Objektumok kapcsolatai

Gregorics Tibor

gt@inf.elte.hu

http://people.inf.elte.hu/gt/oep

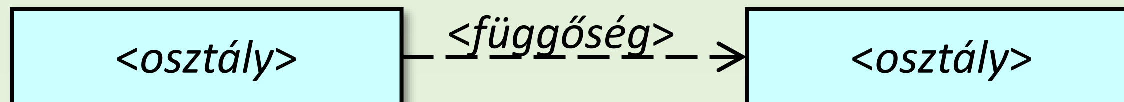
Objektum-kapcsolatok fajtái

- Amikor objektumok egymással kommunikálnak (szinkron vagy aszinkron módon egymás metódusait hívják, egyik a másiknak szignált küld, esetleg közvetlenül a másik adattagjain végeznek műveletet), akkor kapcsolat alakul ki közöttük.
- A kapcsolat fajtája lehet:
  - Függőség (*dependency*)
  - Asszociáció (*association*) vagy társítás
  - Aggregáció (*shared aggregation*) vagy tartalmazás
  - Kompozíció (*composite aggregation*) vagy szigorú tartalmazás
  - Származtatás vagy öröklődés (*inheritence*)
- Az objektumok közötti kapcsolatokat az osztályaik szintjén ábrázolhatjuk (az osztálydiagram az objektumdiagram absztrakciója).

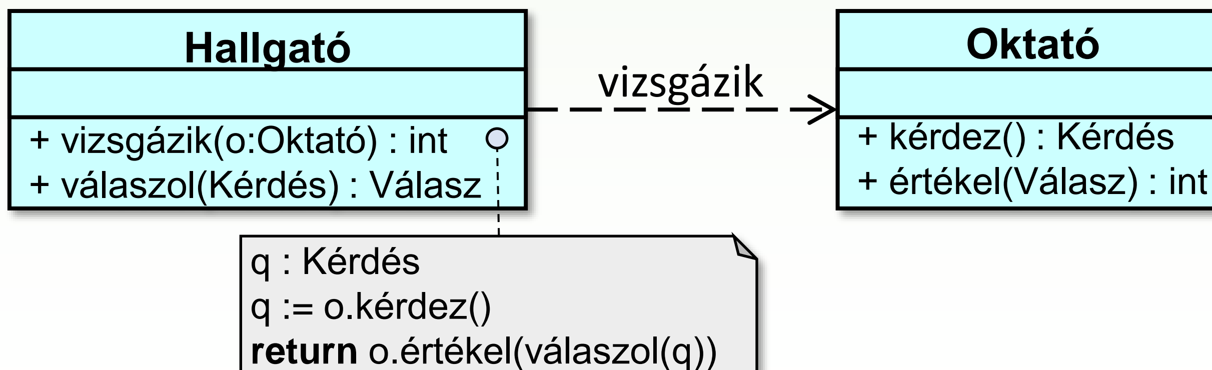
4

Az objektum-orientált nyelvek ismérve a fogalmi szintű absztrakció. (A kapcsolatok leírására azonban csak néhányuknak van nyelvi eleme.)

Függőség

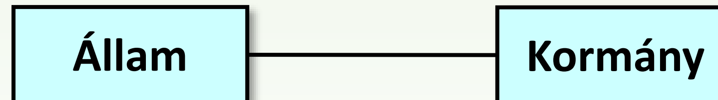


- Egy objektum epizódszerűen (rövid ideig) kerül kapcsolatba egy másik objektummal (amelyet paraméterként kap, vagy maga példányosít), mert
  - annak egyik metódusát hívja a saját metódusából,
  - szignált (üzenetet) küld neki,
  - továbbadja a hivatkozását (pl. kivételkezelésnél),
  - egy lehetséges állapotát (értékét) használja (pl. felsorolt típus értékét.
- Osztályok közötti függőség az, amikor egy metódus (amely lehet osztályszintű is) egy másik osztály osztályszintű metódusát hívja.

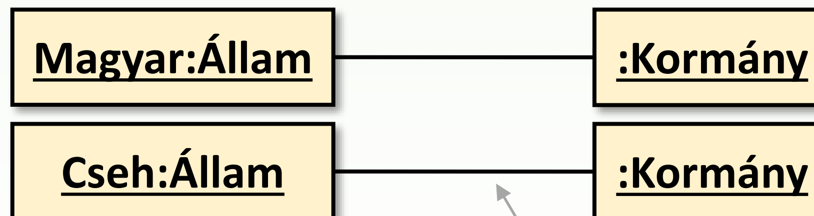


Asszociáció

- Objektumok között hosszabb időszakon keresztül fennálló kapcsolat, amelyben az objektumok sorozatosan küldenek üzenetet egymásnak.
- Matematikai értelemben az asszociáció egy reláció az adott osztályok példányait (objektumait) tartalmazó halmazok direkt szorzatán:
  - egy asszociáció több objektum-kapcsolatot ír le,
  - egy objektum egy asszociáció több kapcsolatában is megjelenhet, de különböző asszociációk kapcsolataiban is szerepelhet.



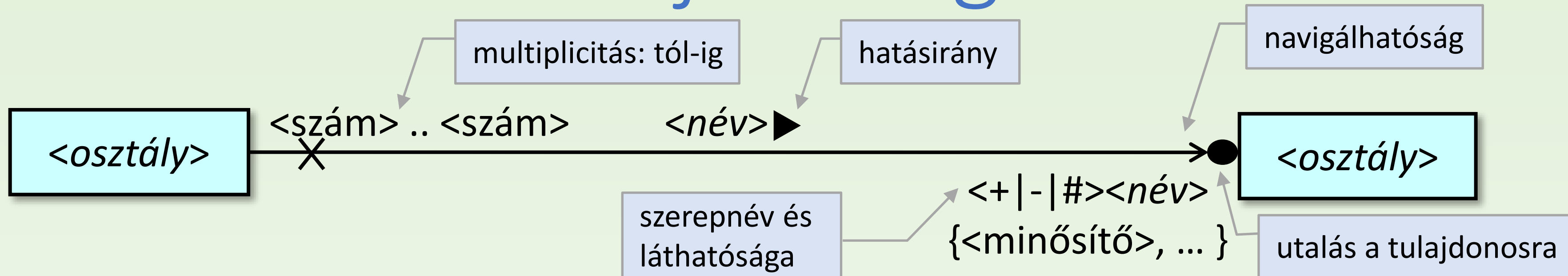
Az osztálydiagram egy lehetséges példányosítása (felpopulálása):



