Objektum Orientált Szoftverfejlesztés

(jegyzet)

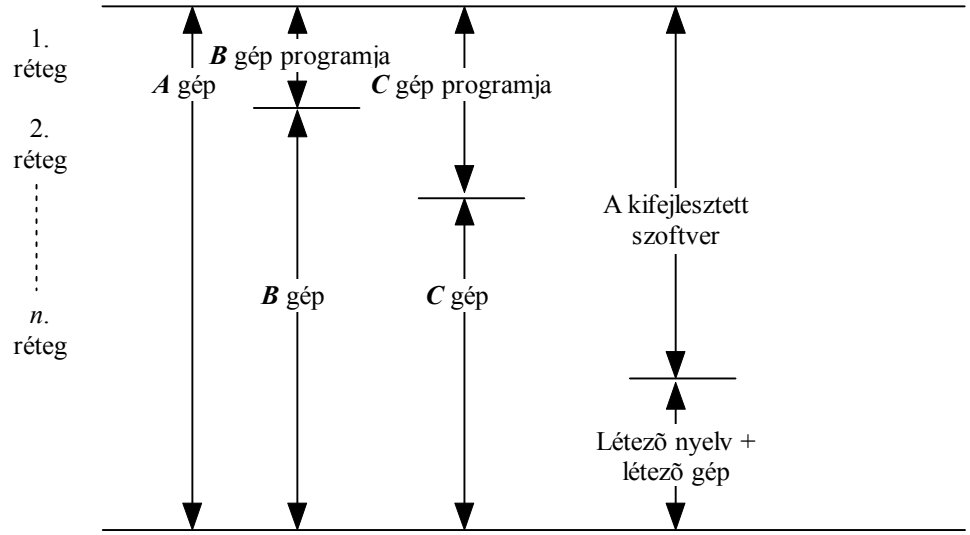
1. Kialakulás

Kísérletek a szoftverkrízisből való kilábalásra:

1.1 Strukturált programozás

Ötlet **(E. W. Dijkstra)**:

1. Elkészítendő programot elgondolhatjuk egy absztrakt gépnek (***A***), melynek egyetlen utasítása van: „oldd meg a feladatot”. Ilyenünk nincs, ezért
2. definiálunk egy egyszerűbb gépet, és ennek utasításkészletével elkészítjük az előző gépet szimuláló programot. Ha ilyenünk nincs → 2. Ha van, akkor készen vagyunk.



Előnyök:

- áttekinthetőség (1-1 réteg önmagában vizsgálható, nem kell az egészet szem előtt tartani)
- könnyű módosíthatóság (egy változtatás hatása csak a következő rétegig fejti ki hatását)
- hordozhatóság (rétegeknél könnyen szétválasztható)

Hátrányok:

- mégsem lett áttekinthető (néhány lépés után a kialakuló utasításkészlet igen bonyolulttá válhat)
- szigorú rétegszerkezet miatt romlik a hatékonyság (ha egy magasabb rétegben olyan műveletre is szükség van, amely csak több réteggel lejjebb válik utasítássá, azt több rétegen keresztül „cipelni” kell)

Mai szemléletünk sok eleme innen származik, legfontosabbak:

- **absztrakció:** részletek eltakarása
- **dekompozíció:** részekre bontás

Ugyancsak itt fogalmazódott meg először a

- felülről lefelé történő tervezés,
- lépésenkénti finomítás,
- információelrejtés.

A strukturált módszertanokban a tervezői döntések domináns fogalma a ***funkcionalitás***, illetve az ezt reprezentáló ***eljárás.***

1.2 Moduláris programozás

Központi fogalma a **modul,** ez a rendszer valamilyen, a fejlesztés során önálló egységként kezelt, cserélhető részét jelöli. Kezdetben a modulokra bontás a méret alapján történt, később megvizsgálták, hogy mit célszerű egy modulba tenni, ennek eredményei:

- A modul belső kötése legyen minél erősebb.
- A modulok közti csatolás legyen minél gyengébb.

2. Funkcionális vs. adatorientált tervezés

Készítsük el a következő feladatot megoldó program vázlatát!
Egy bemeneti szövegfile szavainal abc-sorrendjében nyomtassuk ki, hogy a szavak mely sorokban fordulnak elő, pl.:

Ez a sor az első sor, Ez a sor pedig a második sor.

Ekkor a kimeneten a következő jelenjen meg:

a	1 2 2
az	1
első	1
Ez	1 2
második	2
pedig	2
sor	1 1 2 2

2.1 A funkcionális megoldás

Megállapíthatjuk, hogy először a teljes bemenő file-t be kell olvasnunk, hiszen az utolsó sorában is állhat az a szó, mely a kimenet első sorába kerül.
Így a **funkcióra** koncentrálva a feladat 3 lépésben oldható meg:

