódusát meg fogjuk hívni (azaz annak a CORBA objektumnak a referenciáját, amelynek az előbb említett `_create_request()` nevű metódusát meghívták). Az `operation()` metódus visszaadja a meghívni kívánt metódus nevét. A metódushívás során a metódusnak átadandó paramétereket egy `NVList` osztálybeli objektumban kell összegyűjteni; ezen objektumra vonatkozó referenciát az `arguments()` metódussal kaphatunk vissza. A metódus visszatérési értékét tartalmazó `NamedValue` osztályba tartozó objektumra hivatkozó referenciát a `result()` metódussal szerezhetünk meg.

A metódusoknak a paraméterük mellett átadhatunk információkat az ún. hívási kontextuson keresztül is (megjegyezzük, hogy ennek semmi köze a névszolgáltatónál megismert kontextusokhoz). E hívási kontextusok az operációs rendszer környezeti változóihoz

hasonló szerepet látnak el: különféle azonosítókhoz rendelhetünk benne értéket, amit a hívó megkapva bármilyen célra felhasználhat. Az alkalmazás a `contexts()` metódussal juthat hozzá a metódushívás során átadandó hívási kontextusok nevének listájához, amit egy `ContextList` osztályba tartozó objektum reprezentál, és az alkalmazás ezen osztály metódusaival manipulálhatja e lista tartalmát. A `ctx()` metódusokkal állíthatjuk be, illetve kérdezhetjük le a metódushívás hívási kontextusának a tartalmát (ez hívási kontextusok neveit és a hozzájuk tartozó értékeket tartalmazza). Az `add_in_arg()`, `add_out_arg()`, `add_inout_arg()` metódusokkal fűzhetünk a metódushívás során átadandó paraméterek értékeit tartalmazó listához egy új elemet. E metódusok a lista végére fűzik az új paramétereket. E metódusok nevei tartalmazzák az illető paraméter átadási módját (`in`/`out`/`inout`), valamint e metódusok egyetlen szöveges paraméterében megadható az illető paraméter neve (a paraméternek az interfész IDL specifikációjában használt neve). Ezen metódusok visszatérési értéke egy `Any` osztálybeli objektum, amely a paraméterlistához hozzáfűzött új elemet tartalmazó holder osztályt reprezentálja.

A `set_return_type()` metódussal a CORBA dinamikus metódushívás visszatérési értékének típuskódját lehet beállítani (egy `TypeCode` osztályba tartozó objektumként kell ezt az információt a paraméterben megadni). Magához a visszatérési értékhez a `return_value()` metódussal férhetünk hozzá.

Magát a dinamikus metódushívást többféleképpen is kezdeményezhetjük. A paraméterekkel feltöltött kérést reprezentáló objektum `invoke()` metódusával kezdeményezhetünk egy szinkron metódushívást, amelynek a befejeződéséig a metódust végrehajtó programszál várakozik. Az `invoke()` metódus visszatérésekor - a végrehajtó programszál továbbfutása után - a kimenő paraméterek értékei, valamint a dinamikusan hívott metódus visszatérési értéke a dinamikusan meghívott CORBA objektum végrehajtott metódusa által visszaadott értékekre lesznek beállítva. A "legfeljebb egyszer" szemantikával végrehajtott dinamikus metódushívást a `send_oneway()` metódussal kezdeményezhetjük, míg az aszinkron dinamikus metódushívást a már említett `send_deferred()` metódussal kezdeményezhetjük. Az aszinkron metódushívás esetében a kimenő paraméterek és a visszatérési értékek egészen addig nem állnak a program rendelkezésére, amíg meg nem hívta a `get_response()` metódust, és az vissza nem tért (ez a metódus csak azután tér vissza, miután a dinamikusan meghívott távoli metódus futása befejeződött; e befejeződés tényéről a `poll_response()` metódussal győződhetünk meg: ha e metódus logikai igaz értékkel tér vissza, akkor bizonyosak lehetünk benne, hogy az illető távoli CORBA metódus futása már befejeződött, egyébként nem).

Hátramaradt még az említett `Context` és `ContextList` osztályok ismertetése. A `Context` osztály nevek és hozzájuk rendelt karakterlánc típusú értékpárokat tárolnak. E hívási kontextusoknak egy fa szerkezetű struktúráját hozhatjuk létre, és a hívási kontextusokban tárolt értékekre ekkor minősített nevükkel hivatkozhatunk; a csillag (*) karaktert használhatjuk a keresés során történő összehasonlításkor tetszőleges karaktersorozattal illeszkedő "joker" karakterként.

```
public abstract class Context {
  public abstract Context parent();
  public abstract Context create_child(String gyermek_kontextus_neve);
  public abstract String context_name();
  public abstract void set_one_value(String név, Any érték);
  public abstract void set_values(NVList értékek);
  public abstract void delete_values(String név_keresési_minta);
  public abstract NVList get_values(String kezdőpont, int jelző,
                                    String keresési_minta);
}
```

A `parent()` metódus visszaadja a hívási kontextusok fa szerkezetű hierarchiájában a szülő hívási kontextust reprezentáló objektum referenciáját (annak a hívási kontextusnak a szülőjét, amelyre vonatkozóan e metódust meghívták). A `create_child()` metódus létrehoz egy új hívási kontextust a paraméterében megadott névvel, amely a hívási kontextusok fa szerkezetű hierarchiájában az alá a hívási kontextus alá tartozik, amelyre vonatkozóan ezt a metódust meghívták. Egy hívási kontextus nevét a `context_name()` metódussal kérdezhetjük le. Egy (név,érték) párt egy hívási kontextusba a `set_one_value()` metódussal vehetünk fel; (név,érték) párok egy sorozatát pedig a `set_values()` metódussal (a felvenni kívánt adatokat egy `NVList` objektumban kell összeállítani). Egy adott névhez rendelt értéket a `delete_values()` metódussal törölhetünk egy kontextusból, a metódus paraméterében kell megadni a törölni kívánt nevet. Egy hívási kontextusból a hívási kontextusban tárolt azonosítókhoz rendelt értékeket a `get_values()` metódussal kérdezhetjük le (figyeljünk a többes számra! több kontextus neve is illeszkedhet egy adott név sémára), a keresett nevet a harmadik paraméterben kell megadni (ha ez a név csillag karakterre végződik, akkor az összes olyan névhez rendelt értéket lekérdezi, amely a csillagot megelőző alfanumerikus karaktersorozattal kezdődik). Az első paraméterben adhatjuk meg, hogy mely hívási kontextusokban akarjuk végezni a keresést (ha a keresett név nem található a megadott hívási kontextusban, akkor a keresés a szülő kontextusban folytatódik, addig amíg az első paraméterben megadott nevű kontextusig el nem ér a kontextus-hierarchiában). A második paraméterben vagy nullát, vagy a `CTX_RESTRICT_SCOPE` konstanst adhatjuk meg; ez utóbbi konstans megadása esetén a keresés korlátozva van a paraméterben megadott nevű hívásikontextus-objektumra.

A `ContextList` osztályt arra szokták használni, hogy megnevezzék a dinamikus metódushívás esetén a hívott metódusnak átadandó hívási kontextusokat: egyszerűen karakterláncok sorozatát tartalmazza (az átadandó kontextusok neveit).

```
public abstract class ContextList extends Object {
  public abstract int count() throws SystemException;
  public abstract void add(String context) throws SystemException;
  public abstract String item(int index) throws SystemException, Bounds;
  public abstract void remove(int index) throws SystemException, Bounds;
}
```

A konstruktor létrehoz egy üres listát. A `count()` metódus visszaadja a listában levő elemek számát. Az `add()` metódus egy új elemet fűz fel a listába a metódus paraméterében megadott néven. Az `item()`, valamint a `remove()` metódusokkal rendre megszerezhetjük, illetve letörölhetjük egy adott sorszámú listaelem tartalmát. E metódusok `Bounds` kivételt generálnak, ha nem létező indexű listaelemre akarunk hivatkozni.

Már sokat megismerhettünk a dinamikus metódushívás eszközéről, ideje megnézni egy egyszerű programrészleten azt, hogy hogyan történik mindez a gyakorlatban! Ehhez egy nagyon egyszerű interfészt implementáló objektumon végignézzük, hogy hogyan is történhet meg az illető objektum egy metódusának a dinamikus meghívása. A példában használt interfész neve legyen mondjuk `Negyzetreemelo`, IDL specifikációja pedig - a `Negyzetre` nevű modulban - a következő:

```
module Negyzetre
{
  interface Negyzetreemelo {
    long emel(in long mit);
  };
};
```

Megjegyezzük, hogy ez a példa egy olyan CORBA objektumot definiál, amely nem rendelkezik belső állapottal. Egy négyzetreemelő metódus Java nyelven természtes módon lenne elkészíthető egy `static` módosítóval ellátott osztálymetódus formájában. A fenti példa azonban elég egyszerű ahhoz, hogy ismertetésekor a megoldást támogató mechanizmusokkal foglalkozhassunk, ne pedig a szolgáltatás részleteivel kelljen törődnünk. Később megismerhetünk egy ennél összetettebb szolgáltatás implementációjával is.

A következő egyszerű Java alkalmazás az inicializálása (a megfelelő ORB referencia megszerzése) után összeállít egy `Request` objektumot, majd egy adott nevű CORBA objektumon meghívja a fenti interfész `emel` nevű metódusát (a kívánt CORBA objektumot a CORBA névszolgáltatójából az előbbi pontokban már ismertetett módon keresi ki, a keresett objektum neve: `Negyzetreemelo_objektum`).

```
// NegyzetreemeloTeszt.java
//
// Egy kliens alkalmazás a fenti Negyzetreemelo, IDL nyelven specifikált
// szolgáltatáshoz.

import Negyzetre.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class NegyzetreemeloTeszt {

  public static void main(String args[]) {
```

```
   int mit_emeljen_négyzetre = 5;

   try {
    ORB orbref = ORB.init(args, null); // paraméter: főprogram argumentumai
    org.omg.CORBA.Object nr =
                      orbref.resolve_initial_references("NameService");
    NamingContext ncref = NamingContextHelper.narrow(nr);
    NameComponent nc = new NameComponent("Negyzetreemelo_objektum", "");
    NameComponent keresett_név[] = {nc};
    Negyzetreemelo neref = NegyzetreemeloHelper.narrow(ncref.resolve(
                          keresett_név)); // dinamikus hívás 1. pontja

    // összeállítjuk a bemenő paramétereket tartalmazó listát
    NVList hívási_paraméterek = orbref.create_list(0); // hívás 3. Pontja
    Any mit_emeljen_négyzetre_tartó = orbref.create_any();
    mit_emeljen_négyzetre_tartó.insert_long(mit_emeljen_négyzetre);
    hívási_paraméterek.add_value("mit", mit_emeljen_négyzetre_tartó,
                                 ARG_IN.value); // felvesszük a listába ...

    // összeállítjuk a visszatérési értéket tartalmazó elemet
    Any eredmény = orbref.create_any();
    eredmény.type(orbref.get_primitive_tc(TCKind.tk_long));
    NamedValue eredményNV = orbref.create_named_value(null, eredmény, 0);

    // összeállítjuk a dinamikus hívást reprezentáló objektumot (4. pont)
    Request metódushívás = neref._create_request(null, "emel",
                                 hívási_paraméterek, eredményNV);

    // meghívjuk a metódust ... szinkron módon ... (ez az 5. pont)
    metódushívás.invoke();

    // ezután már hozzáférhetünk a visszatérési értékhez egy
    // "eredmény.extract_long()" metódushívással ... (long típusú!!!)
    System.out.println(new Integer(mit_emeljen_négyzetre).toString()+
                  " négyzete = "+new Integer(eredmény.extract_long()));

   } catch (Exception e) {
    System.out.println("Kivételt jeleztem a végrehajtás végén!");
   }
  }
}
```

A fenti program nem csinál mást, mint négyzetre emeli a program elején levő `mit_emeljen_négyzetre` egész típusú változó tartalmát, a példában ennek értéke 5. Most pedig vázoljuk a program szerkezetét, működésének főbb lépéseit!

- A program kezdetben inicializálja az ORB szoftvert, és megszerez egy referenciát a CORBA névszolgáltatóból a `Negyzetreemelo_objektum` nevű CORBA objektumra vonatkozóan, és ezt a referenciát eltárolja a `neref` nevű változóba.
- Ezután az ORB `create_list()` metódusával létrehoz egy - kezdetben nulla elemet

tartalmazó - paraméterlista objektumot.

- A program ezután létrehoz egy `Any` típusú objektumot, az objektum neve `mit_emeljen_négyzetre_tartó`.
- A létrehozott `Any` típusú objektumba berakja a négyzetre emelendő számot, az 5-öt (`long` típusú értéket).
- A paraméterek listájába felfűzi az előbb létrehozott `Any` típusú objektumot, bemenő (CORBA `in` módú) paraméternek, jelölve, hogy az átadott értéket a hívott metódus IDL specifikációjában megadott `mit` paraméterének kívánja átadni.
- Ezután a program létrehoz egy újabb `Any` típusú objektumot, amin keresztül majd lekérdezhető a hívott CORBA metódus visszatérési értéke.
- Az újonnan létrehozott `Any` típusú objektum `type()` metódusával jelzi, hogy a visszatérési értékként `long` típusú értéket vár vissza.
- Az ORB `create_named_value()` metódusával létrehoz egy `NamedValue` osztályba tartozó objektumot, amely az újonnan létrehozott `Any` típusú objektumot tartalmazza.
- A távoli CORBA objektumra hivatkozó referencia `_create_request()` metódusával létrehoz egy dinamikus metódushívást reprezentáló `Request` objektumot, ahol a `_create_request()` metódusnak átadott paraméterek rendre a következők:
  1. A hívás során átadandó CORBA hívási kontextusra hivatkozó referencia. A példában itt `null` értéket adtunk meg, mivel a metódus IDL specifikációjában nem adtunk meg átvinni kívánt hívási kontextus értékeket.
  2. A második paraméternek karakterlánc formájában kell tartalmaznia a dinamikusan meghívni kívánt metódus nevét (esetünkben ennek neve: `emel`).
  3. A harmadik paraméterben a dinamikusan meghívni kívánt metódusnak átadandó paramétereket tartalmazó listát kell megadni.
  4. A negyedik paraméterben a meghívott CORBA metódus visszatérési értékének kijelölt helyet kell megadni.
- Ezután a program meghívja a dinamikus metódushívást reprezentáló `Request` osztályba tartozó objektum `invoke()` metódusát, végrehajtva ezzel az eddig előkészített távoli CORBA objektum `emel` metódusának szinkron módú meghívását.
- A metódus visszatérésekor a távoli CORBA metódus által visszaadott visszatérési érték az eredménynek kijelölt `NamedValue` osztályba tartozó objektumban lesz, amit az alkalmazás az illető listába felfűzött `Any` osztályú objektum `extract_long()` metódusával szedhet ki onnan.

Figyeljük meg, hogy itt a program vezérlésébe, szerkezetébe be van építve a hívott metódus neve, és annak paraméterezése (ui. a programban létrehozott paraméterlista mindig csak azt az egy elemet fogja tartalmazni). Ezt azért tettük így, mert a Java IDL jelenleg nem támogatja az interfészgyűjtemény elérését.

3.8. Az ORB szoftvert reprezentáló objektum

A már említett, ORB szoftvert reprezentáló objektum metódusait és azok paraméterezését a következő osztályspecifikáció tartalmazza. Megjegyezzük, hogy - legalábbis jelenleg - az itt bemutatott metódusok tekinthetők a Java-alapú ORB szoftvert reprezentáló osztály szabványosított metódusainak; az ORB gyártók ezeken túl tetszésük szerint bővíthetik az ORB szoftverük által nyújtott metódusok halmazát (bár ezen bővítések kihasználása valószínűleg nem hordozható alkalmazásokat fog eredményezni). Érdemes megemlíteni azt is, hogy jelenleg - ahogy azt majd látni fogjuk - az objektumadapter szerepeket is néhány itt definiált metódus látja el.

```
// részletek a Java nyelvi leképezésből

public abstract class ORB {
  String[] list_initial_services();
  org.omg.CORBA.Object resolve_initial_references(String objektum_neve)
                       throws org.omg.CORBA.ORBPackage.InvalidName;
  String object_to_string(org.omg.CORBA.Object objektum_referencia);
  org.omg.CORBA.Object string_to_object(String szöveges_referencia);
  NVList create_list(int elemszám);
  NVList create_operation_list(OperationDef metódus_intfgyűjtbeli_neve);
  NamedValue create_named_value(String név, Any érték, int jelzők);
  ExceptionList create_exception_list();
  ContextList create_context_list();
  Context get_default_context();
  Environment create_environment();
  void send_multiple_requests_oneway(Request[] kérések);
  void sent_multiple_requests_deferred(Request[] kérések);
  boolean poll_next_response();
  Request get_next_response();
  TypeCode create_struct_tc(String azonosító, String név,
                            StructMember[] rekord_típus_tagjai);
  TypeCode create_union_tc(String azonosító, String név,
           TypeCode diszkrimináns_típusa, UnionMember[] tagok);
  TypeCode create_enum_tc(String azonosító, String név,
           EnumMember[] tagok);
  TypeCode create_alias_tc(String azonosító, String név, // typedef
           TypeCode álnév_mögötti_eredeti_típus_kódja);
  TypeCode create_exception_tc(String azonosító, String név,
           StructMember[] tagok);
  TypeCode create_interface_tc(String azonosító, String név);
  TypeCode create_string_tc(int korlát);
  TypeCode create_wstring_tc(int korlát);
```

```
  TypeCode create_sequence_tc(int korlát, TypeCode elemtípusa);
  TypeCode create_recursive_sequence_tc(int korlát, int mettől);
  TypeCode create_array_tc(int hossz, TypeCode elemtípusa);
  Current get_current();
  TypeCode get_primitive_tc(TCKind típuskód);
  Any create_any();
  org.omg.CORBA.portable.OutputStream create_output_stream();
  void connect(org.omg.CORBA.Object objektum_referencia);
  void disconnect(org.omg.CORBA.Object objektum_referencia);
  // ORB inicializáló metódusok
  public static ORB init(Strings[] prog_paraméterek, Properties jellemzők);
  public static ORB init(Applet applet_referencia, Properties jellemzők);
  public static ORB init();
}
```

A fenti listában látható metódusok a megjelölt `init()` metódusok kivételével mind `public abstract` módosítóval vannak deklarálva, de ezt a listában nem jelöltem, mivel úgy a tördelést sokkal kevésbé átlátható módon lehetett volna elkészíteni.

Az ORB szoftver legelső inicializálását az `init()` statikus metódussal végezhetjük. Három változata van, amelyek a következő jellemzőikben különböznek egymástól.

1. A paraméter nélküli metódus mindig ugyanazt az inicializált ORB objektumot adja vissza. Java alkalmazásokból hívva egy teljeskörű ORB szolgáltatásokat nyújtó ORB objektumot kapjuk vissza, míg appletekből meghívva csak a típuskód objektumok létrehozására alkalmas ORB objektumot kapunk vissza (az appletekre vonatkozó szigorú biztonsági megkötések miatt, hiszen a visszaadott ORB objektumot az összes applet megosztva használhatja; appletekkel kapcsolatban lásd a további pontokat is!). Ennek feladata elsősorban a helper osztályok `type()` metódusának támogatása új típuskódok létrehozásakor (erre hamarosan egy példát is láthatunk).

2. Java alkalmazások használhatják az `init()` metódus másik - két paraméteres - változatát, amelynek első paraméterében az alkalmazás indításakor használt programparamétereket kell átadni, második paraméterben pedig Java környezeti jellemzőket (`java.util.Properties` típusú) objektumot adhatunk át. A Java-futtató rendszer ezen helyekről kigyűjti az ORB inicializálásához szükséges információkat, és eszerint végzi az ORB inicializálását (például azt, hogy milyen protokollal, milyen kommunikációs végpont azonosítóval akarjuk az ORB kommunikációs végpontját ellátni). Ez a metódus minden egyes meghívásakor egy új teljes körű ORB szolgáltatásokat nyújtó ORB objektumpéldányt hoz létre, és az erre vonatkozó referenciát adja vissza.

3. A Java appletek más formában kapják meg a paramétereiket (ehhez például az illető appletet reprezentáló `java.applet.Applet` osztályba tartozó objektum `getParameter()` metódusát lehet használni), ezért az appletek az ORB inicializálására az `init()` metódusnak egy olyan változatát használhatják, amelynek az első

paraméterében az inicializálást kérő applet objektum referenciáját kell megadni (az illető `java.applet.Applet` osztálytól származtatott objektum a `this` speciális referenciával hivatkozhat önmagára), a második paraméterében pedig az előző pontban megismertekhez hasonlóan Java környezeti jellemzők adhatók meg. Az ORB inicializálása itt is a paraméterekben kapott információk alapján történik, a metódus minden egyes hívásakor egy-egy teljes körű ORB szolgáltatásokat nyújtó ORB objektum példány lesz létrehozva.