az objektumok (object) az osztály példányai

a kapcsolatok (link) az asszociáció példányai

Asszociáció tulajdonságai



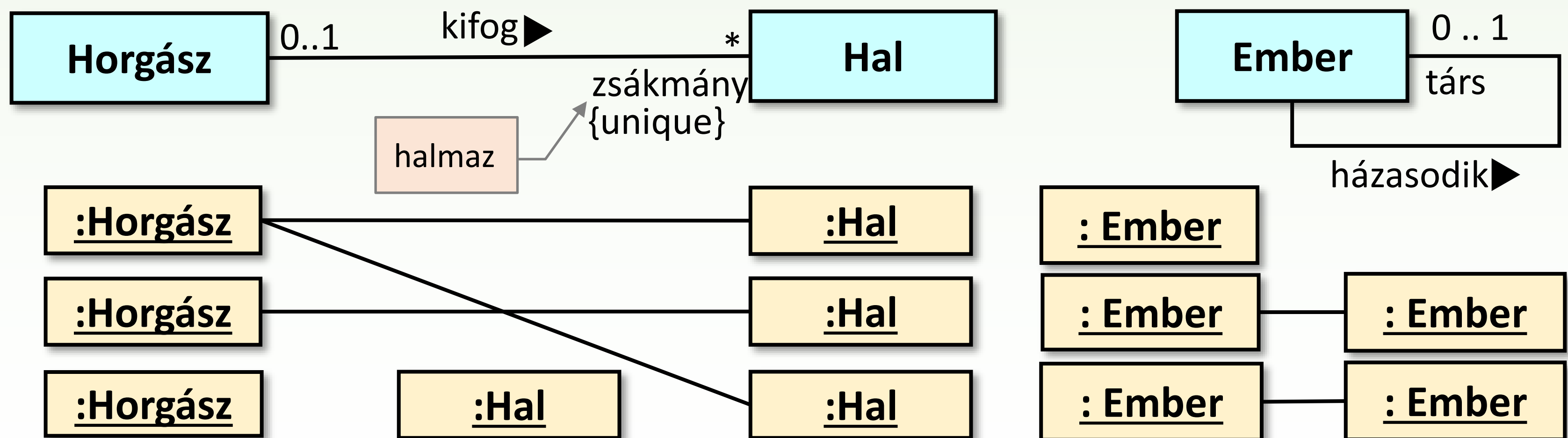
- Az asszociáció tulajdonságai az általa leírt kapcsolatokat jellemzik:
  - név: a kapcsolatok közös megnevezése
  - hatásirány: kapcsolódó objektumok közti fogalmi viszony
  - multiplicitás: egy objektumhoz kapcsolható objektumok száma
  - aritás: egyetlen kapcsolatban résztvevő objektumok száma
  - navigálhatóság: a kapcsolat melyik objektumát kell gyorsan elérni
  - asszociációvég nevek: a kapcsolatban álló objektumok azonosítói
  - asszociációvégek neveinek láthatósága
  - asszociációvégek neveinek tulajdonosa
  - egy objektumhoz kapcsolódó több objektum gyűjteményének minősítése

Nevek és a hatásirány

- Egy UML modell kifejezőképességén javít, ha a különböző asszociációkat, és annak kapcsolataiban megjelenő objektumokat nevekkel látjuk el: az asszociációkra a megoldandó feladat szövege olyan egyszerű bővített mondatokkal utal, amelynek állítmánya (néha a tárgya) az asszociáció neve, a mondat többi (nem állítmány, nem jelző) eleme pedig az asszociációvégek nevei az ún. szerepnevek lesznek.
- A bináris (két objektum kapcsolatát leíró) asszociációk neve mellé rajzolt fekete háromszög hegye az asszociáció hatásiránya, amely mindig az asszociációt jellemző mondat alanyát adó objektum felől mutat a másik (sokszor ez a mondat tárgya) irányába. Az alany osztályában többnyire megjelenik az asszociáció nevével megegyező nevű metódus is.

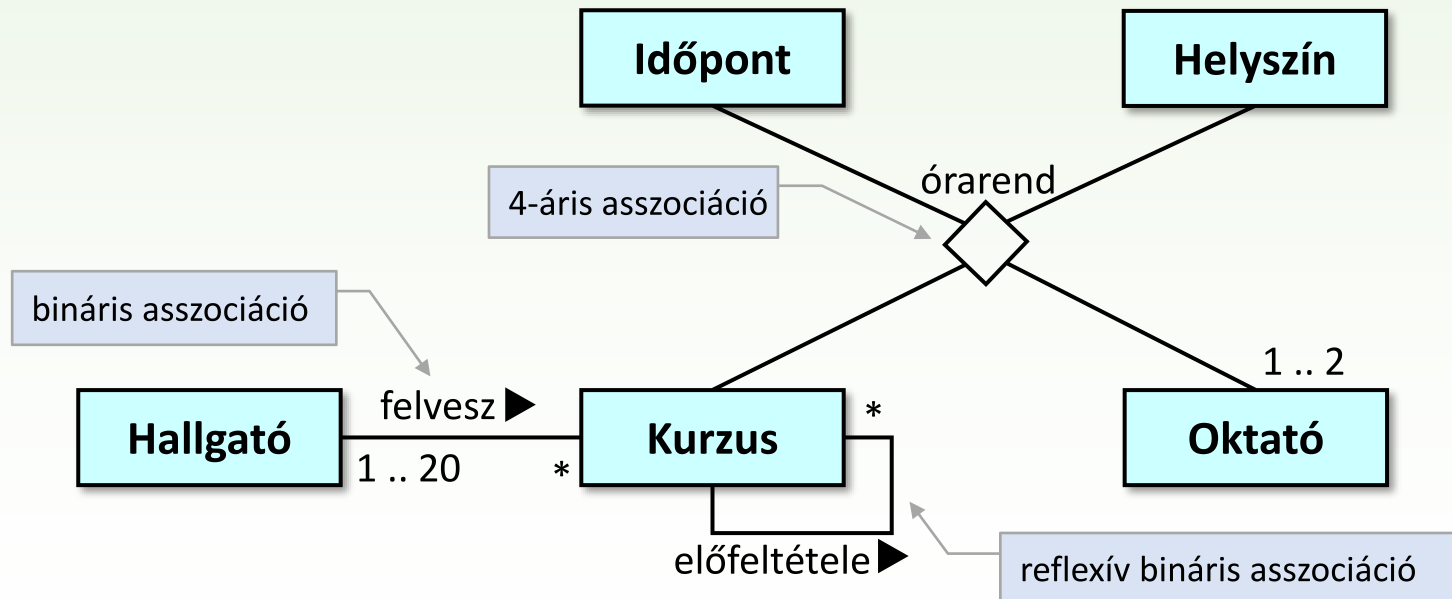
Multiplicitás

- Az asszociáció multiplicitása azt mutatja, hogy az asszociáció szerint egy objektum hány másik objektummal létesíthet egyszerre kapcsolatot. Ez lehet egy természetes szám (alapértelmezés szerint 1), vagy természetes számok min..max intervalluma. (Ha a szám, illetve a max helyén * áll, akkor az az adott érték tetszőleges voltára utal.)
- Előírhatjuk hogy egy objektumhoz kapcsolt objektumok
  - mind különbözzenek egymástól {unique},
  - megadott sorrendben legyenek felsorolhatók {ordered}.



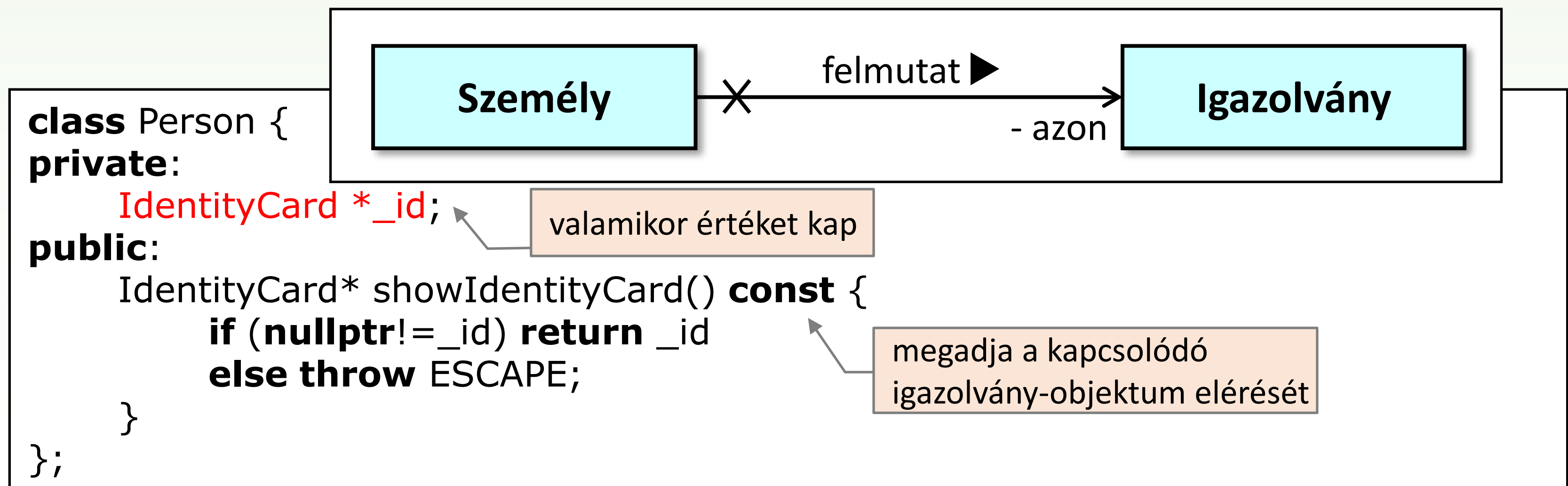
Aritás

- Az asszociáció aritása arra utal, hogy az asszociáció egyetlen kapcsolata hány objektumot köthet össze. (Eddig csak bináris asszociációkra láttunk példákat, ahol egy kapcsolat mindig két objektum között jött létre.)
- Ne keverjük össze az aritás és a multiplicitás fogalmait!