1. Beolvassuk a file-t egy táblázatba
2. Rendezzük a táblázatot.
3. Kinyomtatjuk a rendezett táblázatot.

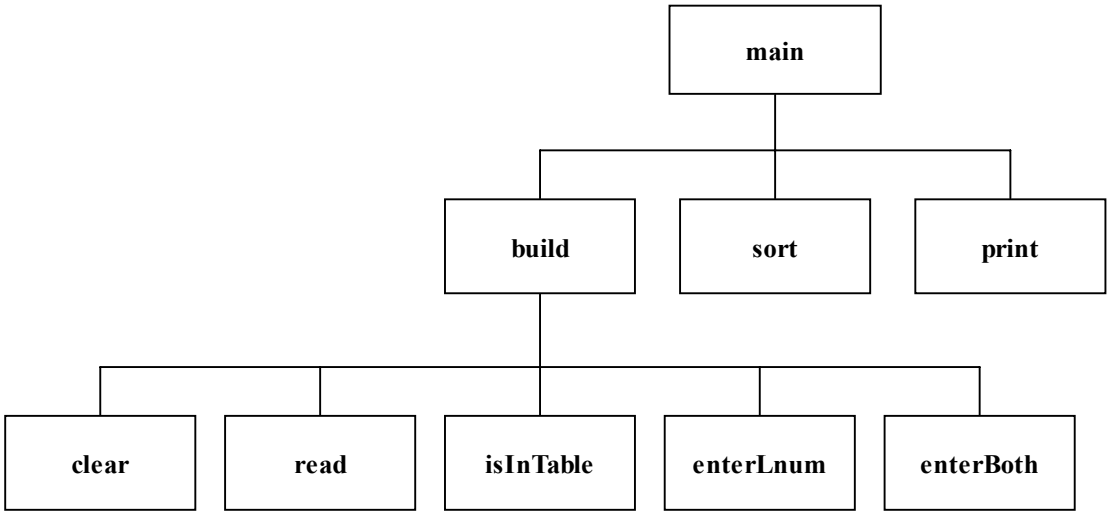
A program pszeudonyelven a következőképpen nézne ki:

```
main() {
build (infile, table);
sort (table);
print (outfile, table);
}
```

A `build` függvényt kifejtve:

```
void build (infile_type& infile, table_type& table) {
    int linenumber = 0;
    clear (table);
    while (!eof(infile)) {
        linenumber++;
        while (true) {
            if (read(infile, word) == EOL) break;
            else
            if (isInTable(word,table))
                enterLnum(word, linenumber, table);
            else enterBoth(word, linenumber, table);
        }
    }
}
```

A következő ábra azt mutatja, hogy egy függvény milyen más függvényeket hív meg.



A funkcionalitásra koncentrálva az algoritmusokat elrejtettük, viszont döntéseket hoztunk az adatstruktúrát illetően. (Pl.: a szavakat csak egyszer tároljuk a táblázatban.). Ez a döntés látható lesz a `build` függvény hatáskörén kívül is, meghatározza a `sort` és a `print` függvényeket is:

- Amennyiben a 3 részt (`build, sort, print`) egymástól függetlenül, párhuzamosan akarjuk fejleszteni, szükséges a (bonyolult) table adaszerkezet pontos definiálása.
- Az adatszerkezeten történ legkisebb változás is valamennyi programrészre hatást gyakorol.

2.2 Adatszerkezet orientált megoldás

Feladatunk:

- **adatstruktúrák azonosítása,**
- azon **absztrakt műveletek meghatározása**, melyek ezen struktúrák használatához szükségesek.

Nyilvánvalóan a table egy adatszerkezet lesz. A rajta végrehajtandó műveleteket *a feladat megoldása szempontjából* határozzuk meg, nem pedig azzal, hogy hogyan valósítjuk meg.
3 adatszerkezetünk lesz:

1. InFile bemeneti file
2. Table táblázat
3. OutFile kimeneti file

InFile adatszerkezetet definiáljuk a következő műveletekkel:

```
void get (word, linenumber);
BOOL morePairs();
```

Table

```
void store (word, linenumber);
void retrieve (word, linenumber);
BOOL moreData();
```

OutFile

```
void print (word, linenumber);
```

Ezzel sikerült az adatszerkezetek fizikai és logikai adatszerkezetét eltakarni (információrejtés).
Ezeket felhasználva a program a következőképpen alakul:

```
while (InFile.morePairs()) {
    InFile.get(word, linenumber);
    Table.store(word, linenumber);
}
while (Table.moreData()) {
    Table.retrieve(word, linenumber);
    OutFile.print(word, linenumber);
}
```

Ez igen egyszerű. 3 részre bontva párhuzamosan fejleszthető, mindössze a szót és a sorszámot kell egyeztetni, nem egy bonyolult táblázatot. (*részek közti kapcsolat gyenge*)

3. Objektum orientáltság

A szisztematikus programtervezés alkalmazása során tudatosult, hogy egy információ-feldolgozási probléma két oldalról közelíthető meg, az algoritmusok, illetve az adatok oldaláról. Ez a két oldal szorosan összefügg, egyik a másik nélkül nem áll meg. A feladatok jellegétől függően egyik vagy másik oldal nagyobb hangsúlyt kaphat (pl: adatbázis-kezelés vs. gyors Fourier transzformáció). Ezeket a tapasztalatokat összefoglaló módszertanoknak nagy sikere volt/van.

Az objektum orientáltság a legelfogadottabb paradigma a számítástechnikában.

Paradigma: egy világszemlélet, látás- és gondolkodásmód, amelyet az adott fogalomkörben használt elméletek, modellek, és módszerek összessége jellemez.