Az ORB inicializálásához szükséges információkat az `init()` metódus a következő helyekről a következő sorrendben veszi (ha egy adott inicializálási paramétert a program több helyen is megtalálna, akkor az alábbi sorrend rögzíti az egyes helyek prioritását).

(a) Az alkalmazás vagy az applet paraméterei (az `init()` metódus első paramétere).

(b) Az `init()` metódus második paraméterében megadott környezeti jellemzők.

(c) A Java rendszer környezeti jellemzőit (legkisebb prioritású, azaz ha egy adatot az előző két hely valamelyikén már megtalált, akkor itt nem nézi).

A Java IDL ORB szoftver jelenleg a következő paraméterek értékét veszi figyelembe a rendszer inicializáláskor (ha e paraméterek értéke a fenti helyek valamelyikén nincs megadva, akkor a Java-futtató rendszer egy implementációfüggő alapértelmezés szerinti értéket használ; megjegyezzük, hogy Java alkalmazások esetén az egyes jellemzők parancssorban történő definiálása esetén a név `org.omg.CORBA` prefixét el kell hagyni):

`org.omg.CORBA.ORBClass` : Az `org.omg.CORBA` interfészeket implementáló Java osztály neve. Az alkalmazások itt adhatják meg az általuk használt ORB implementációt tartalmazó Java osztály nevét (a futás elején ennek az inicializáló metódusa lesz végrehajtva ...).

`org.omg.CORBA.ORBInitialHost` : Az alkalmazás által használni kívánt CORBA névszolgáltatót - és egyéb CORBA inicializálási szolgáltatásokat - futtató számítógép neve. Alapértelmezés szerint az alkalmazások a futtató számítógépen elérhető inicializálási szolgáltatásokat, míg az appletek a letöltési helyükön[12] levő szolgáltatókat érik el.

`org.omg.CORBA.ORBInitialPort` : Az inicializálási szolgáltató eléréséhez használandó kommunikációs végpont azonosítója (alapértelmezés szerinti értékként általában 900-at használják).

[12] Emlékezzünk rá, hogy egy applet letöltési helyére a `getCodeBase().getHost()` módon hivatkozhatunk.

Tekintsük az alábbi programrészletet, amelyben az inicializáláskor megadjuk a kívánt ORB osztályának a nevét. A példa egy Java applet részlete, így az annak megfelelő `init()` metódust használjuk[13]!

```
public class TeszApplet1 extends java.applet.Applet {

  public void init() {
    Properties p = new java.util.Properties();
    p.put("org.omg.CORBA.ORBClass","ORBosztályneve");
    ORB orbRef = ORB.init(this, p);
    //
    // Egyéb inicializáló részek
    // ...
  }

}
```

Az objektumadapter szolgáltatások közül az újonnan létrehozott CORBA objektum példányoknak az ORB szoftvernél történő bejegyzésére, illetve az ottani nyilvántartásból történő törlésére a `connect()`, illetve a `disconnect()` metódust használhatjuk. E metódusok paraméterében az illető - újonnan létrehozott - CORBA objektumra vonatkozó metódushívási kérelmek kiszolgálásáért felelős ún. kiszolgáló objektum (helyi) Java referenciáját kell megadni.

A CORBA névszolgáltató elérését bemutató pontban már említettük az ORB osztály `resolve_initial_references()` nevű metódusát, amellyel egy CORBA kliens (vagy szerver) alkalmazás elindulása után hozzájuthat a környezetében elérhető néhány fontos CORBA objektum referenciájához. Egyetlen szöveges típusú paraméterében kell megnevezni azt a CORBA objektumot (általában valamilyen rendszerszolgáltatást nyújtó CORBA objektumot), amelynek az objektumreferenciáját meg akarjuk kapni. E metódus a visszatérési értékében adja vissza a paraméterében megnevezett objektum vagy szolgáltatás CORBA objektumreferenciáját, illetve egy `InvalidName` kivételt vált ki, ha a keresett nevű szolgáltatás nincs megnevezve a kezdeti szolgáltatások adatbázisában. Jelenleg a CORBA névszolgáltatás az egyetlen itt megnevezett szolgáltatás, és erre

[13] A programban az `org.omg.CORBA.ORBClass` jellemző értékének annak az ORB osztálynak a nevét adjuk meg, amelyet a programunkban használni akarunk. Ha például a JOE ORB-t akarjuk használni, akkor a `com.sun.CORBA.iiop.ORB` osztálynevet adjuk meg. Vannak olyan web-böngésző programok, amelyekbe beépítettek egy CORBA ORB-t (például a Netscape böngészőibe a VisiBroker ORB-k vannak beépítve). Ez a módszer akkor lehet különösen érdekes, ha egy web-böngészőben a beépített ORB szoftver helyett egy másik ORB szoftvert akarunk használni. Felhívjuk a figyelmet arra, hogy ha egy applet egy olyan ORB osztályra hivatkozik, amely nincs beépítve az appletet futtató böngészőbe, akkor a böngésző a futáshoz szükséges osztályokat megpróbálja letölteni az appletet tároló számítógép HTTP-szerveréről, az applet letöltési helyéről ugyanúgy, mint az applet működéséhez szükséges egyéb osztályokat. Ha például a JOE ORB-vel készítünk el egy appletet, és arra számítunk, hogy azt a Netscape böngészőivel is futtatni akarják, akkor feltehetjük a JOE ORB-t alkotó osztályokat az applet letöltési helyén az applet mellé, és amikor az alkalmazás inicializáláskor a már említett `com.sun.CORBA.iiop.ORB` osztályt próbálja meg letölteni, akkor a letöltés az appleteknél megismert osztályletöltési mechanizmusokkal fog végbemenni. Ugyanez nemcsak a Netscape böngészőire igaz, hanem mindazokra, amelyekbe a JOE ORB nincs beépítve.

a `NameService` szöveges névvel hivatkozhatunk. A kezdetben elérhető szolgáltatások neveinek listáját a `list_initial_services()` metódussal kaphatjuk vissza egy karakterláncokból álló tömbben.

A `resolve_initial_references()` metódus paraméterében megadhatjuk akár az `InterfaceRepository` szövget is. Ekkor hozzájutunk a szerveroldalon használt interfészgyűjteményt reprezentáló CORBA objektum referenciájához (persze csak ha van ilyen, ugyanis például a JOE ORB szerveroldala egyelőre nem támogatja ezt a lehetőséget, de JOE ORB-vel működő kliensekből hozzáférhetünk más ORB-vel működő szerveroldali interfészgyűjteményekhez.

Felmerülhet a kérdés, hogy mi a teendő, ha egy alkalmazás két névszolgáltatót is el akar érni. A válasz erre a kérdésre is egyszerű: példányosíthatunk akár két ORB objektumot is, az egyikkel az egyik névszolgáltatót elérve, a másikkal a másik névszolgáltatót elérve. Mivel a program paramétereiben csak egy elérni kívánt névszolgáltatót nevezhetünk meg (az `ORBInitialHost` és `ORBInitialPort` paraméterekkel), ezért a további névszolgáltatók elérési paramétereit az új ORB-t létrehozó `init()` metódusok `java.util.Properties` típusú paramétereiben adhatjuk meg (és az ismertetett prioritási sorrend miatt a programparamétereket ne is adjuk meg, ha az erre vonatkozó paraméterek még nem lettek kitörölve).

Mivel két vagy több ORB létrehozása kicsit nehézkes folyamat, ezért azt tanácsoljuk, hogy ezt ne követeljük meg minden alkalmazástól. Ehelyett alakítsunk ki egy olyan névszolgáltató-struktúrát, ahol egyetlen olyan több ORB-t elérő program van, amely az egyes névszolgáltatók IOR-referenciáit bejegyzi a többi névszolgáltatóba[14], ezzel lényegében összekapcsolva azokat, és az alkalmazás ennek ismeretében már kényelmesen elérheti bármely névszolgáltatóban tárolt adatokat egyetlen ORB segítségével.

Már említettük, hogy egy CORBA objektumreferenciát szöveges reprezentációjára az `object_to_string()` metódussal alakíthatunk, míg a fordított irányú konverziót a `string_to_object()` metódussal végezhetjük. Egy objektumreferencia karakterlánc reprezentánsának előállítását a következő programrészlet szemlélteti[15]:

```
String szövegesRef = orbRef.object_to_string(CORBA_referencia);
```

Ha a referenciát például ki akarjuk írni egy fájlba, azt megtehetjük a következőképpen:

```
OutputStream os = new FileOutputStream(
                        System.getProperty("user.home")+
                        System.getProperty("file.separator")+
                        "IORfileneve.ior"); // tetszőleges fájlnév
DataOutputStream dos = new DataOutputStream(os);
String ior = orbRef.object_to_string(CORBA_referencia);
dos.writeBytes(ior);
```

[14] Egy névszolgáltató bejegyezhető egy másikba akár egy új kontextusként is, így az egyes névtartományok teljesen összekapcsolhatók.

[15] Az `orbRef` változóban az ORB objektumra hivatkozó referencia van.

```
dos.close();
os.close();
```

A fordított irányú konverzió szemléltetésére pedig tekintsük a következő példát (ne felejtsük el, hogy a CORBA típuskényszerítési eszközeivel ezt a referenciát megfelelő típusúra kell konvertálni, mert így még csak `CORBA::Object` osztálybeli objektumként használhatnánk):

```
org.omg.CORBA.Object CORBA_referencia = orbRef.string_to_object(szövegesRef);
```

A dinamikus metódushívási interfész ismertetésekor említettük a `CORBA::NVList` objektumok szerepét, mint a dinamikusan meghívott metódus számára átadandó paraméterértékek kijelölésére szolgáló helyet. Egy ilyen lista létrehozását az ott már említett `create_list()` metódussal végezhetjük, a metódus paraméterében kell megadni a létrehozandó lista elemeinek számát (a későbbiekben a listához még fűzhetünk további elemeket is). A `create_operation_list()` metódussal az előbbihez hasonlóan egy ilyen listát hozhatunk létre, de a lista elemeinek a számát a `create_operation_list()` metódus paraméterében megadott metódus által várt paraméterek számához igazítja (a paraméterben a kívánt metódust az azt leíró `OperationDef` struktúrával kell megadni, amelynek a szerkezetével nem foglalkozunk). Egy - szintén a dinamikus metódushívási interfész ismertetésekor említett - `NamedValue` objektumot, a `create_named_value()` metódussal hozhatunk létre, a metódus paramétereinek részletes ismertetését lásd a dinamikus metódushívást bemutató részben.
A `send_multiple_request_deferred()` és a `send_multiple_requests_oneway()` metódusok is a dinamikus metódushívási interfésszel kapcsolatosak: ezek az ott említett `send_deferred()`, illetve `send_oneway()` metódusoknak olyan változatai, amelyek több dinamikus metódushívási kérés elküldését is meg tudják oldani (e metódusok paramétereiben kell megadni a dinamikus metódushívásokat reprezentáló objektumokat; felhasználhatók akár többszörözött objektumpéldányok elérésekor is, igaz csak úgy, hogy meg kell adjuk az összes objektumpéldányra vonatkozóan az egyes objektumpéldányok elérését reprezentáló kérésobjektumot; a CORBA önmagában nem támogatja többszörözött objektumpéldányok létrehozását és kezelését). Ilyen esetben a `poll_next_response()` metódussal kaphatunk vissza információkat arról, hogy ezen (késleltetett) metódushívások bármelyikére érkezett-e már válasz. Ha ez a metódus egy logikai igaz visszatérési értékkel azt jelzi, hogy érkezett már válasz, akkor az érkezett válaszról a `get_next_response()` metódussal tudhatunk meg többet (ez visszaadja az azt reprezentáló `Request` objektumot).

A `create_any()` metódus létrehoz és visszaad egy új IDL `Any` típusú objektumot, amely kezdetben a `TCKind.tc_null` típuskóddal van inicializálva.

A `create_exception_list()`, `create_context_list()` és a `create_environment()` metódusok létrehoznak rendre egy `ExceptionList`, `ContextList`, illetve `Environment` osztályba tartozó objektumot. E három metódus közül a `ContextList` szerepét láttuk; az

`ExceptionList` objektumot a dinamikus metódushívási interfész egy IDL művelet által kiváltható kivételek listájának tárolására használja (`count` adattagjával kérdezhető le e lista elemeinek a száma, `add()`, illetve `remove()` metódusával rendre hozzáadhatunk, illetve leszedhetünk egy kivételt reprezentáló objektumot ebből a listából (a hozzáadandó kivételnek a `TypeCode` reprezentánsát, míg a törlendő kivételnek a listabeli indexét kell megadni); az `item()` metódussal pedig visszakaphatjuk a lista egy adott indexű elemében tárolt kivétel `TypeCode` reprezentánsát); az `Environment` osztályba tartozó objektumok pedig egy metódushívás során kiváltott kivételekről tárolnak információkat (például az aszinkron dinamikus metódushívás esetén kifejezetten zavaró lenne, ha a program a metódushívás befejeződésekor váratlanul egy kivételt jelezne; ehelyett egy `Environment` osztálybeli objektum `exception()` metódusaival[16] lehet beállítani és lekérdezni a meghívott metódus által kiváltott kivételt; a beállítandó kivételt reprezentáló `TypeCode` objektumot a metódus paraméterében kell átadni, míg a kiváltott kivétel lekérdezését a paraméter nélküli metódus végzi, és a kiváltott kivétel típuskódját a visszatérési értékében adja meg; a `clear()` metódus pedig törli a tárolt típuskód-információt). A `get_default_context()` metódus visszaadja az alapértelmezés szerinti hívásikontextus-objektumot.

```
package org.omg.CORBA;

public abstract class Environment {
  void exception(java.lang.Exception kiv);
  java.lang.Exception exception();
  void clear();
}

public abstract class ExceptionList {
  public abstract int count();
  public abstract void add(TypeCode exc);
  public abstract TypeCode item(int index)
    throws org.omg.CORBA.Bounds;
  public abstract void remove(int index)
    throws org.omg.CORBA.Bounds;
}
```

A többi (nevében `_tc` karakterekre végződő) metódus (például a példaprogramjainkban korábban már használt `get_primitive_tc()` metódus) szerepe az attribútumok, illetve a paraméterek típusait reprezentáló újabb típuskód objektumok létrehozása. Az IDL fájlokban specifikált adattípusokat reprezentáló típuskódot az IDL fordító generálja. Néhány - felhasználói alkalmazásokban ritkán előforduló - esetben futás közben új típuskódokat kell generálni (például különböző, ORB feladatokat ellátó szoftverek közötti híd kialakításakor). Ilyenkor használhatóak ezek a metódusok az új típuskódok létrehozására, a metódusok paraméterében az új típusról kell további információkat

[16] Megjegyezzük, hogy a Java IDL korábbi kiadásában `except()` volt az itt bemutatott `exception()` metódus neve. A C++ nyelvi leképezéshez igazodva azonban áttértek az `exception()` névre.

megadni. Mivel ezekkel a metódusokkal a legtöbb Java/CORBA alkalmazásnál nem fogunk találkozni, ezért ezekkel a továbbiakban egy egyszerű példától eltekintve nem fogunk foglalkozni.

Az alábbi példában megmutatjuk egy IDL konstrukcióhoz a hozzá tartozó típuskód felépítésének lépéseit. Tekintsük az alábbi IDL deklarációt.

```
struct Datum {
  short Ev;
  short Honap;
  short Nap;
};
```

A fent definiált rekord típushoz tartozó típuskódot a következő programrészlet állítja elő.

```
import org.omg.CORBA.*;

TypeCode DatumTC; // ebben hozzuk létre a típuskódot
ORB orbRef;       // az ORB szoftvert reprezentáló objektum

... // egyebek

StructMember[] tagok = null;
tagok = new StructMember[3];
tagok[0] = new StructMember( // "Ev" rekordkomponens típuskódja
  "Ev", orbRef.init().get_primitive_tc(TCKind.tk_short), null);
tagok[1] = new StructMember(
  "Honap", orbRef.init().get_primitive_tc(TCKind.tk_short), null);
tagok[2] = new StructMember(
  "Nap", orbRef.init().get_primitive_tc(TCKind.tk_short), null);
DatumTC = orbRef.init().create_struct_tc(
        "IDL:Elfoglaltsagi_programcsomag/Naptar/Datum:1.0",
        "Datum", tagok);
```

3.9. Szerveroldali elemek

A fejezet eddigi részeiben elsősorban a kliensoldali Java leképezéssel foglalkoztunk. Most áttekintjük, hogy milyen módon készíthetjük el a CORBA objektumok szerveroldali részének implementációját (vagyis az illető objektumok által nyújtott szolgáltatásokat milyen módon implementálhatjuk, illetve azt, hogy hogyan hozhatunk létre és jelenthetünk be objektumokat az ORB szoftvernek, hogy az azután a kliensektől érkező metódushívási kérelmeket az illető objektumnak továbbíthassa).

Az objektumadapterek szerepének bemutatásakor láthattuk, hogy a CORBA objektumokat élettartamuk (illetve aktivációs módjuk) szerint két fő csoportra oszthatjuk: tranziens (azaz nem perzisztens) és perzisztens objektumok csoportjára. A tranziens objektumok élettartama megegyezik az őket létrehozó CORBA szerver folyamat élettartamával (vagy esetleg még annál rövidebb is lehet, ha a folyamat még a

befejeződése előtt megszünteti az illető objektumot), míg egy perzisztens objektum élettartama független az őt kezelő szerver folyamat(ok) élettartamától (hiszen az objektumot kezelő szerver szükség esetén bármikor újra-, illetve elindítható, ha az ORB egy metódushívási kérelmet kap az illető perzisztens objektumra vonatkozóan). Megjegyezzük, hogy az objektumreferenciák a hivatkozott objektumnak megfelelően lehetnek perzisztensek és tranziensek, ahol a perzisztens referencián az olyan referenciát értjük, amely az illető objektumot kezelő szerver futásának befejeződése után is alkalmas az illető perzisztens objektum azonosítására.

Megjegyezzük, hogy e sorok írásakor a CORBA Java nyelvi leképezését specifikáló szabvány nem tartalmazza a perzisztens objektumok Java nyelven történő elkészítéséhez szükséges elemeket, csak a tranziens objektumok készítésének támogatása megoldott (és az erre vonatkozó szabványt sem fogadták még el, de a kialakított szabványvázlat a szabványosítás során várhatóan már nem fog változni, ezt fogjuk itt röviden bemutatni). Természetesen a más - például C vagy C++ - nyelven írt perzisztens objektumok Java programokból történő hívására vonatkozóan semmiféle korlátozás nincs, egy Java kliensnek mindegy az, hogy a rendelkezésére álló CORBA objektumreferencia perzisztens vagy tranziens CORBA objektumra hivatkozik; a lényeg csak az, hogy a hivatkozott objektum CORBA objektum.

Egy CORBA IDL interfészt implementáló - szerveroldalon levő - objektum elkészítése úgy történik, hogy meg kell írni az illető interfészt implementáló ún. kiszolgáló osztályt: ennek az osztálynak a példányai implementálják az illető CORBA objektumok metódusait (és azt is állíthatjuk, hogy mindegyik CORBA objektumot egy-egy kiszolgáló osztály egy példánya implementál). E kiszolgáló osztályokat az ún. kiszolgáló bázisosztályból való származtatással lehet előállítani, amit az `idltojava` segédprogram tud generálni. A kiszolgáló bázisosztály feladata a kliens metódushívási kérésében érkező adatok (hívási paraméterek) kicsomagolása, majd a megfelelő szolgáltatást implementáló kiszolgáló metódus meghívása, és a kiszolgáló metódus által visszakapott visszatérési érték, illetve kimenő paraméterek visszajuttatása a metódushívást kezdeményező kliens alkalmazáshoz. A kiszolgáló osztály a kiszolgáló bázisosztályt általában az IDL interfészben megnevezett metódusok implementációjával, valamint az IDL fájlban specifikált attribútumok elérését és módosítását lehetővé tevő metódusokkal bővíti (azt ezekkel kell bővítenie).