Navigálhatóság

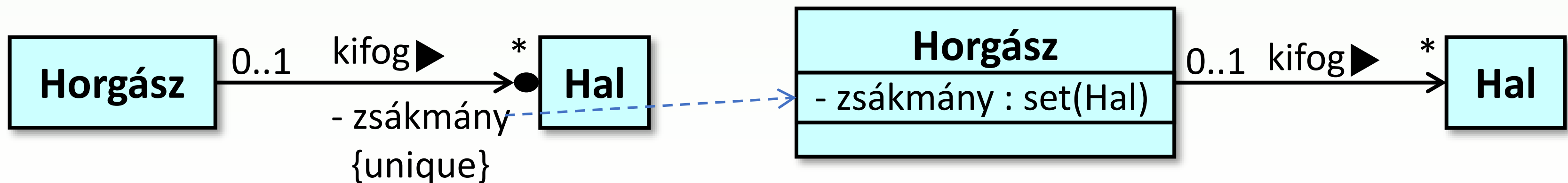
- A navigálhatóság azt jelzi, hogy az asszociáció kapcsolataiban melyik objektumot kell a kapcsolat másik (többi) objektumának hatékonyan elérni. Az asszociáció végén álló jel lehet
  - nyíl: a hatékonyan elérendő objektumok osztályára mutat.
  - „×”: a jelzett osztály objektumait nem kell hatékonyan elérni.
  - jelöletlen: nincs eldöntve vagy lényegtelen a jelzett osztály objektumainak elérhetősége.
- A hatásirány és a navigálhatóság iránya különböző fogalmak: az előbbi a modell fogalmi képét erősíti, az utóbbi az implementációra ad előírást.




```
class Person {
private:
    IdentityCard *_id;
public:
    IdentityCard* showIdentityCard() const {
        if (nullptr!=_id) return _id
        else throw ESCAPE;
    }
};
```

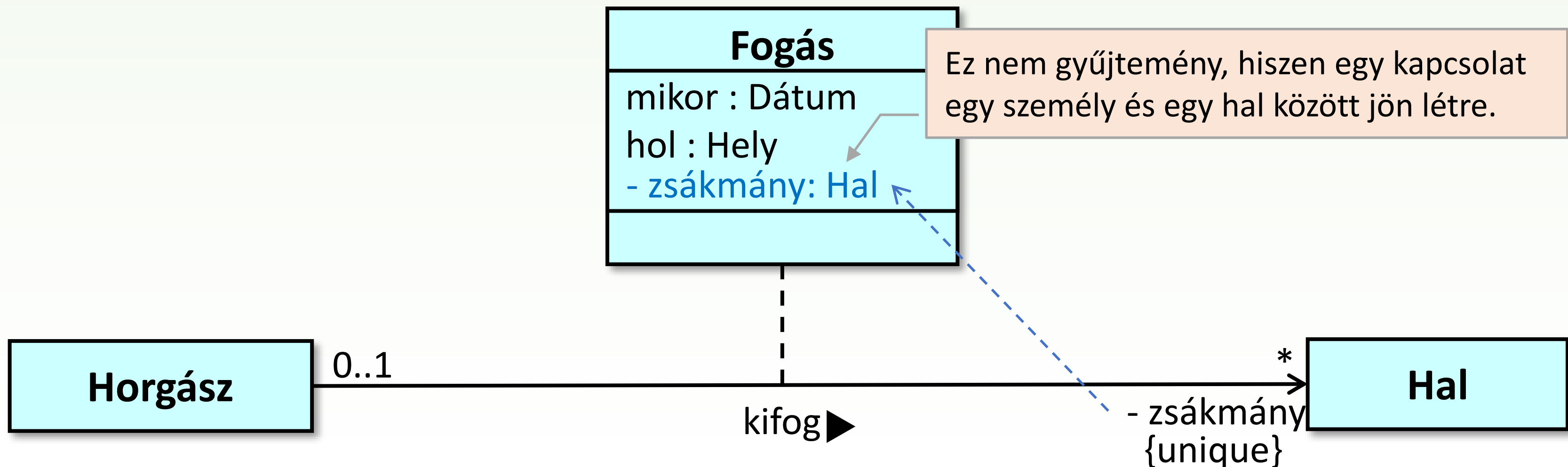
Szerepnév tulajdonosa

- Egy kapcsolat mentén zajló navigációban el kell érni a kapcsolat egyik objektumának a másik objektum hivatkozását. Hol tároljuk ezt el?
- Ki legyen egy kapcsolatban egy szerepnév által azonosított objektum hivatkozásának (röviden a szerepnévnek) a tulajdonosa?
  - Maga a kapcsolat, amennyiben a kapcsolat rendelkezik önálló tárhellyel, amelyet a kapcsolat objektumai elérnek.
  - A kapcsolat másik (többi) objektuma a saját tárhelyén. Erre utal a *fekete pötty* az asszociációs vonal végén, az adott szerepnévnél.
- A szerepnév láthatósága (private, protected, public) mutatja, hogy ez a név publikus, vagy sem; multiplicitása pedig azt, hogy gyűjtemény-e.