Az objektum orientált módszertan alkalmazásával a kifejlesztendő rendszert együttműködő objektumokkal modellezzük, a tervezés és az implementáció során pedig ezen objektumokat „szimuláló" programegységeket alakítunk ki. Pl.: objektumok a körülöttünk lévő világban: egyetem, tanár, diák

Objektumok absztrakciót tükröznek, azaz a valóság egy szeletét reprezentálják. Az emberek a világ dolgait objektumként kezelik. Pl: autó (sokan nem tudják, hogy hogyan működik az autó, mégis használják)

Objektum: egy rendszer egyedileg azonosítható szereplője, amelyet a külvilág felé mutatott viselkedésével, belső struktúrájával és állapotával jellemezhetünk.

Külső szemlélő nem lát bele az objektumba, nem látja a belső működését. Ez az információrejtés elve, vagyis az egységbe zárás (encapsulation).

Objektum egységbe zárja:

- állapotát tároló adatokat és azok szerkezetét,
- az előzőeken végrehajtott, az objektum viselkedését meghatározó műveleteket.

3.1 Az objektum felelőssége

Elvárás: objektum képes legyen elvégezni a rá szabott feladatot és csak annyi ismerettel rendelkezzen, amivel azt a feladatot el tudja végezni.

Objektumhoz felelősség rendelhető. A felelősség vállalása és teljesítése az objektum feladata.

Lokális felelősség elve: minden objektum felelős önmagáért (azaz a feladatát maradéktalanul hajtsa végre).

3.2 Az objektum viselkedése

Az objektum viselkedése az általa végrehajtott tevékenységsorozatban nyilvánul meg.

Viselkedésének jellege alapján az objektum lehet:

- passzív (nem tesz semmit, amíg valamilyen környezeti hatás nem éri)
- aktív (folyamatosan működik)

3.3 Üzenetek

Objektumok egymásra hatása üzeneteken keresztül valósul meg.

Üzenetek szerepe:

- adatcsere
- objektumok vezérlése

Üzenetek komponensei:

- név (meghatározza az üzenet fajtáját)
- paraméter

Különböző üzenetek különböző reakciót váltanak ki az objektum részéről.

A paraméterek alkalmasak a működés közbeni dinamikus információk átadására.

3.4 Események

Az esemény azonosítható pillanatszerű történés. Az esemény bevezethető az üzenet helyett, az objektumok kölcsönhatásainak, együttműködésének modellezésére.

Az eseményhez rendelhető: *név* (statikus info) és *paraméter* (dinamikus info)

Az eseményeket az objektumok hozzák létre.

Elképzelhető az üzenet és az esemény együttes használata, amivel finom különbségeket lehet tenni.
pl: posta

Küldemény típusa: levél, csomag… (üzenet neve)
Ennek megérkezése esemény, ami független a levél tartamától.
A levél tartalma lehet az üzenet paraméterei.

3.5 Metódusok

Az objektum a beérkező üzenetre valamilyen cselekvéssel (metódus végrehajtásával) reagál.
Üzenet megmondja hogy MIT, a metódus pedig hogy HOGYAN.
Azonos tartalmú üzenetre az objektum különféleképpen reagálhat.

3.6 Állapot

Az objektumban nyoma marad a korábbi üzeneteknek. Ezt a nyomot az objektum *állapotának* nevezzük.
Az állapotváltozókat kívülről közvetlenül nem látjuk és nem is érhetjük el (információelrejtés). Az állapotot az objektum attribútumai tárolják (szoftverben ezek változók).

3.7 Polimorfizmus

Különböző objektumok is kaphatnak azonos üzeneteket.
Polimorfizmus: ha egy objektum úgy küldhet üzenetet egy másiknak, hogy nem kell figyelemmel lennie arra, hogy ki az üzenet vevője.

3.8 Osztályok és példányok

Osztály: megegyező viselkedésű és struktúrájú objektumok.

Pl.: ember, autó, ház

Az objektum a viselkedését és struktúráját definiáló osztály egy példánya.
Minden objektum ismeri a saját osztályát. Objektum és példány szinonim szavak.
Ábrázolás: UML
Osztály megadása:

Osztálynév
attributumnév_1:adattípus_1=alapérték_1 ... attributumnév_n:adattípus_n=alapérték_n
metódusnév(paraméterlista):eredménytípus_1 ... metódusnév(paraméterlista):eredménytípus_n

Az objektumok egymás felé csak a metódusaikat mutatják, biztosítva az információelrejtést.

Az objektum és környezete közti csatolás gyengítésére kell törekedni.
Ez leggyengébb akkor, ha teljesül a *Demeter törvény*:

Egy objektum a metódusain belül csak az alábbi elemekre hivatkozhat:

- metódus paramétereire, eredményeire
- metódust tartalmazó osztály attribútumaira
- program globális változóira
- metódusban definiált lokális változókra

