1. Objektum

1. Az objektum azonosítható, az objektumok egymástól megkülönböztethetők, függetlenül azok állapotától.
2. Tulajdonságok, jellemzők, *attribútumok* tartoznak hozzá. Ezek között formális paraméterek is lehetnek.
3. Állapot tartozik hozzá. Az attribútumok konkrét értékei az objektum mindenkori állapotát határozzák meg.
4. Műveletek (leképezések, tevékenységek, események) tartoznak hozzá.
5. Korlátolt láthatósággal rendelkezik, azaz van *látható része*, amelyet a felhasználó ismer, és van *láthatatlan része*, amelyet a felhasználó nem ismer.
    - A látható rész az objektum külső felületét (interfészt), azaz az objektumhoz tartozó export és import műveleteknek a formáját és jelentését írja le. Ebből megtudhatjuk, hogy az objektum létezik, és hogy milyen műveletek tartoznak hozzá. (Deklaráció.)
    - A láthatatlan rész (elrejtett, burkolt rész) írja le az objektum ábrázolásának részleteit, a szolgáltatások megvalósítását. (Reprezentáció.)

6. Az objektumnak van absztrakt és konkrét megjelenési formája:
   - Az *absztrakt forma* az absztrakció valamelyik szintjének megfelelően leírt forma, azaz a konkrét ábrázolástól és megvalósítástól független forma.
   - A *konkrét forma* egy konkrét ábrázolása az objektumnak, és a hozzáférés műveleteinek ebben a formában való leírását jelenti.
7. Az objektum az osztály egy példánya.
8. Az objektumot szabványos felületek veszik körül, amelyek a hozzáférések engedélyezését határozzák meg.

objektum = identitás + megnyilvánulás + állapot.

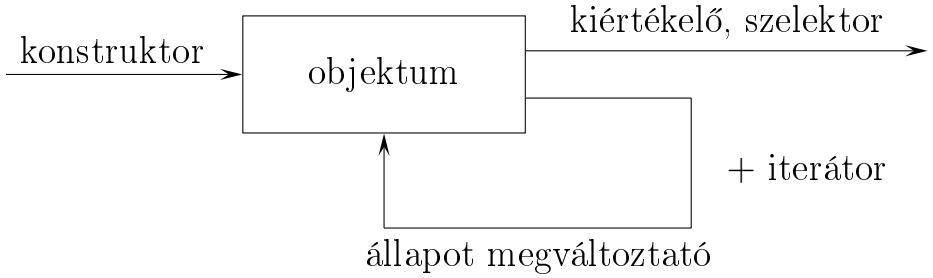
Objektumok azonosításán azt értjük, hogy a típushalmaz elemei, mint objektumok megkülönböztethetők. Ezen belül az azonosítás

- történhet névvel,
- történhet olyan állításra adott válasszal, amely az adott objektumra, és csakis arra igaz.

Az objektumok műveleteit két csoportba oszthatjuk:

1. *Export műveletek* azok, amelyeket az objektum magára nézve megenged, amelyeket más objektumok végezhetnek rajta.
2. *Import műveletek* azok, amelyeket az objektum másokon végez, amelyeket igényel ahhoz, hogy az export szolgáltatásokat nyújtani tudja.

Az export műveletek csoportjai:



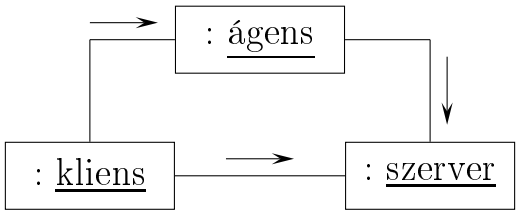
- A *konstruktor* műveletek az objektum létrehozására, felépítésére szolgálnak.
- A *kiértékelő* műveletek az objektum bizonyos jellemzőit kérdezik le.
- A *szelektor* műveletek az objektum bizonyos részét kiemelik.
- Az *állapot megváltoztató* művelet az objektum attribútumainak az értékét változtatja meg.
- Az *iterátor* az objektum felépítésében részt vevő komponensek bejárására szolgáló eljárás.

Az objektumok osztályozása viselkedésük (a műveletek típusai) alapján:

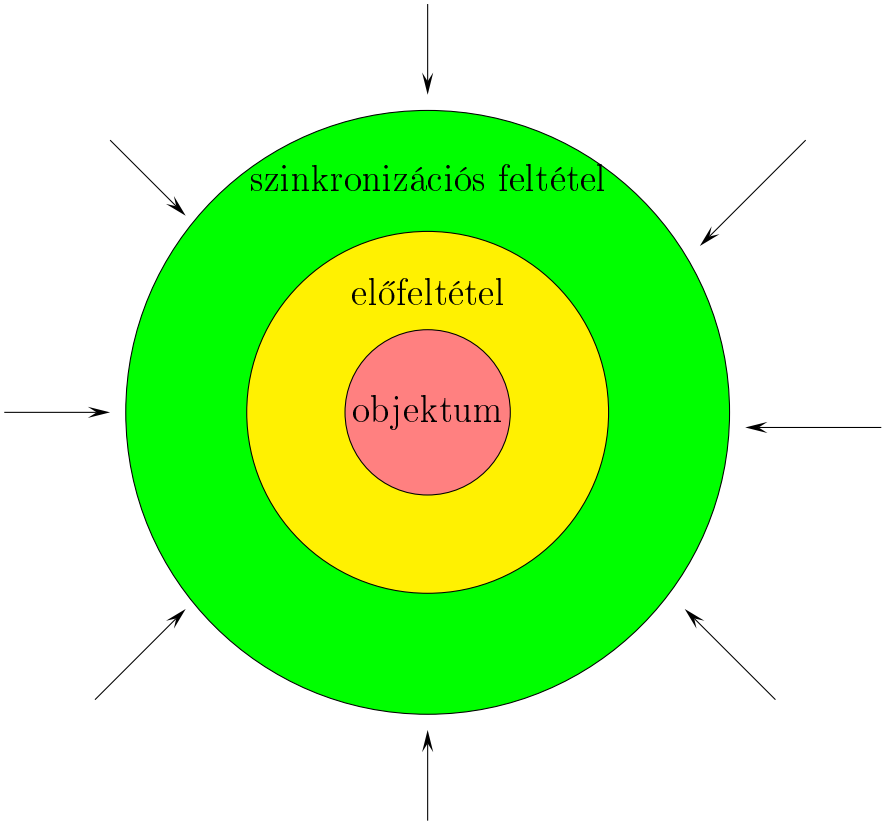
Kliens: olyan *aktív objektum*, amely csak másik objektumon végez műveleteket, de rajta mások nem végeznek műveleteket. A kliens más objektum létrejöttét, működését, megszűnését vezérelheti, de az ő működését mások nem vezérelhetik. A kliens mások szolgáltatását igénybe veszi, de másnak nem nyújt szolgáltatást. A kliensnek tehát *nincs export felülete*.

Szerver: olyan *passzív objektum*, amelynek csak export felülete van, azaz amelyen csak mások végeznek műveleteket, de ő másokon nem. A szerver másoktól érkező üzenetre vár, amelyben a működését kezdeményezik, szolgáltatását igénylik. A szerver másoknak nyújt szolgáltatást, de mások szolgáltatását nem veszi igénybe. A szervernek tehát *nincs import felülete*.

Ágens: általános objektum modell, amely *mind export, mind import felülettel* rendelkezik. Az ágens közvetítő szerepet tölt be a kliens és a szerver között.



Az objektumot körülvevő szabványos felületeletek. Erős összetartó erő érvényes belül, gyenge kapcsolódás kifelé.



2. Objektumosztály, osztály

1. *Hasonló tulajdonságú objektumok egy halmaza.* A hasonlóság az implementáció szempontjából egységesen kezelhető szerkezeti és viselkedésbeli jellemzőket jelenti.
2. *Az osztálynak van neve*, amelyet az osztályba tartozó összes objektum örököl.
3. Az osztálynak lehetnek *attribútumai*, *paraméterei*, amelyek az objektumoknak is közös építőkövei. Az attribútumok lehetnek:
   - Objektum szintű adatok, amelyek az osztály objektumaira egyesével vonatkoznak.
   - Az osztály egészére vonatkozó adatok, amelyeket az osztály objektumai közösen használnak. Ezeket objektumtól függetlenül is lehet használni.
4. Tartoznak hozzá *szolgáltatások, operációk, műveletek*, amelyek lehetnek:
   - Objektum szintű műveletek, amelyek az osztály minden objektumára különkülön vonatkoznak.
   - Az osztály egészére vonatkozó műveletek, amelyek objektumtól függetlenül is használhatóak. Ezek csak osztályszintű attribútumokat használhatnak.

5. Az osztályhoz tartozhat *import felület*, amely az általa igényelt szolgáltatások definícióját tartalmazza.

6. Az osztály lehet *absztrakt osztály*. Az absztrakt osztály megvalósítása nem teljes, bizonyos műveletek esetén csak a szolgáltatások absztrakt formáját és jelentését tartalmazza. Absztrakt osztály nem példányosítható.