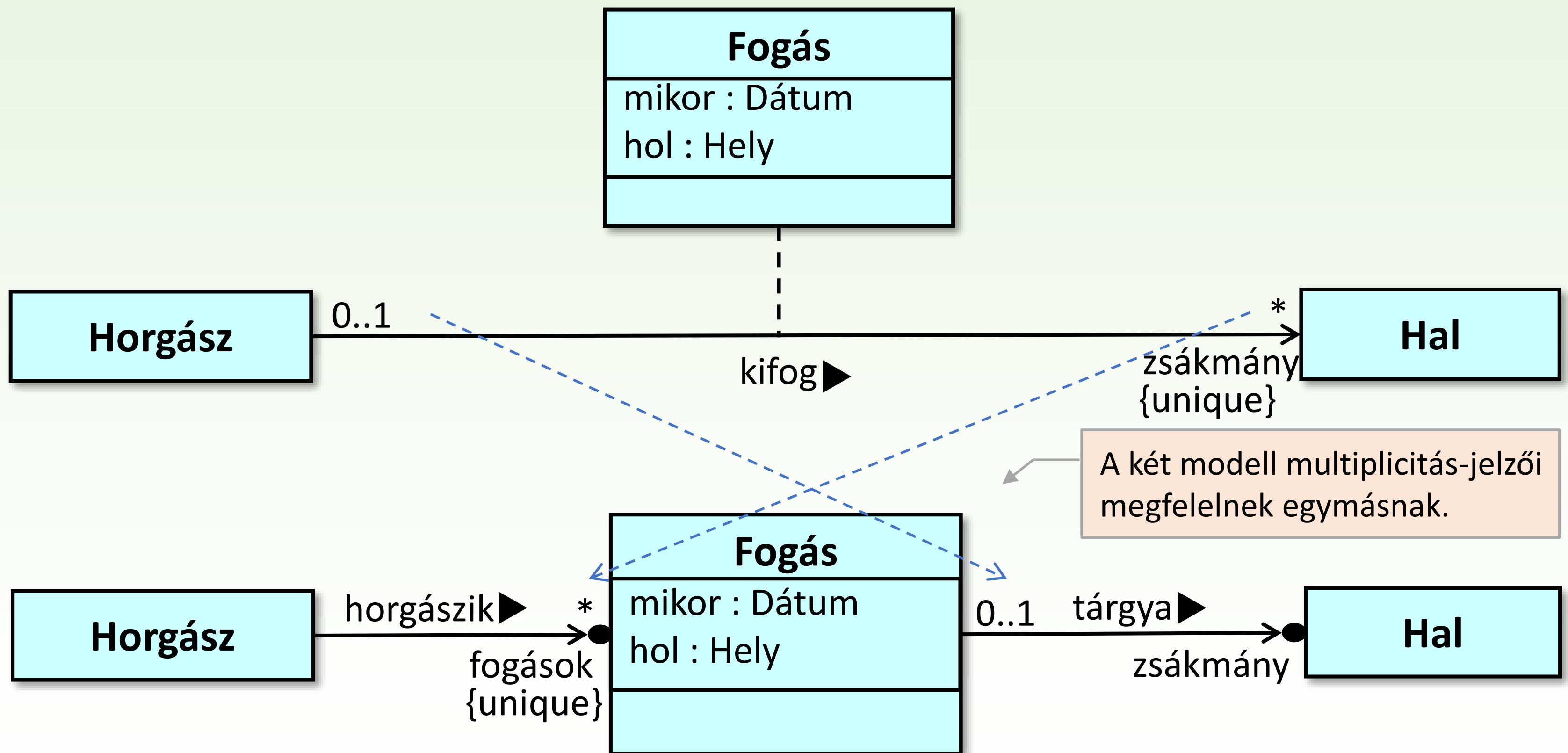
Asszociációs osztály

- Az UML lehetőséget ad arra, hogy az asszociációk kapcsolatainak egyéb tulajdonságait is leírhassuk. Erre az asszociációk mellé rendelt ún. asszociációs osztály szolgál, amelynek a példányai lesznek az általa jellemzett asszociáció tulajdonképpeni kapcsolatai, és egy-egy ilyen kapcsolatot közvetlenül elérnek az általa összekötött objektumok.
- Amikor az asszociáció a tulajdonosa egy szerepnévnek, akkor ez a szerepnév az asszociációs osztálynak (és persze az abból példányosított kapcsolatoknak) lesz az adattagja.



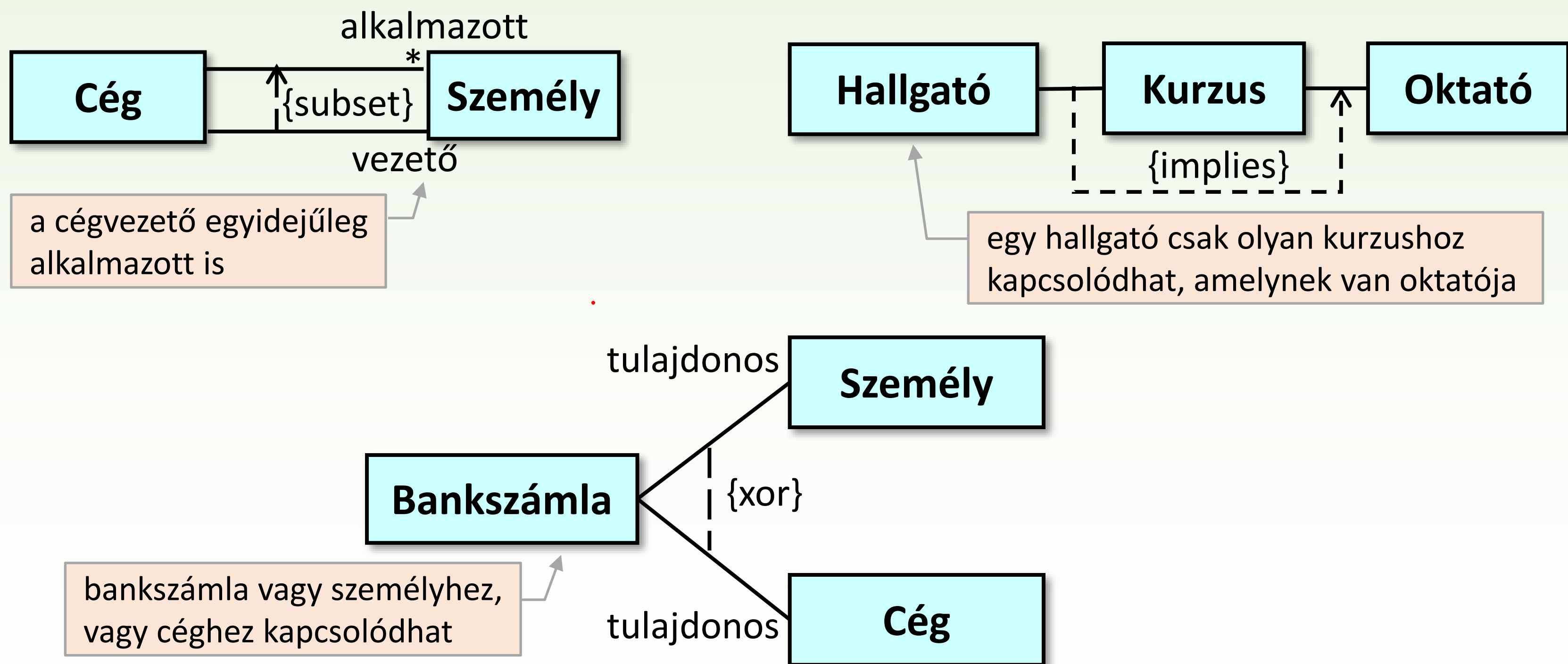
Asszociációs osztály kiküszöbölése

- A programozási nyelvek többsége az asszociációs osztály fogalmát nem támogatja: a kapcsolatokhoz nem rendelnek önálló tárhelyet. Ha erre szükség van, akkor közönséges osztály segítségével helyettesíthetjük.