Egy `xxxyyz` nevű IDL interfészhez generált kiszolgáló bázisosztály neve általában `_xxxyyzImplBase` (ez természetesen egy Java osztály, nem pedig egy interfész), így a kiszolgáló osztályokat úgy kell megírni, hogy ezt az osztályt bővítsék, és az IDL nyelvű interfészspecifikációban rögzített metódusokat mind implementálják (e metódusokat nyilvános láthatósággal, `public` módosítóval kell ellátni). Ha egy IDL interfész egy - esetleg több - másik IDL interfésztől örököl metódusokat, akkor az abból létrehozott Java interfész az IDL interfésze szülőinterfészeinek Java interfész megfelelőitől örökli azok összes metódusát. Mivel egy adott IDL interfész alapján generált `ImplBase` osztály

implementálja az IDL interfész alapján generált Java interfész összes metódusát, ezért az IDL interfészek többszörös öröklődési mechanizmusai nem okoznak problémát a Java leképezés elkészítésénél (ui. a Java az osztályokkal kapcsolatban nem, de az interfészeivel kapcsolatban igenis támogatja a többszörös öröklődést).

Egy CORBA objektumot egyszerűen az interfészét implementáló Java kiszolgáló osztály példányosításával hozhatunk létre, majd az ORB szoftvert reprezentáló, ORB osztályba tartozó objektum `connect()` metódusával kell bejelentenünk azt az ORB szoftvernek, hogy az a későbbiekben az illető objektumra vonatkozóan érkező távoli CORBA metódushívási kéréseket el tudja juttatni az illető CORBA objektum kiszolgáló osztályához (illetve a kiszolgáló osztály példányaira hivatkozó referencia bármely olyan metódus paramétereként átadható legyen, amely metódus egy ilyen CORBA objektumot vár a paraméterében). Megjegyezzük, hogy a `connect()` metódus meghívása nem szükséges minden esetben az illető CORBA objektum aktiválásához, mivel abban az esetben ez automatikusan megtörténik, ha az illető CORBA objektumot egy nem helyi CORBA objektum metódusának paraméterében átadjuk.

Egy CORBA objektumot az ORB szoftvertől az ORB szoftvert reprezentáló objektum `disconnect()` metódusával lehet "lekapcsolni". Az ORB szoftver a lekapcsolás után nem közvetít további metódushívási kérelmeket a kliensektől a CORBA objektum kiszolgáló osztályához (az ezután érkező metódushívási kérelmeket az ORB szoftver egy `CORBA::OBJECT_NOT_EXIST` kivételt generálva utasítja vissza).

Tekintsük a következő példát egy egyszerű IDL fájlra:

```
module Probamodul {
  interface Probainterface {
    long novel(in long param);
  };
};
```

Az ehhez az interfészhez megírt kiszolgáló osztály szerkezete e következő:

```
class ProbainterfaceKiszolgalo extends _ProbainterfaceImplBase {

  public int novel (int param) { // az IDL specifikációbeli metódus implem.
    // ki kell számolni a visszatérési értéket, majd visszaadni ...
    return (kiszámolt_végeredmény); // itt adjuk vissza
  }

}
```

A fenti kiszolgáló osztály elkészítése után a következőkben bemutatott Java utasításokkal létrehozhatunk egy új objektumpéldányt, és bejelenthetjük az ORB szoftvernél, hogy a kliensek meg tudják hívni a metódusát.

```
ProbainterfaceKiszolgalo pikRef = new ProbainterfaceKiszolgalo();
orbRef.connect(pikRef);
// ... itt a kliensek elérhetik létrehozott CORBA objektum metódusait
```

```
// (persze ehhez szükséges, hogy például a névszolgáltatón keresztül
//  elérhessék az illető objektum referenciáját)
orbRef.disconnect(pikRef);
```

Megjegyezzük, hogy a bemutatott módszerrel problémás lenne például Java appletek közvetlen bővítése CORBA szerver feladatokkal. Itt a problémát az okozza, hogy a fentiek alapján a kiszolgáló osztályt a kiszolgáló bázisosztályból való származtatással kell létrehozni, így az nem származtatható az appletek szokásos ősosztályától. Ezt a problémát a következőképpen oldhatjuk meg a delegáció-alapú szervercsonkok alkalmazásával.

A szerverfeladatok ellátását a korábban ilyen célra is használt kiszolgáló bázisosztály helyett rábízhatjuk a delegáció-alapú (ún. tie) szervercsonk osztályra, amit általában az `idltojava` segédprogrammal generáltathatunk egy `-ftie` programargumentum megadásával[17] (lásd a 3.6. fejezetet). Ekkor a kiszolgáló osztály szerkezete lényegesen megváltozik: a kiszolgáló osztályt ekkor bármilyen ősosztálytól származtathatjuk, de a kiszolgáló osztálynak implementálnia kell az `_elemnévOperations` interfészt, amely az `elemnév` nevű IDL interfész metódusainak Java leképezését tartalmazza. A kiszolgáló osztály példányosítása után létre kell hoznunk egy tie osztályt (ennek neve `_elemnévTie`), a tie osztály konstruktorában a kiszolgáló osztály referenciáját adjuk meg. Ezután a létrehozott tie objektumot bejelenthetjük az ORB szoftvernél az ORB `connect()` metódusával.

A kiszolgáló osztály szignatúrája ekkor például a következő séma szerint épül fel:

```
public class ProbainterfaceKiszolgáló extends java.applet.Applet
       implements _ProbainterfaceOperations {

  // Egyéb applet-metódusok ...

  public int novel (int param) { // az IDL specifikációbeli metódus
    // ki kell számolni a visszatérési értéket, majd visszaadni ...
    return (kiszámolt_végeredmény);
  }

}
```

A szerver objektumot a következőképpen jelenthetjük be az ORB-nél:

```
ProbainterfaceKiszolgáló pikK = this; // Ez az applet osztálya, abból
                                      // pedig már van egy példány,
                                      // maga a példányosított applet
                                      // (a   this   referencia).
Probainterface pikRef = new _ProbainterfaceTie(pikK);
```

[17] Emlékezzünk rá, hogy a delegáció-alapú szervercsonkok generálása két fontos Java elem generálását jelenti: az `_elemnévOperations` interfész, valamint az `_elemnévTie` osztály generálását.

```
orbRef.connect(pikRef);
// ... itt a kliensek elérhetik létrehozott CORBA objektum metódusait
// (persze ehhez szükséges, hogy például a névszolgáltatón keresztül
//  elérhessék az illető objektum referenciáját)
orbRef.disconnect(pikRef);
```

Mivel a fenti példában maga a futó applet osztálya implementálta a CORBA műveleteket (a `_ProbainterfaceOperations` interfészt), amiből a webböngésző már létrehozott egy példányt, ezért nekünk magunknak nem kellett egy új kiszolgálópéldányt létrehoznunk, hanem megadhattuk a már létrehozott appletpéldányra hivatkozó objektumreferenciát (a `this` referenciát). Ha valamilyen oknál fogva újabb példányokat kellene létrehozni, akkor azt megtehetjük a szokásos módon (lásd az alábbi sort).

```
ProbainterfaceKiszolgáló pikK = new ProbainterfaceKiszolgáló();
```

Fontos látni, hogy a fenti delegáció-alapú példában egy appletnek CORBA objektumszerver feladatokat kellett ellátnia. Ha egy appletnek csak kliens feladatokat kell ellátnia, akkor egyszerűen példányosítania kell egy ORB objektumot, és azon keresztül megszerezheti más CORBA objektumok referenciáit, és meghívhatja az illető objektumok metódusait.

3.10. A dinamikus szerveroldali csonk-interfész

Az előző pontban megismert (statikus) szervercsonk objektumok általában úgy működnek, hogy valamilyen táblázatban össze van gyűjtve, hogy az illető szervercsonk milyen nevű CORBA objektummetódusokat tud kiszolgálni, illetve az egyes metódusok feladatainak elvégzéséért mely helyi objektummetódusok felelősek, és ha egy klienstől egy CORBA metódushívási kérés érkezik, akkor a táblázatból vissza lesz keresve, hogy a kliens által kért CORBA objektummetódus végrehajtásáért melyik helyi metódus felelős, majd az illető metódus meg lesz hívva a klienstől érkező kérésben megadott paraméterekkel. Fontos látni, hogy az említett metódustáblát az `idltojava` generátor hozza létre a CORBA objektumok interfészeinek IDL-nyelvű specifikációja alapján, és létrehozása után - futásidőben - ez a tábla nem bővíthető további elemekkel (csak a megfelelő IDL interfész módosításával, majd a szervercsonk újragenerálásával).

A CORBA DSI (dinamikus szerveroldali csonk-interfész) komponense ennél nagyobb rugalmasságot biztosít azzal, hogy nincs szüksége egy ilyen előre elkészített metódustáblára, hanem a használatához egyetlen metódust kell megírni, amely megkapja a klienstől érkező metódushívási kérésben meghívni kívánt metódus nevét, és a meghívott metódus paramétereit (a paramétereket a dinamikus metódushívási interfész bemutatásakor már ismertetett `CORBA::NVList` típusú adatszerkezetbe csomagolva). Ez a metódus aztán bármit csinálhat ezekkel ez információkkal, például meghívhatja a megfelelő kiszolgáló metódust, vagy éppen maga is nyújthatja az illető szolgáltatást (a lényeg az, hogy tegye meg azt, amit a kliens elvár). A visszatérési értéket, valamint

a kimenő paramétereket ehhez hasonló módon megadva az ORB szoftver azokat visszajuttatja majd a kliens implementációhoz (a kliensnek eközben persze fogalma sincs arról, hogy a szerver a dinamikus csonk-interfésszel van implementálva, vagy a statikussal - számára az a fontos, hogy visszakapja a szerver válaszát olyan formában, amilyen formában azt várja).

A dinamikus szerveroldali csonk-interfész támogatását két Java osztály biztosítja: az `org.omg.CORBA.DynamicImplementation` és az `org.omg.CORBA.ServerRequest` osztályok. Ezen Java osztályok specifikációját a következő lista tartalmazza.

```
public abstract class DynamicImplementation
                      extends org.omg.CORBA.portable.ObjectImpl {

  public abstract void invoke(org.omg.CORBA.ServerRequest request);

}

public abstract class ServerRequest {
  public abstract String op_name();
  public abstract Context ctx();
  public abstract void params(NVList parms);
  public abstract void result(Any a);
  public abstract void except(Any a);
}
```

A dinamikus szerveroldali csonkokat (pontosabban az illető objektum metódusainak a kiszolgálását végző kiszolgáló osztályt) a `DynamicImplementation` osztálytól kell származtatni[18], és a programozónak a származtatott kiszolgáló osztály `invoke()` metódusának implementációját kell elkészítenie. Ez az a metódus, amelyet az ORB szoftver egy klienstől érkező metódushívási kérés esetén meghív, és az ORB szoftver ennek a metódusnak ez egyetlen paraméterében adja át a kliens által meghívott metódus nevét és paramétereit. Az így elkészített kiszolgáló osztályt adjuk aztán majd át az ORB szoftvert reprezentáló objektum `connect()`, illetve `disconnect()` metódusainak a CORBA objektumnak az ORB szoftvernél történő bejelentése céljából. A szerveralkalmazás a klienstől érkező paramétereket az előbb említett `invoke()` nevű metódusának az egyetlen, `ServerRequest` osztályba tartozó paraméterében kapja meg. Ennek az osztálynak a metódusaival lehet a klienstől kapott, illetve a kliensnek visszaadandó adatokat lekérdezni, illetve beállítani. A szóban forgó metódusok a következők:

[18] Látható, hogy az `org.omg.CORBA.DynamicImplementation` osztály ősosztálya egy `org.omg.CORBA.portable.ObjectImpl` nevű osztály. Ezzel az osztállyal részletesebben nem fogunk foglalkozni, csak annyit említünk meg róla, hogy ez minden kliens- és szerveroldali csonk ősosztálya. Ennek az osztálynak a bevezetésével tudták azt elérni, hogy a kliens- és szerveroldali csonkok függetlenek legyenek a fejlesztésükre felhasznált ORB-szoftvertől. Úgy van implementálva, hogy az egyes metódusokat egy ORB-függő segédosztályokhoz delegálja – bár ezzel az alkalmazói programok készítőinek nincs dolguk. A Java leképezés szerveroldali része a közeljövőben várhatóan módosulni fog annyiban, hogy egy másik ősosztályt kapnak ezek a csonkok, de az alkalmazó programok készítőinek ezzel nem kell törődniuk.

`op_name()` : szöveges formában visszaadja a kliens által meghívni kívánt metódus nevét.

`ctx()` : a klienstől érkezett hívási kontextus objektumhoz férhetünk hozzá ennek segítségével (attribútumot - azaz adattagot - elérő automatikusan generált metódus esetén ez nem tartalmaz értékes információkat).

`params()` : a kliens által a metódushívási utasításban megadott metódushívási paraméterekhez férhetünk hozzá ennek segítségével. Az ORB szoftver a `params()` metódus paraméterében átadott `NVList` osztályba tartozó objektumba teszi bele a kliens által küldött paraméterértékeket (a paramétereknek a kliensoldali metódus interfészét definiáló IDL specifikációban rögzített sorrendjében, "balról jobbra"). Megjegyezzük, hogy az átadott `NVList` objektumot a metódus által várt paraméterek száma és típusa szerint előre fel kell tölteni (a várt paraméter típuskódját reprezentáló `TypeCode` értéket be kell állítani), és a visszakapott `NVList` lista objektum minden egyes `NamedValue` osztálybeli eleme egy `Any` osztályba tartozó értéket tartalmaz, ami a hívási paraméterek szerint lesz beállítva. A szervercsonk az `NVList` paraméterben átadott lista összeállításához felhasználhatja az `op_name()` metódus visszatérési értékeként megkapott metódusnevet; onnan tudhatja, hogy a kliens melyik metódust hívta, és a paraméterek lekérdezéséhez mennyi és milyen típusú elemet kell az említett listába betenni.

`result()`**, illetve** `except()` : ezekkel a metódusokkal rendre a kliensnek visszaadandó visszatérési értéket, illetve sikertelen végrehajtás esetén a sikertelenség okát jelző kivételt kell beállítani. Mindkét metódus a paraméterében egy `Any` típusú objektumot vár, amit helyesen ki kell tölteni a metódus visszatérési értékének típusának megfelelően, illetve a kiváltott kivételről a szükséges információkat szintén itt kell megadni.

Az `invoke()` metódus általában a következő feladatokat végzi el:

1. Az `op_name()` metódussal megvizsgálja, hogy mi a meghívott metódus neve.
2. Felépít egy `NVList` osztályba tartozó objektumot, amelybe a kliens által küldött paraméterértékeket várja (az `NVList` objektum szerkezetét az előbb már láthattuk; az `NVList` objektumban a várt paraméterek számát és a paraméterek típusát a program vagy kikövetkeztetheti például a hívott metódus neve alapján, vagy felhasználhatja ehhez az interfészgyűjteményt is, ahol van ilyen[19]).
3. Lekérdezi a paraméterek értékét az ORB szoftvertől. Ehhez meghívja az előbb említett `params()` metódust, a paraméterében átadja az előbb összeállított `NVList`

[19] Megjegyezzük, hogy a JavaSoft JOE ORB szoftvere nem tartalmaz interfészgyűjtemény implementációt, de a JOE-val is elérhetünk más interfészgyűjteményeket, hiszen ezeknek az elérési módja is szabványosítva van.

objektumot. Az ORB szoftver a `params()` metódus paraméterében átadott `NVList` objektumba beleteszi a klienstől érkezett paraméterértékeket.

4. Elvégzi a kliens kérésében kért műveletet, majd a kimenő - azaz `out` és `inout` módú - paraméterek értékeit beállítja az említett `NVList` objektumban.

5. Létrehoz egy `Any` osztálybeli objektumot, amelybe bemásolja a CORBA metódus visszatérési értékét, vagy a kiváltott kivétel azonosítóját (annak típuskódjával együtt).

6. Meghívja vagy az `except()`, vagy pedig a `result()` metódust (aszerint, hogy kell-e egy CORBA kivételt generálni a kliens oldal felé)[20].

A dinamikus szervercsonk osztályoknak implementálniuk kell a szülőosztályuktól örökölt - ott absztrakt - `_ids()` metódust. Ennek szignatúrája a következő kell legyen (figyeljünk rá, hogy az implementációnál már ne legyen absztrakt!):

```
public String[] _ids();
```

Az `_ids()` metódust úgy kell implementálni, hogy az az implementált CORBA interfész interfészgyűjteménybeli nevét adja vissza. A metódus implementálható például úgy, hogy egy karakterlánc-vektorban elhelyezzük a visszaadandó nevet, és az _ids() metódusnak egyszerűen ezt a vektort kell visszaadnia. A következő példa egy ilyen karakterlánc-vektor deklarációját mutatja be:

```
String[] ifaceIds = {"IDL:Rozsika/SzerverCsonkPelda:1.3"};
```

Ekkor az említett `_ids()` metódus implementációja egyszerűen a következő:

```
public String[] _ids() {
  reeturn ifaceIds;
}
```

Az interfészek elnevezési konvenciójaként a CORBA specifikáció az alábbi megkötéseket említi:

- Egy interfésznév három komponensből áll, a komponenseket kettőspont karakterek választják el egymástól.
- Az első komponens az `IDL` karaktersorozat legyen.
- A második komponens `/` karakterekkel elválasztott azonosítókból álljon. Az azonosítók betűk, számok, aláhúzás karakterek, kivonás (azaz -) és pont karakterekből álljanak. Az azonosítókat tetszésünk szerint szabadon választhatjuk meg;

[20] A `params()` metódust feltétlenül hívjuk meg (legalább a paraméterek konzisztenciájának ellenőrzése céljából), és csak ezután hívjuk meg a `result()`, vagy az `except()` metódusok egyikét (csak egyszer). Ha ezeket a korlátozásokat megszegjük, akkor az ORB szoftver egy `CORBA::BAD_INV_ORDER` kivételt generál.

a kialakult konvenciók szerint az azonosítók első része egy világszerte egyedi - például egy szoftverprojektet, vagy szoftvergyártót megnevező - prefix legyen, amit az igényeknek megfelelően további komponensek követhetnek.

- A harmadik komponens az interfész verziószámát tartalmazza. A fő-, illetve alverziószámok[21] decimális számok, és egy pont karakter válassza el őket egymástól.

[21] Ha két interfésznév csak az alverziószámban különbözik egymástól, akkor feltételezhető, hogy a magasabb verziószámú interfész az alacsonyabb verziószámú interfész bővítése.

4. Fejezet

A négyzetreemelő programunk további részei

Már megvannak a szükséges ismereteink ahhoz, hogy elkészíthessük a dinamikus metódushívási interfész ismertetésekor megírt `Negyzetreemelo` interfészt implementáló objektumunkat.

4.1. A szerveroldal implementációja

Lássuk tehát a feladatot implementáló program forráskódját, majd utána néhány rövid magyarázatot is olvashatunk.

```
// NegyzetreSzerver.java
//
// Szerveroldali egység a négyzetreemelő példaprogramunkhoz.
//
// A program a Negyzetre modul Negyzetreemelo IDL interfészének
// implementációját biztosító objektumot hoz létre és jegyez be
// a CORBA névszolgáltatóba

import Negyzetre.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class NegyzetreSzerver extends _NegyzetreemeloImplBase {

  // Ez a Negyzetreemelo IDL interfész egyetlen emel() nevű metódusa.
  public int emel(int mi) { // Ez az IDL interfész metódusa
    return (mi*mi);
  }

  public static void main(String[] args) {
```

```
    try {
      // A következő objektumot kölcsönös kizárás implementációjara
      // fogjuk használni
      java.lang.Object holtpont = new java.lang.Object();

      ORB orbRef = ORB.init(args, null); // Létrehozzuk az ORB referenciát
      NegyzetreSzerver neoRef = new NegyzetreSzerver(); // Új objektum
      orbRef.connect(neoRef); // A létrehotott objektumot bejelentjük
                              // az ORB szoftvernél.

      org.omg.CORBA.Object ncObjRef =
                        orbRef.resolve_initial_references("NameService");
      NamingContext ncRef=NamingContextHelper.narrow(ncObjRef);
      NameComponent nc = new NameComponent("Negyzetreemelo_objektum", "");
      NameComponent útvonal[] = { nc } ;
      ncRef.rebind(útvonal, neoRef); // A névszolgáltatóba is betesszük

      synchronized (holtpont) { // Itt a végtelenségig várunk, hátha
        holtpont.wait();        // lesznek klienseink ...
      }
    } catch (Exception e) {
      System.out.println("Kivétel: "+e.getMessage());
    }
  }
}
```

A fenti programban definiált Java osztály - amint azt láthatjuk - az `idltojava` fordító által generált `_NegyzetreemeloImplBase` nevű szervercsonk ősosztálytól származik. Az IDL interfész `emel()` nevű metódusát az IDL specifikációnak megfelelően nyilvános - azaz `public` - módosítóval implementálja.