3.9 Objektum típusa

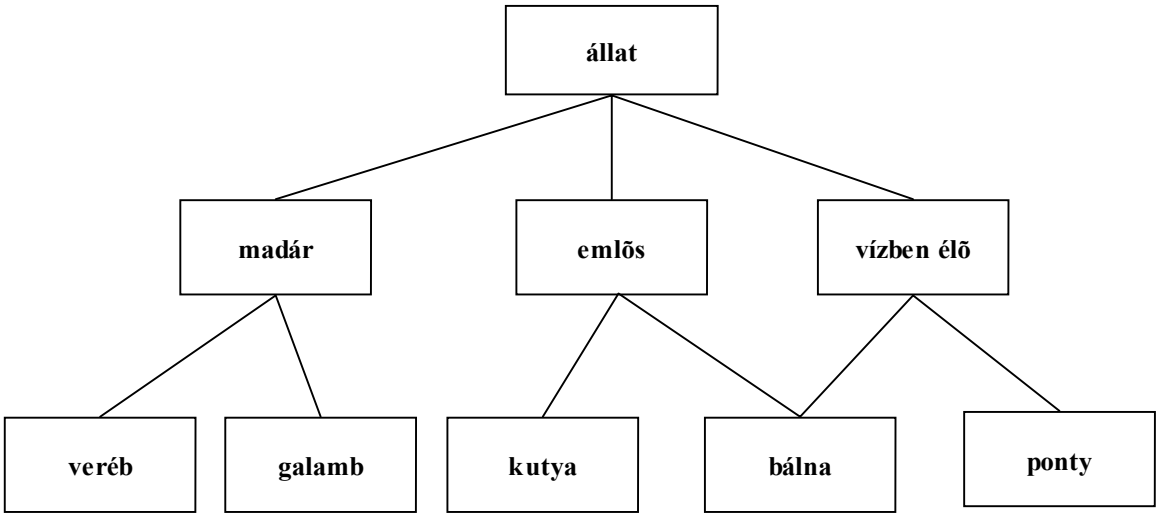
A *típus* egy objektum halmaz viselkedését specifikálja. A típus definiálja az objektum által értelmezett üzeneteket és operációk szemantikáját.

Típusokon definiálható a *kompatibilitási reláció*.

T típus kompatíbilis U-val, ha a T típusú objektum bármikor és bárhol alkalmazható, ahol az U típusú objektum használata megengedett (minden U típusú objektum által megértett M üzenetet a T típusú objektumnak is meg kell értenie)
Reláció:

- *reflexív:* T kompatíbilis önmagával
- *nem szimmetrikus*
- *tranzitív:* T kompatíbilis U-val és U kompatíbilis V-vel akkor T kompatíbilis V-vel

Felépíthető egy hierarchia:



Ha T kompatíbilis U-val akkor U a T szupertípusa, T az U szubtípusa.

3.10 Az objektumváltozó

Objektum egy konkrét osztály konkrét példánya.
Változó alkalmas értékek befogadására.
Igény, hogy legyen olyan változó amely alkalmas az objektumok hordozására.
változó:

- statikusan tipizált: típus eleve meghatározott
- dinamikusan tipizált: változónak nincs előre maghatározott típusa

Változón végrehajtható műveleteket meghatározza a változóban tárolt érték és a változó.
Művelet és változó kapcsolata: *kötés*

- statikus kötés: változó típusa határozza meg a műveleteket
- dinamikus kötés: változó értéke határozza meg a műveleteket

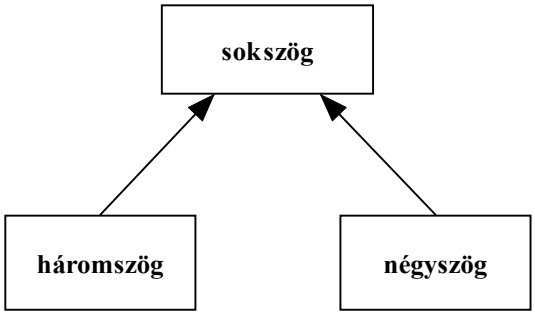
3.11 Öröklés

Az öröklés olyan implementációs és modellezési eszköz, amelyik lehetővé teszi, hogy egy osztályból olyan újabb osztályokat származtassunk, amelyek rendelkeznek az eredeti osztályban már definiált tulajdonságokkal, szerkezettel és viselkedéssel.

Öröklést tekinthetjük a kompatibilitás reláció implementációjának.

Alaposztály: az osztály, amelyből öröklünk

Származtatott osztály: az osztály, amely örökli a struktúrát és a viselkedést



Az öröklési hierarchiában egy adott osztály fölötti valamennyi alaposztályt az adott osztály ősének tekintjük.

Egy adott osztályból közvetlenül vagy közvetve származtatott valamennyi osztályt leszármazottnak nevezzük.

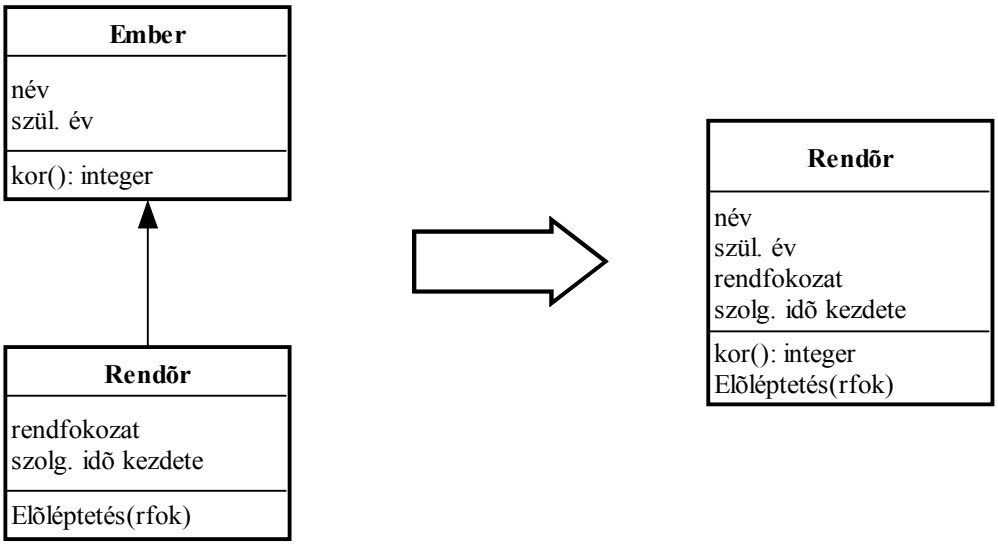
Az öröklés mögött álló relációt *általánosításnak*, vagy *specializációnak* nevezzük.

