Objektumok kapcsolatai

Gregorics Tibor

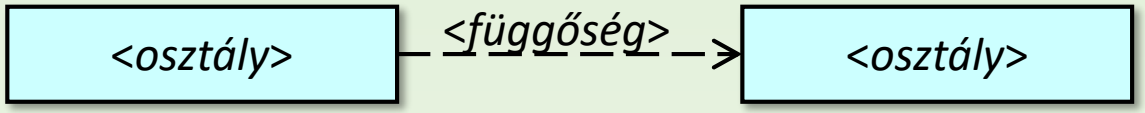
gt@inf.elte.hu

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

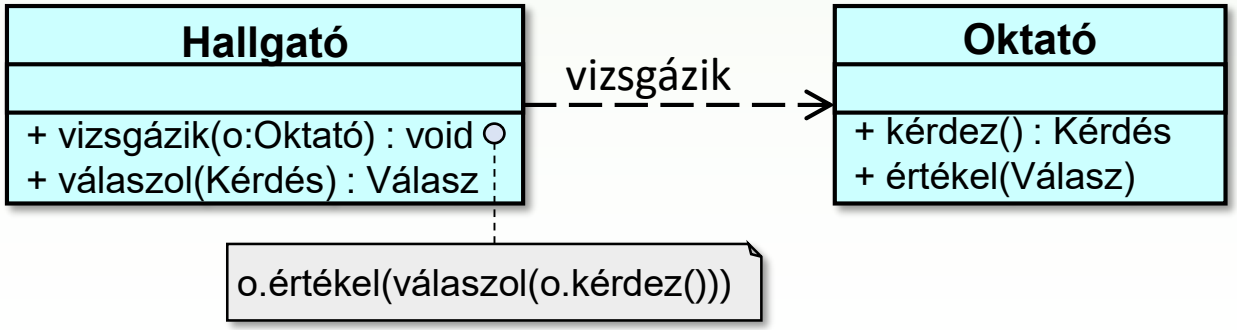
Objektum-kapcsolatok fajtái

- 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.
- Az objektumok közötti kapcsolatokat az osztályaik szintjén ábrázoljuk (hiszen az osztálydiagram az objektumdiagram absztrakciója).
- A kapcsolat fajtája lehet:
  - Függőség (*dependency*)
  - Asszociáció (*association*) vagy társítás
  - Aggregáció (*aggregation*, *shared aggregation*) vagy tartalmazás
  - Kompozíció (*composition*, *composite aggregation*) vagy szigorú tartalmazás
  - Származtatás vagy öröklődés (*inheritence*)

Függőség

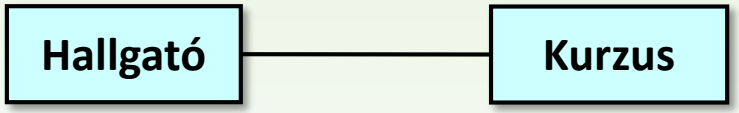


- Amikor egy metódus (ez lehet osztályszintű is) epizód szerűen kerül kapcsolatba egy másik osztály objektumával, amelyet paraméterként kap meg, vagy lokálisan hoz létre, azért hogy
  - annak egyik metódusát meghívja,
  - szignált (aszinkron üzenetet) küldjön neki,
  - továbbadja a hivatkozását (pl. kivételkezelésre szánt objektumot),
  - hivatkozzon az állapotára (pl. felsorolt típus egyik értékére).
- Amikor egy metódus (lehet osztályszintű is) egy másik osztály osztályszintű metódusát hívja.