7. Az osztály lehet *konkrét osztály*. A konkrét osztály minden szolgáltatásához definiált annak megvalósítása is.

8. Az osztály objektumainak attribútumaihoz és operációihoz való *hozzáférési módok*:

   - *Public*, az objektumhoz kívülről történő hozzáférést engedélyező mód.
   - *Private*, az objektumhoz kívülről történő hozzáférést nem engedélyező mód. Az attribútumok, az operációk az osztályon kívül kívülről láthatatlanok.
   - *Protected*, az objektumhoz csak az osztályon kívülről történő hozzáférést tiltó hozzáférési mód, a származtatott osztályokon belül ezek az attribútumok, operációk láthatók.
   - *Package*, az elemet csak a csomagon belül lehet látni, elérni.

9. Az osztály lehet *paraméteres osztály (sablon)*. A sablon közös formával rendelkező osztályok egy osztályát definiálja. A definíciót olyan formális paraméterekkel adjuk meg, amelyeknek

    - sem típusa,
    - sem korlátozása nincs meghatározva.

10. Egy osztály lehet *aktív* osztály. Ennek objektumai birtokolnak egy vagy több folyamatot, illetve szálat, így képesek vezérlési tevékenység kezdeményezésére.

Láthatóság

Az osztálydiagramban megadhatjuk az attribútumok és a műveletek láthatóságát, elérhetőségét a következő jelekkel:

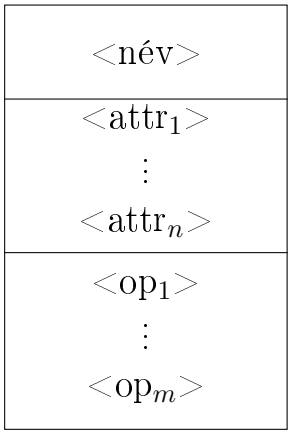
$+$ public,

$\#$ protected,

$-$ private,

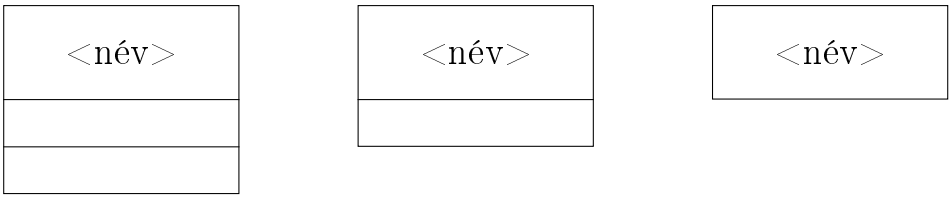
$\sim$ csomag (package).

2.1. Osztályok jelölése UML diagramokban

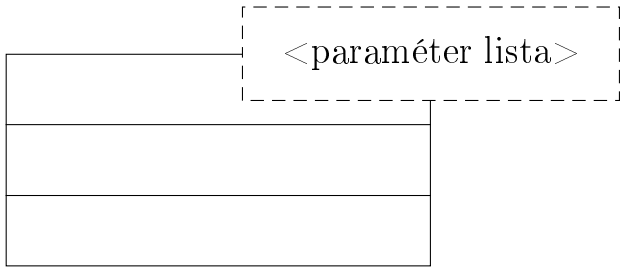


Az osztály neve félkövér betűkkel szedett.
Absztrakt osztály: a név félkövér dőlt betűkkel szedett.

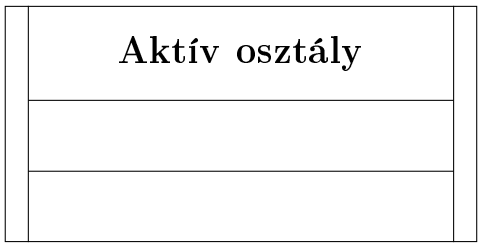
Egyszerűsített jelölések:



Sablon osztály jelölése

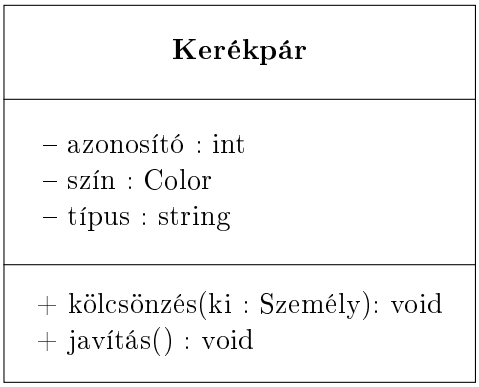


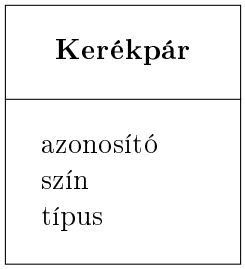
Aktív osztály jelölése



Ha egy osztálydiagramban nem szerepel aktív osztály, akkor az aktivitásról nem tehetünk fel semmit (nem kötelező megadni). Ellenkező esetben csak a jelölt osztályok lehetnek aktívak.

Tekintsük konkrét példaként a kerékpárok osztályát, ahol ismerjük az egyes kerékpárok színét, típusát és azonosítóját; a lehetséges műveletek pedig a kölcsönzés és a javítás!

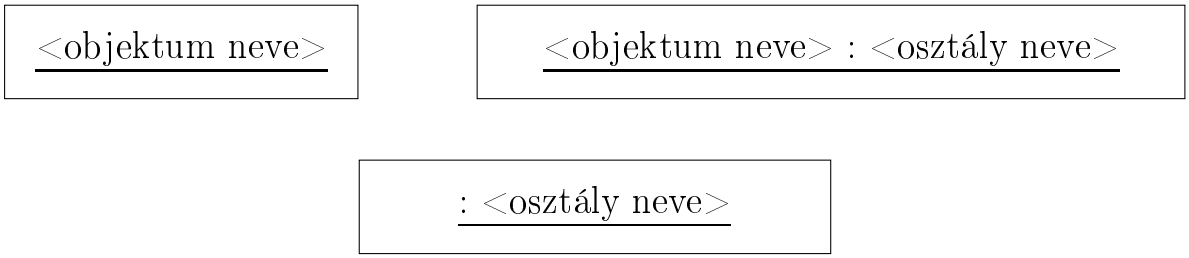




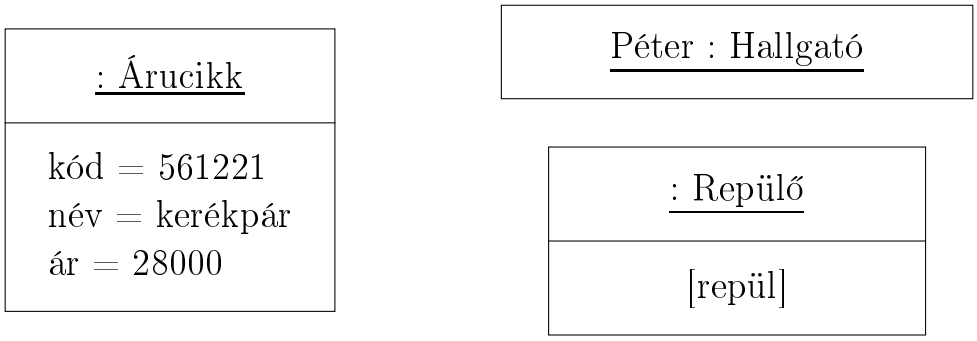

```
class Kerekpar
{
public:
    Kerekpar();
    void kolcsonzes(Szemely &ki);
    void javitas();
private:
    int    azonosito;
    string tipus;
    Color  szin;
};
```

```
public class Kerékpár
{
    private int    azonosito;
    private String tipus;
    private Color  szin;

    public Kerékpár()   { ... }
    public void kölcsönzés(Személy ki)  { ... }
    void javítás()      { ... }
}
```

2.2. Objektumok jelölése



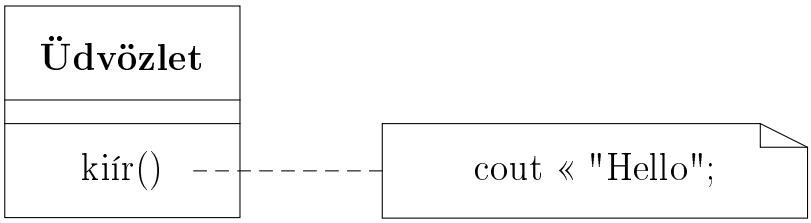
Példák:



2.3. Annotáció

A műveletek argumentumainak meghatározása, végrehajtásának tisztázása az implementáció kérdéskörébe tartozik. Ezt támogatja az UML-ben az *annotáció*. Az annotáció az UML nyelv szemantikai kiterjesztését teszi lehetővé. Jelölése:

Példa:



2.4. Megszorítások

Az attribútumok lehetséges értékeire tehetünk *megszorításokat*. Ekkor az attribútum mellé, kapcsos zárójelek közé írhatjuk a megszorítást kifejező feltételt. Például bizonyos számlák esetén a számla egyenlege nem lehet negatív.

Számla
egyenleg {egyenleg ≥ 0}

Az UML diagramokban értelemszerűen máshol is elhelyezhetünk megszorításokat. A megszorítást kifejező feltételt minden esetben kapcsos zárójelek között kell megadni.