Örökléskor lehetőségünk van az ősöktől örökölt struktúra és viselkedés bizonyos vonatkozásaink megváltoztatására:

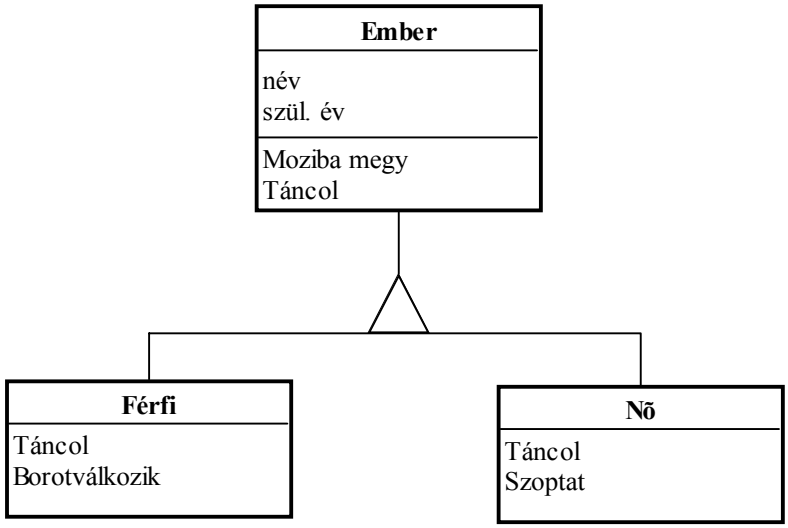
- új attribútumok hozzáadása
- új metódusok hozzáadása
- örökölt metódusok átdefiniálása

3.11.1 Absztrakt osztály

Absztrakt osztály:(nincs belőle példány) csak azért van, hogy öröklés révén átadja az attribútumait és metódusait a származtatott osztálynak.

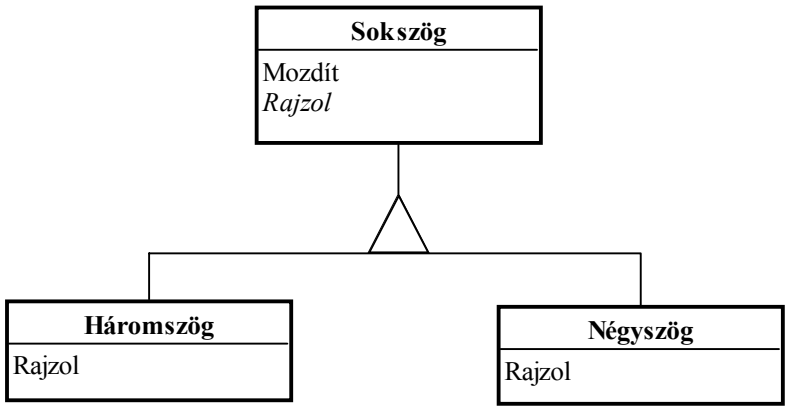


Öröklött metódus *átdefiniálható*:



A férfi és a nő osztályokban más a Táncol metódus implementálása, hiszen a tánclépések különböznek.

3.11.2 Virtuális metódus



A Mozdít-ban hivatkozni kell a Rajzol-ra. De melyik objektuméra?

Virtuális Rajzol metódus:

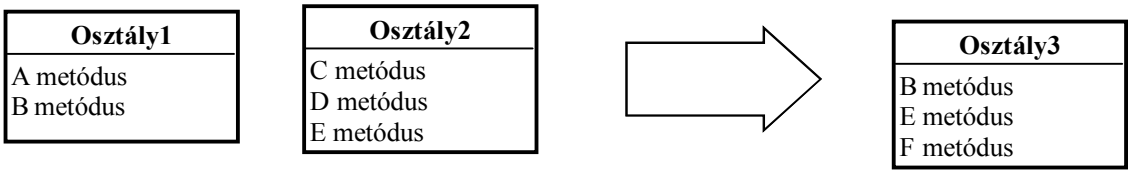
Sokszög-ben szereplő Mozdít metódusban előforduló Rajzol metódust futás közben az aktuális osztály azonos nevű metódusával helyettesíti