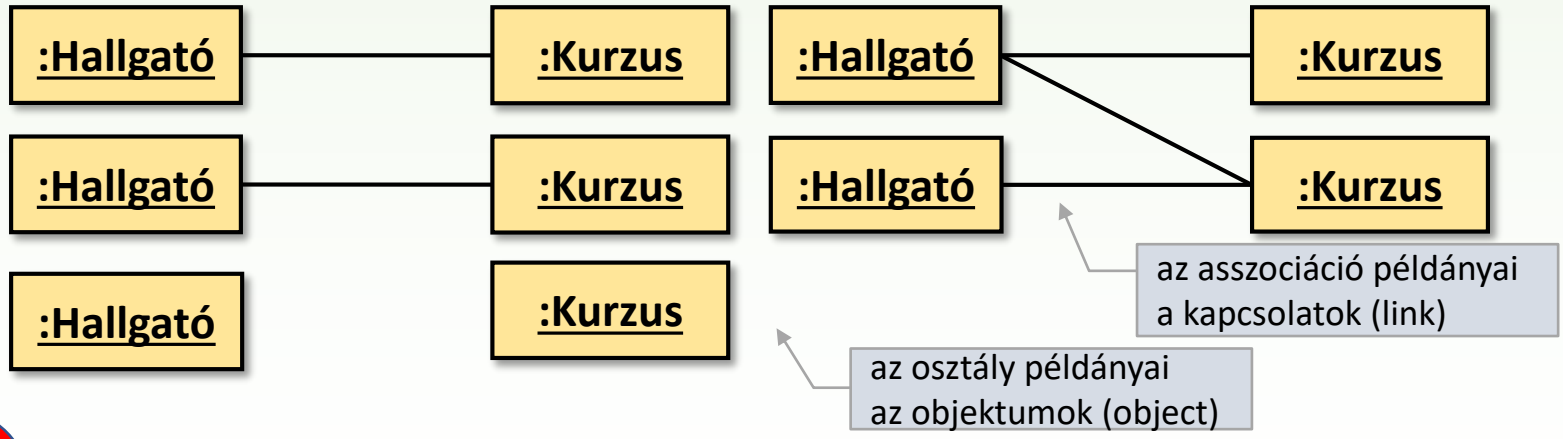


Asszociáció

- Az objektumok között hosszabb időszakon keresztül fennálló kapcsolat. (Az objektumok között állandósult függőségi kapcsolat.)
  - Egy asszociáció több objektum-kapcsolatot ír le
  - Egy objektum egy asszociáció több kapcsolatában is szerepelhet.
  - Egy objektum több asszociációs kapcsolatban is megjelenhet.

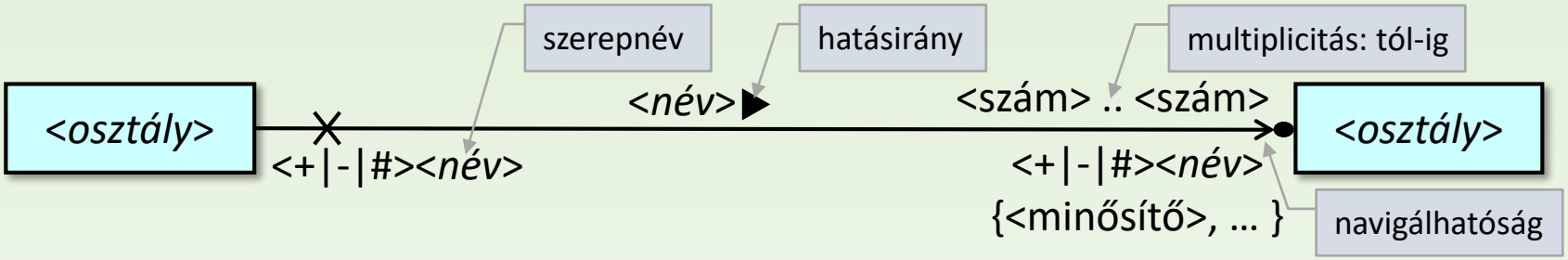


Az osztálydiagramm egy lehetséges példányosítása (felpopulálása):