2.5. Interfészek

Az UML-ben *interfésznek* (interface) nevezzük egy elem viselkedését jellemző műveletek névvel ellátott csoportját.

Egy interfész egy osztály (komponens) látható műveleteinek leírása, a belső szerkezet megadása nélkül. Egy interfész gyakran egy tényleges osztály viselkedésének csak egy korlátozott részét írja le. Egy osztály több (diszjunkt vagy átfedő) interfészt is támogathat. Az interfészekhez nem tartozik implementáció, nincsenek attribútumaik, állapotaik, csak műveletekkel rendelkeznek. Az interfészek között öröklődési kapcsolat fennállhat. Ekkor a származtatott interfész átveszi az ős interfész összes műveletét, és azokat újakkal egészítheti ki. Egy interfész összes művelete publikus.

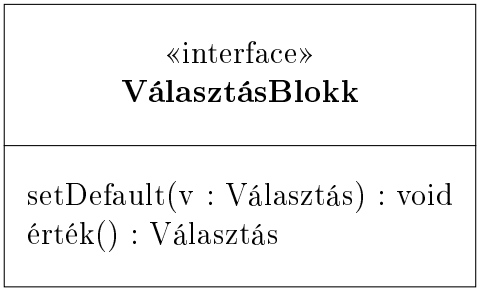
Az interfész lényegében megegyezik egy olyan absztrakt osztállyal, amelyhez nem tartoznak attribútomok és megvalósított műveletek.

Ha egy osztály megvalósít (implementál) egy interfészt, akkor az interfész összes műveletét deklarálnia kell, illetve meg kell valósítania. Ha több interészt valósít meg, akkor az összes interfész műveletére ennek kell teljesülnie. Ha ugyanaz a művelet több interfészben is szerepel, akkor a jelentésüknek meg kell egyezniük, különben konfliktust okoznak, illetve a modell helytelen.

Interfészek csak úgy vehetnek részt asszociációs kapcsolatban, ha a navigálhatóság az interfész felé mutat.

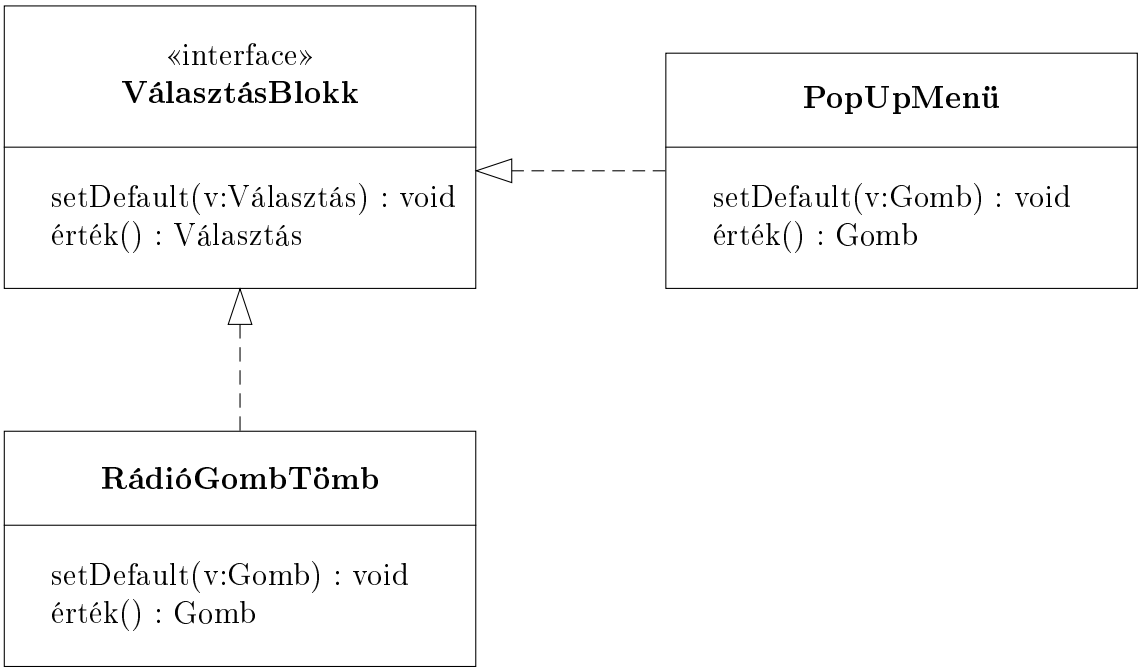
Jelölés:

Az osztályhoz hasonlóan téglalappal az `«interface»` kulcsszó feltüntetésével.

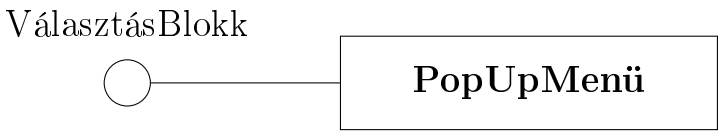



```
public interface VálasztásBlokk
{
    public abstract void setDefault(Választás v);
    public abstract Választás érték();
}
```

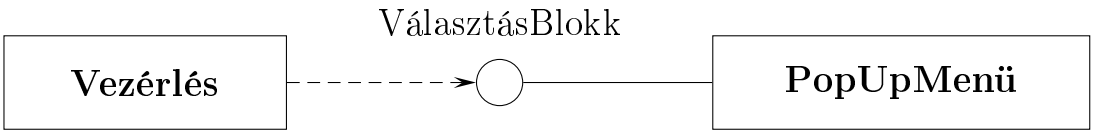
A megvalósítás jelölése:



A megvalósítás egyszerűsített jelölése:



Interfész használata:



3. Osztálydiagram

Az osztálydiagram a problématérben a megoldás szerkezetét leíró, összefüggő gráf, amelynek

- csomópontjaihoz az osztályokat,
- éleihez pedig az osztályok közötti relációkat rendeljük.

Az osztályok között a következő relációk állhatnak fenn:

- asszociáció,
- aggregáció,
- kompozíció,
- öröklődés.

Az öröklődés osztályok közötti kapcsolat, a másik három a részt vevő osztályok objektumait kapcsolja össze.

4. Objektumdiagram

Az objektumdiagram egy gráf, amelynek

- csomópontjaihoz az objektumokat,
- éleihez pedig az objektumok közötti relációkat rendeljük.

A rendszerhez egy osztálydiagram tartozik, ugyanakkor egy osztálydiagramhoz több objektumdiagram tartozhat. Mindegyik objektumdiagramnak meg kell felelnie az osztálydiagramnak. A rendszer működése során dinamikusan jönnek létre, változnak és szűnnek meg objektumok, ezért az idő függvényében változhat az objektumdiagram. Az osztálydiagram a rendszer egész idejére jellemző, az objektumdiagram egy pillanathoz köthető.

Az objektumdiagramban az osztályok helyébe azok példányai, az objektumok kerülnek. A relációk is az osztálydiagramban szereplő relációk példányai, és átveszik a megfelelő tulajdonságokat. Az objektumdiagram relációi multiplicitás nélküliek, mert az osztáldiagramban szereplő relációk multiplicitásának megfelelő számú objektum jelenik meg az adott helyen. Az öröklődési reláció nem jelenik meg az objektumdiagramban, hiszen abban konkrét objektumok szerepelnek.

5. Példa

Ipari környezetekben gyakori követelmény, hogy követni lehessen a raktározott alkatrészeket, és azok felhasználását. A példa különböző alkatrészek nyilvántartásával és felhasználásával foglalkozik. A felhasználás ebben az esetben az alkatrészek beépítése tetszőleges bonyolultságú szerelvényekbe.

A program olyan információt kezel, amely a rendszer által ismert alkatrészek leírását tartalmazza. Ezt az információt például egy gyári katalógusban lehet hozzáférhetővé tenni. A példa szempontjából egy alkatrészről elegendő a következőket ismernünk:

- hivatkozási szám (egész),
- név (string),
- ár (egész).

Az alkatrészek összeszerelhetők bonyolultabb szerkezetekbe, amelyeket szerelvényeknek nevezünk. Egy szerelvény tetszőleges számú alkatrészből állhat, és hierarchikus szerkezete lehet, vagyis tartalmazhat szerelvényeket is. A tartalmazott szerelvényt a továbbiakban részszerelvénynek nevezzük. Egy részszerelvényben lehetnek alkatrészek, illetve további részszerelvények.

Egy programot, amely ilyen információkat kezel fel lehet használni például katalógus- és leltárkezelésre, a gyártott szerelvények szerkezetének tárolására, a szerelvényeken végrehajtott különböző műveletek támogatására. Ilyen művelet lehet például egy szerelvény anyagárának a kiszámítása, amely a benne található alkatrészek árainak összege. Egy másik lehetséges művelet az összes alkatrészt tartalmazó lista készítése. A példában az előző, anyagárat meghatározó műveletet fogjuk vizsgálni.

Első lépésben a rendszer objektumait fogjuk azonosítani. A tervezés során az egyik legnehezebb feladat a rendszer adatainak felosztása objektumok halmazába úgy, hogy az objektumok sikeresen működjenek együtt a rendszer teljes működésének megvalósításában.