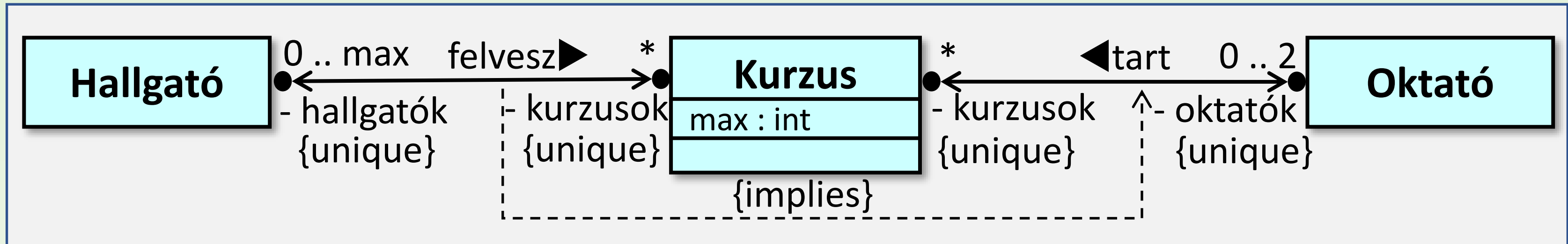


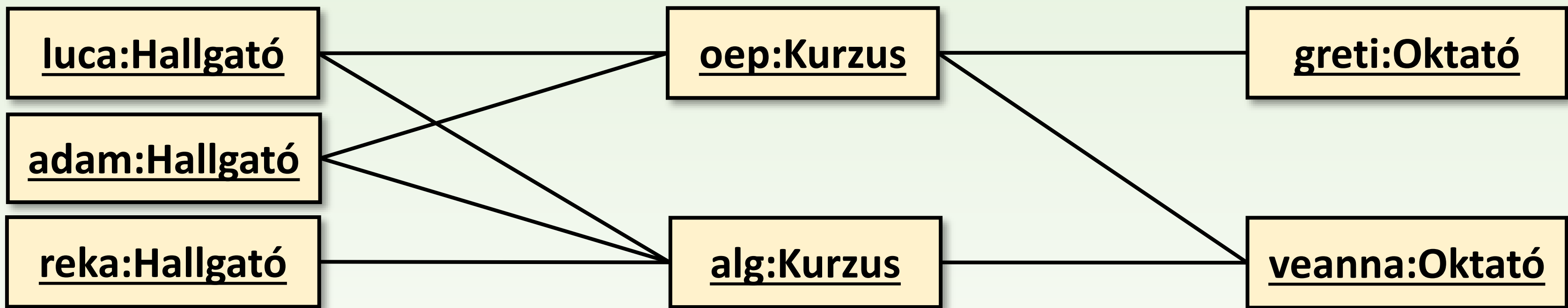
Asszociációk közötti feltételek

- Megadhatunk az asszociációk között logikai feltételeket (subset, and, or, xor, implies, …), amelyek különböző asszociációk kapcsolatai közötti korlátozásokat fogalmazzák meg.



Példa: Kurzusok



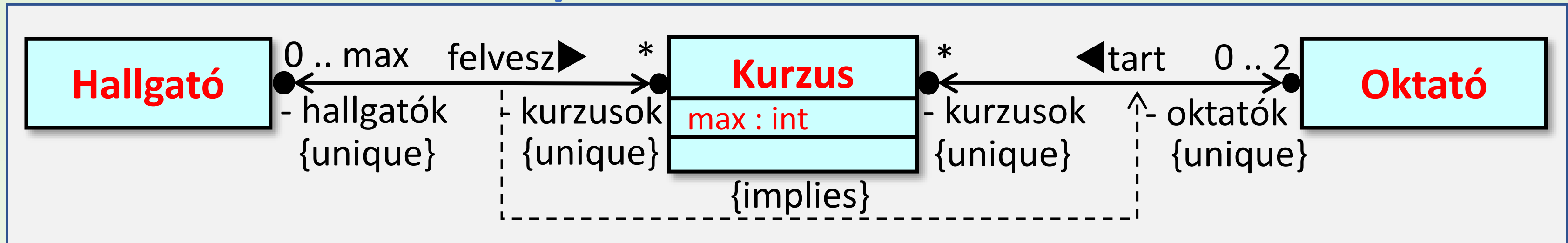



```
Student luca, adam, reka;
Teacher greti, veanna;
Course oep(20), alg(22);

greti.undertakes(&oep);
veanna.undertakes(&oep); veanna.undertakes(&alg);

luca.signs_up(&alg); luca.signs_up(&oep);
adam.signs_up(&alg); adam.signs_up(&oep);
reka.signs_up(&alg);
```

Először az objektumokat példányosítjuk.

A kapcsolatokat az asszociációk nevével azonos nevű metódusok hozzák létre.

Példa osztályai



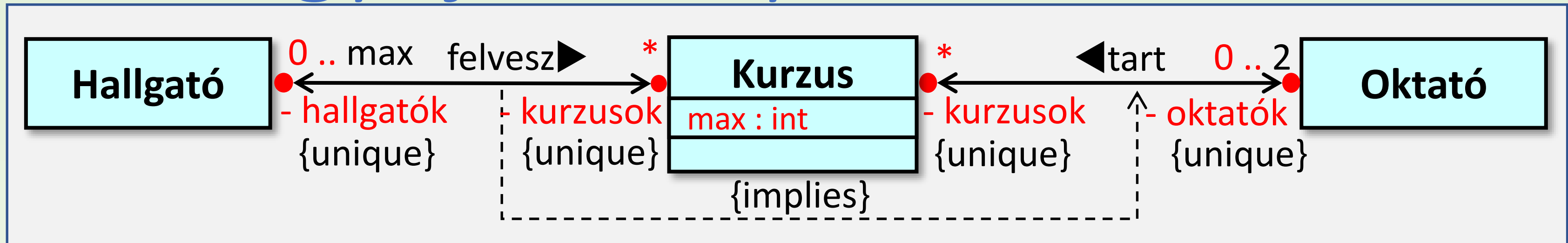
Felvesszük a jelzett osztályokat.

```
class Course {
private:
        int _max;
        ...

public:
        Course(int a) : _max (a) { }
};
```

```
class Student {
private:
        ...
public:
        ...
};
```

```
class Teacher {
private:
        ...
public:
        ...
};
```

Példa gyűjteményei



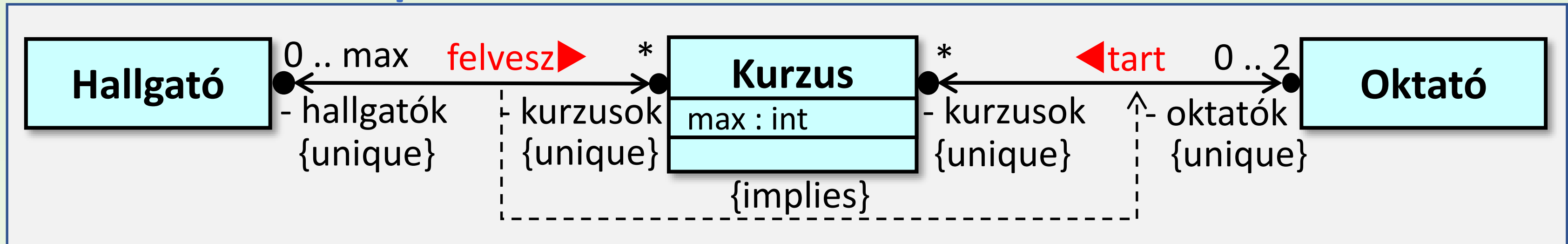
Gyűjtemények tárolják egy objektumhoz kapcsolódó objektumok hivatkozásait. Ez hatékonyan támogatja a navigációt.

```
class Course {
private:
    int _max;
    std::vector<Teacher*> _teachers;
    std::vector<Student*> _students;
public:
    Course(int a) : _max (a) { }
};
```

```
class Student {
private:
    std::vector<Course*> _courses;
public:
    …
};
```

```
class Teacher {
private:
    std::vector<Course*> _courses;
public:
    …
};
```

Példa kapcsolatainak létrehozása



Az asszociáció neve egy ugyanilyen nevű metódust feltételez az asszociációval összekötött osztályokban.

```
class Course {
private:
        int _max;
        std::vector<Teacher*> _teachers;
        std::vector<Student*> _students;
public:
        Course(int a) : _max (a) { }
};
```

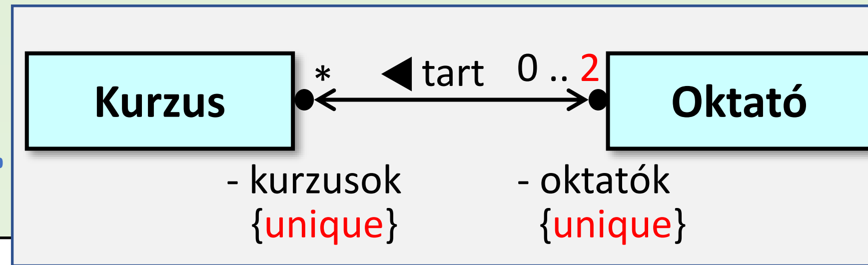
```
class Student {
private:
        std::vector<Course*> _courses;
public:
        void signs_up(Course *pc);
};
```

A felvesz() metódus hoz létre egy újabb hallgató-kurzus kapcsolatot.

```
class Teacher {
private:
        std::vector<Course*> _courses;
public:
        void undertakes(Course *pc);
};
```

A tart() metódus hoz létre egy újabb oktató-kurzus kapcsolatot.

Példa metódusai 1.