4 Az objektum-orientáltság ismérve a fogalmi szintű absztrakció.

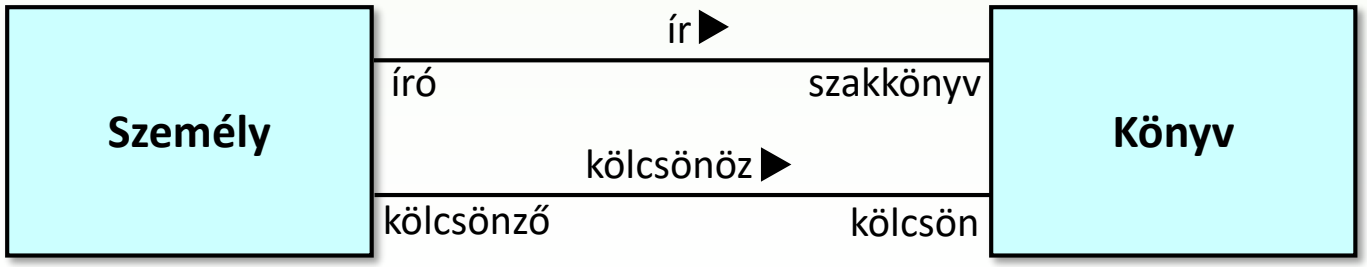
Asszociáció tulajdonságai



- Az asszociáció tulajdonságai az általa leírt kapcsolatokat jellemzik:
  - név: a kapcsolatok közös megnevezése
  - hatásirány: kapcsolódó objektumok egymáshoz való viszonya
  - 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 szerepnevei,
  - asszociációvégek neveinek láthatósága,
  - asszociációvégek neveinek tulajdonosa,
  - egy objektumhoz kapcsolódó több objektum gyűjteményének minősítése.

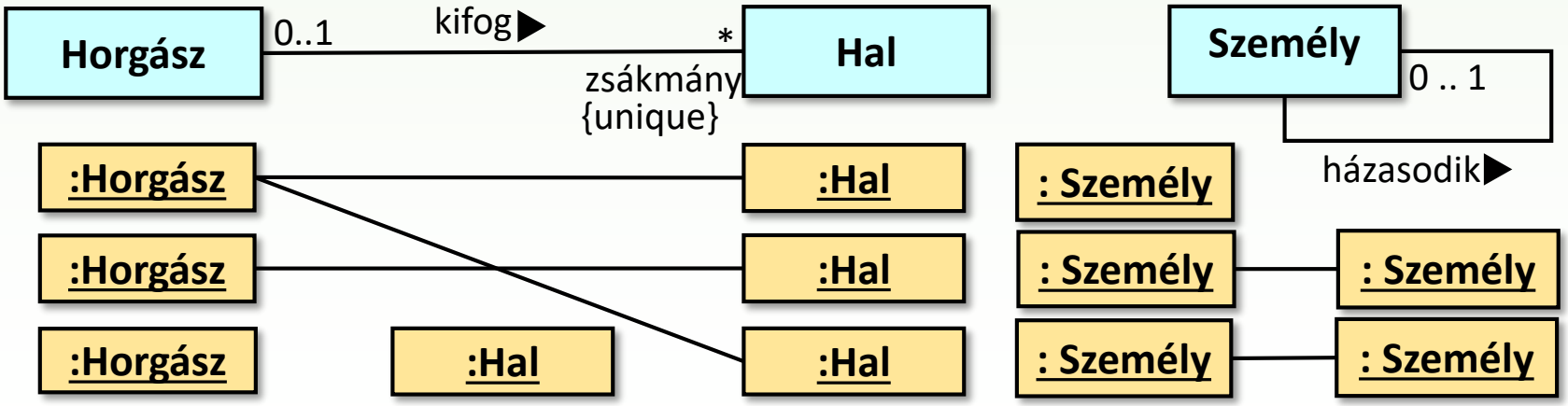
Nevek és hatásirány

- Az asszociációkat gyakran egy egyszerű bővített mondattal írhatjuk le, amelynek állítmánya (néha a tárgya) lesz az asszociáció neve, a mondat többi (nem állítmány, nem jelző) eleme pedig az asszociációvégek nevei az ún. szerepnevek lesznek.
- Egy kapcsolatban az objektumokra ezekkel a szerepnevekkel tudunk majd hivatkozni.
- 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. Sokszor az alany osztályában megjelenik asszociáció nevével megegyező nevű metódus is.