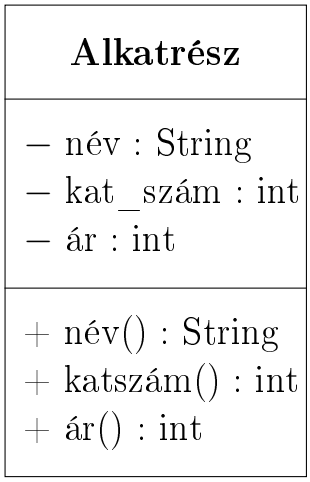
5.1. Objektumok

Egy gyakori ökölszabály az objektumok kiválasztására, hogy a valóságos objektumoknak a modellben objektum feleljen meg. Rendszerünk egyik fő feladata, hogy nyomon kövesse az összes alkatrészt, amelyeket a gyártó raktáron tart. Ezért adódik, hogy minden alkatrészt objektumként kezeljünk a rendszerben.

Sokféle alkatrész-objektum fordulhat elő, amelyek különböző alkatrészeket írnak le, de mindegyiknek ugyanaz lesz a szerkezete. Egy ugyanazt a valóságelemet kifejező objektumhalmaz közös szerkezetét osztállyal írjuk le. Az objektumhalmaz minden eleme egy példánya lesz az osztálynak. Az osztály egyrészt tartalmazza a közös szerkezetet (adatok), másrészt az objektumokon végrehajtható műveleteket. Esetünkben a következő `Alkatrész` osztály létrehozása lehet a tervezés első lépése:

```
public class Alkatrész
{
    public Alkatrész(String n, int ksz, int a)
    {
        név = n; kat_szám = ksz; ár = a;
    }

    public String   név()         { return nev; }
    public int      katszám()     { return kat_szám; }
    public int      ár()          { return ár; }

    private String  név;
    private int     kat_szám;
    private int     ár;
}
```

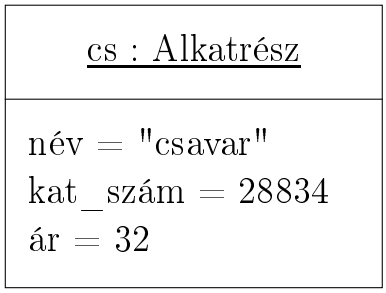
Az UML osztály a konstruktor nélkül:



Az osztályok fordítási időben lesznek meghatározva, az objektumok viszont futási időben jönnek létre, mint az osztályok példányai. Az

```
Alkatrész cs = new Alkatrész("csavar", 28834, 32);
```

művelet végrehajtása után egy új objektum jön létre. A memóriában egy terület tartozik az objektumhoz, amely a megfelelő értékekkel rendelkezik.



Azonosság

Egy harmadik lényeges eleme az objektum definíciójának, hogy az objektumok megkülönböztethetők egymástól, azaz bármely objektum megkülönböztethető bármely más objektumtól. Ez akkor is teljesül, ha két objektum pontosan ugyanazokat az adatokat tartalmazza és felületük is megegyezik. Például a következő programrészlet eredménye két objektum, amelyek állapota megegyezik, az objektumok mégis megkülönböztethetők.

```
Alkatrész cs1 = new Alkatrész("csavar", 28834, 32);
Alkatrész cs2 = new Alkatrész("csavar", 28834, 32);
```

Az objektumelvű modell feltételezi, hogy minden objektumhoz tartozik egy „azonosság", amely egyfajta címkeként megkülönbözteti az objektumot másoktól. Ez az azonosság egy belső, lényeges része az objektumelvű modellnek, és különbözik az objektumban tárolt adatok mindegyikétől. (Objektumelvű nyelvek esetén az objektum memóriabeli címe használható erre a célra. Ez nyilvánvalóan eltérő különbözö objektumok esetén.)

Az UML lehetővé teszi, hogy az objektumokat névvel lássuk el az osztálynév mellett, és így biztosítsuk az objektum egyediségét. Ezeket a neveket a modellen belül használhatjuk, és lehetőséget adnak, hogy az objektumra egyedileg hivatkozzunk a modellben. Az objektumnév nem felel meg semmilyen adategységnek. Az objektumnév különbözhet annak a változónak a nevétől, amellyel a programban hivatkozunk az objektumra. Viszont gyakran kényelmes és praktikus, ha a két név megegyezik. Ugyanakkor több, mint egy változó hivatkozhat ugyanarra az objektumra, és egy változó az élettartama során több objektumra is hivatkozhat. Ezért a névegyezés nem mindig valósítható meg.

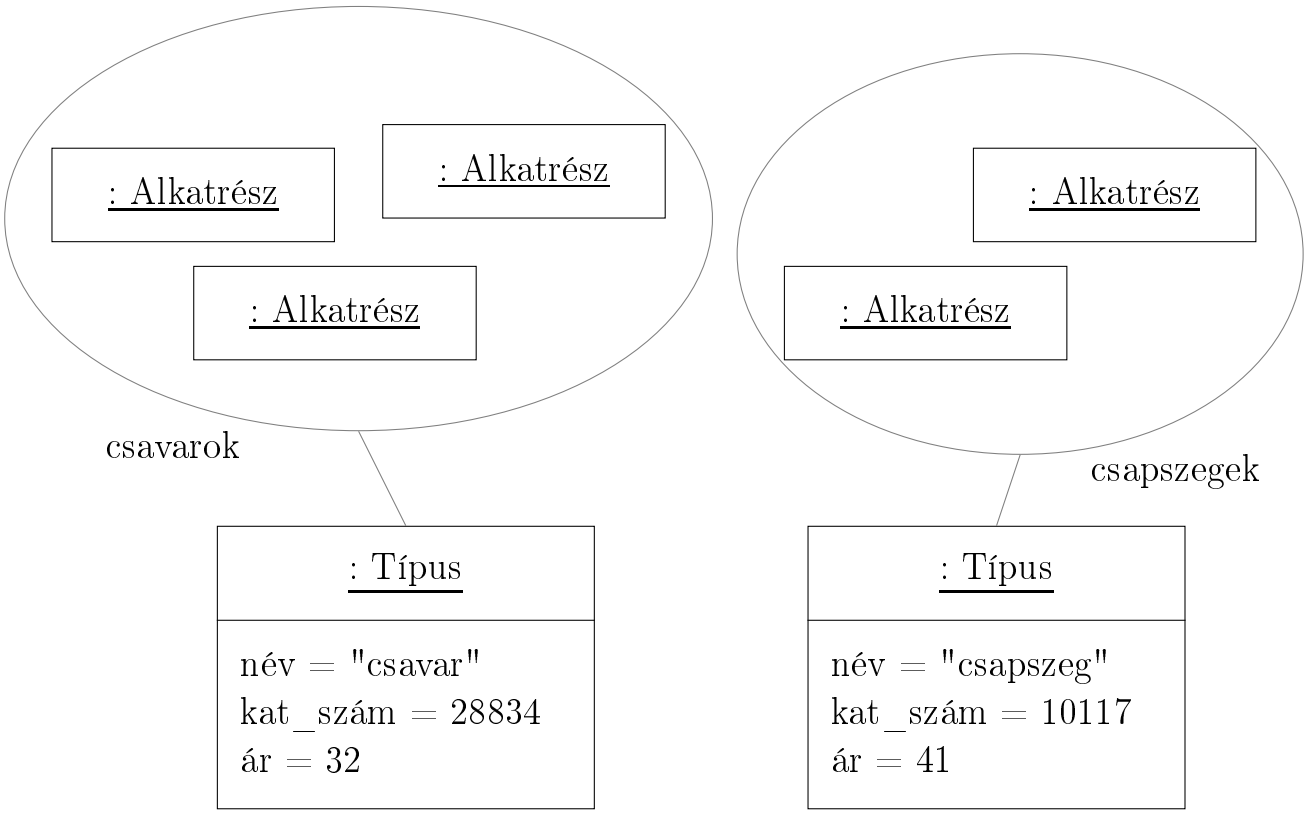
5.2. Az adatismétlés elkerülése

A modell ugyan kézenfekvő, azonban egy alkatrésztípust leíró adatokat megismételjük minden egyes alkatrész esetén, ugyanis a leírásokat az objektumokban tároljuk. Azaz, ha a rendszerben kettő vagy több alkatrész van ugyanabból a fajtából, akkor annyi objektum jön létre, és mindegyik tartalmazza ugyanazokat az adatokat.

- Jelentős redundanciát eredményez. Lehetséges, hogy a rendszerben több ezer ugyanolyan alkatrész (pl. csavar) fordul elő, amelyeknek megegyezik a neve, katalógusszáma és ára. Ha ezeket minden objektumban tároljuk, akkor jelentős mennyiségű helyet igénylünk feleslegesen.
- Az ár ismétlése várhatóan karbantartási problémához vezet. Ha egy alkatrész ára megváltozik, akkor ezt minden egyes objektum esetén el kellene végezni. Ez egyrészt csökkentené a hatékonyságot, másrészt nehéz lenne biztosítani azt, hogy minden érintett objektum esetében elvégezzük a változtatást és más objektumot nem változtatunk meg.
- Egy alkatrészre vonatkozó információt tartósan kell tárolni. Előfordulhat, hogy az adott alkatrészből éppen nincs egy sem a rendszerben, azaz nem tartozik hozzá objektum, így az információ is elveszne.