A főprogram először létrehoz egy segédobjektumot, amellyel az elindult főprogram "olcsó" végtelen ciklusban történő futtatását támogatja (a fent bemutatott végtelen ciklus lényegében egy holtpont-szerű helyzet eredménye, és azért neveztük olcsónak, mert a program a várakozása alatt nem csinál semmit, nem foglalja a CPU-időt, amint azt esetleg egy `while(true) { }` felépítésű végtelen ciklus alkalmazása esetén tenne).
A program ezután inicializálja az ORB szoftvert reprezentáló objektumot, majd létrehoz egy példányt a négyzetreemelési szolgáltatást biztosító objektumunkból, és bejelenti azt az ORB szoftvernél (a kliensek az objektumot csak ezután érhetik el, illetve néhány fejlesztői környezetben - mint azt már említettük - csak ezután tudjuk az illető objektumot CORBA metódushívások paramétereiben átadni; vagy akár a CORBA névszolgáltatóba bejegyezni). Ezután a program bejegyzi a létrehozott objektumot a CORBA névszolgáltatóba, és az előbb már említett végtelen ciklusba jut, amikor a kliensek már szabadon hívhatják a CORBA objektumainak a metódusait.

Megjegyezzük, hogy a fent bemutatott végtelen ciklus úgy működik, hogy a `synchronized()` blokkon belül megpróbálja megszerezni a `synchronized()` utasítás által lefoglalt objektumzárat (angol nevén lock), és egészen addig vár, amíg ez nem

sikerül. A program fordítása előtt készíttessük el az `idltojava` segédprogrammal a kliens- és szerveroldali csonkokat a következő parancssorral:

```
idltojava -fno-cpp -fclient -fserver Negyzetre.idl
```

Fordítsuk is le a `Negyzetre` könyvtárban létrehozott Java forrásfájlokat!

A program fordításakor pedig ne feledjük megadni a `-encoding Cp852` programargumentumot a Java fordítónknak, hogy helyesen értelmezze a magyar nyelvű ékezeteket is. A fordítást én egy felinstallált Java IDL környezetben az alábbi parancs beadásával végeztem:

```
javac -encoding Cp852 NegyzetreSzerver.java
```

4.2. Egy statikus modellre épített kliens példaprogram

Most pedig bemutatjuk, hogy az előbbi négyzetreemelő szolgáltatásunkat milyen módon vehetjük igénybe a CORBA statikus metódushívási interfészén keresztül (emlékezzünk rá, hogy korábban csak a dinamikus metódushívási interfész elemeit használó programot mutattuk be).

```
// SNegyzetreemeloTeszt.java
//
// Egy kliens program a négyzetreemelő szerverünkhöz.
// A kliensben a CORBA statikus metódushívási interfészét használjuk a
// szerver metódusának meghívására.

import Negyzetre.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class SNegyzetreemeloTeszt {

  public static void main(String args[]) {
   int mit_emeljen_négyzetre = 15;

   try {
    ORB orbref = ORB.init(args, null); // paraméter: főprogram argumentumai
    org.omg.CORBA.Object nr =
                      orbref.resolve_initial_references("NameService");
    NamingContext ncref = NamingContextHelper.narrow(nr);
    NameComponent nc = new NameComponent("Negyzetreemelo_objektum", "");
    NameComponent keresett_név[] = {nc};
    Negyzetreemelo neref = NegyzetreemeloHelper.narrow(ncref.resolve(
                           keresett_név));

    int eredm = neref.emel(mit_emeljen_négyzetre);
```

```
    // Kiírjuk az eredményt
    System.out.println(new Integer(mit_emeljen_négyzetre).toString()+
                  " négyzete = "+new Integer(eredm).toString());
   } catch (Exception e) {
    System.out.println("Kivétel lépett fel a hívás során!");
   }
  }
}
```

A fenti program szerkezete nagyon egyszerű: inicializálja az ORB szoftvert reprezentáló objektumot, majd a más megismert módon a névszolgáltatóból megszerez egy objektumreferenciát a négyzetreemelő szolgáltatást nyújtó CORBA objektumra, majd meghívja a négyzetreemelő metódusát és kiírja a négyzetreemelés eredményét.

4.3. A négyzetreemelő példaprogram futtatása

A négyzetreemelő programunk a következő komponensekből áll: a szerveroldalon a `NegyzetreSzerver` osztály példányai nyújtják a négyzetreemelési szolgáltatást, míg ezeknek a kliensei lehetnek akár az `SNegyzetreemeloTeszt`, akár a korábban - a dinamikus metódushívási interfészről szóló részben - bemutatott `NegyzetreemeloTeszt` osztály példányai.

A program egyes komponensei a CORBA névszolgáltatóján keresztül találják meg egymást, így azt is el kell indítani az előbb említett komponensek előtt (nyilván azt csak egy példányban, egyszer kell elindítani).

A névszolgáltatót a JOE ORB szoftver használata esetén a következő parancssorok valamelyikével indíthatjuk a JavaIDL rendszer konfigurálása után:

```
tnameserv
```

vagy pedig a

```
tnameserv -ORBInitialPort TCPportsorszám
```

Az első sor szerint a névszolgáltató az alapértelmezés szerinti 900-as TCP-porton lesz elérhető, míg a második példában a `TCPportsorszám` paraméterben megadott TCP-porton lesz elérhető. Megjegyezzük, hogy az előbbi forma használatához rendszergazdai jogkör szükséges, mivel a 900-as a foglalt TCP-portok közé tartozik. Egy számítógépen természetesen több névszolgáltató is futhat, ilyenkor mindegyiket más-más TCP porton lehet elérhetővé tenni. A névszolgáltató elindulása után kiírhatja például az elérésére használt IOR referencia karakterlánc reprezentációját, vagy kiírhat akár másféle adatokat is. A névszolgáltatót a korábban elérhető fejlesztői rendszereken (a JDK 1.2 megjelenése előtt) a `nameserv` paranccsal lehetett elindítani. Az újabb változatokban a parancs neve elé írt `t` betű a névszolgáltató tranziens (nem-perzisztens) jellegére utal.

Megjegyezzük, hogy ha az Olvasó más ORB szoftvert használ, akkor lehet, hogy a rendszerprogramok - így a névszolgáltató - indítása az Olvasó saját számítógépén

máshogyan történik. Hasonlóan változhat az is, hogy a fenti parancs végrehajtása után visszakapjuk-e a parancsértelmező promptját (ekkor a névszolgáltatót háttérben futó valamilyen démon folyamatként elindítva), vagy nem kapjuk vissza a promptot (és ekkor például U*X/Linux rendszerekben magunknak kell gondoskodnunk a névszolgáltató háttérbeli elindításáról, a legtöbb parancsértelmező shell esetén ehhez a parancs végére egy & karaktert kell írni).

Miután fut a névszolgáltató, elindíthatjuk az alkalmazásainkat. Mivel a szerverprogramot úgy írtuk meg, hogy az végtelen ciklusban fusson, ezért azt vagy a háttérben kell elindítanunk (az előbb említett módon), vagy külön ablakokban kell futtatni a kliens- és a szerveralkalmazásokat.

A szerver indítása a következő parancssorral történhet:

```
java NegyzetreSzerver
```

A kliens indítására pedig a következő parancsot használjuk:

```
java NegyzetreemeloTeszt
```

Megjegyezzük, hogy az alkalmazások így az alapértelmezés szerinti, azaz a 900-as TCP-porton keresik a névszolgáltatót, de ezt a `-ORBInitialPort` a névszolgáltatónál bemutatott módon egy programargumentummal felülbírálhatjuk.

4.4. A szerveroldal dinamikus implementációja

A program két osztály implementációjából áll: a `DSINegyzetreSzerver.java` fájl tartalmazza a főprogramot, a `DSINegyzetreKiszolgalo.java` fájl pedig a dinamikus szervercsonk implementációt, a tulajdonképpeni kiszolgáló osztályt tartalmazza.

```
// DSINegyzetreSzerver.java
//
// Szerveroldali egység a négyzetreemelő példaprogramunkhoz.
// Dinamikus szerveroldali-csonk bemutatására
//
// A program a Negyzetre modul Negyzetreemelo IDL interfészének
// implementációját biztosító objektumot hoz létre és jegyez be
// a CORBA névszolgáltatóba. Kivételt generál és leáll, ha a
// névszolgáltató nem érhető el.

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class DSINegyzetreSzerver {

  public static void main(String[] args) {

    try {
```

```
      // A következő objektumot kölcsönös kizárás implementációjara
      // fogjuk használni
      java.lang.Object holtpont = new java.lang.Object();

      ORB orbRef = ORB.init(args, null); // Létrehozzuk az ORB referenciát
      // A következő sor létrehoz egy példányt a dinamikus szerveroldali
      // kiszolgáló objektumból, amit azután ugyanúgy használhatunk, mint
      // a statikus párjait.
      DSINegyzetreKiszolgalo neoRef = new DSINegyzetreKiszolgalo(orbRef);
      orbRef.connect(neoRef); // A létrehotott objektumot bejelentjük
                              // az ORB szoftvernél.

      org.omg.CORBA.Object ncObjRef =
                        orbRef.resolve_initial_references("NameService");
      NamingContext ncRef=NamingContextHelper.narrow(ncObjRef);
      NameComponent nc = new NameComponent("Negyzetreemelo_objektum", "");
      NameComponent útvonal[] = { nc } ;
      ncRef.rebind(útvonal, neoRef); // A névszolgáltatóba is betesszük

      synchronized (holtpont) { // Itt a végtelenségig várunk, hátha
        holtpont.wait();        // lesznek klienseink ...
      }
    } catch (Exception e) { // pl. ha névszolgáltató nem érhető el
      System.out.println("Kivétel: "+e.getMessage());
    }
  }
}
```

Az előbbi osztály szerkezete hasonlít a statikus szerveroldal implementációnál bemutatott program szerkezetéhez (itt a `main()` metódusról van szó), azzal a különbséggel, hogy itt egy másik kiszolgáló lesz példányosítva és a CORBA névszolgáltatóba bejegyezve. A kiszolgáló itt is ugyanúgy egy végtelen ciklusban fut, mint a statikus esetben.

Számunkra most a dinamikus kiszolgáló osztály szerkezete a legérdekesebb. Ezt tartalmazza a következő programlista.

```
// DSINegyzetreKiszolgalo.java
//
// Ez a négyzetreemelő példankhoz szükséges dinamikus
// szerveroldali csonk implementációja.

import org.omg.CORBA.*;

public class DSINegyzetreKiszolgalo
                    extends DynamicImplementation {

  ORB orbV; // Az ORB-re hivatkozó referenciát tároló változó

  public DSINegyzetreKiszolgalo(ORB o) {
      this.orbV = o; // Eltároljuk az ORB-re hivatkozó referenciát
  }
```

```
  private final String ifaceIds[] = {
    // Ay implementált IDL interfész azonosítója
    "IDL:Negyzetre/Negyzetreemelo:1.0"
  };

  public String[] _ids() {
    // Visszaadom az implementált interfész azonosítóját
    // (pontosabban annak egy másolatát, nehogy mások felülírják)
    return (String[]) ifaceIds.clone();
  }

  public void invoke(ServerRequest kérés) {
     if (kérés.op_name().equals("emel"))
     {
        NVList paramLista = orbV.create_list(0); // kezdetben üres

        // Minden paraméternek lefoglalok egy Any objektumot
        Any mit = orbV.create_any();  // 1. paraméter
        mit.type(ORB.init().          // 1. paraméter típusa
                 get_primitive_tc(TCKind.tk_long));
        paramLista.add_value("mit", mit, ARG_IN.value);
        // Ennek a metódusnak csak ez az egy paramétere van

        kérés.params(paramLista); // Lekérdezem a paramétereket az ORB-től

        int mit_emeljen, eredmény;
        mit_emeljen = mit.extract_long(); // Megvan az első paraméter
        eredmény = mit_emeljen*mit_emeljen; // Négyzetreemelem
        Any visszatérési_érték = orbV.create_any();
        visszatérési_érték.insert_long(eredmény); // Beállítom a vissza-
        kérés.result(visszatérési_érték);         // térési értéket és
                                                  // annak típusát
     } else {
        // System.out.println("Ismeretlen metódus ...");
        throw new BAD_OPERATION(0, CompletionStatus.COMPLETED_MAYBE);
     }
  }
}
```

A kiszolgáló osztálynak egyetlen adattagja van, amelyben egy ORB-referenciát tárol. Ezen keresztül férhet majd hozzá a klienstől érkező metódushívási kérelmek jellemzőihez. Az ORB-referenciát az osztály konstruktorának a paraméterében kell kijelölni (amit az előbbi program meg is tett).

A kiszolgáló osztály implementálja az `_ids()` metódust, amely a dinamikusan implementált interfész interfészgyűjteménybeli azonosítóját adja vissza a hívónak. A korábban bemutatott IDL-specifikáció alapján az `idltojava` program a forrásprogramba írt interfészazonosítót generálná, így mi is ezt írtuk be a programba. Ezt az azonosítót egy konstans formájában tároltuk el a programban, amiről egy objektummásolat lesz vissza-

adva az `_ids()` metódus meghívásakor.

Az `invoke()` metódus a dinamikus szerveroldali csonk lényegi része: a szerverobjektum itt kapja meg a klienstől érkező kérések jellemzőit - azt, hogy a kliens milyen nevű metódust hívott, és hogy milyen értékeket adott át paraméterként. Az `invoke()` metódus paraméterében kapja meg azt a kulcsobjektumot, amelynek metódusait meghívva hozzáférhet a kliens kérésének az előbb említett részleteihez. A paramétert a fenti példában `kérés` néven neveztük. Mint azt a dinamikus szerveroldali csonk-interfészről szóló pontban láthattuk, `kérés.op_name()` néven érhetjük el a kliens által meghívni kívánt metódus nevét. Az `invoke()` metódus szerkezete általában olyan, hogy elágazik a kliens által meghívni kívánt metódus neve alapján, és mindegyik metódushoz tartalmaz egy megfelelő megvalósítást, amely eltárolja a klienstől érkező paramétereket, feldolgozza azokat, majd visszaadja a megfelelő visszatérési értéket.

Az fenti példában az `invoke()` metódus megvizsgálja, hogy a klienstől érkező kérésben megadott metódusnév `emel` tartalmú-e. Ha igen, akkor elvégzi a megfelelő műveleteket, míg ha nem, akkor egy ezt jelző `CORBA::BAD_OPERATION` kivételt vált ki. Az `emel` nevű műveletet feldolgozó programrész lefoglal egy `CORBA::NVList` objektumot, feltölti olyan típusú üres elemekkel, amilyen típusú paramétereket a klienstől vár. A fenti példában ez a lista egyetlen elemet tartalmaz, egy IDL `long` típusú értéket, amit a klienstől a `mit` nevű paraméter értékeként várunk. Miután ez a lista elkészült, meghívjuk a kliens kérését reprezentáló objektum `params()` metódusát, paraméterként átadjuk az előbb létrehozott listát. A `params()` metódus visszatérése után az ORB kitöltötte az előbb említett listát, a kliens által megadott `mit` nevű paraméter értékét berakta az ezt reprezentáló listaelembe. Ezt a listaelemet a `mit` nevű Java referenciával érhetjük el, és mivel ez egy `Any` típusú objektum, ezért a benne tárolt értéket a program az objektum `extract_long()` metódusával érheti el. A program ezután négyzetre emeli a kapott értéket, és létrehoz egy `Any` típusú objektumot az IDL `long` típusának megfelelő típuskóddal, és beleteszi a kapott eredményt, majd a `result()` metódussal visszaadja a kliensnek a visszatérési értéket tartalmazó `Any` típusú objektumot.

A fenti dinamikus szerveroldali implementáció klienseként használhatjuk az eddig bemutatott bármelyik négyzetreemelő-kliens alkalmazást.

5. Fejezet

Egy CORBA-alapú kliens/szerver példaprogram

A bemutatott négyzetreemelő alkalmazás több területen - különösen a dinamikus metódushívási interfész területén - a lehetőségeknek csak a legegyszerűbb használati módját szemléltette (ott nem voltak például sem kimenő paraméterek, sem kivétel kezelési elemek, amelyeknek használata nem triviális).

Most bemutatunk egy másik példaprogramot, amely a távoli metódushívásról szóló fejezetben bemutatott példaprogramokból már megismert egyszerű naptár szolgáltatást nyújtja (csak naptárlekérdezési és -felviteli műveleteket támogat, nem használunk semmiféle tranzakciókezelési elemet, és a szinkronizációval sem fogunk komolyabban foglalkozni). Az alkalmazás komponenseit összekapcsoló interfészt az alábbi IDL fájl definiálja:

```
// Elfoglaltsagi.idl

module Elfoglaltsagi_programcsomag
{
  interface Naptar
  {

    exception ErvenytelenDatum {
      string oka;
    };

    struct Datum {
      short Ev;
      short Honap;
      short Nap;
    };

    void Beallit(in Datum mikor, in string mi_van_akkor)
                   raises (ErvenytelenDatum);
```

```
    boolean Lekerdez(in Datum mikor, out string mi_van_akkor)
                    raises (ErvenytelenDatum);
    string Tulajdonos_neve();
  };

};
```

Mi egy `Naptar` interfészt biztosító objektumot szolgáltató szervert és egy azt lekérdezni és módosítani képes kliens alkalmazást fogunk kifejleszteni (pontosabban két klienst: az egyikben a statikus, a másikban pedig a dinamikus metódushívási interfésszel fogjuk megvalósítani a naptár objektumszerver elérését).

Most röviden az interfészünk specifikációjáról fogok írni. Az interfészünknek három metódusa van. A `Beallit()` metódussal felvehetünk a naptárba egy megadott dátumhoz egy elfoglaltságot leíró szöveget (a dátumot az első, az elfoglaltságot leíró szöveget a metódus második paraméterében kell megadnunk; mindkét paraméter `in` módúan lesz átadva, vagyis a hívott nem módosíthatja azokat). A metódus egy `ErvenytelenDatum` kivételt vált ki, ha 1996 előtti évet, vagy egy már foglalt dátumot adunk meg (az 1996-os határ megválasztása teljesen önkényesen történt, a példaprogram szempontjából ennek lényege az, hogy ilyen módon könnyen tanulmányozhatjuk a kivételek klienstől szerver felé történő terjedését). Megjegyezzük, hogy a szerveroldal implementációja nem ellenőrzi a megadott dátum tényleges helyességét (vagyis hogy például nem február 31-ét adtak-e meg); az implementáció egyszerűen berakja azt egy hash-táblába.
A `Lekerdez()` metódussal kérdezhetünk le egy adott napi elfoglaltságot. Az első paraméterében kell megadni az elfoglaltság dátumát, a másodikban pedig az elfoglaltság szöveges leírását fogjuk majd visszakapni (ui. ez egy szöveges kimenő - `out` módú - paraméter). A metódus visszatérési értéke logikai típusú: igaz értéket akkor és csak akkor ad vissza, ha a kérdéses időpontra már bejegyeztek egy elfoglaltságot. A metódus 1996 előtti dátum megadása esetén `ErvenytelenDatum` kivételt generál. A `Tulajdonos_neve()` metódussal egy naptár tulajdonosának nevét kérdezhetjük le.