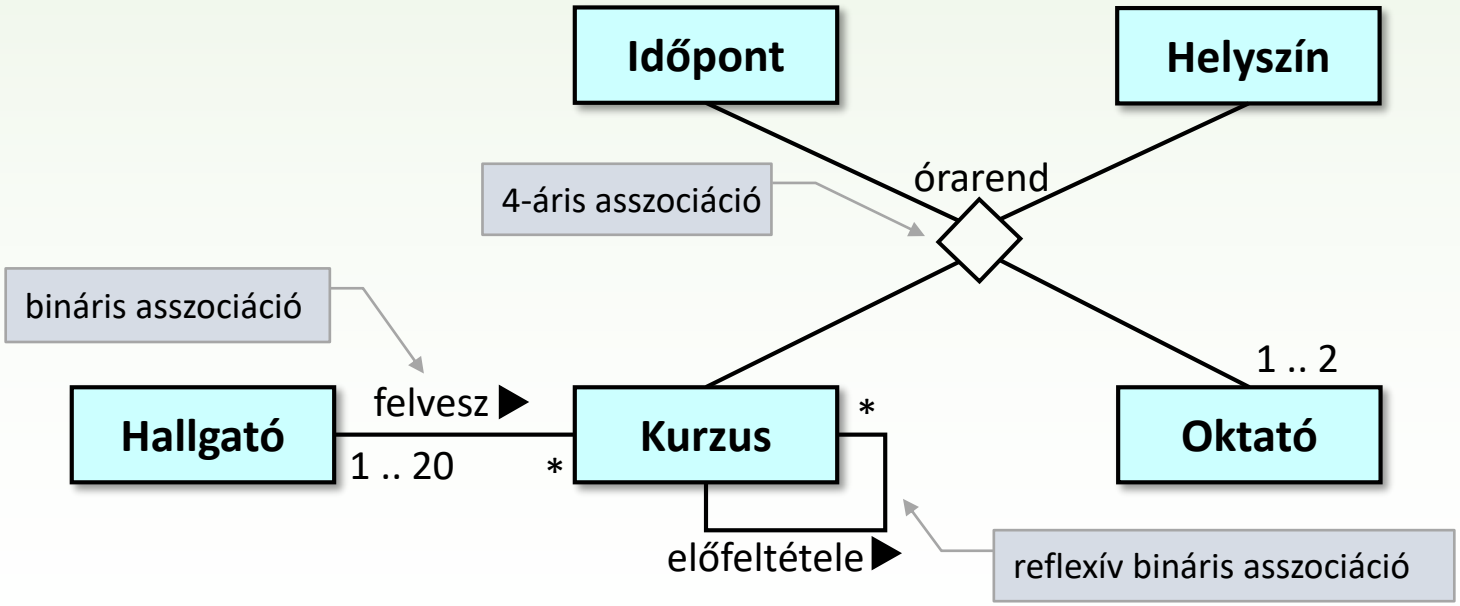
Multiplicitás

- Az asszociáció multiplicitása azt mutatja, hogy az asszociációnak a multiplicitással ellátott végén levő osztálynak hány (min .. max) objektuma létesíthet egyszerre kapcsolatot az asszociáció másik (többi) osztályának egy objektumával.
  - az 1 multiplicitás jelölését gyakran elhagyjuk
  - a 0 .. * helyett, ahol * tetszőleges természetes szám, a * jelölést használjuk
- Előírhatjuk egy „sok” multiplicitású asszociációnál, hogy egy objektumhoz kapcsolt „sok oldali” objektumok
  - mind különbözzenek egymástól {unique},
  - megadott sorrendben legyenek felsorolhatók {ordered}.



Aritás

- Az asszociáció aritása arra utal, hogy az asszociáció egy kapcsolata hány objektumot köt össze.
- Eddig csak bináris asszociációkra láttunk példákat, ahol a kapcsolat két objektum között jött létre. (Reflexív asszociáció is bináris: ugyanazon osztály két objektuma közötti kapcsolatot ír le.)