Egy jobb tervezési megközelítés, ha az azonos típusú alkatrészeket leíró közös információt egy elkülönített objektumban tároljuk. Ezek a „leíró” objektumok nem képviselnek egyedi alkatrészeket, hanem egy csatolt információt tartalmaznak, amely megadja egy alkatrész típusát. Nevezzük ezeket az objektumokat *típusnak*.

Minden egyes, a rendszerben megtalálható alkatrésztípushoz tartozik egy egyedi típus objektum, amelyben nyilvántartjuk a nevet, katalógusszámot és árat. Az alkatrészeket reprezentáló objektumokban ezek nem jelennek meg. Ezeket az adatokat a megfelelő típustól kérhetjük le, ezért ismernie kell minden alkatrésznek a megfelelő típust, hivatkoznia kell arra.


: Alkatrész
: Alkatrész
: Alkatrész
csavarok
: Típus
név = "csavar"
kat_szám = 28834
ár = 32
: Alkatrész
: Alkatrész
csapszegek
: Típus
név = "csapszeg"
kat_szám = 10117
ár = 41

Ezzel a módszerrel meg tudjuk oldani az előzőekben felsorolt problémákat.

- Az adatokat csak egy helyen tartjuk nyilván, így megszűnik a redundancia.
- Egy adott alkatrész adatainak változtatása egyszerű, csak egy típus adatait kell módosítani.
- A típus mindig létezhet, függetlenül attól, hogy mennyi alkatrész objektum található a rendszerben. Így az információ tárolható még az objektum létrejötte előtt.

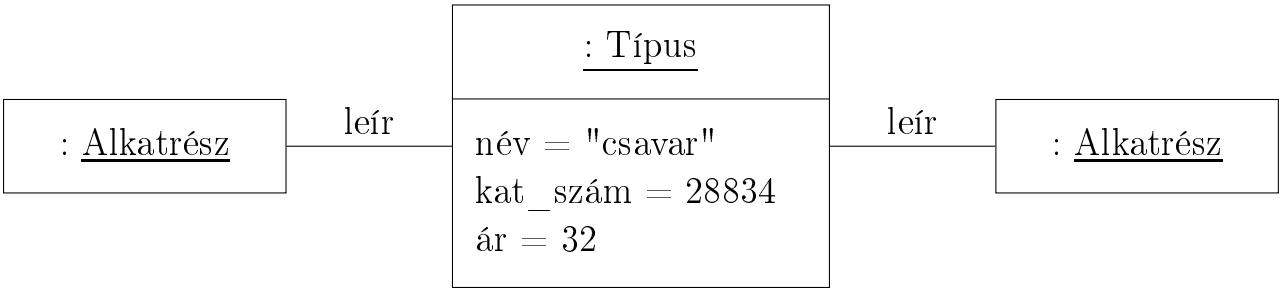
5.3. Kapcsolatok

Az új tervben két különböző osztályhoz tartozó objektumok szerepelnek. A típus objektumok azt a statikus információt tartalmazzák, amely minden adott típusú alkatrészre vonatkozik, az alkatrész objektumok egy-egy létező alkatrészt képviselnek.

A kétféle objektum között egy fontos kapcsolatot azonosíthatunk, nevezetesen, hogy egy alkatrész objektumot csak a megfelelő típus objektummal együtt lehet használni. A kapcsolat a két objektum között az, hogy a típus leírja az alkatrészt.

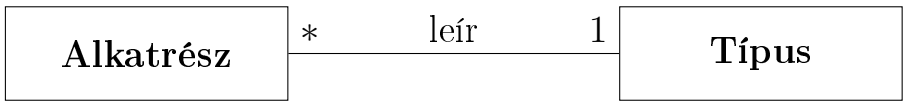
Az objektumok közötti kapcsolatokat az UML-ben asszociációnak nevezzük. Ezt a két kapcsolatban álló objektumot összekötő vonallal ábrázoljuk.

Az ábra két csavart képviselő objektumot mutat, amelyek kapcsolatban állnak a csavarokat leíró típus objektummal.



A kapcsolatok címkézhetőek egy kifejezéssel, amely kifejezi azt az egyedi kapcsolatot, amelyet a tervezés során modellezünk. Ezek a címkék tetszőlegesek, és ha nem áll fenn a félreértés veszélye, akkor gyakran el is hagyják ezeket. A címkék rendszerint igék, amelyeket úgy választunk, hogy az összekapcsolt objektumok osztályainak neveivel összeolvasva természetes nyelven kifejezzék a kapcsolat értelmét.

Az objektumok közötti kapcsolatot az osztályok között is feltüntethetjük. Ez kifejezi, hogy egy adott osztály objektuma és egy másik osztály objektuma kapcsolatban áll egymással. Ennek jelölése megegyezik az objektumok esetében megismerttel, csak további jellemzőket tüntethetünk fel a két osztály közötti él mellett. Ezek közül az egyik legfontosabb a multiplicitás, azaz, hogy egy adott osztályból mennyi objektum vesz illetve vehet részt a kapcsolatban.



A kapcsolatok implementációja

A legtöbb programozási nyelv nem definiálja a kapcsolatok implementációjának módját. A legegyszerűbb megközelítés a kapcsolatok kifejezésére általában az, hogy egy összekapcsolt objektumon (osztályon) belül biztosítani kell annak a lehetőségét, hogy az objektum tudja milyen más objektummal áll kapcsolatban.

A kapcsolat tulajdonságaitól és a használt nyelvtől függően különféleképpen érhetjük ezt el. Az egyik legegyszerűbb módszer, ha az objektumok hivatkozásokat (pointer) tartalmaznak azokra az objektumokra, amelyekkel kapcsolatban állnak. Példánkban ez megvalósítható, ha egy alkatrész hivatkozik egy típusra.

```
public class Típus
{
    public Típus(String n, int ksz, int a)
    {
        név = n; kat_szám = ksz; ár = a;
    }

    public String   név()         { return név; }
    public int      katszám()     { return kat_szám; }
    public int      ár()          { return ár; }

    private String  név;
    private int     kat_szám;
    private int     ár;
}

public class Alkatrész
{
    public Alkatrész(Típus t)   { tip = t; }
    public String név()         { return tip.név(); }
    public int    katszám()     { return tip.katszám(); }
    public int    ár()          { return tip.ár(); }
    private Típus tip;
}
```

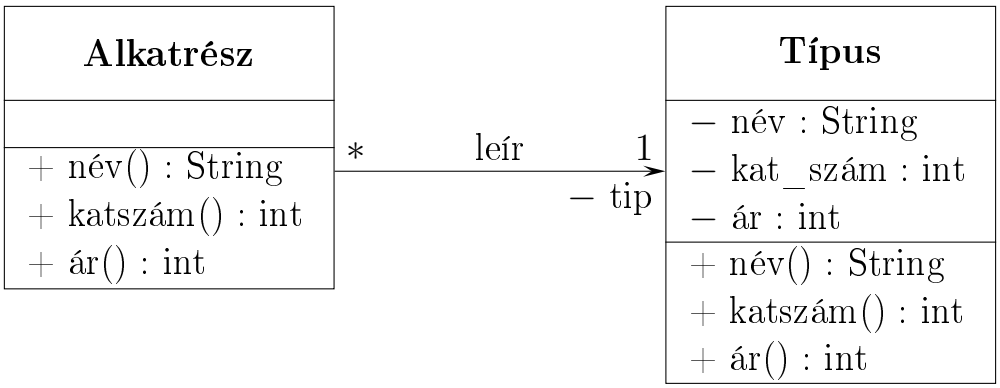
Ebben az implementációban egy alkatrész létrehozásakor biztosítani kell egy hivatkozást a megfelelő típusra. Ez nem okozhat gondot, hiszen úgyis csak meghatározott típusú alkatrészeket szeretnénk létrehozni. A következő kódsorok mutatják miként hozhatunk létre két objektumot, egy típust és egy megfelelő alkatrészt. Az alkatrészosztály konstruktora biztosítja, hogy az objektum létrejöttekor a kapcsolat is kialakuljon a megfelelő típus objektummal.

```
Típus csavar = new Típus("csavar", 28834, 32);
Alkatrész cs1 = new Alkatrész(csavar);
```

A bemutatott megvalósításban az alkatrészek osztálya tartalmaz egy mezőt, amely a megfelelő típus objektumra hivatkozik, de fordítva ez nincs meg. Azaz, a típusok osztályában nincs utalás az alkatrészekre. Ez azt jelenti, hogy egy alkatrész számára elérhető a saját típusa a mutató segítségével, azonban arra nincs lehetőség, hogy közvetlenül megkapjuk egy típushoz kapcsolt alkatrészek halmazát. (Erre a fordított irányú hivatkozásra a használat során nincs is szükség.)