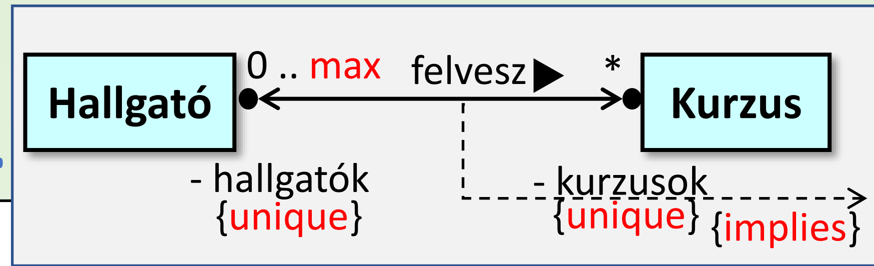

```
class Course {
private:
    int _max;
    std::vector<Teacher*> _teachers; // 0 .. 2
    std::vector<Student*> _students;
public:
    Course(int a) : _max (a) { }
    bool can_lead(Teacher *pt) {
        if (_teachers.size()>=2) return false;
        for (Teacher *p : _teachers) {
            if ( p==pt ) return false;
        }
        _teachers.push_back(pt);
        return true;
    }
};
```

Itt vizsgáljuk, hogy egy újabb kapcsolattal nem lépjük túl a multiplicitás felső korlátját.

Ez a lineáris keresés vizsgálja meg, hogy az újabb kapcsolattal nem sérülnek-e a unique feltételek.

```
class Teacher {
private:
    std::vector<Course*> _courses;
public:
    void undertakes(Course *pc){
        if ( pc==nullptr ) return;
        if ( pc->can_lead(this) ) {
            _courses.push_back(pc);
        }
    }
};
```

Példa metódusai 2.




```
class Course {
private:
    int _max;
    std::vector<Teacher*> _teachers; // 0 .. 2
    std::vector<Student*> _students; // 0 .. max
public:
    Course(int a) : _max (a) { }
    bool can_lead(Teacher *pt) { … }
    bool has_teacher() const { return _teachers.size()>0; }
    bool can_sign_up(Student *ps) {
        if ( _students.size()>=unsigned(_max ) ) return false;
        for (Student *p : _students) {
            if ( p==ps ) return false;
        }
        _students.push_back(ps);
        return true;
    }
};
```

Itt ellenőrizzük a multiplicitás felső korlátját.

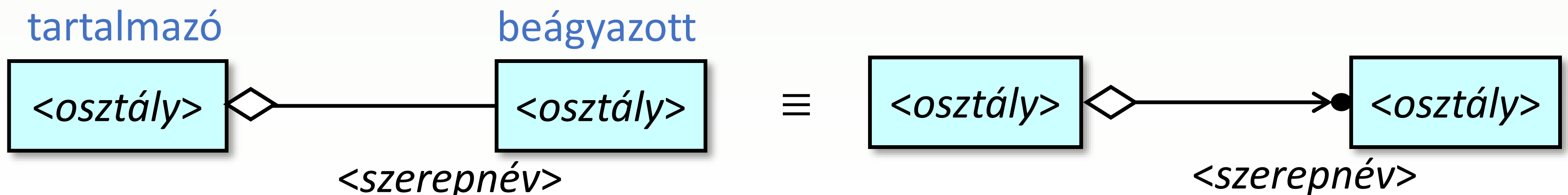
Itt ellenőrizzük az implies feltételt.

Ez a lineáris keresés vizsgálja meg, hogy az újabb kapcsolattal nem sérülnek-e a unique feltételek.

```
class Student {
private:
    std::vector<Course*> _courses;
public:
    void signs_up(Course *pc){
        if ( pc==nullptr  || !pc->has_teacher() )
            return;
        if ( pc->can_sign_up(this) )
            _courses.push_back(pc);
    }
};
```

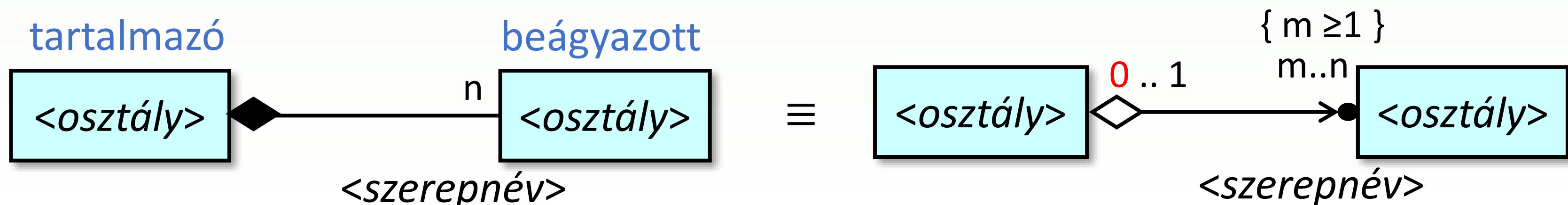
Aggregáció

- Egész-rész kapcsolatot kifejező bináris asszociáció, amely azt írja le, hogy egy objektumnak része, tulajdona egy másik:
  - Ez egy aszimmetrikus, nem reflexív, tranzitív binér reláció a tartalmazó és a beágyazott objektumok között, tehát nem alkothat irányított kört (azaz egy objektum még közvetett módon sem lehet önmaga része, tulajdona).
  - Tartalmazó objektumnak nem kell mindig rendelkeznie beágyazott objektummal; a beágyazott objektum pedig létezhet önmagában is, vagy egyidejűleg több tartalmazó objektumnak is része lehet.
- Megállapodás szerint a beágyazott objektum
  - irányába hatékonyan navigálhatunk a tartalmazó objektumból,
  - szerepneve a tartalmazó objektum tulajdona.

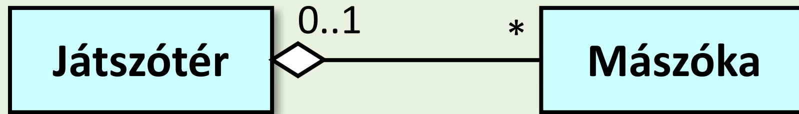


Kompozíció

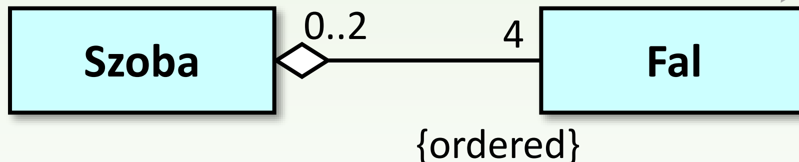
- Speciális aggregáció (tehát aszimmetrikus, nem reflexív , tranzitív binér reláció), de
  - a tartalmazó objektum nem létezhet beágyazott objektum nélkül,
  - a beágyazott objektum egyszerre csak egy objektum része lehet.
- A kompozíció fogalmán sokan az alábbi, a beágyazott objektumra tett, egyre szigorodó megszorításokat is hozzáértik:
  - van tartalmazó objektuma: beágyazandó önmagában nem létezhet
  - a tartalmazó objektuma nem változik: a beágyazott létrehozása és megszüntetése a tartalmazó feladata
  - élettartama azonos a tartalmazó objektumával: a tartalmazó konstruktora példányosítja, destruktora törli a beágyazottat.



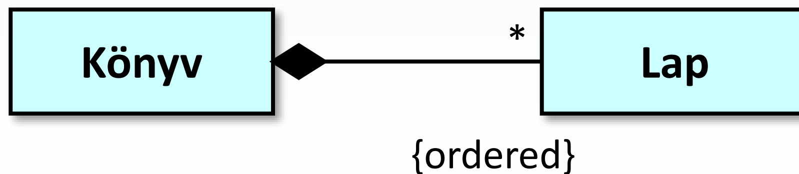
Példák: aggregáció vs. kompozíció



A mászókák a játszótér részei, de egy játszótér mászóka nélkül is használható. (Habár egy mászóka egyszerre csak egy játszótérhez tartozhat, de át lehet vinni másik játszótérre, sőt önmagában is használható.)