Navigálhatóság

- A navigálhatóság azt jelzi, hogy egy kapcsolatban melyik objektumot kell hatékonyan elérni a többi (másik) objektumból.
  - A hatékony navigálási irányt az asszociáció megfelelő végén elhelyezett nyíl jelöli.
  - A nyíl helyett a kereszt a navigálás nem támogatott irányát mutatja.
  - A jelöletlen asszociációvég a nem-definiált navigálhatóságra utal.
- A navigálhatóság iránya és a hatásirány különböző fogalmak, ezért irányításuk különbözhet.

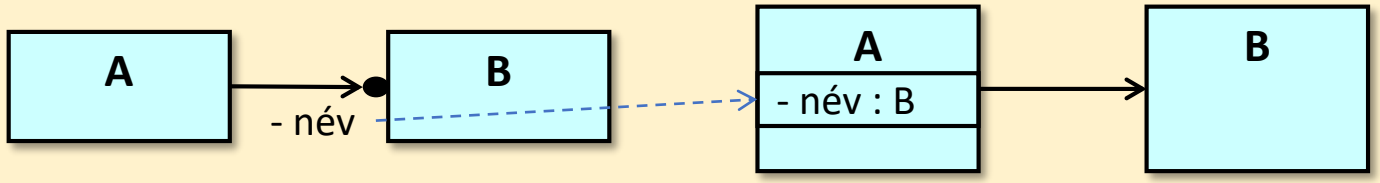

```
class Person {
private:
      IdentityCard *_id;
public:
      void makeIdentityCard() { … ; _id = new IdentityCard (…); }
      IdentityCard* showIdentityCard() const { return _id; }
};
```

felépíti a kapcsolatot

megadja a kapcsolódó igazolvány objektum elérését

Szerepnév tulajdonosa

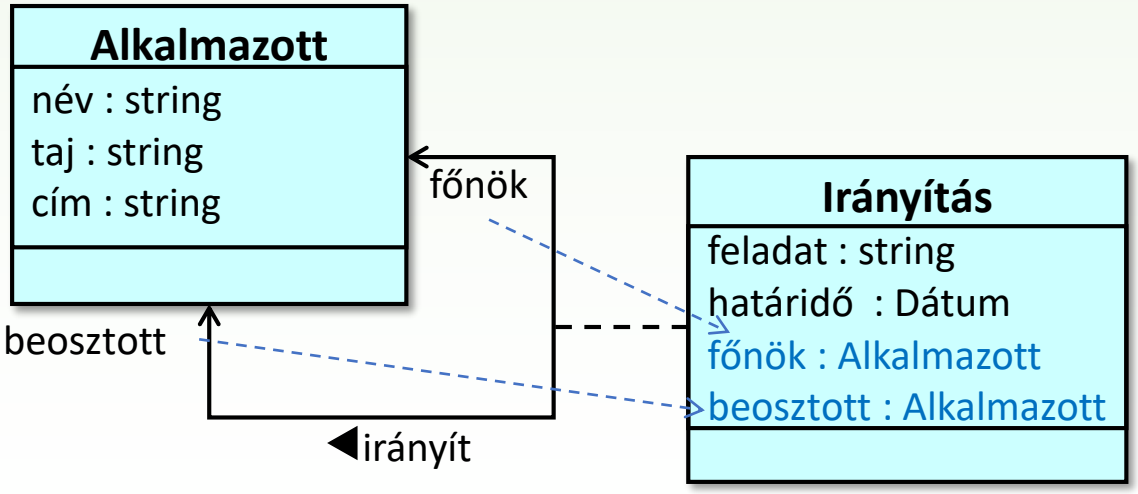
- A navigálás érdekében minden kapcsolatban el kell tárolni a hatékonyan elérendő objektum(ok) hivatkozását (szerepnevét).
  Ki legyen a szerepnév tulajdonosa?
  - Maga a kapcsolat is tárolhatja az általa összekapcsolt objektumok közül a hatékonyan elérendő hivatkozását (szerepnevét).
  - Egy hatékonyan elérendő objektum hivatkozását (szerepnevét) a kapcsolatban levő másik (többi) objektum is tárolhatja. Erre az osztálydiagrammban a szerepnévnél feltüntetett *fekete pötty* utal.