Ez a kapcsolatot asszimetrikussá teszi, ami nem szerepelt az előző osztálydiagramban. Ez egy hiányosságnak tűnik, de a legtöbb esetben (akárcsak most) nincs szükség a hivatkozásokra mindkét irányban. Az egyirányú hivatkozással ugyanakkor jelentős egyszerűsítéseket lehet elérni.

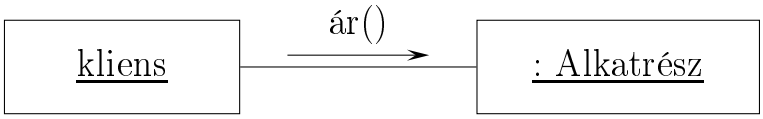
Azt a tényt, hogy a hivatkozásokat csak az egyik irányba lehet követni úgy fejezzük ki, hogy a kapcsolat csak egy irányba navigálható. A navigálhatóság egy diagramban úgy jelölhető, hogy egy nyílhegyet teszünk a kapcsolat egyik végére, amely mutatja a navigáció irányát. Ha nincs nyílhegy, akkor feltételezzük, hogy a kapcsolat tetszőleges irányban navigálható.


Alkatrész
+ név() : String
+ katszám() : int
+ ár() : int
*
leír
1
− tip
Típus
− név : String
− kat_szám : int
− ár : int
+ név() : String
+ katszám() : int
+ ár() : int

5.4. Üzenetátadás

Amint már említettük egy lehetséges felhasználása az eddigieknek egy szerelvény árának kiszámítása. Ennek a megvalósításához kell egy művelet, amellyel lekérdezhetjük egy alkatrész árát.

Objektumelvű megközelítés esetén az alapmechanizmus egy objektum adatainak lekérdezésére az üzenetküldés. (Ugyanez a helyzet minden más információ, illetve működési igény esetén is.) Az üzeneteket objektumok küldik más objektumoknak. Az objektumok közötti kapcsolatok az üzenetek közlekedési csatornái, és az üzeneteket a kapcsolatok melletti címkézett nyilakkal jelöljük. A következő ábrán egy kliens objektum üzenetet küld egy alkatrésznek, hogy lekérdezze annak árát. (Az üzenetet a megfelelő művelettel fejezzük ki.)



A kliensnek az ábrán van objektumneve, de nincs osztályneve. Az *ár* üzenetet bármilyen osztályhoz tartozó objektum elküldheti egy alkatrészhez, és az üzenet valamint az arra küldött válasz megértéséhez lényegtelen ismerni a küldő objektum osztályát. Ezért egyszerűségi és kényelmi szempontból nem tüntetjük fel a kliens osztályát.

Amikor egy objektum átvesz egy üzenetet, akkor normális esetben arra valamilyen módon reagál. Esetünkben az elvárt reakció az, hogy az alkatrész visszaküldi saját árát a lekérdező kliensnek. Azonban az ár nem attribútuma az alkatrésznek, így ez esetünkben nem egy egyszerű visszaküldést jelent.

Ez a példa megvilágítja az objektumelvű rendszereknek azt a jellemző vonását, miszerint az adatok szét vannak osztva az összekapcsolt objektumok hálózatában. Az adatok egy része attribútumként áll rendelkezésre, míg más adatokat más objektumoktól kell lekérdezni, amelyekkel az objektum kapcsolatban áll.

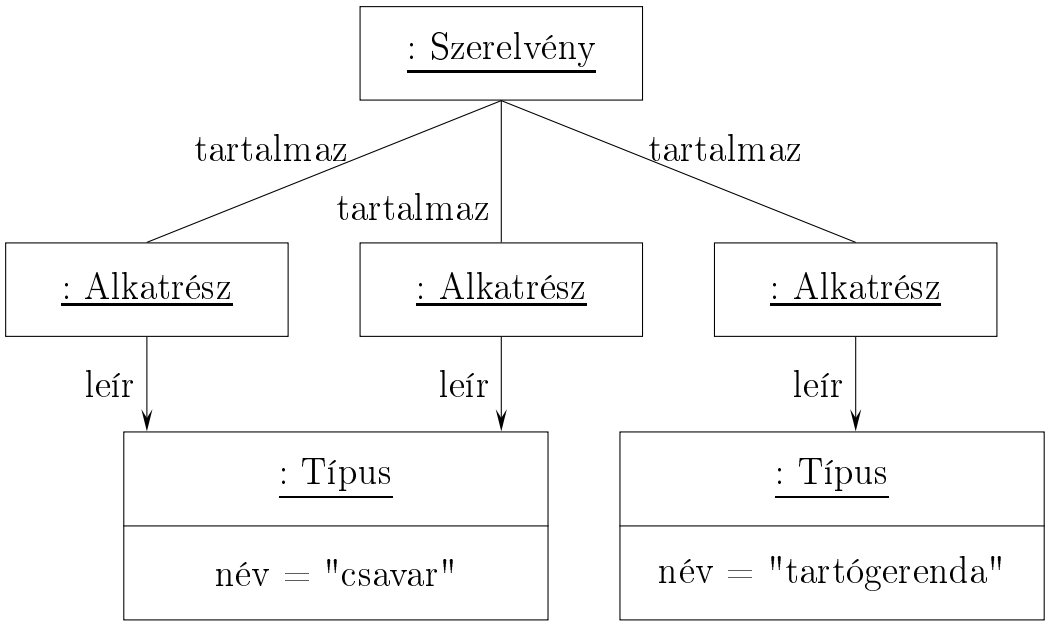
Esetünkben az ár a típusnak egy attribútuma, és ez kapcsolatban áll az alkatrésszel. Ezért az üzenet átvételekor az alkatrész egy további üzenetet küld a kapcsolt típusnak, hogy lekérje a szükséges adatot. Ezt aztán már továbbítani tudja a kliensnek.

Az üzenetek implementációja egyszerűbb esetekben megegyezik a szokásos procedurális művelethívással. Azaz a megfelelő üzenet elküldésekor a küldő objektum az adott művelet meghívásával átadja a vezérlést a hívottnak. Ennek megfelelően implementáltuk esetünkben is a megfelelő műveleteket. (Az `Alkatrész` osztály a `Típus` osztály megfelelő műveletét hívja meg.)

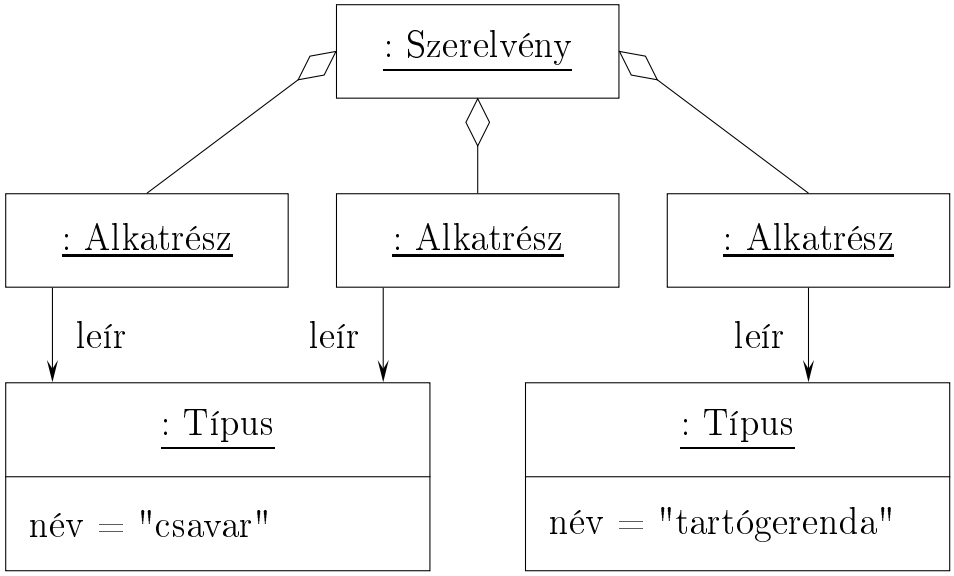
5.5. Polimorfizmus

A programnak szerelvényekkel is kell dolgoznia, ezért az alkatrészek adatainak nyilvántartása mellett, regisztrálnia kell azt is, hogy az alkatrészekből miként épülnek fel a szerelvények.