5.1. A szerveroldal implementációja

Először a szerveroldal implementációját fogjuk áttekinteni. Tekintsük először a szerverprogram forráskódját!

```
// NaptarSzerver.java
//
// A Naptár interfészünk implementációját megvalósító szerver.
// Egyetlen argumentumot vár, a naptár tulajdonosának nevét (azonosítóját),
// amit a névszolgáltatóba bejegyez (ui. a továbbiakban a naptárat azon
// keresztül érhetjük el)
// Más paramétereket (pl. -ORBInitialPort) csak ez után írjunk.

import Elfoglaltsagi_programcsomag.*;
```

```
import Elfoglaltsagi_programcsomag.NaptarPackage.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.Hashtable;

public class NaptarSzerver extends _NaptarImplBase {

  String tulajneve; // Eltároljuk a tulajdonos nevét
  Hashtable elfoglaltság; // Ebben tároljuk az elfoglaltságokat

  private String Szöveg_Dátumból(Datum mi) { // Ilyenek a hashtábla kulcsok
    return(new Short(mi.Ev).toString()+"/"+
           new Short(mi.Honap).toString()+"/"+
           new Short(mi.Nap).toString());
  }

  public NaptarSzerver(String tulajneve) { // A konstruktor
    this.tulajneve=tulajneve;
    this.elfoglaltság = new Hashtable(); // Üres hashtáblát hozok létre
  }

  // Beír egy elfoglaltságot a naptárba
  public void Beallit(Elfoglaltsagi_programcsomag.NaptarPackage.Datum
         mikor, String mi_van_akkor)
         throws Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
  {

    if (mikor.Ev < 1996) { // Ekkor dobunk kivételt
      throw new Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
                (" 1996 elotti ");
    }
    synchronized (elfoglaltság) {
      // Ha a hashtáblában az illető dátumnál már van valami -> kivétel
      if (elfoglaltság.containsKey(Szöveg_Dátumból(mikor))) {
       throw new Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
           (" Már foglalt ");
      }
      elfoglaltság.put(Szöveg_Dátumból(mikor),mi_van_akkor); // bejegyezzük
    }
  }

  // Lekérdezzük egy dátum alapján az akkori elfoglaltságot
  public boolean Lekerdez(Elfoglaltsagi_programcsomag.NaptarPackage.Datum
         mikor, org.omg.CORBA.StringHolder mi_van_akkor)
         throws Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
  {

    if (mikor.Ev < 1996) { // Ekkor dobunk kivételt
      throw new Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
                (" 1996 elotti ");
```

```
    }

    mi_van_akkor.value="Nincs adat";
    synchronized (elfoglaltság) {
      if (!elfoglaltság.containsKey(Szöveg_Dátumból(mikor))) {
        return false; // Még nem foglalt az időpont
      }
      mi_van_akkor.value=(String) elfoglaltság.get(Szöveg_Dátumból(mikor));
      return true; // Már foglalt ...
    }
  }

  public String Tulajdonos_neve() { // Egyszerűen visszaadjuk a
    return tulajneve;               // tulajdonos nevét
  }

  public static void main(String[] args) {

    if (args.length >= 1) { // Pontosan egy paraméter van
     try {
      java.lang.Object holtpont = new java.lang.Object();
      ORB orb = ORB.init(args, null); // ORB inicializálása
      NaptarSzerver MNaptárRef = new NaptarSzerver(args[0]);
      orb.connect(MNaptárRef); // Létrehozott új objektum bejelentése

      org.omg.CORBA.Object ncObjRef =
                orb.resolve_initial_references("NameService");
      NamingContext ncRef=NamingContextHelper.narrow(ncObjRef);
      NameComponent nc = new NameComponent(args[0],"");
      NameComponent útvonal[] = { nc } ;
      ncRef.rebind(útvonal, MNaptárRef); // Bejegyezzük a névszolgáltatóba

      synchronized (holtpont) {
        holtpont.wait(); // várakozunk a kliensekre
      }

     } catch (Exception e) {
      System.out.println("Kivételt kaptam, program leállt."+
                         " A kivétel oka: "+e.getMessage());
     }
    } else {
      System.out.println("Indítás: java NaptarSzerver "+
                         "naptártulajdonos_neve");
    }
  }
}
```

A szerver osztály implementációja nagyon egyszerű: az elfoglaltságok egy hash-táblában lesznek nyilvántartva; az elfoglaltságok időpontjait használjuk kulcsként, és ezekhez fogjuk az elfoglaltságok szöveges leírását tartalmazó karakterláncokat asszociálni. Láthatjuk, hogy a naptár elérését biztosító metódusok az IDL fájlban definiált dátum

típust kapják meg paraméterükben az elfoglaltság dátumaként, de a hash-táblában nem ezt tároljuk el kulcsként, hanem egy ebből előállított karakterláncot. E karakterláncokat a `Szöveg_Dátumból()` metódussal állíthatjuk elő az IDL fájlban definiált dátum típusú objektumokból, és e karakterláncok szerkezete nagyon egyszerű: a dátum év, hónap, nap komponensei egy `/` karakterrel lesznek elválasztva, és így áll elő a kulcs (azaz például 1997 január 2-át a `1997/1/2` szöveggel fogjuk reprezentálni). Már említettük a feladat bevezetésénél, hogy a dátumok valódisága nem lesz ellenőrizve (vagyis a program nem ellenőrzi, hogy ha például május kilencvenhatodikát írtunk, akkor létezik-e olyan nap, vagy sem), és megfigyelhetjük a szerverprogramban azt is, hogy az IDL fájlban definiált kivételeket a Java nyelvben megszokott módon kezelhetjük (lásd a programban az 1993 előtti érvénytelen dátumok kezelésénél; emlékeztetőül megismételjük, hogy ha az IDL fájlban egy kivétel - ami szintén egy Java osztályra lesz leképezve - egy IDL interfészen belül lett deklarálva, akkor az IDL Java nyelvre történő leképezési szabályai szerint a beágyazott kivételt az `idltojava` fordító egy olyan csomagba helyezi, amely csomagnak a neve az interfész nevével kezdődik, és a `Package` szócskával végződik, azaz a példában a beágyazott kivételek és struktúrák a `NaptarPackage` Java csomagba kerülnek).

Felhívjuk még a figyelmet a `Lekerdez()` nevű metódus második paraméterére, amely egy kimenő típusú karakterlánc paraméter alapján lett generálva, így egy `StringHolder` osztály van ott megadva.

A program elindításakor egyetlen paramétert kell megadnunk: a naptár tulajdonosának a nevét, amin keresztül a naptár objektumot el akarjuk érni, illetve amilyen néven a naptárat be akarjuk jegyezni a CORBA névszolgáltatóba. A `main()` metódus egyszerűen inicializálja az ORB szoftvert, majd létrehoz egy új naptár CORBA objektumot, és bejegyzi azt a névszolgáltatóba. Ezután a már megismert holtpontba jutási technikával felfüggeszti a futását, és kiszolgálhatja a kliensek metódushívási kéréseit.

A program eléggé szigorúan veszi a paramétereinek a sorrendjét és jelentését (ennek részletes leírását megtalálhatjuk a program elején levő megjegyzés sorokban), így ha más paramétereket is meg akarunk adni (például az `-ORBInitialPort` paramétert is az ORB alapvető névszolgáltatásának elérési helyének megadására), akkor ezeket csak a paraméterlista végére (a program által elvárt paraméterek mögé) tegyük.

A következő parancssorral elindíthatjuk a programunkat, és létrehozhatjuk Mariska nevű felhasználónk naptárát.

```
java NaptarSzerver Mariska
```

illetve ha például a névszolgáltatót az 1900-as TCP-portra tettük, akkor a következő paranccsal indítsuk a programunkat:

```
java NaptarSzerver Mariska -ORBInitialPort 1900
```

Megjegyezzük, hogy egyes projektekben a kialakult konvenciók megtilthatják objektumoknak a névszolgáltató gyökérkontextusába való bejegyzését (ez egyébként is

csak ritkán lehet indokolt). Ekkor egyszerűen az alábbi módon - összetett nevek alkalmazásával - jegyezhetünk be egy objektumreferenciát (illetve névkontextust) a névszolgáltatóba:

```
NameComponent nc1 = new NameComponent("Naptarprojekt", "");
NameComponent nc2 = new NameComponent("Mariskakontextusa", "");
NameComponent útvonal[] = {nc1, nc2};
NamingContext bejegyzettkontextus = ncRef.bind_new_context(útvonal);
```

A fenti példában láthatjuk egy újonnan létrehozott névkontextus objektumnak a CORBA névszolgáltatóba történő bejegyzésének módját.
Az új `Naptarprojekt;Mariskakontextusa` kontextus a `Naptarprojekt` kontextust tároló CORBA névszolgáltatóban lesz létrehozva. Megjegyezzük, hogy egy új kontextust létrehozhattunk volna a `new_context()` metódussal is, és ezt bejegyezhettük volna a `bind_context()` metódussal (ekkor az új kontextus abban a CORBA névszolgáltatóban lesz létrehozva, amely azt a `NamingContext` objektumot kezeli, amelyre vonatkozóan a `new_context()` metódust végrehajtottuk).

5.2. Egy statikus hívási modell alapú kliens alkalmazás

Tekintsük meg a statikus modellel megírt kliens alkalmazásunk listáját!

```
// NaptarKliens.java
//
// args[0] : naptár tulajdonosának a neve
// args[1] : naptáron végrehajtandó művelet (lekerdez/beallit)
// args[2], args[3], args[4] : elfoglaltság ideje éééé/hh/nn
// args[5] : beállításnál a beállítandó elfoglaltság megnevezése
// Más paramétereket (pl. -ORBInitialPort) csak ezek mögé írjunk!

import Elfoglaltsagi_programcsomag.*;
import Elfoglaltsagi_programcsomag.NaptarPackage.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.Calendar;

public class NaptarKliens {

  // Kiírjuk a program hívási szintaxisát
  private static void hivas() {
    System.out.println("Szabálytalan hívás!");
    System.out.println("A helyes hívásmód szintaxisát lásd alább:");
    System.out.println("java NaptarKliens tulaj_neve beallit "+
                       "éééé hh nn elfoglaltságleíró_szöveg");
    System.out.println("java NaptarKliens tulaj_neve lekerdez éééé hh nn");
```

```
    System.out.println("Más paraméterek csak ezek mögött legyenek!");
  }

  // Ez végzi a tényleges feladatokat, azt feltételezve, hogy az args[]
  // paramétervektorban megvan a megadott művelethez szükséges
  // számú paraméter.
  public static void csinald(String[] args) {

    try {
      ORB orb = ORB.init(args, null);
      org.omg.CORBA.Object ncObjRef =
                    orb.resolve_initial_references("NameService");
      NamingContext ncRef=NamingContextHelper.narrow(ncObjRef);
      NameComponent nc = new NameComponent(args[0],"");
      NameComponent útvonal[] = { nc } ;
      Naptar MNaptarRef = NaptarHelper.narrow(ncRef.resolve(útvonal));

      String tulajneve = MNaptarRef.Tulajdonos_neve(); // ki a tulajdonos?
      System.out.println("A talált naptár tulajdonosának neve:"+tulajneve);

      Datum d1 = new Datum(Short.parseShort(args[2]), // IDL dátum típus
                           Short.parseShort(args[3]), // összeállítása
                           Short.parseShort(args[4]));
      if (args[1].equals("beallit")) { // Ha beírni akarunk ...
        MNaptarRef.Beallit(d1,args[5]); //  akkor a Beallit() metódust
      } else {
        // Létrehozzuk a kimenő paraméter számára a paraméter típusának
        // megfelelő "Holder" osztályt.
        StringHolder sh = new StringHolder();
        try {
          // Meghívjuk a távoli metódust a statikus hívási interfésszel
          boolean b = MNaptarRef.Lekerdez(d1,sh); // egyébként a Lekerdez()
          if (b) {                                 // metódust hívjuk meg.
            System.out.println("Eredmény/lekérdez:"+sh.value);
          } else {                                // és kiírjuk az eredményt
            System.out.println("Nincs semmi.");
          }
        } catch (ErvenytelenDatum ede) {
          System.out.println("ErvenytelenDatum kivétel: "+ede.oka);
        } catch (Exception e) {
          System.out.println("Kivételt kaptam az eredmény"+
                             " kinyerésekor: "+e+"/"+e.getMessage());
        }
      }
    } catch (ErvenytelenDatum ede) {
      System.out.println("ErvenytelenDatum kivétel: "+ede.oka);
    } catch (org.omg.CORBA.ORBPackage.InvalidName e) {
      System.out.println("A név érvénytelen: "+e.getMessage());
    } catch (org.omg.CosNaming.NamingContextPackage.NotFound e) {
      System.out.println("A következő név nincs a névszolgáltatóban: "+
                         (e.rest_of_name)[0].id);
```

```
    } catch (java.lang.NumberFormatException e) {
      System.out.println("A megadott szám (dátum) formátuma hibás: "+
                         e.getMessage());
    } catch (Exception e) {
      System.out.println("Egyéb kivétel: "+e.getMessage());
    }
  }

  // A főprogram - main() - metódusa egyszerűen ellenőrzi, hogy a programot
  // megfelelő számú paraméterrel hívtuk-e meg, és ha minden rendben, akkor
  // végrehjtja a fent bemutatott csinald() metódust, átadva neki a
  // program paramétereit.
  public static void main(String[] args) {
    // Argumentumok ellenőrzése
    if (args.length < 5) {
      hivas();
    } else {
      if (args[1].equals("beallit") || args[1].equals("lekerdez")) {
        if ((args[1].equals("beallit") && args.length >= 6) ||
            (args[1].equals("lekerdez") && args.length >= 5)) {
          csinald(args);
        } else {
          hivas();
        }
      } else {
        System.out.println("A második paraméter csak beallit vagy "+
                           "lekerdez lehet!");
      }
    }
  }
}
```

A program indulása után ellenőrzi, hogy megfelelő számú paramétert kapott-e, és ha igen, akkor meghívja a `csinald()` nevű metódust, átadva neki a program paramétereit. Ez megkeresi a naptárat a tulajdonosának neve alapján a CORBA névszolgáltatóból, meghívja a tulajdonos nevét lekérdező `Tulajdonos_neve()` metódust, és kiírja a metódus által visszaadott szöveget (feltehetően ez a tulajdonos neve, amit egyébként az első programparaméterben is megadtunk).

Ezután a program megvizsgálja, hogy a `beallit` vagy a `lekerdez` nevű művelet azonosítót adták-e meg a második paraméterében; ha az előbbit, akkor meghívja a távoli objektum szerverének `Beallit()` metódusát az összeállított dátum objektummal és eseményleírással; ha pedig a `lekerdez` nevű művelet azonosítóját adtuk meg, akkor létrehozza a második, `out` átadási módú szöveges típusú paraméter tárolására szolgáló `StringHolder` objektumot, és meghívja a távoli `Lekerdez()` metódust a megfelelő paraméterezéssel, és kiírja a metódus által visszaadott információkat (illetve kiírja a meghívott metódus által kiváltott kivételekről rendelkezésre álló információkat; láthatjuk, hogy ennek a kezelése ugyanúgy megy, mint ahogyan ezt más - helyben kiváltott - Java kivételek esetén tennénk).

Ez a program is eléggé szigorúan veszi a paramétereinek a sorrendjét és a jelentését (ennek részletes leírását megtalálhatjuk a program elején levő megjegyzés sorokban), így ha más paramétereket is meg akarunk adni (például az `-ORBInitialPort` paramétert is az ORB alapvető névszolgáltatásának elérési helyének megadására), akkor ezeket csak a paraméterlista végére (a program által elvárt paraméterek mögé) tegyük.

A következő parancssorral elindíthatjuk a programunkat, és bejegyezhetünk egy eseményt Mariska nevű felhasználónk naptárába.

```
java NaptarKliens Mariska beallit 1997 10 20 Esemény
```

illetve ha például a névszolgáltatót az 1900-as TCP-portra tettük, akkor a következő paranccsal indítsuk a programunkat:

```
java NaptarKliens Mariska beallit 1997 10 20 Esemény -ORBInitialPort 1900
```

A naptár lekérdezését pedig a következő hívással tehetjük meg:

```
java NaptarKliens Mariska lekerdez 1997 10 20
```

5.3. Egy dinamikus hívási modell alapú kliens alkalmazás

Most elkészítünk egy olyan programot, amely a távoli naptár szolgáltatást nyújtó objektumot csak lekérdezni tudja, módosítani nem. A távoli objektum elérésére most a CORBA dinamikus metódushívási interfészét fogjuk használni. A négyzetreemelő példaprogramnál megismertekhez képest annyival bonyolultabb a helyzet, hogy a meghívott távoli metódus (a `Lekerdez()`) akár egy kivételt is kiválthat, és ennek kezeléséről is gondoskodnunk kell; illetve figyelni kell arra is, hogy a metódushívás második paramétere `out` módon lett átadva, vagyis abban egy értéket fogunk visszakapni.

```
// Lekerdez.java
//
// args[0] : a naptár tulajdonosának neve
// args[1], args[2], args[3] : elfoglaltsag ideje
// Más paramétereket (pl. -ORBInitialPort) csak ezek mögé írjunk!

import Elfoglaltsagi_programcsomag.*;
import Elfoglaltsagi_programcsomag.NaptarPackage.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.Calendar;

public class Lekerdez {
```

```
public static void main(String[] args) {

 if (args.length >= 4) {
  try {
    ORB orb = ORB.init(args, null);
    org.omg.CORBA.Object ncObjRef =
                         orb.resolve_initial_references("NameService");
    NamingContext ncRef=NamingContextHelper.narrow(ncObjRef);
    NameComponent nc = new NameComponent(args[0],"");
    NameComponent útvonal[] = { nc } ;
    Naptar MNaptarRef = NaptarHelper.narrow(ncRef.resolve(útvonal));

    Datum d1 = new Datum(Short.parseShort(args[1]), // IDL dátum típus
                         Short.parseShort(args[2]), // összeállítása
                         Short.parseShort(args[3]));

    // Először összeállítjuk a kiváltható kivételek listáját
    ExceptionList exl=orb.create_exception_list();
    exl.add(ErvenytelenDatumHelper.type());

    NVList hivasi_parameterek=orb.create_list(0);  // üres paraméterlista
    // 1. paraméter
    Any mikori_datumot_kerdezzen_le = orb.create_any();
    DatumHelper.insert(mikori_datumot_kerdezzen_le,d1); // Datum átadása
    hivasi_parameterek.add_value("mikor",  // "mikor" nevű paraméterben
                       mikori_datumot_kerdezzen_le,ARG_IN.value);
    // 2. paraméter
    Any vart_esemeny = orb.create_any();
    vart_esemeny.type(orb.get_primitive_tc(TCKind.tk_string));
    hivasi_parameterek.add_value("mi_van_akkor",
                       vart_esemeny,ARG_OUT.value);

    // Összeállítjuk azt az objektumot, amiben majd az eredményt várjuk
    Any eredmeny = orb.create_any();
    eredmeny.type(orb.get_primitive_tc(TCKind.tk_boolean));
    NamedValue eredmenyNV=orb.create_named_value(null,eredmeny,0);

    // Összeállítjuk a dinamikus metódushívást reprezentáló kérést
    Request keres = MNaptarRef._create_request(null,"Lekerdez",
                    hivasi_parameterek, eredmenyNV,
                    exl, null);
    // Törlöm a hívott metódus által kiváltott kivételek listáját
    keres.env().clear();

    keres.invoke(); // Végrehajtom a távoli metódust

    if (keres.env().exception() != null) {
      // A hívott metódus váltott ki kivételt
      if (keres.env().exception() instanceof UnknownUserException) {
        // IDL fájlban definiált felhasználói kivétel?
        Any kiv=((UnknownUserException)keres.env().exception()).except;
```

```
          if (kiv.type().equals(ErvenytelenDatumHelper.type())) {
            System.out.println("ErvenytelenDatum kivételt kaptam!");
            System.out.println("Valószínűleg olyan dátumot adtál be,"+
                               "amelyet a szerver nem kezel (<1996)!");
            ErvenytelenDatum javaKivetel =
                                    ErvenytelenDatumHelper.extract(kiv);
            System.out.println("OKA:"+javaKivetel.oka);
          } else {
            System.out.println("Nem várt kivételt kaptam vissza!");
          }
        }
        System.out.println("Visszakapott kivétel osztálya:"+
                           keres.env().exception());
      } else {
        // Nem váltott ki kivételt
        boolean b = eredmeny.extract_boolean(); // eredményt megszerezzük
        if (b) { // a szerveren valami be van jegyezve
          System.out.println("A szerveren az alábbi elfoglaltság van"+
                             " bejegyezve:"+vart_esemeny.extract_string());
        } else { // a szerveren nincs bejegyezve semmi az adott napra
          System.out.println("A szerver válasza alapján a kérdéses"+
                             " időpontra semmi sincs bejegyezve.");
        }
      }
    } catch (org.omg.CORBA.ORBPackage.InvalidName e) {
      System.out.println("A név érvénytelen: "+e.getMessage());
    } catch (org.omg.CosNaming.NamingContextPackage.NotFound e) {
      System.out.println("A következő név nincs a névszolgáltatóban:"+
                         (e.rest_of_name)[0].id);
    } catch (java.lang.NumberFormatException e) {
      System.out.println("A megadott szám (dátum) formátuma hibás: "+
                         e.getMessage());
    } catch (Exception e) {
      System.out.println("Egyéb kivétel: "+e.getMessage());
    }
   } else {
     System.out.println("Hívás:java Lekerdez tulajdonos_neve éééé hh nn");
   }
  }
}
```

A fenti program a távoli objektumreferencia megszerzése után először összeállítja a `Lekerdez()` távoli metódus első paraméterében átadni kívánt dátum objektumot.

A következő lépésben a program összeállítja a kiváltható kivételek listáját, és felveszi bele az egyetlen ilyen kivételt, az érvénytelen dátumot jelző kivételét.

Ezután létre lesz hozva a távoli metódusnak átadandó paraméterlista: először létrehozunk egy üres listát, és ehhez hozzáadjuk a távoli metódus első (`mikor` nevű) paraméterének átadandó értéket tároló `Any` objektumot (belemásolva az átadni kívánt dátum értéket), majd ezután hozzáfűzzük a távoli metódus (`mi_van_akkor` nevű)

paraméterének átadandó értéket tároló `Any` típusú objektumot (megjelölve, hogy milyen típusú értéket várunk benne, ui. az egy kimenő paraméter).