3.11.3 Többszörös öröklés

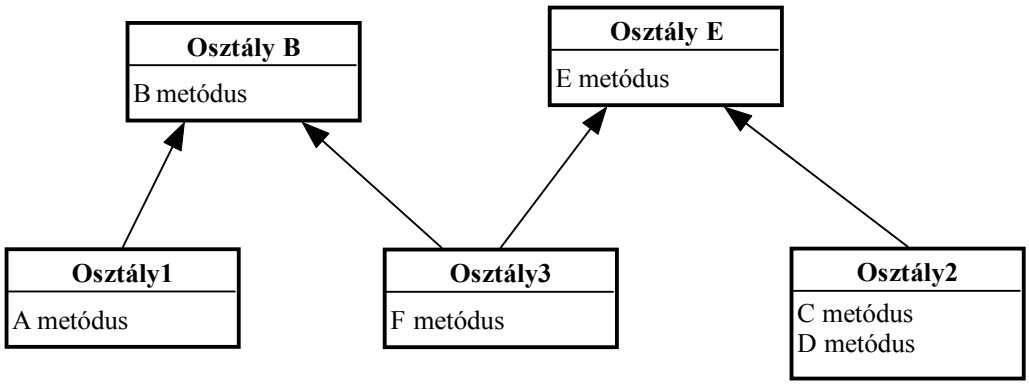
Többszörös öröklés, amikor új osztályt kívánunk definiálni, amely két vagy több meglévő osztályra épül.

Többszörös öröklésnél egy osztálynak több közvetlen őse van.

Az *Osztály3* osztályt a következőképp szeretnénk létrehozni:



Megoldás: új osztályokat kell létrehozni. Pl.:



Jelentős probléma: közös őstől ugyanazok a metódusok és attribútumok örökölhetők mindegyik öröklési ágon ám esetleg különböző utakon különbözőképpen átdefiniálva. Ennek feloldására általános szabály nincs.

4. Objektummodellezés

Az objektummodellezés célja, hogy a világot egymással kapcsolódó objektumokkal írja le. Ehhez különböző nézőpontokat használhatunk. Software esetében 3 nézőpontból vizsgáljuk a rendszert. Koncentrálhatunk az:

1. adatokra,
2. műveletekre,
3. vezérlésre.

1. Az *objektummodell* az „adat”-ot tekintve dominánsnak, írja le a rendszer statikus tulajdonságait és struktúráit.
2. A *dinamikus modell* az időbeliséget rögzíti a vezérlés nézőpontjából.
3. A *funkcionális modell* középpontjában a rendszer által végrehajtandó funkciók állnak.

Ez a három modell kielégítő képet ad ahhoz, hogy tervezni és implementálni tudjunk.

4.1 Objektummodell

Az objektummodell leírja a rendszerbeli objektumok struktúráit, attribútumait és metódusait, valamint az objektumok közti viszonyokat, kapcsolatokat, relációkat.

Megadja az alapot, amihez a funkcionális és dinamikus modellek kapcsolódnak (ezeknek csak akkor van értelmük). Osztálydiagrammal ábrázoljuk.

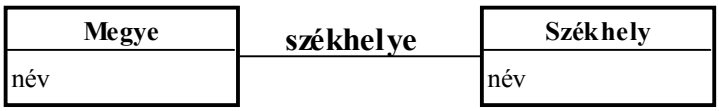
4.1.1 Attribútumok

Az *attribútumok* az objektum tulajdonságait és állapotát határozzák meg. Ezek száma és típusa definiálja az objektum struktúráját. Az attribútumoktól a következő tulajdonságokat várjuk el:

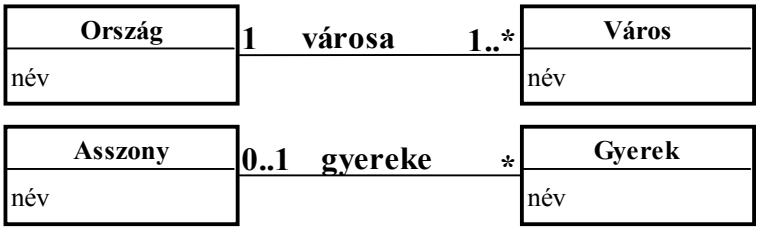
1. Legyen *egyedi* nevük.
2. Az attribútumhalmaz legyen *teljes*, azaz a modellezés szempontjából az összes jellemző tulajdonságot írja le.
3. Legyenek *függetlenek*. (Pl.: ne tároljuk egyszerre: szül. év - életkor)