Tekintsünk példaként egy egyszerű szerelvényt, amely egy tartógerendából és két csavarból áll. A megfelelő objektumdiagram (a típusok lényegtelen attribútumait elhagytuk):


: Szerelvény
tartalmaz
tartalmaz
tartalmaz
: Alkatrész
: Alkatrész
: Alkatrész
leír
leír
leír
: Típus
név = "csavar"
: Típus
név = "tartógerenda"

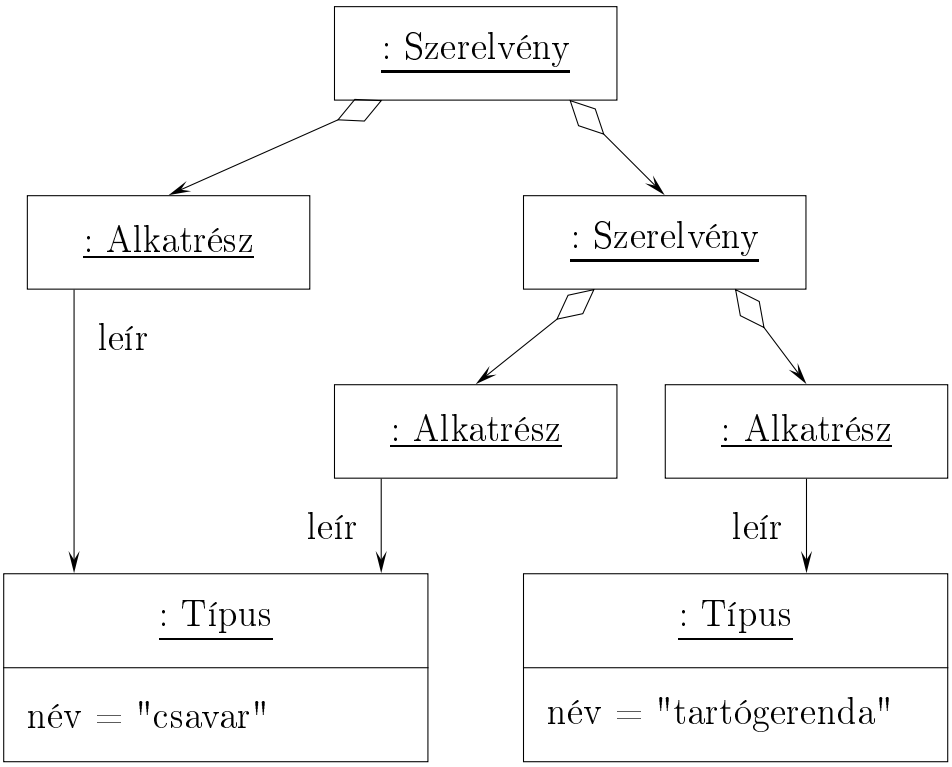
A tartalmazás jellegű kapcsolatok kifejezésére szolgál az aggregáció. Ezt felhasználva a következő diagramhoz jutunk:



A szerelvény felépítését és a benne részt vevő alkatrészeket a diagram kapcsolatai adják meg, amelyek összekötik az alkatrész objektumokat a szerelvény objektummal. Ha ezeket a kapcsolatokat az előzőeknek megfelelően implementáljuk, akkor egy szerelvénynek hivatkoznia kell az összes benne szereplő alkatrészre.

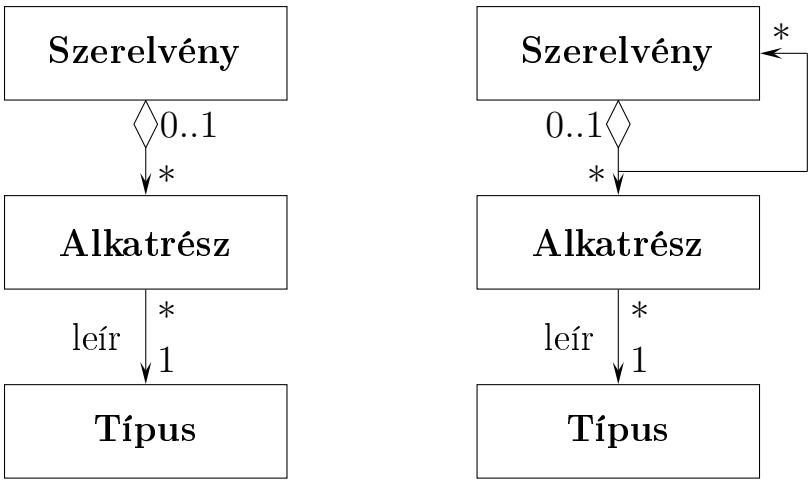
Ennek egy lehetséges módja, ha a szerelvény olyan adatszerkezetet tartalmaz, amely alkalmas több objektumra mutató hivatkozás tárolására, és vannak olyan műveletei, amelyek segítségével hivatkozásokat vehetünk fel, illetve törölhetünk. A továbbiakban feltesszük, hogy a kapcsolatok csak egy irányban navigálhatóak, így fordított irányú hivatkozásokra nincs szükség.

Ugyanakkor nem elég, hogy egy szerelvényt egyszerűen alkatrészek gyűjteményeként modellezzünk. A feladat ismertetésében szerepel, hogy egy szerelvény szerkezete hierarchikus is lehet, azaz szerepelhetnek benne részszerelvények. Az előzőleg bemutatott szerelvény másképp is felépíthető, noha ez nem feltétlen életszerű. Ebben az esetben a tartógerenda és egy csavar alkot egy részszerelvényt, és ezeket egy csavar egészíti ki, így jutunk a teljes szerelvényhez.


: Szerelvény
: Alkatrész
: Szerelvény
leír
: Alkatrész
: Alkatrész
leír
leír
: Típus
név = "csavar"
: Típus
név = "tartógerenda"

A hierarchikus szerkezet megvalósítása érdekében egy szerelvénynek alkalmasnak kell lennie arra, hogy más szerelvényeket is magába foglaljon. Azaz az aggregációs (tartalmaz) kapcsolatok nem csak egy szerelvényt kapcsolhatnak össze egy alkatrésszel, hanem két szerelvény között is kapcsolatot teremthetnek, mint azt a második lehetőség objektumdiagramja mutatja.

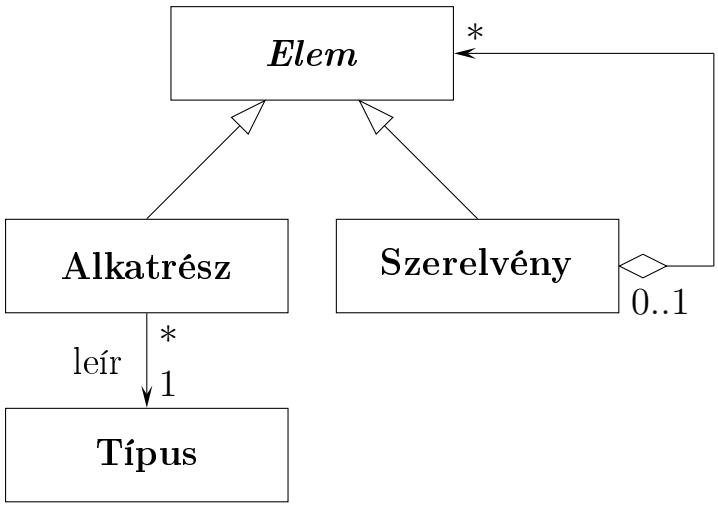
Az UML-ben a kapcsolatok típusosak, azaz egyező címkéjű kapcsolatoknak ugyanolyan típusú objektumokat kell összekötniük. Ez az első eset objektumdiagramja esetében fennáll, azonban nem teljesül a második esetben. Ez azt jelenti, hogy az aggregációs kapcsolat egyik felén nem egyetlen osztályhoz tartozó objektumok szerepelhetnek, hanem több osztály objektumai fordulhatnak ott elő. Ez különösen nyilvánvaló, ha osztálydiagramokat próbálunk felrajzolni.



Ez a helyzet egy példa a polimorfizmusra. A polimorfizmus jelentése több forma, és most azt fejezi ki, hogy több osztály objektumai kapcsolhatóak össze ugyanolyan típusú kapcsolatokkal.

Az UML típusosságát azonban fenn kell tartani, ezért meg kell határoznunk azon osztályok halmazát, amelyek részt vehetnek a kapcsolatban, és létre kell hoznunk egy általános osztályt, amelynek speciális esetei lehetnek a meghatározott halmazba tartozó osztályok. Ezután ezt az osztályt használhatjuk a kapcsolat megfelelő oldalán.

A konkrét példában vezessük be az `Elem` osztályt, amelynek specializációja lehet az `Alkatrész` vagy a `Szerelvény` osztály.



Ebben a kapcsolat végén szereplő kis háromszög fejezi ki, hogy a kapcsolat másik végén szereplő osztályok specializációi a háromszögnél szereplő osztálynak.

A program futása során a tényleges objektumok, amelyeket egy szerelvénybe teszünk az alkatrészek vagy szerelvények osztályába tartoznak, és nem az elemek közé. Ez a származtatásnál elmondottak miatt nem okoz gondot, hiszen az alkatrészek és a szerelvények osztályát az elemekből származtattuk.

Az elemek osztálya egy speciális osztály, ugyanis a program futása során nem jön létre az osztályhoz tartozó objektum. Csak alkatrész és szerelvény objektumok keletkezhetnek. Ennek oka, hogy az elemek osztálya egy fogalmat képvisel, miszerint az alkatrészek és szerelvények speciális esetei ennek az általánosabb fogalomnak. Nem azért vezettük be ezt az osztályt, hogy ilyen objektumokat képezzünk, hanem azért, hogy biztosítsuk az alkatrészek és szerelvények kicserélhetőségét bizonyos körülmények között.