Ezután létrehozunk egy új `NamedValue` osztályba tartozó objektumot, amibe szintén egy Any objektumot másolunk, ami majd a visszatérési értéket fogja tárolni. Ezen is be kell állítani azt, hogy milyen típusú értéket várunk a távoli metódustól (a példánkban itt a logikai típust azonosító típuskódot adjuk meg).

Ezután létrehozzuk a dinamikus metódushívást reprezentáló kérésobjektumot az előbb létrehozott adatok alapján. A metódushívás ekkor még nem lesz végrehajtva.

A metódushívás végrehajtása előtt töröljük a kiváltott kivételek listáját, és a metódushívást reprezentáló objektum `invoke()` metódusának meghívásával ekkor végrehajthatjuk a távoli metódus meghívását.

A távoli metódus visszatérése után megvizsgáljuk, hogy váltott-e ki kivételt. Ha igen (ezt onnan tudhatjuk, hogy a kérés környezetének a kivételleíró komponense egy `CORBA::UnknownUserException` osztály példánya vagy annak egy leszármazottja), akkor megvizsgáljuk, hogy milyen kivétel lett kiváltva. Ez úgy történhet, hogy az előbb említett kivételleíró-komponens objektumban tárolt állapotváltozó lekérdezésével megkaphatjuk a kiváltott kivételről információkat tároló `Any` osztálybeli objektumot, és ennek típuskódját összehasonlítva a "várható" kivételek típuskódjának valamelyikével megtudhatjuk, hogy milyen kivételt váltott ki a hívott metódus. A programban láthatjuk azt is, hogy hogyan szerezhetjük meg az `Any` objektumból a kiváltott kivétel Java kivétel reprezentánsát.

```
package org.omg.CORBA;

public class UnknownUserException extends UserException {

    public Any except; // lekerdezes
    public UnknownUserException(); // konstruktor
    public UnknownUserException(Any a); // except legyen = a

}
```

Ha a távoli metódus nem váltott ki kivételt, akkor az előbbi lépésekben a visszatérési értéknek, illetve a kimenő paraméterek értékeinek lefoglalt `Any` osztálybeli objektumokban tárolt adatokból hozzájuthatunk mind a metódus visszatérési értékéhez, mind pedig a kimenő paraméterek értékéhez.

5.4. A szerveroldal implementációja applet formájában

Ebben a pontban bemutatjuk azt, hogy hogyan készíthető el a szerveroldal egy Java applet formájában. Az elkészült applet egy elfoglaltságokat nyilvántartó CORBA objektumot jegyez be a CORBA névszolgáltatóba. Az implementált naptárszolgáltatás interfész megegyezik a korábban már bemutatottal, így az applet teszteléséhez a már

bemutatott kliens implementációk bármelyikét felhasználhatjuk. Az applet felhasználói felülete egyetlen grafikus elemet tartalmaz, amely egy listában felsorolja a naptárba bejegyzett elfoglaltságokat.

Az applet megvalósítását egy `NaptarAppletSzerver` nevű osztály tartalmazza. Használatához az appletet tartalmazó HTML-oldalon néhány paramétert be kell állítani. A beállítandó paraméterek a következők (az ORB paraméterezéséről korábban már írtunk):

`org.omg.CORBA.ORBInitialHost` : az elérni kívánt névszolgáltatót tartalmazó számítógép neve

`org.omg.CORBA.ORBInitialPort` : az elérni kívánt névszolgáltató eléréséhez használandó TCP port

`tulajdonos` : a naptár tulajdonosának a neve, a naptárszolgáltató objektum ezen a néven lesz bejegyezve a CORBA névszolgáltatójába (ezt be kell írni a HTML-lapra).

Ügyeljünk arra, hogy az appletek gyakran csak a letöltési helyükön levő számítógéppel építhetnek fel TCP kapcsolatot, ezért az `org.omg.CORBA.ORBInitialHost` paraméterben általában annak a számítógépnek a nevét kell megadni, ahonnan az appletet letöltik (és összeköttetés létesítési kérelmeket is csak azoktól a számítógéptől fogadhat, ahonnan letöltötték). Ez eléggé korlátozza az applet használhatóságát, hiszen ez azt jelenti, hogy a kliens alkalmazásokat csak az applet letöltési helyén lehet elindítani. Léteznek ún. IIOP- átjáró szoftverek, amelyek - hasonlóan a Java RMI-nél megismert hálózati tűzfal átjáróhoz - a más számítógépekről érkező kliens kéréseket továbbítják az objektumimplementációhoz, de hosszabb távon valószínűleg az a megoldás fog elterjedni, hogy az appleteket digitálisan aláírják, és ezek az appletek nem lesznek korlátozva abban, hogy mit csinálhatnak.

Az applet forráskódja a következő listában látható:

```
// NaptarAppletSzerver.java
//
// A Naptár interfészünk implementációját megvalósító szerverapplet.
// Paraméterében kell megadni a naptár tulajdonosának nevét (azonosítóját),
// amit a névszolgáltatóba bejegyez (ui. a továbbiakban a naptárat azon
// keresztül érhetjük el)
// Példa felhasználására:
// ... html fájlban ...
//
// <HTML>
// ...
// <APPLET CODE=NaptarAppletSzerver.class
//         WIDTH=500
//         HEIGHT=300>
// <PARAM name="org.omg.CORBA.ORBInitialHost" value=rozsika>
// <PARAM name="org.omg.CORBA.ORBInitialPort" value=900>
```

```
// <PARAM name="tulajdonos" value="ember2">
// </APPLET>
// ...
// </HTML>
//
//
// A tulajdonos nevét a "tulajdonos" applet-paraméterben adtuk meg.

import Elfoglaltsagi_programcsomag.*;
import Elfoglaltsagi_programcsomag.NaptarPackage.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.Hashtable;

import java.applet.Applet;
import java.awt.*;

public class NaptarAppletSzerver extends Applet
             implements _NaptarOperations {

  String tulajneve; // Ebben tároljuk a tulajdonos nevét
  Hashtable elfoglaltság; // Ebben tároljuk az elfoglaltságokat

  // ***** Applet mivolttal kapcsolatos részek ************************

  List elfogl; // Az elfoglaltságokat kiíró felhasználói felület objektum

  public void init()
  {
     elfogl=new List(5,false); // Ötelemű, egy elem választható ki
     this.elfoglaltság = new Hashtable(); // Üres hashtáblát hozok létre

     tulajneve = getParameter("tulajdonos"); // Lekérdezi az
                                             // appletparamétert
     System.out.println("Tulajdonos neve:"+tulajneve);
     try {
      java.util.Properties p = new java.util.Properties();
      p.put("org.omg.CORBA.ORBClass","com.sun.CORBA.iiop.ORB"); // JOE ORB
                                                                // osztálya
      ORB orb = ORB.init(this, p); // ORB inicializálása
      NaptarAppletSzerver MN_Kiszolg = this; // Mi magunk implementáljuk a
                                             // kiszolgáló funkcionalitást,
                                             // belőlünk pedig már van egy
                                             // példány, a  this
      Naptar MNaptárRef = new _NaptarTie(MN_Kiszolg);
      orb.connect(MNaptárRef); // Létrehozott új objektum bejelentése

      org.omg.CORBA.Object ncObjRef =
                orb.resolve_initial_references("NameService");
      NamingContext ncRef=NamingContextHelper.narrow(ncObjRef);
```

```
    // A tulajdonos nevén jegyezzük be a naptárat a névszolgáltatóba
    NameComponent nc = new NameComponent(this.tulajneve,"");
    NameComponent útvonal[] = { nc } ;
    ncRef.rebind(útvonal, MNaptárRef); // Bejegyezzük a névszolgáltatóba

    add(elfogl);  // A felhasználói felület lista alkotóelemét
                  // felrakjuk az ablakba
    validate();   // A böngésző itt kiszámolja, hogy mi mekkora legyen
                  // a képernyőn

   } catch (Exception e) {
    System.out.println("Kivételt kaptam, program leállt."+
                       " A kivétel oka: "+e+"/"+e.getMessage());
   }
}

// ***** CORBA+objektumimplementációval kapcsolatos részek **********

private String Szöveg_Dátumból(Datum mi) { // Ilyenek a hashtábla kulcsok
  return(new Short(mi.Ev).toString()+"/"+
         new Short(mi.Honap).toString()+"/"+
         new Short(mi.Nap).toString());
}

// Beír egy elfoglaltságot a naptárba
public void Beallit(Elfoglaltsagi_programcsomag.NaptarPackage.Datum
       mikor, String mi_van_akkor)
       throws Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
{

  if (mikor.Ev < 1996) { // Ekkor dobunk kivételt
    throw new Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
              (" 1996 elotti ");
  }
  synchronized (elfoglaltság) {
    // Ha a hashtáblában az illető dátumnál már van valami -> kivétel
    if (elfoglaltság.containsKey(Szöveg_Dátumból(mikor))) {
     throw new Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
         (" Már foglalt ");
    }
    // A következő utasításban levő addItem() metódushívás a Java 1.2
    // változat megjelenésétől kezdődően elavultnak van minősítve.
    // Helyette ugyanilyen paraméterezésű  add()  metódust használhatunk.
    // Mivel a legtöbb webböngésző még nem támogatja a Java 1.1-et vagy
    // Java 1.2-t, ezért mi az  addItem()  metódust használjuk.
    // Újabb Java változatoknál egyszerűen írjuk át a metódus nevét
    // add-ra.
    elfogl.addItem(Szöveg_Dátumból(mikor)+": "+mi_van_akkor);
         // megjelenítettük
    elfoglaltság.put(Szöveg_Dátumból(mikor),mi_van_akkor); // bejegyezzük
  }
```

```
  }

  // Lekérdezzük egy dátum alapján az akkori elfoglaltságot
  public boolean Lekerdez(Elfoglaltsagi_programcsomag.NaptarPackage.Datum
         mikor, org.omg.CORBA.StringHolder mi_van_akkor)
         throws Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
  {

    if (mikor.Ev < 1996) { // Ekkor dobunk kivételt
      throw new Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
                (" 1996 elotti ");
    }

    mi_van_akkor.value="Nincs adat";
    synchronized (elfoglaltság) {
      if (!elfoglaltság.containsKey(Szöveg_Dátumból(mikor))) {
        return false; // Még nem foglalt az időpont
      }
      mi_van_akkor.value=(String) elfoglaltság.get(Szöveg_Dátumból(mikor));
      return true; // Már foglalt ...
    }
  }

  public String Tulajdonos_neve() { // Egyszerűen visszaadjuk a
    return tulajneve;               // tulajdonos nevét
  }

}
```

Mivel a fenti példában egy applet lát el CORBA szerver feladatokat, és az appletet a szokásos java.applet.Applet osztálytól kell származtatni, ezért kénytelenek vagyunk a CORBA szerver feladatokat delegációs (TIE) módon megoldani. Mivel a CORBA szerverfeladatokat ugyanebben az osztályban implementáltuk, ezért a delegáció ugyanehhez az osztályhoz történik (az applet osztálya implementálja a `_NaptarOperations` interfészt, ezt jeleztük is a specifikációjában).

A forráskódban szétválasztottuk az applet inicializálással kapcsolatos részeket a CORBA objektumszerver feladatait ellátó metódusoktól.

Az applet inicializáló metódusa létrehoz egy lista felhasználói felület objektumot, amiben az elfoglaltságok a későbbiekben meg lesznek jelenítve, majd létrehoz egy hashtáblát az elfoglaltságok tárolására. Ezután lekérdezi a `tulajdonos` appletparaméter értékét. A következő lényeges lépés az ORB inicializálása, ami az appleteknél megszokott módon történik, de a példában explicite kijelöltük, hogy a `com.sun.CORBA.iiop.ORB` osztályban implementált ORB-t akarjuk használni. Ha az appletet olyan böngészőben akarjuk futtatni, amelybe nem a JOE ORB van beépítve, akkor az applet futása közben letölti a működéséhez szükséges osztályokat (ezért ne felejtsük el a megfelelő osztályokat felmásolni az applet letöltési helyére). Ezután létrehozunk egy tie-osztályt, amelynek a paraméterében megadjuk az applet objektumnak a `this` referenciáját (kiszolgáló

osztályként, ui. ez az osztály implementálja a CORBA objektum metódusait). Az ORB `connect()` metódusának a paraméterében a tie-osztály referenciáját adjuk meg, majd ezt jegyezzük be a névszolgáltatóba is. A CORBA alapú naptármetódusok implementációjával most nem foglalkozunk, mivel ezt korábban már láthattuk. Annyit azért megemlítünk, hogy a `Beallit` metódust kibővítettük egy olyan utasítással, amivel a bejegyzett elfoglaltságot felírjuk a felhasználói felületre. Tekintsük a következő HTML-lapot, az appletünk egy lehetséges felhasználási lehetőségével. A fájl neve legyen `Ember2Naptara.html`.

```
<HTML>
<HEAD>
 <TITLE>NaptarApplet szerver peldaprogram ("ember2" felhasznalo)</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">

<H1 ALIGN=CENTER>Az "ember2" felhasznalo naptarja</H1>
<HR>

<APPLET CODE=NaptarAppletSzerver.class
        WIDTH=500
        HEIGHT=300>
<PARAM name="org.omg.CORBA.ORBInitialHost" value=rozsika>
<PARAM name="org.omg.CORBA.ORBInitialPort" value=900>
<PARAM name="tulajdonos" value="ember2">
</APPLET>

</BODY>
</HTML>
```

A lefordított appletet és a fenti HTML-lapot tegyük fel a számítógépünk webszerverére, a futáshoz szükséges ORB osztályokkal együtt (egyszerűen kicsomagolhatjuk ide az ezeket tartalmazó JAR fájlokat, vagy az `ARCHIVE` HTML appletparaméterben kijelölhetjük az ezeket is tartalmazó JAR fájlt).

Megjegyezzük, hogy a böngészőprogramnak a fenti HTML-lap elérési címét `http://rozsika/Ember2Naptara.html` formában kell megadni, amiből a lényeg a `rozsika` számítógépnéven van (hiszen könnyen eszünkbe juthatna ide `localhost` nevet is írni, ha e kettő ugyanazt a számítógépet azonosítja). Azért kell erre vigyázni, mert csak az egyik mögött levő IP-cím egyezik meg az applet letöltési helyével, és mivel a HTML-lapban a névszolgáltató elérésére a `rozsika` címet adtuk meg, ezért az csak akkor tudja elérni a névszolgáltatót, ha ez egyben az applet letöltési helye is.

Az alábbi parancs segítségével bejegyezhetünk egy elfoglaltságot az `ember2` felhasználó naptárába.

```
java NaptarKliens ember2 beallit 1998 05 25 Konferencia
```

5.5. Egy lekérdező applet kliens

Most megnézünk egy egyszerű appletet, amelynek segítségével lekérdezhetjük egy naptár tartalmát akár a WWW-n keresztül is. Az applet felhasználói felülete nagyon egyszerű, mivel a célunk itt elsősorban annak a bemutatása, hogy hogyan érhetünk el CORBA objektumszervereket Java appletekből, nem pedig a Java nyelv felhasználói felület készítési csomagjának a bemutatása a cél.

```
// LekerdezoApplet.java
//
// Lekérdezési feladatokat támogató Java applet.

import Elfoglaltsagi_programcsomag.*;
import Elfoglaltsagi_programcsomag.NaptarPackage.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.Calendar;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class LekerdezoApplet extends Applet implements ActionListener {

  Button lekerdezoGomb = new Button(" Lekérdezés ");
  TextField tulajNeve = new TextField();
  TextField datumEv = new TextField(4);
  TextField datumHonap = new TextField(2);
  TextField datumNap = new TextField(2);
  Label mivanakkor = new Label("_________________________");
  ORB orb;

  public void init() {
    java.util.Properties p = new java.util.Properties();
    p.put("org.omg.CORBA.ORBClass","com.sun.CORBA.iiop.ORB"); // JOE ORB
    orb = ORB.init(this, p); // ORB inicializálása

    Panel panel=new Panel();
    panel.setLayout(new GridLayout(6,1));
    Panel tulajpanel=new Panel();   // tulajdonos neve
    tulajpanel.setLayout(new BorderLayout());
    tulajpanel.add("West", new Label("Tulajdonos neve:"));
    tulajpanel.add("Center", tulajNeve);
    panel.add(tulajpanel);

    Panel evpanel=new Panel(); // elfoglaltság éve
    evpanel.setLayout(new BorderLayout());
    evpanel.add("West", new Label("Elfoglaltság éve:"));
    evpanel.add("Center", datumEv);
    panel.add(evpanel);
```

```
  Panel honappanel=new Panel(); // elfoglaltság hónapja
  honappanel.setLayout(new BorderLayout());
  honappanel.add("West", new Label("Elfoglaltság hónapja:"));
  honappanel.add("Center", datumHonap);
  panel.add(honappanel);

  Panel nappanel=new Panel(); // elfoglaltság napja
  nappanel.setLayout(new BorderLayout());
  nappanel.add("West", new Label("Elfoglaltság napja:"));
  nappanel.add("Center", datumNap);
  panel.add(nappanel);

  panel.add(lekerdezoGomb);
  lekerdezoGomb.addActionListener(this);

  panel.add(mivanakkor);

  add(panel);
  validate();
}

public void actionPerformed(ActionEvent evt) {
  if (evt.getSource() == lekerdezoGomb) {
   try {
    org.omg.CORBA.Object ncObjRef =
                orb.resolve_initial_references("NameService");
    NamingContext ncRef=NamingContextHelper.narrow(ncObjRef);
    NameComponent nc = new NameComponent(tulajNeve.getText(),"");
    NameComponent útvonal[] = { nc } ;
    Naptar MNaptarRef = NaptarHelper.narrow(ncRef.resolve(útvonal));

    Datum d1 = new Datum(Short.parseShort(datumEv.getText()),
                         Short.parseShort(datumHonap.getText()),
                         Short.parseShort(datumNap.getText()));

      // Létrehozzuk a kimenő paraméter számára a paraméter típusának
      // megfelelő "Holder" osztályt.
      StringHolder sh = new StringHolder();
      try {
        // Meghívjuk a távoli metódust a statikus hívási interfésszel
        boolean b = MNaptarRef.Lekerdez(d1,sh); // egyébként a Lekerdez()
        if (b) {                                 // metódust hívjuk meg.
          mivanakkor.setText(sh.value);
        } else {                                // és kiírjuk az eredményt
          mivanakkor.setText("Nincs semmi.");
        }
      } catch (ErvenytelenDatum ede) {
        mivanakkor.setText("!!!! "+ede.oka);

      }
      catch (Exception e) {
```

```
          System.out.println("Kivételt kaptam az eredmény"+
                             " kinyerésekor: "+e+"/"+e.getMessage());
        }

     } catch (Exception e) {
          System.out.println("Kivételt kaptam az eredmény"+
                             " kinyerésekor: "+e+"/"+e.getMessage());
     }
    }
  }
}
```

A fenti applet `init()` metódusa inicializálja az ORB szoftvert reprezentáló objektumot, majd létrehozza a program felhasználói felületét. A felhasználói felület hat sorból áll, amely a következőképpen épül fel.

A felhasználói felület első sora tartalmaz egy feliratot és egy szövegbeadási mezőt. A feliraton a `Tulajdonos neve:` szöveg látható. A felhasználónak a felirat melletti szövegbeadási mezőben kell megadni annak a naptárnak a tulajdonosának a nevét, amely naptár tartalmát le akarja kérdezni. A felhasználói felület második sora szintén egy feliratot és egy szövegbeadási mezőt tartalmaz. A feliraton az `Elfoglaltság éve:` szöveg látható. A felhasználónak a felirat melletti szövegbeadási mezőben egy évszámot kell megadni (például 1998). A felhasználói felület következő két sorának szerkezete ehhez hasonló, de az azokban levő szövegbeadási mezőkben egy hónapsorszámot (például 5), illetve egy napsorszámot (például 24) kell megadni. Vigyázzunk, nehogy felesleges karaktereket (például szóközöket) írjunk az adatbeadási mezőkbe, mert akkor a numerikusra konvertálásuk sikertelen lesz, és egy ezt jelző kivétel lesz kiváltva, amiről a program a Java konzolra ír ki részletesebb információkat. A felhasználói felület következő sorában egy nyomógomb található `Lekérdezés` felirattal. A nyomógomb megnyomásához hozzárendeltük az applet objektumának az `actionPerformed()` metódusát. A felhasználói felület hatodik sorában kezdetben aláhúzás jelek vannak. Később itt lesz kiírva a lekérdezés eredményeként visszakapott szöveg.