4.1.2 Reláció

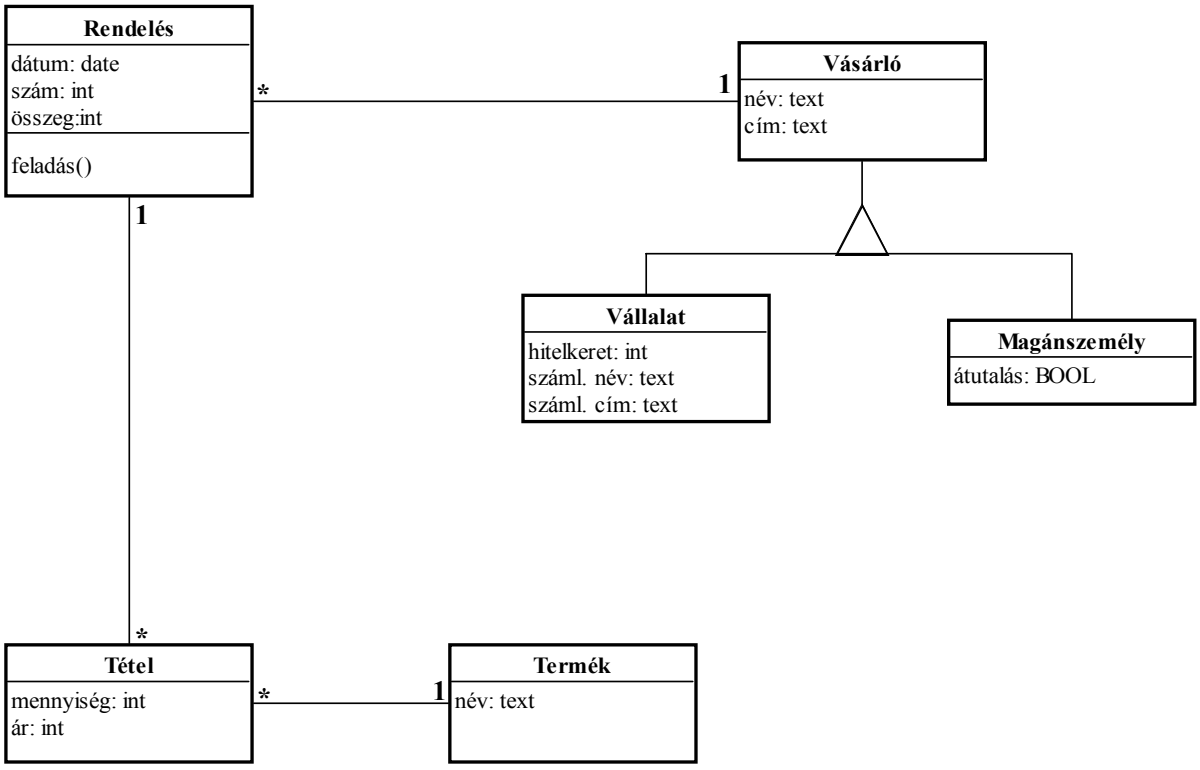
A *reláció* az objektumok, illetve osztályok közti kapcsolatot jelenti.
reláció: láncolás(objektumok közt), asszociáció (osztályok közt)
bináris reláció: ha a reláció pontosan két objektumot (osztályt) kapcsol össze.
Pl.:



Az asszociáció *multiplicitása* megadja, hogy az asszociáció az egyik osztály adott példányát a másik osztály hány példányával kapcsolja, vagy kapcsolhatja össze. (Ezt mindkét irányra meg kell adni.)
Pl.:



Az objektumokat és az objektumok közti kapcsolatokat jeleníti meg az osztálydiagram:

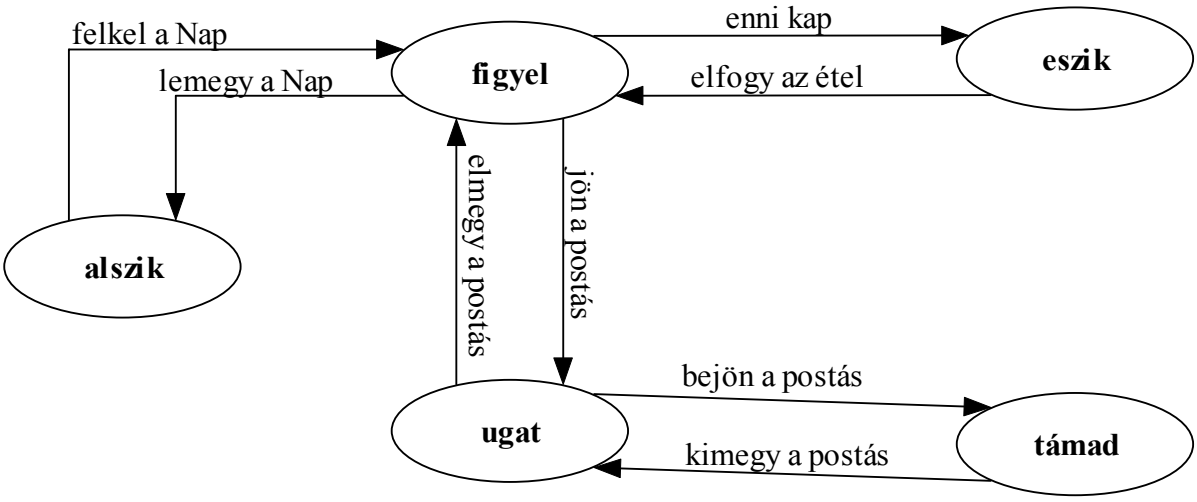


4.2 Dinamikus modell

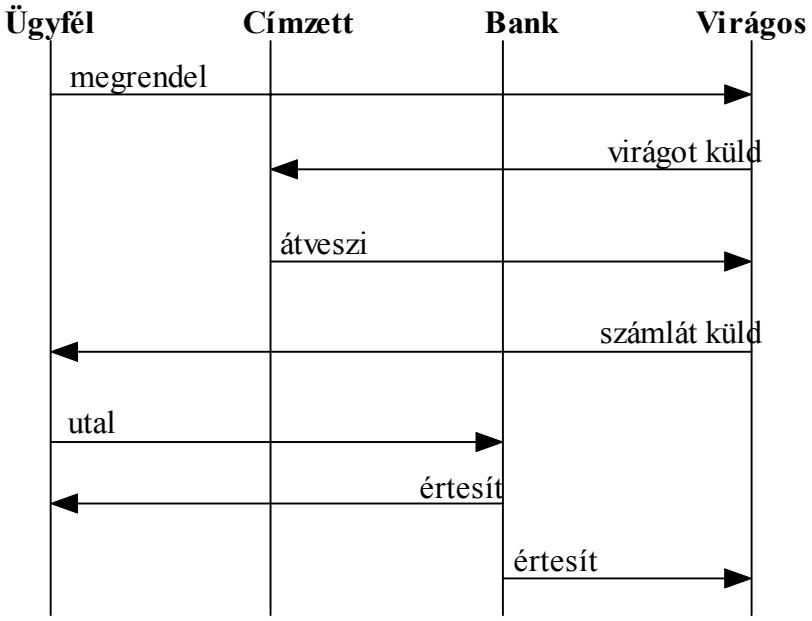
A *dinamikus modell* a rendszer időbeli viselkedését írja le. Ez gyakran sorrendiséget jelent. (a rendszert érő hatások, események sorrendjét) Azoknak az objektumoknak a viselkedését és együttműködését írja le, melyek az objektummodellben szerepelnek. Ennek ábrázolására alkalmazhatóak:

- folyamatábra
- állapotdiagram
- kommunikációs diagram

Az állapotdiagram az objektumosztály példányának, a külső események hatására történő állapotváltozását és a válaszul adott reakcióinak sorrendiségét adja meg. Pl.: kutya állapotváltozásai



Elemezhető az is, hogy a rendszer objektumai milyen üzeneteket milyen sorrendben küldenek egymásnak → *kommunikációs diagram*. Pl.: virágküldés kommunikációs diagramja



4.3 Funkcionális modell

A *funkcionális modell* leírja, hogy mi történik a rendszerben és megmutatja, hogy a rendszerbe belépő adatokból milyen lépésekkel lehet a kimenet értékeit meghatározni. Arra koncentrál, hogy a rendszer MIT csinál, figyelmen kívül hagyja, hogy MIKOR és HOGYAN. *Adatfolyam ábrákkal* írható le.

Az *adatfolyam ábra* a rendszer által előállított adatok és a bejövő értékek, illetve az adattárak közti funkcionális viszonyt írja le. Ez egy irányított gráf, ahol

- élek: adatutak
- csúcsok:
  - adattranszformáló folyamatok
  - adattárak
  - adatok forrásai és nyelői

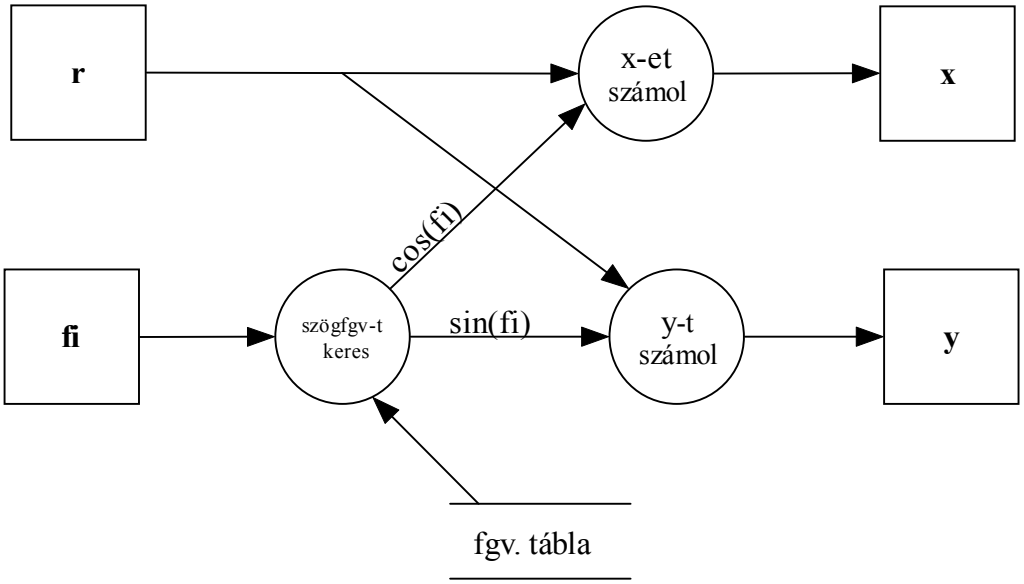
Jelölések:

adattranszformáló folyamatok: ◯

adattárak: ═══

adatok forrásai és nyelői: □

Pl.: Síkvektor polárkoordinátáiból Descartes koordinátákat számító rendszer adatfolyam ábrája:



Az *objektum* a *dinamikus* és *funkcionális* modellek az adat a vezérlés és a funkcionalitás szempontjait érvényesítő leírások. Ezek birtokában már elegendő ismeretünk van a rendszerről ahhoz, hogy tervezni és implementálni tudjunk.

Felhasznált irodalom:

[1] Dr. Kondorosi Károly, Dr. László Zoltán, Dr. Szirma-Kalos László Objektum-Orientált Szoftverfejlesztés, Computerbooks 1999.