Az ilyen célból bevezetett osztályokat absztrakt osztályoknak nevezzük. Az absztrakt osztálynak biztosítania kell azt a felületet, amellyel a belőle származtatott osztályok rendelkeznek. Ez a műveletekre jelent megszorítást, hiszen minden műveletnek meg kell jelennie itt is. Miután nem jönnek létre absztrakt osztályhoz tartozó objektumok, ezért a műveletek implementációjára nincs is szükség. Erre lehetőség van Javaban, sőt a Java azokat az osztályokat tekinti absztraktnak, amelyekben van implementáció nélküli művelet. Ekkor a nyelv biztosítja, hogy ne is lehessen ilyen objektumot létrehozni. Ha minden származtatott osztályban ugyanaz a művelet implementációja, akkor azt természetesen meg lehet valósítani az absztrakt osztályban is. Egy másik lehetőség az alapértelmezés megadása, amitől el lehet térni a származtatás során.

A specializáció implementálása

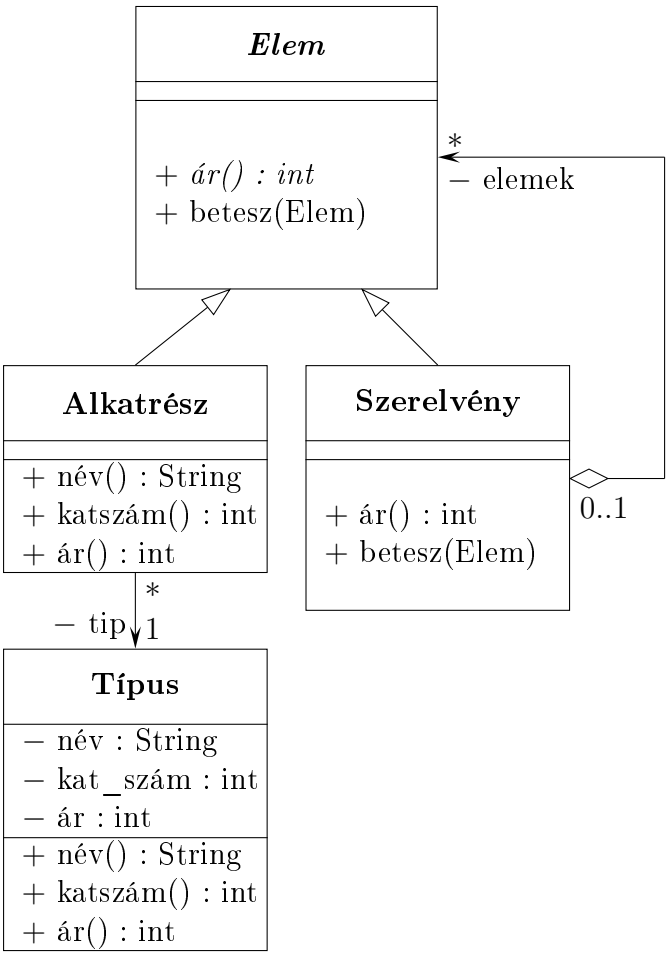
A specializáció megvalósítására az objektumelvű nyelvekben az öröklődés szolgál. Definiálhatunk egy osztályt, majd ebből az osztályból öröklődéssel származtathatunk újabb osztályokat. A származtatott osztályok példányai bárhol előfordulhatnak, ahol az eredeti osztály példányai szerepeltek.

Esetünkben tehát készítenünk kell egy `Elem` osztályt, és ebből kell származtatni az `Alkatrész` és a `Szerelvény` osztályt. A `Típus` osztály nem változik.

```
public abstract class Elem
{
    protected Elem()        {}

    public abstract int ár();

    public void betesz(Elem e)  {}
}
```

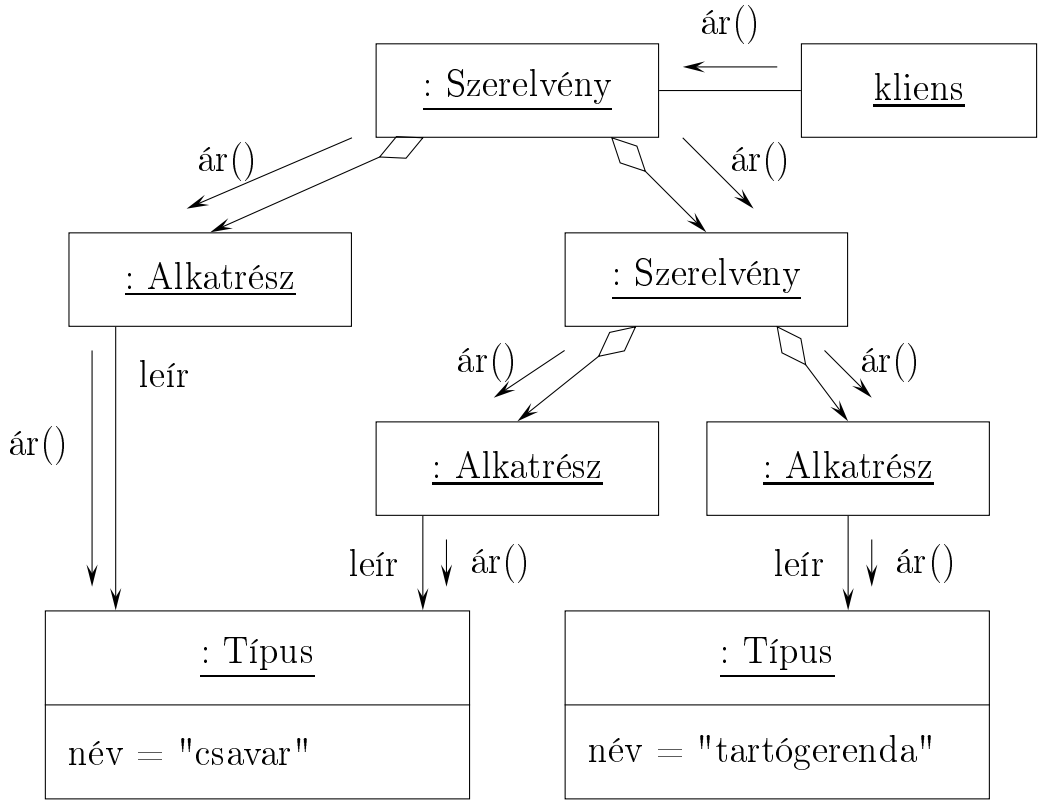
```
public class Alkatrész extends Elem
{
    private Típus   tip;

    public Alkatrész(Típus t)
    {
        tip = t;
    }

    public String név()     { return tip.név(); }
    public int katszám()    { return tip.katszám(); }
    public int ár()         { return tip.ár(); }
}
```

```
import java.util.Vector;

public class Szerelvény extends Elem
{
    private Vector<Elem>    elemek;

    public Szerelvény()
    {
        elemek = new Vector<Elem>();
    }

    public int ár()
    {
        int sum = 0;
        for ( int i = 0; i < elemek.size(); i++ )
            sum += elemek.get(i).ár();
        return sum;
    }

    public void betesz(Elem e)  { elemek.add(e); }
}
```


Elem
+ ár() : int
+ betesz(Elem)
*
− elemek
Alkatrész
+ név() : String
+ katszám() : int
+ ár() : int
Szerelvény
+ ár() : int
+ betesz(Elem)
0..1
*
− tip
1
Típus
− név : String
− kat_szám : int
− ár : int
+ név() : String
+ katszám() : int
+ ár() : int

5.6. Dinamikus összekapcsolás

Vizsgáljuk meg az ár meghatározásának mechanizmusát! Ha egy kliens lekérdezi egy szerelvény objektum árát a megfelelő üzenet elküldésével, akkor a szerelvény eleget tud tenni ennek a kérésnek, ha lekérdezi elemeinek az árát, és az összeget visszaküldi a kliensnek. Azok az elemek, amelyek önmaguk is szerelvények hasonló módon járnak el. Amint azt már láttuk, az alkatrész típusú alkotóelemek egy típus objektumtól kérdezik le az árukat.



(Látható, hogy az üzenetküldések iránya minden esetben megfelel a navigálhatóságnak.)

Az alkatrészek árának meghatározását már megtárgyaltuk: a saját `ár()` műveletében meghívja a típus `ár()` műveletét. Egy szerelvény az `ár()` műveletében az összetevő elemektől lekért árakat összegzi. Ennek során az alkatrészek és a részszerelvények felé ugyanazt az üzenetet küldi, és nem tudhatja, hogy a konkrét címzett objektum milyen típusú. Valójában nem is kell tudnia, ha a megfelelő választ kapja.

Ezt a viselkedést garantálja a dinamikus összekapcsolás. Ennek lényege, hogy nem az üzenet küldője, hanem az átvevő határozza meg mi kerül végrehajtásra az üzenet átvételekor. Polimorfizmus esetén csak a program futási idejében, a konkrét objektum ismeretében hozható meg ez a döntés. Erre szolgálnak felüldefiniált műveletek.

A példában ez azt jelenti, hogy az elemek osztályában definiálni kellett az `ár()` műveletet, amelyet az alkatrészek és a szerelvények újra definiáltak. (Az alkatrészek esetén az eddigi implementáció megfelel.) Miután elemek esetén semmi értelmeset sem tudunk mondani erről a műveletről, ezért ez egy implementáció nélküli, absztrakt művelet.