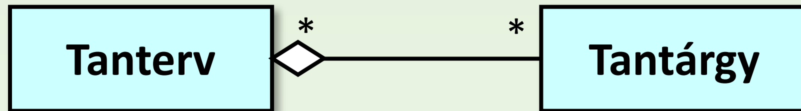


A falak a szobák részei, de ugyanaz a fal egyszerre két szobához is tartozhat. (Egy szoba kidőlt falakkal talán már nem szoba, viszont egy fal önmagában is fal.)

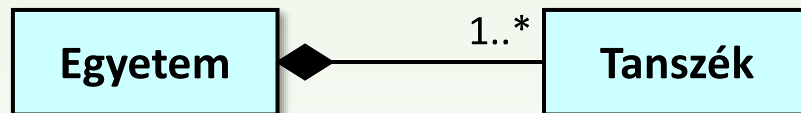


Egy könyv adott számú lapból áll, és a lapjai mindig csak az adott könyvhöz tartozhatnak. (Ha egy lap kiesik, a könyv már nem használható jól, de egy-egy lap önmagában használható.)

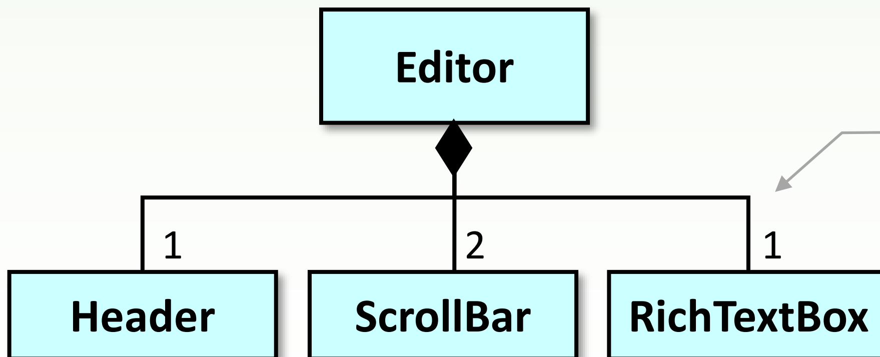
Példák: aggregáció vs. kompozíció



Tanterv ugyan nincs tantárgyak nélkül, de egy tantárgy akár több tanterv része is lehet, sőt nem kell tantervhez sem tartoznia.



Nincs egyetem tanszékek nélkül, és nincs tanszék egyetem nélkül. Egy tanszék nem kerülhet át másik egyetemre, és ha bezárják az egyetemet, akkor tanszékei megszűnnek.

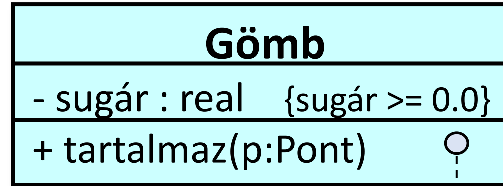


Egy szerkesztő ablak létrehozásakor létrejön annak fejléce, gördítősávjai, és szerkesztő területe is, amelyek megszűnnek az ablak megszűnésekor.

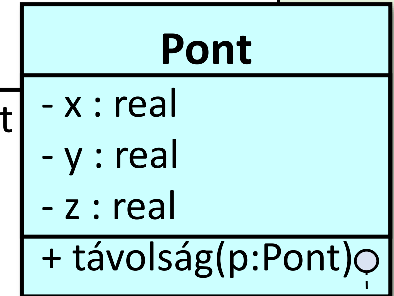
Példa: pont a gömbben (1.)

Egy gömbnek része a középpont, amely a gömbbel együtt születik és szűnik meg.

```
class Point {
private:
      double _x, _y, _z;
public:
      Point(double a, double b, double c) : _x(a),_y(b),_z(c){}
      double distance(const Point &p) const {
            return sqrt(pow(_x-p._x,2) + pow(_y-p._y,2) + pow(_z-p._z,2));
      }
};
```



- középpont



return középpont.távolság(p) ≦ sugár

return sqrt((x–p.x)²+(y–p.y)²+(z–p.z)²)

```
class Sphere{
private:
      Point _centre;
      double _radius;
public:
      enum Errors{ILLEGAL_RADIUS};
      Sphere(const Point &c, double r): _centre(c), _radius(r) {
            if (_radius<0.0) throw ILLEGAL_RADIUS;
      }
      double contains(const Point &p) const {
            return _centre.distance(p) <= _radius;
      }
};
```

tartalmazás

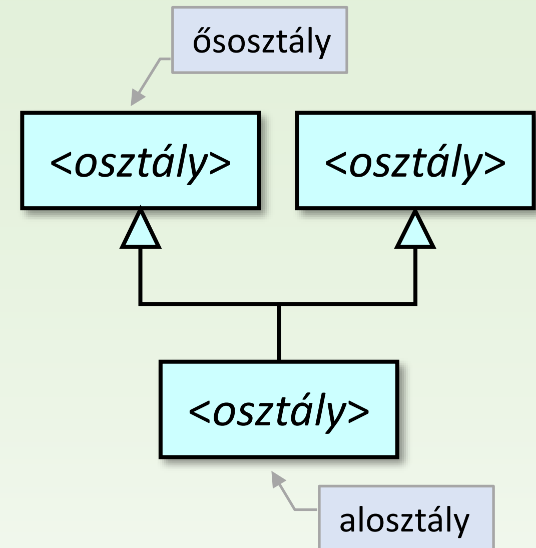
A konstruktor másolja le a c pontot az automatikusan létrejövő középpontnak. Ez önálló objektum lesz, amely a gömb megszűnésével együtt eltűnik.

```
Point  p(-12.0, 0.0, 23.0);
Point  c(-12.3, 0.0, 23.4);
Sphere g(c, 1.0);
cout << g.contains(p) << endl;
cout << c.distance(p) << endl;
```

Kapcsolatok létrehozásának helye

- Asszociáció esetén egy kapcsolatot a kapcsolat egyik objektuma hozza létre egy metódusával (ez gyakran az asszociáció nevét viselő metódus, de lehet a konstruktor). Ez a metódus paraméterként kapja meg a kapcsolat másik (többi) objektumának hivatkozását, vagy maga példányosítja azt (azokat), és gondoskodik a hivatkozásaik eltárolásáról.
- Aggregáció esetén a kapcsolat létrehozása a tartalmazó objektum felelőssége.
- Kompozíció esetén többnyire a tartalmazó objektum konstruktora építi ki a kapcsolatot a beágyazott objektummal.
  - Ha a beágyazott objektum lecserélhető, akkor azt a tartalmazó objektum egy erre alkalmas metódusa végzi.
  - Szigorúbb értelmezés mellett a konstruktor példányosítja a beágyazott objektumot, a destruktor törli.

Származtatás, öröklődés



- Ha egy objektum más objektumokra hasonlít, azokkal megegyező adattagjai és metódusai vannak, akkor az osztálya a vele hasonló objektumok osztályainak mintájára írható fel, azaz belőlük származtatható. Más szóval örököli azok tulajdonságait, amelyeket azonban módosíthat is, és ki is egészíthet.

- A modellezés során kétféle okból használunk származtatást:
  - Általánosítás: már meglévő, egymáshoz hasonló osztályoknak a közös tulajdonságait leíró ősosztályát (szuperosztály) hozzuk létre.
  - Specializálás: egy osztályból származtatással hozunk létre egy alosztályt.