A számunkra érdekes rész az `actionPerformed()` metódusban van. Itt a program megvizsgálja, hogy a lekérdezés feliratú nyomógombot nyomták-e meg, és ha igen, akkor a korábbi naptárlekérdező kliens példaprogramban megismert módon elvégzi a felhasználói felületen beadott nevű tulajdonos naptárának a lekérdezését, és a lekérdezés eredményét kiírja a felhasználói felület már említett hatodik sorában. Látható, hogy a program a nyomógomb minden egyes megnyomásakor lekérdezi a névszolgáltatót, majd eléri a megadott nevű CORBA naptárobjektumot. Az alkalmazott CORBA objektumelérési mechanizmus megegyezik a statikus hívási modell alapú kliens alkalmazásunk lekérdező részével, azzal a különbséggel, hogy itt az appleteknél használandó ORB inicializáló metódust hívtuk meg az applet `init()` metódusában.

Az appletet elindíthatjuk akár egy webböngésző programban, akár az `appletviewer` programmal. Én ez utóbbival próbáltam ki, mivel az általam használt webböngésző nem

támogatja a Java 1.1-es eseménykezelési mechanizmusait (ha egy ilyen böngészőben akarjuk a programot futtatni, akkor át kell írni a Java 1.0 eseménykezelési mechanizmusaira épülve). Az applet futtatásához kell egy HTML fájl is, amelyben hivatkozunk az appletre. Egy ilyen fájlra tekintsük a következő példát:

```
<HTML>
<HEAD>
   <TITLE>Naptarlekerdezo applet</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">

<H1 ALIGN=CENTER>Naptarlekerdezo applet</H1>
<HR>

<APPLET CODE=LekerdezoApplet.class
        WIDTH=300
        HEIGHT=200>
<PARAM name="org.omg.CORBA.ORBInitialHost" value=rozsika>
<PARAM name="org.omg.CORBA.ORBInitialPort" value=900>
</APPLET>

</BODY>
</HTML>
```

Megjegyezzük, hogy ha a CORBA objektumszerver egy érvénytelen dátumot jelző kivételt vált ki, akkor a lekérdezés eredményét jelző feliratra a kivétel okát megjelölő szöveget írjuk ki négy darab felkiáltójelet és egy szóközt követően.

A fenti applet csak azon a számítógépen levő naptárobjektumokat tudja elérni, amelyről le lett töltve, hacsak nincs megfelelő módon digitálisan aláírva.

5.6. A szerveroldal dinamikus megvalósítása

A következő példaprogram egy naptárobjektum szerver, amely a dinamikus szerveroldali csonk-interfész segítségével tartja a kapcsolatot az ORB szoftverrel.

A program egyetlen Java osztályból áll, melynek neve `DSINaptarKiszolgalo`. A korábban már látott statikus naptárszerver implementációhoz képest ebben a példában a legfontosabb metódus az `invoke()` metódus. Ez a metódus a klienstől érkező kérések kiszolgálását végzi, paramétere egy kliens kérését reprezentáló objektum.

Az `invoke()` metódus egyszerűen megvizsgálja, hogy a kliens milyen nevű metódust kíván meghívni, majd annak megfelelően összeállít egy listát. Később a kliens által átadott paramétereket ebben a listában kaphatja meg. A paraméterlistába megfelelő számú `Any` típusú objektumot fűzünk fel, az egyes paramétereknél megadva azok típusát és átadási módját (az illető paraméterek várt típusát). A paraméterek értékeihez a kliens kérését reprezentáló objektum `params()` metódusának meghívásával juthatunk. Az értékek az előbb említett `Any` objektumokba lesznek berakva, az egyes értékeket az

illető objektum `extract()`, illetve `extract_<típusnév>()` metódusával kaphatjuk vissza. A program a paraméterek kicsomagolása után meghívja a megfelelő szolgáltató metódust (ezek a metódusok ugyanazok, mint amiket a statikus szerveroldali megoldásnál láthattunk). Ha a meghívott szolgáltató metódus kivételt váltott ki, akkor a hívás helyén azt elkapjuk, és az `except()` metódussal visszaküldjük a hívás helyére. Ha viszont a szolgáltató metódus nem váltott ki kivételt, akkor a visszatérési értéknek létrehozunk egy `Any` típusú objektumot, és belecsomagoljuk abba, majd a `result()` metódussal visszaküldjük azt a hívás helyére.

```
// DSINaptarKiszolgalo.java
//
// A Naptár interfészünk implementációját megvalósító szerver.
// Dinamikus szervercsonkra épül.
// Egyetlen argumentumot vár, a naptár tulajdonosának nevét (azonosítóját),
// amit a névszolgáltatóba bejegyez (ui. a továbbiakban a naptárat azon
// keresztül érhetjük el)
// Más paramétereket (pl. -ORBInitialPort) csak ez után írjunk.

import Elfoglaltsagi_programcsomag.*;
import Elfoglaltsagi_programcsomag.NaptarPackage.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.Hashtable;

public class DSINaptarKiszolgalo extends DynamicImplementation {

  ORB orbV; // Az ORB szoftvert reprezentáló objektum
  String tulajneve; // Eltároljuk a tulajdonos nevét
  Hashtable elfoglaltság; // Ebben tároljuk az elfoglaltságokat

  public DSINaptarKiszolgalo(ORB o, String tulajneve) { // A konstruktor
    this.orbV = o;
    this.tulajneve=tulajneve; // Eltárolom a tulajdonos nevét
    this.elfoglaltság = new Hashtable(); // Üres hashtáblát hozok létre
  }

  private final String ifaceIds[] = {
    "IDL:Elfoglaltsagi_programcsomag/Naptar:1.0"
  };

  public String[] _ids() { // Visszaadja, hogy melyik IDL interfészeket
                           // implementálja ez az osztály
    return (String[]) ifaceIds.clone();
  }

  public void invoke(ServerRequest kérés) {
     NVList paraméterlista = orbV.create_list(0);
     if (kérés.op_name().equals("Beallit")) {
       Any mikori_dátum = orbV.create_any();
```

```
  mikori_dátum.type(DatumHelper.type()); // a paraméter várt típusa
  paraméterlista.add_value("mikor", mikori_dátum, ARG_IN.value);
  Any elfoglaltság = orbV.create_any();
  elfoglaltság.type(ORB.init().get_primitive_tc(TCKind.tk_string));
  paraméterlista.add_value("mi_van_akkor", elfoglaltság,ARG_IN.value);
  kérés.params(paraméterlista); // Elkérjük a paramétereket az ORB-től
  Datum mikor = DatumHelper.extract(mikori_dátum);
  String mi_van_akkor = elfoglaltság.extract_string();
  try {
    Beallit(mikor, mi_van_akkor);
  }
  catch (ErvenytelenDatum ede) {
    // Ha a metódus implementációja kivételt váltott ki, akkor
    // elkapjuk a kiváltott kivételt, és ezt adjuk vissza a hívónak
    Any kiváltott_kivétel = orbV.create_any();
    ErvenytelenDatumHelper.insert(kiváltott_kivétel, ede);
    kérés.except(kiváltott_kivétel); // Eredmény: kivétel
    return;
  }
  // Előkészítjük a visszatérési értéket (ami most nincs, azaz void)
  Any visszaad = orbV.create_any();
  visszaad.type(orbV.get_primitive_tc(TCKind.tk_void));
  kérés.result(visszaad); // Eredmény: nincs visszatérési érték
} else if (kérés.op_name().equals("Lekerdez")) {
  Any mikori_dátum = orbV.create_any();
  mikori_dátum.type(DatumHelper.type()); // a paraméter várt típusa
  paraméterlista.add_value("mikor", mikori_dátum, ARG_IN.value);
  Any elfoglaltság = orbV.create_any();
  elfoglaltság.type(ORB.init().get_primitive_tc(TCKind.tk_string));
  paraméterlista.add_value("mi_van_akkor",elfoglaltság,ARG_OUT.value);
  kérés.params(paraméterlista); // Elkérjük a paramétereket az ORB-től
  Datum mikor = DatumHelper.extract(mikori_dátum);
  StringHolder mi_van_akkor = new StringHolder();
  boolean van_bejegyezve;
  try {
    van_bejegyezve = Lekerdez(mikor, mi_van_akkor);
  }
  catch (ErvenytelenDatum ede) {
    // Visszaadjuk a kiváltott kivételt
    Any kiváltott_kivétel = orbV.create_any();
    ErvenytelenDatumHelper.insert(kiváltott_kivétel, ede);
    kérés.except(kiváltott_kivétel); // Eredmény: kivétel
    return;
  }
  // Előkészítjük a visszatérési értéket
  elfoglaltság.insert_string(mi_van_akkor.value);
  Any visszaad = orbV.create_any();
  visszaad.insert_boolean(van_bejegyezve);
  kérés.result(visszaad); // Eredmény: foglalt-e az adott nap
} else if (kérés.op_name().equals("Tulajdonos_neve")) {
  kérés.params(paraméterlista); // Meghívjuk az ORB metódusát, de
```

```
                                    // itt nem várunk paramétereket, ui.
                                    // a paraméterlista egy üres lista
      String visszaadandó_név = Tulajdonos_neve();
      // Előkészítjük a visszatérési értéket
      Any visszaad = orbV.create_any();
      visszaad.insert_string(visszaadandó_név);
      kérés.result(visszaad); // Eredmény: tulajdonos neve
    } else {
      throw new BAD_OPERATION(0, CompletionStatus.COMPLETED_MAYBE);
    }
  }

  // Szolgáltatómetódusok

  private String Szöveg_Dátumból(Datum mi) { // Ilyenek a hashtábla kulcsok
    return(new Short(mi.Ev).toString()+"/"+
           new Short(mi.Honap).toString()+"/"+
           new Short(mi.Nap).toString());
  }

  // Beír egy elfoglaltságot a naptárba
  public void Beallit(Elfoglaltsagi_programcsomag.NaptarPackage.Datum
         mikor, String mi_van_akkor)
         throws Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
  {

    if (mikor.Ev < 1996) { // Ekkor dobunk kivételt
      throw new Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
                (" 1996 elotti ");
    }
    synchronized (elfoglaltság) {
      // Ha a hashtáblában az illető dátumnál már van valami -> kivétel
      if (elfoglaltság.containsKey(Szöveg_Dátumból(mikor))) {
       throw new Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
           (" Már foglalt ");
      }
      elfoglaltság.put(Szöveg_Dátumból(mikor),mi_van_akkor); // bejegyezzük
    }
  }

  // Lekérdezzük egy dátum alapján az akkori elfoglaltságot
  public boolean Lekerdez(Elfoglaltsagi_programcsomag.NaptarPackage.Datum
         mikor, org.omg.CORBA.StringHolder mi_van_akkor)
         throws Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
  {

    if (mikor.Ev < 1996) { // Ekkor dobunk kivételt
      throw new Elfoglaltsagi_programcsomag.NaptarPackage.ErvenytelenDatum
                (" 1996 elotti ");
    }
```

```
    mi_van_akkor.value="Nincs adat";
    synchronized (elfoglaltság) {
      if (!elfoglaltság.containsKey(Szöveg_Dátumból(mikor))) {
        return false; // Még nem foglalt az időpont
      }
      mi_van_akkor.value=(String) elfoglaltság.get(Szöveg_Dátumból(mikor));
      return true; // Már foglalt ...
    }
  }

  public String Tulajdonos_neve() { // Egyszerűen visszaadjuk a
    return tulajneve;               // tulajdonos nevét
  }

  // Főprogram - innen indul a végrehajtás

  public static void main(String[] args) {

    if (args.length >= 1) { // Egy paraméter van
     try {
      java.lang.Object holtpont = new java.lang.Object();
      ORB orb = ORB.init(args, null); // ORB inicializálása
      DSINaptarKiszolgalo MNaptárRef =
                          new DSINaptarKiszolgalo (orb,args[0]);
      orb.connect(MNaptárRef); // Létrehozott új objektum bejelentése

      org.omg.CORBA.Object ncObjRef =
                orb.resolve_initial_references("NameService");
      NamingContext ncRef=NamingContextHelper.narrow(ncObjRef);
      NameComponent nc = new NameComponent(args[0],"");
      NameComponent útvonal[] = { nc } ;
      ncRef.rebind(útvonal, MNaptárRef); // Bejegyezzük a névszolgáltatóba

      synchronized (holtpont) {
        holtpont.wait(); // várakozunk a kliensekre
      }

     } catch (Exception e) {
      System.out.println("Kivételt kaptam, program leállt."+
                         " A kivétel oka: "+e.getMessage());
     }
    } else {
      System.out.println("Indítás: java DSINaptarKiszolgalo "+
                         "naptártulajdonos_neve");
    }
  }
}
```

Az osztály lefordítása után a következő paranccsal indíthatjuk el például az `ember4` felhasználó naptárát.

```
java DSINaptarKiszolgalo ember4
```

A. Függelék

Az OMG IDL nyelv módosított BNF nyelvtana

(Java vonatkozású részek, az OMG CORBA 2.0 specifikációja alapján.)

```
<IDL specifikáció>    ::= <definíció> { <definíció> }
<definíció>           ::= <típusdeklaráció> ";" | <konstansdeklaráció> ";"
                        | <kivételdeklaráció> ";" | <interfészdeklaráció> ";"
                        | <moduldeklaráció> ";"
<moduldeklaráció>     ::= "module" <azonosító> "{"  <definíció>
                          { <definíció> } "}"
<interfészdeklaráció> ::= <interfészleírás> | <FORWARDdeklaráció>
<interfészleírás>     ::= <interfészfejléc> "{" <interfésztörzs> "}"
<FORWARDdeklaráció>   ::= "interface" <azonosító>
<interfészfejléc>     ::= "interface" <azonosító> [ <öröklés_specifikálása> ]
<interfésztörzs>      ::= { <export_specifikáció> }
<export_specifikáció> ::= <típusdeklaráció> ";" | <konstansdeklaráció> ";"
                        | <kivételdeklaráció> ";" | <attribútumdeklaráció> ";"
                        | <műveletdeklaráció> ";"
<öröklés_specifikálása> ::= ":" <sznév> { "," <sznév> }
<sznév>               ::= <azonosító> | "::" <azonosító>
                        | <sznév> "::" <azonosító>
<konstansdeklaráció>  ::= "const" <konstans_típus> <azonosító> "="
                          <konstans_kifejezés>
<konstans_típus>      ::= <integer_típus> | <karakteres_típus>
                        | <boolean_típus> | <lebegőpontos_típus>
                        | <karakterlánc_típus> | <sznév>
<konstans_kifejezés>  ::= <logikai_vagy_kifejezés>
<logikai_vagy_kifejezés> ::= <logikai_kizáró_vagy_kifejezés>
                        | <logikai_vagy_kifejezés> "|"
                          <logikai_kizáró_vagy_kifejezés>
<logikai_kizáró_vagy_kifejezés> ::= <logikai_és_kifejezés>
                        | <logikai_kizáró_vagy_kifejezés> "^"
                          <logikai_és_kifejezés>
<logikai_és_kifejezés> ::= <shift_kifejezés>
                        | <logikai_és_kifejezés> "&" <shift_kifejezés>
```

```
<shift_kifejezés>      ::= <additív_kifejezés>
                         | <shift_kifejezés> ">>" <additív_kifejezés>
                         | <shift_kifejezés> "<<" <additív_kifejezés>
<additív_kifejezés>    ::= <multiplikatív_kifejezés>
                         | <additív_kifejezés> "+" <multiplikatív_kifejezés>
                         | <additív_kifejezés> "-" <multiplikatív_kifejezés>
<multiplikatív_kifejezés> ::= <unáris_kif>
                         | <multiplikatív_kifejezés> "*" <unáris_kif>
                         | <multiplikatív_kifejezés> "/" <unáris_kif>
                         | <multiplikatív_kifejezés> "%" <unáris_kif>
<unáris_kif>           ::= <unáris_művelet> <kifejezés> | <kifejezés>
<unáris_művelet>       ::= "-" | "+" | "~"
<kifejezés>            ::= <sznév> | <literál> | "(" <konstans_kifejezés> ")"
<literál>              ::= <integer_literál>
                         | <karakterlánc_literál>
                         | <karakter_literál>
                         | <lebegőpontos_literál>
                         | <boolean_literál>
<boolean_literál>      ::= "TRUE" | "FALSE"
<pozitív_egész_konstans> ::= <konstans_kifejezés>
<típusdeklaráció>      ::= "typedef" <típusdeklarátor> | <rekord_típus>
                         | <unió_típus> | <felsorolási_típus>
<típusdeklarátor>      ::= <típusspecifikáció> <deklarátorok>
<típusspecifikáció>    ::= <egyszerű_típus_spec.> | <összetett_típus_spec.>
<egyszerű_típus_spec.> ::= <alap_típus_spec> | <templét_típus_spec> | <sznév>
<alap_típus_spec>      ::= <lebegőpontos_típus> | <integer_típus>
                         |  <karakteres_típus> | <boolean_típus>
                         |  <oktett_típus> | <any_típus>
<templét_típus_spec>   ::= <szekvencia_típus> | <karakterlánc_típus>
<összetett_típus_spec.>::= <rekord_típus> | <unió_típus>
                         | <felsorolási_típus>
<deklarátorok>         ::= <deklarátor> { "," <deklarátor> }
<deklarátor>           ::= <egyszerű_deklarátor> | <összetett_deklarátor>
<egyszerű_deklarátor>  ::= <azonosító>
<összetett_deklarátor> ::= <tömb_deklarátor>
<lebegőpontos_típus>   ::= "float" | "double"
<integer_típus>        ::= <előjeles_egész> | <előjelnélküli_egész>
<előjeles_egész>       ::= <előjeles_long> | <előjeles_short>
<előjeles_long>        ::= "long"
<előjeles_short>       ::= "short"
<előjelnélküli_egész>  ::= <előjelnélküli_long> | <előjelnélküli_short>
<előjelnélküli_long>   ::= "unsigned" "long"
<előjelnélküli_short>  ::= "unsigned" "short"
<karakteres_típus>     ::= "char" | "wchar"
<boolean_típus>        ::= "boolean"
<oktett_típus>         ::= "octet"
<any_típus>            ::= "any"
<rekord_típus>         ::= "struct" <azonosító> "{" <mezőlista> "}"
<mezőlista>            ::= <mező> { <mező> }
<mező>                 ::= <típusspecifikáció> <deklarátorok> ";"
<unió_típus>           ::= "union" <azonosító> "switch" "(" <szelektor_típusa>
```

```
                          ")" "{" <unió_törzse> "}"
<szelektor_típusa>     ::= <integer_típus> | <karakteres_típus>
                         | <boolean_típus> | <felsorolási_típus> | <sznév>
<unió_törzse>          ::= <ág> { <ág> }
<ág>                   ::= <ág_címke> { <ág_címke> } <elem_spec> ";"
<ág_címke>             ::= "case" <konstans_kifejezés> ":" | "default" ":"
<elem_spec>            ::= <típusspecifikáció> <deklarátor>
<felsorolási_típus>    ::= "enum" <azonosító> "{" <felsorolási_konstans>
                           { "," <felsorolási_konstans> } "}"
<felsorolási_konstans> ::= <azonosító>
<szekvencia_típus>     ::= "sequence" "<" <egyszerű_típus_spec.>
                           "," <pozitív_egész_konstans> ">"
                           | "sequence" "<" <egyszerű_típus_spec.> ">"
<karakterlánc_típus>   ::= "string" "<" <pozitív_egész_konstans> ">"
                           | "string" | "wstring"
                           | "wstring" "<" <pozitív_egész_konstans> ">"
<tömb_deklarátor>      ::= <azonosító> <rögzített_méretű_tömb>
                           { <rögzített_méretű_tömb> }
<rögzített_méretű_tömb>::= "[" <pozitív_egész_konstans> "]"
<attribútumdeklaráció> ::= [ "readonly" ] "attribute" <paraméter_típus_spec.>
                           <egyszerű_deklarátor> { "," <egyszerű_deklarátor> }
<kivételdeklaráció>    ::= "exception" <azonosító> "{" { <mező> } "}"
<műveletdeklaráció>    ::= [ <módosító> ] <visszatérési_érték> <azonosító>
                           <param_dekl.> [ <kiváltható_kivételek> ]
                           [ <kontextus_záradék> ]
<módosító>             ::= "oneway"
<visszatérési_érték>   ::= <paraméter_típus_spec.> | "void"
<param_dekl.>          ::= "(" <egyparaméter> { "," <egyparaméter> } ")"
                           | "(" ")"
<egyparaméter>         ::= <paraméter_attribútum> <paraméter_típus_spec.>
                           <egyszerű_deklarátor>
<paraméter_attribútum> ::= "in" | "out" | "inout"
<kiváltható_kivételek> ::= "raises" "(" <sznév> { "," <sznév> } ")"
<kontextus_záradék>    ::= "context" "(" <karakterlánc_literál> { ","
                           <karakterlánc_literál> } ")"
<paraméter_típus_spec.>::= <alap_típus_spec> | <karakterlánc_típus> | <sznév>
```

Megjegyzések

1. A nyelvtan megadására a hagyományos BNF eszköz egy módosításának tekinthető nyelvtandefiníciós eszközzel felépített szabályrendszert használunk.
   A `::=` a nyelvtani szabályokat definiáló egyenlőség jele.
   Az egyes nyelvtani jelek alternatíváit függőleges vonal (|) karakterrel választjuk el egymástól.
   A terminális - azaz nem nyelvtani - karaktereket idézőjelek közé írtuk.
   A nyelvtani karaktereket "csibeszájak" közé írtuk.
   A szögletes zárójelek közé írt szintaktikai elemek elhagyhatóak, de legfeljebb egyszer fordulhatnak elő.
   A kapcsos zárójelek közé írt szintaktikai egységek nullaszor, vagy egyszer, vagy

tetszőlegesen sokszor ismétlődhetnek egymás után.