5

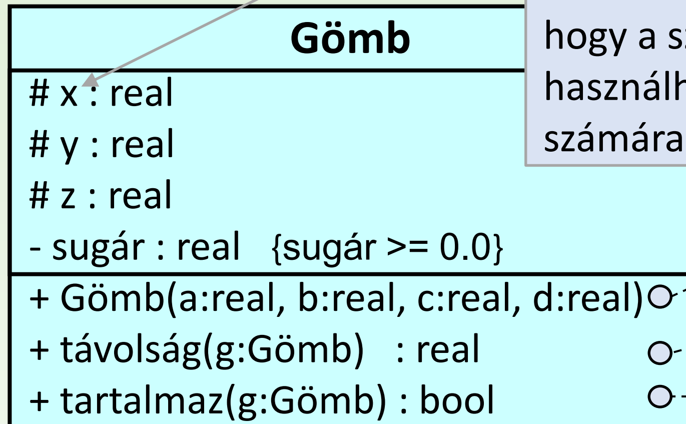
Az objektum-orientált nyelvek támogatják az öröklést: osztályok már meglévő osztályokból származtathatók.
Ősosztály változójának értékéül adható az alosztályának objektuma.

Származtatás és láthatóság

- Egy alosztályban hivatkozhatunk az ősosztályában definiált publikus és védett tagokra, de nem érjük el az ősosztály privát tagjait, azokhoz csak indirekt módon, az ősosztálytól örökölt metódusokkal férhetünk hozzá.
- A származtatás módja maga is lehet
  - publikus (*public*): ekkor az ősosztály publikus és védett tagjai az ősosztályban definiált láthatóságukkal együtt öröklődnek az alosztályra. (Az UML szerint ez a default, de a C++ nyelvben nem.)
  - védett (*protected*): ekkor az ősosztály publikus és védett tagjai mind védettek lesznek az alosztályban.
  - privát (*private*): ekkor az ősosztály publikus és védett tagjai privátok lesznek az alosztályban.

Példa: gömbből pont (2.)

Single responsibility
O
Liskov's substitution
I
D



A védett (protected) láthatóság lehetővé teszi, hogy a származtatott osztály objektumai használhassák ezeket az adattagokat, de mások számára továbbra is rejtve maradjanak.

x, y, z, sugár := a, b, c, d

return sqrt((x–g.x)²+(y–g.y)²+(z–g.z)²) – this.sugár – g.sugár

return távolság(g) + 2 · g.sugár ≦ 0

1. A Gömb metódusainak Gömb típusú paraméterváltozói pontok is lehetnek.
   a : Gömb, p : Pont;
   l := a.tartalmaz(p)

2. Sőt, a Gömb metódusai gömb helyett pontra is meghívhatók :
   p, q : Pont;
   d := p.távolság(q)

Pont
{sugár = 0.0}

+ Pont(a:real, b:real, c:real)

A Pont osztály nem látja a sugár tagot, de van neki, és el kell érnünk, hogy ez 0.0 értékű legyen. (típus invariáns)

Gömb(a, b, c, 0.0)

Az ősosztály konstruktora nem öröklődik, viszont a leszármazott osztály konstruktora meghívja.

C++ : gömbből pont (2.)

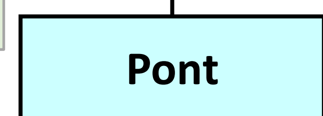
```
Point  p(0,0,0);
Sphere s(1,1,1,1);

cout << s.contains(p) << endl;
cout << s.distance(p) << endl;
cout << s.contains(s) << endl;
cout << s.distance(s) << endl;
cout << p.contains(s) << endl;
cout << p.distance(s) << endl;
cout << p.contains(p) << endl;
cout << p.distance(p) << endl;
```

```
class Sphere{
protected:
        double _x, _y, _z;
private:
        double _radius;
public:
        enum Errors{ILLEGAL_RADIUS};
        Sphere(double a, double b, double c, double d = 0.0):
                _x(a),_y(b),_z(c),_radius(d){ if (_radius<0.0) throw ILLEGAL_RADIUS; }
        double distance(const Sphere g) const
                { return sqrt(pow((_x-g._x),2) + pow((_y-g._y),2) + pow((_z-g._z),2))
                    -_radius - g._radius; }
        bool contains(const Sphere g)  const
                { return distance(g) + 2 * g._radius <= 0; }
};
```

publikus származtatás

Itt tehát használhatnánk az öröklött védett tagokat.

```
class Point : public Sphere {
public:
        Point(double a, double b, double c) : Sphere(a, b, c, 0.0) {}
};
```



Példa: pontból gömb (3.)

Hogyan döntse el a fordító program, hogy itt melyik sugár() metódus működését kell lefordítani? Fordítási időben még nem, csak futási időben dől el, hogy itt pontra vagy gömbre hivatkozik-e a p (illetve a this).

Pont

\# x : real
\# y : real
\# z : real

\+ Pont(real, real, real)
\+ távolság(p:Pont) : real
\+ sugár() : real { virtual }

return sqrt($(x-p.x)^2+(y-p.y)^2+(z-p.z)^2$) − sugár() − p.sugár()

return ~~sqrt($(x-p.x)^2+(y-p.y)^2+(z-p.z)^2$) − sugár − p.sugár~~

return ~~sqrt($(x-p.x)^2+(y-p.y)^2+(z-p.z)^2$)~~

Pontnak nincs sugár adattagja

return 0.0

Ha gömbre (vagy gömbbel) hívjuk meg, rossz eredményt ad: sérül a Liskov-féle elv.

Ha egy ősosztályban bevezetett és annak alosztályaiban felülírt metódus virtuális, akkor e metódus hívásakor ennek azon osztálybeli változata fut le, amelyik osztály példányára az a referencia- vagy pointer változó hivatkozik, amelyikre a metódust meghívják.

Gömb

\# sugár : real {sugár >= 0.0}

\+ Gömb(a:real, b:real, c:real, d:real)
\+ tartalmaz(p:Pont) : bool
\+ sugár() : real { override }

Pont(a, b, c); sugár := d

return ~~távolság(p) ≦ sugár~~

return távolság(p) + 2 · p.sugár() ≦ 0

return sugár

C++ : pontból gömb (3.)

```
Point  p(0,0,0);
Sphere g(1,1,1,1);

cout << p.distance(p) << endl;
cout << g.distance(p) << endl;
cout << p.distance(g) << endl;
cout << g.distance(g) << endl;
cout << g.contains(p) << endl;
cout << g.contains(g) << endl;
```

```
class Point {
protected:
    double _x, _y ,_z;
public:
    Point(double a, double b, double c) : _x(a), _y(b), _z(c) {}
    double distance(const Point &p) const
        { return sqrt(pow((_x-p._x),2)+pow((_y-p._y),2)+pow((_z-p._z),2))
            - radius() - p.radius();}
    virtual double radius() const { return 0.0; }
};
```

referencia változó

this->radius()

virtuális metódus

```
class Sphere : public Point {
private:
    double _radius;
public:
    enum Errors{ILLEGAL_RADIUS};
    Sphere(double a, double b, double c, double d) : Point(a,b,c), _radius(d)
        { if (_radius<0.0) throw ILLEGAL_RADIUS; }
    bool contains(const Point &p)const{return distance(p)+2*p.radius() <= 0;}
    double radius() const override { return _radius; }
};
```

felülírt metódus

Pont

Gömb

Dinamikus altípusos polimorfizmus

- Ha egy ősosztály metódusát a leszármazott osztályban felülírjuk (override), akkor ez a metódus több alakkal is rendelkezik (polimorf).
- Mivel egy ősosztály típusú változónak mindig értékül adható az alosztályának egy példánya, ezért csak futási időben derülhet ki, hogy ez a változó az ősosztály egy példányára vagy alosztályának egy példányára hivatkozik-e. (késői vagy futási idejű vagy dinamikus kötés).
- Ha egy referencia- vagy pointer változóra egy polimorf virtuális metódust hívunk meg, akkor e metódusnak azon osztálybeli változata fut majd le, amelyik osztálynak a példányára hivatkozik a referencia- vagy pointer változó hivatkozik (dinamikus altípusos polimorfizmus).

6 Az objektum-orientált nyelvek ismérve a dinamikus altípusos polimorfizmus.