2. Az IDL fájl feldolgozása során keresztülmegy egy makro preprocesszoron; a preprocesszornak szóló direktívák felépítése hasonlít a C++ nyelvnél megismertekhez (egy # jellel kezdődnek az őket tartalmazó sorok).

3. Megjegyzéseket írhatunk egyrészt /* és */ szimbólumok közé (az említett szimbólumok közti megjegyzés tetszőleges hosszúságú lehet, és az ilyen megjegyzések nem ágyazhatók egymásba), illetve egysoros - a sor végéig tartó - megjegyzéseket a // jelekkel kezdhetjük.

4. Az azonosítókkal kapcsolatban annyit érdemes megemlíteni, hogy azok alfanumerikus, valamint aláhúzás (azaz _) karakterek tetszőleges hosszúságú sorozata lehetnek. Az első karakter betű kell legyen; az azonosítók összes karaktere szignifikáns (azaz nem az első hat vagy nyolc karaktere határozza meg egyértelműen az azonosítót, mint azt néhány programnyelvben esetleg láthattuk); a kis és a nagybetűk összehasonlításkor azonosnak tekintendők.

5. Az OMG IDL nyelv kulcsszavait a fenti grammatikában idézőjelek közt írtuk.

6. A fenti BNF szabályok között feltüntettük az UNICODE karakter alapú karakterláncok adattípusát leíró `wchar`, valamint `wstring` típusneveket, bár ezek a CORBA specifikáció 2.0 változatában még nem találhatóak meg.

7. A fenti IDL specifikáció nem tartalmazza néhány literál (konstans) BNF szabályát. Ezek a következők:

   - `<integer_literál>` : nem nullával kezdődő számjegyek sorozata tízes számrendszerben értendő; nullával kezdődő számjegyek sorozata oktális (nyolcas) számrendszerben, míg a `0x` vagy `0X` karakterekkel kezdődő számjegysorozat hexadecimális (tizenhatos) számrendszerben értendő.
   - `<karakter_literál>` : aposztrófok közti egy karakter hosszú konstans; a `\ooo` az `ooo` oktális, míg `\xhh` a `hh` hexadecimális konstanssal megadott kódú karaktert azonosítja, és a következő speciális karakterek vannak még - a "szokásos" jelentéssel - definiálva: `\n, \t, \v, \b, \r, \f, \a, \\, \?, \', \"`.
   - `<karakterlánc_literál>` : karakterek sorozata (ld. `<karakter_literál>` nyelvtani elemnél) idézőjelek között. Egymást követő karakterlánc literálok egyetlen karakterlánc literállá lesznek konkatenálva (azaz az "AB" "C" egymás után írva és "ABC" ugyanazt reprezentálják). Nem tartalmazhatja a `\0` karaktert.
   - `<lebegõpontos_literál>` : egy egészrészt követ egy tizedespont, azt egy tizedes rész, majd egy "e" vagy "E" karakter, és ezt követi egy előjeles egész

kitevő kifejezés. Vagy az egészrész, vagy a tizedes rész hiányozhat (mindkettő nem), és vagy a tizedespont, vagy az "e" (ill. "E") és az azt követő kitevő hiányozhat (de mindkettő nem).

B. Függelék

Irodalom

Ebben a könyvben az elosztott alkalmazások készítésének CORBA és Java eszközét ismerhettük meg. Elsősorban a gyakorlati alkalmazások szükségleteinek szemszögéből. Tárgyalásunk az Internet hálózati modelljére épült, mivel jelenleg a TCP/IP-alapú protokollok sokkal elterjedtebbek, mint az OSI hálózati protokolljai, és nem valószínű, hogy a közeljövőben komolyabb változásra számíthatnánk ezen a téren.

Az irodalomjegyzékben ismertetni fogom az - általam - legfontosabbnak ítélt műveket, amelyekben az Olvasó érdeklődési körének megfelelően tovább mélyítheti ismereteit.

Az itt bemutatott könyvek három fő csoportba sorolhatók, eszerint fogom csoportokba rendezni az ismertetésüket.

1. A Java programozási nyelvvel kapcsolatos olvasnivalók.
2. Az alapszoftverrel kapcsolatos olvasnivalók.
3. Hálózatokkal kapcsolatos érdekesebb olvasnivalók.

B.1. A Java programozási nyelvvel kapcsolatos irodalmak

1. James Gosling et al.: Java Programming Language
   SunSoft Press, 1996

   A Java programozási nyelv tervezői írták a Java nyelvről. A könyv inkább haladók számára készült, és a Java nyelv részletes leírását tartalmazza.

2. David Flanagan: Java in a Nutshell, 2nd Edition
   O'Reilly & Associates, 1997

Ezt a könyvet elsősorban azoknak a C, illetve C++ programozási nyelvet ismerő programozóknak ajánljuk, akik gyorsan - korábbi ismereteikre alapozva - szeretnék megismerni a Java nyelv rejtelmeit. Bemutatja a Java 1.1 nyelv és környezet lényegesebb elemeit, és kényelmesen használható referenciaként a Java 1.1 szabványban definiált osztályokhoz is.

3. Nyékyné et al.: JAVA útikalauz programozóknak
   Kalibán BT., 1996

   Ez a könyv jelenleg a Java nyelvet legrészletesebben ismertető magyar nyelvű könyv. Tartalmazza mind a nyelvnek, mind pedig a Java fejlesztői környezetekkel együtt szállított "szabványos" osztályok leírását. Használható akár tankönyvként, akár referencia kézikönyvként.

4. Robert Orfali, Dan Harkey: Client/Server Programming with JAVA and CORBA
   John Wiley & Sons, Inc., 1997

   Ez a könyv a Java és a CORBA, mint két meghatározó ipari technológia helyét keresi a hálózati technológiák között. Számos alternatív technológiát is bemutat - egy-egy egyszerű példaprogramon keresztül, és az egyes technológiák hatékonyságát is megvizsgálja az elkészített példaprogramok futásidejének összehasonlítása alapján. A könyv jó stílusban, és olvasmányosan van megírva, így elolvasását mindenkinek ajánlom.

5. Andreas Vogel, Keith Duddy: Java Programming with CORBA
   John Wiley & Sons, Inc., 1997

   Ez a könyv elsősorban a Java nyelv CORBA kapcsolatát vizsgálja, bemutatva a ma elérhető három Java alapú CORBA ORB szoftver implementáció, a VisiBroker, az OrbixWEB és a Joe ORB szoftver használatát.

B.2. Az alapszoftverrel kapcsolatos irodalmak

1. Andrew S. Tanenbaum: Modern Operating Systems
   Prentice-Hall International, Inc., 1992

   Ez a könyv - a szerzőjétől megszokott módon - lényegre törő és olvasmányos stílusban ismerteti a hagyományos és a modern operációs rendszerekkel kapcsolatos fontosabb tudnivalókat. Komoly értéke a könyvnek a gyakorlati szemléletmódja: a tárgyalt témakörök között elsősorban azok gyakorlati jelentőségük alapján súlyoz,

és több operációs rendszer belső szerkezetét is bemutatja az esettanulmányaiban (UNIX, MS-DOS, Amoeba, Mach operációs rendszereket).

2. Marshall Kirk McKusick et al.: The Design and Implementation of the 4.4BSD Operating System
Addison-Wesley, 1996

   A könyv bemutatja a 4.4BSD operációs rendszer belső szerkezetét, amelyet ma is sokan használnak akár PC- ken, akár nagyobb munkaállomásokon is (ezen alapulnak a 386BSD, FreeBSD, NetBSD és OpenBSD operációs rendszerek is, amelyek akár egy PC-n is képesek működni). A történeti áttekintés mellett számunkra különösen a hálózati protokollok - illetve azok tervezési és implementációs szempontjainak - bemutatását tartalmazó részek az érdekesek; választ ad a Java környezet specifikációjának olvasása közben felmerülő számos kérdésre.

3. M. Ben-Ari: Principles of Concurrent Programming
Prentice-Hall, 1982

   Ez a könyv olvasmányos stílusban egy alapos bevezetést nyújt a párhuzamos programozási technikák alkalmazásába. Ismerteti az alapvető szinkronizációs mechanizmusokat (szemaforok, monitorok, randevúk és üzenetküldés), és bemutat néhány alapvető fontosságú párhuzamos programozási problémát, és megoldásukat.

4. Grady Booch: Object-Oriented Analysis and Design /with Applications 2nd Edition/
Addison-Wesley, 1994

   Ez a könyv egy ma igen elterjedt objektum-orientált tervezési módot ismertet, mind fogalmi, mind pedig gyakorlati szempontból fontos szempontok alapján. Külön értéke a sok esettanulmánya, amelyen keresztül valóséletbeli, nagyméretű objektum-orientált rendszerek tervezésének problémáiba is betekinthet az Olvasó.

B.3. Hálózatokkal kapcsolatos irodalmak

1. Andrew S. Tanenbaum: Számítógépes hálózatok
Panem - Prentice-Hall, 2004

   A könyv számítógépes hálózatokkal kapcsolatos elméleti és gyakorlati kérdéseket tárgyal. Központjában az OSI referenciamodellje áll, amit szintről szintre a szolgáltatásokkal együtt részletesen ismertet (a legújabb angol nyelvű kiadásban a szerző már részletesebben foglalkozik az igen elterjedt Internet, és a ma még kevésbé

elterjedt, de rohamosan terjedő ISDN és ATM hálózati technológiákkal, és kevesebb részletességgel tárgyalja az OSI szabvány elemeit).

2. Richard W. Stevens: TCP/IP Illustrated, Volume I. /The Protocols/
Addison-Wesley, 1994

Bemutatja a TCP/IP protokollcsalád elemeit. Komoly értéke a könyvnek, hogy a protokolloknak nem csak a specifikációit ismerteti, hanem az életből vett példákon keresztül szemlélteti azok működését is (vagyis azt, hogy mi is megy át a hálózaton a csomagokban, amikor egyes résztvevők az illető protokollal kommunikálnak egymással).

3. Object Management Group: The Common Object Request Broker: Architecture and Specification
Object Management Group, 1995

Ez az OMG CORBA 2.0 specifikációja, az OMG objektum-orientált osztott infrastruktúrájának az elemeit ismerteti. Részletesen elemzi az OMG IDL szerkezetét, elemeit, a statikus és a dinamikus metódushívási interfészt, a szerveroldal implementációjának - BOA és DSI - eszközeit, és az interfészgyűjteményt. Olvashatunk benne az ORB szoftverek együttműködéséről (GIOP, IIOP és DCE-ESIOP protokollokról), valamint az OMG IDL nyelvnek a C, C++, valamint a Smalltalk programozási nyelvekre történő leképezésére. Az OMG legtöbb specifikációja szabadon elérhető a WWW szerverükről, a `http://www.omg.org` URL-címen.

4. Object Management Group: The Common Object Request Broker: Architecture and Specification
Object Management Group, 1998

Ez az OMG CORBA 2.2 specifikációja. Ez tartalmazza a szabványosított IDL/Java leképezési szabályokat.

5. Object Management Group: CORBAservices: Common Object Services Specification
Object Management Group, 1997 november

Ez az általános CORBA objektumszolgáltatások specifikációja. Több, mint ezer oldalon keresztül ismerteti a szabványosított objektumszolgáltatásokat. Néhol kicsit nehezen érthető, nem is tankönyvnek szánták, hanem az egyes szabványok pontois leírására.

6. Remote Method Invocation Specification
   Sun Microsystems, 1996, 1997

   Ez a Java távoli metódushívási rendszerének a specifikációja. Bemutatja a távoli metódushívásnak mind a programozói modelljét, mind pedig az architektúráját. Részletesen ismerteti a távoli metódushívás megvalósítását támogató interfészeket és osztályokat, valamint a kliens- és szerver csonkokkal kapcsolatos fontosabb ismereteket.

7. RFC dokumentumok: az IETF által kidolgozott, legtöbbször az Internet hálózattal kapcsolatos szabványok

   Ma már több, mint 2100 ilyen RFC dokumentumot elkészítettek, a legkülönbözőbb szabványok, ajánlások és ötletek dokumentálására. Az RFC dokumentumok a legtöbb anonim FTP szerverről szabadon elérhetők. A könyvünkben hivatkozott RFC dokumentumok a következők:

```
RFC 767: Az UDP protokoll specifikációja.
RFC 791: Az Internet Protokoll (IP) specifikációja.
RFC 793: A TCP protokoll specifikációja.
RFC 821: Az Internet SMTP levelezési protokolljának specifikációja.
RFC 822: Az Internet szöveges üzeneteinek szabványos formátuma.
RFC 854: A TELNET távoli terminál szolgáltatás protokollja.
RFC 867: Aktuális dátumot és időt visszaadó alkalmazói szolgáltatás.
RFC 959: Az FTP fájlátviteli protokoll specifikációja.
RFC 992: Hibatűrő működésű folyamat-csoportok kommunikációja.
RFC1006: OSI transzport protokoll működtetése TCP protokoll felett.
RFC1013: Az X Window Rendszer protokollja - 11-es változat.
RFC1034: A DNS Internet névszolgáltató fogalmi rendszerét és lehetőségeit
         ismerteti.
RFC1035: A DNS Internet névszolgáltató specifikációját és implementációját
         ismerteti.
RFC1055: A soros vonal feletti Internet protokoll (SLIP).
RFC1075: A DVMRP multicast útvonalkijelölési protokoll.
RFC1301: Egy multicast transzport protokollt ismertet.
RFC1320: Az MD4 algoritmus.
RFC1341: MIME specifikáció - az Internet üzenetek formátumának specifikációs
         eszköze.
RFC1350: A TFTP egyszerű fájlátviteli protokoll 2. változata.
RFC1425: Az SMTP levélküldési szolgáltatás kibővítése.
RFC1426: Az SMTP kibővítése 8 bites MIME alapú átvitelre.
RFC1436: Az INTERNET Gopher protokoll
RFC1630: URI-azonosítók a WWW-n.
RFC1737: URN nevekkel szemben támasztott követelmények.
RFC1738: URL-azonosítók formája.
RFC1808: Relatív URL-azonosítók formája.
RFC1866: HTML 2.0 specifikációja.
RFC1896: A text/enriched MIME típus specifikációja.
```

RFC1945: HTTP 1.0 protokoll specifikáció
RFC2044: UTF-8 kódolási formátum specifikációja.
RFC2045: MIME specifikáció 2.0 I. - A MIME üzenetek törzsének formátuma.
RFC2046: MIME specifikáció 2.0 II. - A MIME dokumentumtípusok.
RFC2047: MIME specifikáció 2.0 III. - Fejlécmezők kiterjesztése nem-ASCII karakterekkel.
RFC2068: HTTP 1.1 protokoll specifikáció

8. Jigsaw - a WWW Consortium HTTP szerverének forráskódja
WWW Consortium, 1997

A Jigsaw a WWW Consortium HTTP szervere.
Az egész szerver Java nyelven készült, a forráskódja szabadon elérhető a `http://www.w3.org/pub/WWW/Jigsaw/` címen. Ezt a szervert kutatási projektek segítésére készítették el, és tették bárki számára elérhetővé. Az Olvasónak azt ajánljuk, hogy e könyv elolvasása után tanulmányozza át a Jigsaw programot, annak forráskódját, mivel tartalmilag jól illeszkedik e könyv témájához, és több valós életbeli példaprogramon keresztül vizsgálhatóak a könyvünkben bemutatott eszközök (akár a servletek is).

B.4. Elosztottság és irodalma: informatika és matematika?

Informatikai témájú irodalmat olvasván felmerülhet az Olvasóban az a nagy kérdés, hogy mennyire kapcsolódik az informatika a matematikához.

Hogyan könnyebb elosztott alkalmazást készíteni? Java nyelven CORBA eszközökkel VAGY PEDIG papíron, ceruzával néhány formulát és mellé hieroglifákat felírva? Előbbihez sokat kell tudni (ennek a könyvnek az anyagát például kimondottan hasznos ismerni), utóbbihoz pár tollvonás is elég!

Milyen egyszerű az informatika! – mondhatnánk egy matematikai megközelítésből. De ez valóban egy szigorú matematikai megközelítés lenne. És a matematikusok ehhez hasonló érveket hoznak fel arra, hogy miért is ilyen egyszerű az informatika, miért is tekinthető az a matematika egy tudományágának.

A számítástudományban használt matematikai programok modelljét nem azért tettük be a könyv ezen részébe, hogy bebizonyítsuk, azt a sokak szerint[1] igaznak vélt vagy áhitott állítást, hogy az informatika csak egy ága a matematikának. Természetesen az informatikának és a matematikának számos közös vonásai (sőt, közös gyökerei) vannak, de a különbségek sem elhanyagolhatóak.

Az informatika és a matematika közös vonásai közül néhány:

- Az absztraktság, az absztrakciók készítése a problémamegoldás során.

[1] Ez alatt leginkább sok matematikust értünk.

- Precíz, logikus gondolatmenetek kialakítása és követése.
- Mennyiségi elemzések a megoldással kapcsolatban.
- Széles alkalmazási környezet.

Akadnak viszont nagy számban különbségek is az informatika és a matematika mint tudományágak között, amelyek miatt nem mondhatjuk azt, hogy az informatika csupán a matematika tudományok egy ága[2].

A különbségek közül kiemeljük, hogy a matematikában az időnek, a folyamatok lefutásának nincs központi szerepe. A változók ebből következően csupán értékeket reprezentáló szimbólumok. Ezzel szemben az informatikát a dinamizmus jellemzi: a vizsgálatok középpontjában a program állapotának időbeli változása áll: a bemeneti adatok kimeneti adatokká lesznek transzformálva. A változók itt értékeket reprezentálnak, melyek az idő változásával maguk is változhatnak. A metematikában a végtelen fogalma központi szerepet játszik mind a folytonos számításoknál (például integrálszámítás), mind pedig a diszkrét számításoknál (végtelen sorok, sorozatok, halmazelmélet). Az informatikában a végesség dominál (kivéve a programhibák eredményeként felmerülő végtelen ciklusokat ☺). Az informatika véges szerkezetekkel foglalkozik. Míg mindkét tudományág általános szabályok felállításán dolgzik, addig az informatikának mindemellett különösen sok kivételes esettel kell foglalkoznia az algoritmusok megadása során (a kivételek forrása a valós élet – az bizony tele van kivételekkel). Az informatika jellemzői között soroljuk fel a hatékonysággal kapcsolatos problémákat is. Egy matematikus örül, ha egy problémára talál egy megoldást – függetlenül a megoldás bonyolultságától, míg az informatikus feladata ezen felül a lehető leghatékonyabb megoldás megkeresése, szem előtt tartva a megoldás egyszerűségét is. A matematika nem foglalkozik az informatikában oly fontos szereppel bíró párhuzamos végrehajtás és a megoldás komponensei közötti kommunikáció problémájának a kérdésével. Ezek a különbségek figyelemreméltóak, nem is beszélve arról, hogy az informatikának technikai oldala is van. Ennek tudatában megdőlnek azok a próbálkozások, amelyek az informatika dinamizmusából eredő különbségeket visszavezetik a matematika statikus világának állapotváltozásaira egy állapotváltozás-leíró függvény segítségével, illetve a párhuzamosságot és a kommunikációt azzal vezetik vissza a matematikára, hogy megkísérlik visszavezetni a szekvenciális végrejahtásra (ahogyan az egy időosztásos operációs rendszerrel ellátott egyprocesszoros gépen történhet, hiszen a párhuzamosságot ott csak szimuláljuk). Ez a módszer azonban nem alkalmazható valódi párhuzamos végrehajtás esetén.

Ezeket ne felejtsük el, mikor informatikai témájú könyveket írunk vagy olvasunk...

[2]Niklaus Wirthtől, a Pascal és számos más nyelv és programozási környezet atyjától, származik az a mondás, hogy egy – informatikus – mérnök az az ember, aki egy centből meg tudja valósítani azt, amit más (informatikával foglalkozó ember) csak egy dollárból. (Forrás: Böszörményi László – magánbeszélgetés.)

Tárgymutató