- A szerepnév láthatósága (private, protected, public) mutatja, hogy a név publikus, vagy kizárólag csak a tulajdonosa láthatja.

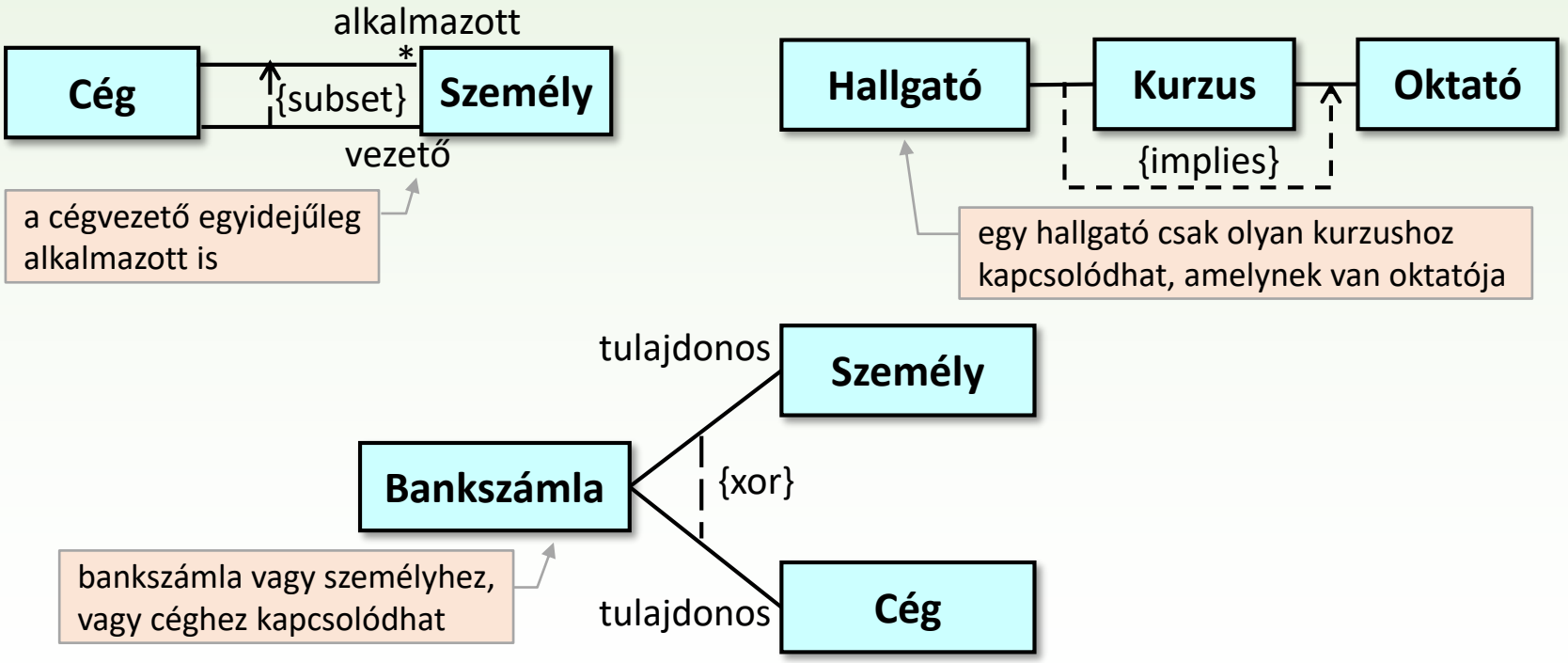
Asszociációs osztály

- Az UML lehetőséget ad egy asszociációhoz tartozó kapcsolatok tulajdonságait leíró osztály definiálására Ennek példányai a kapcsolatok, amelyekhez az általuk összekapcsolt objektumok hozzáférnek, és így elérik az abban tárolt információt.
- Amikor egy szerepnévnek maga az asszociáció a tulajdonosa, akkor a szerepnév az asszociációt leíró asszociációs osztálynak az adattagja.
- Ezt a fogalmi absztrakciót az ismertebb OO nyelvek nem támogatják.

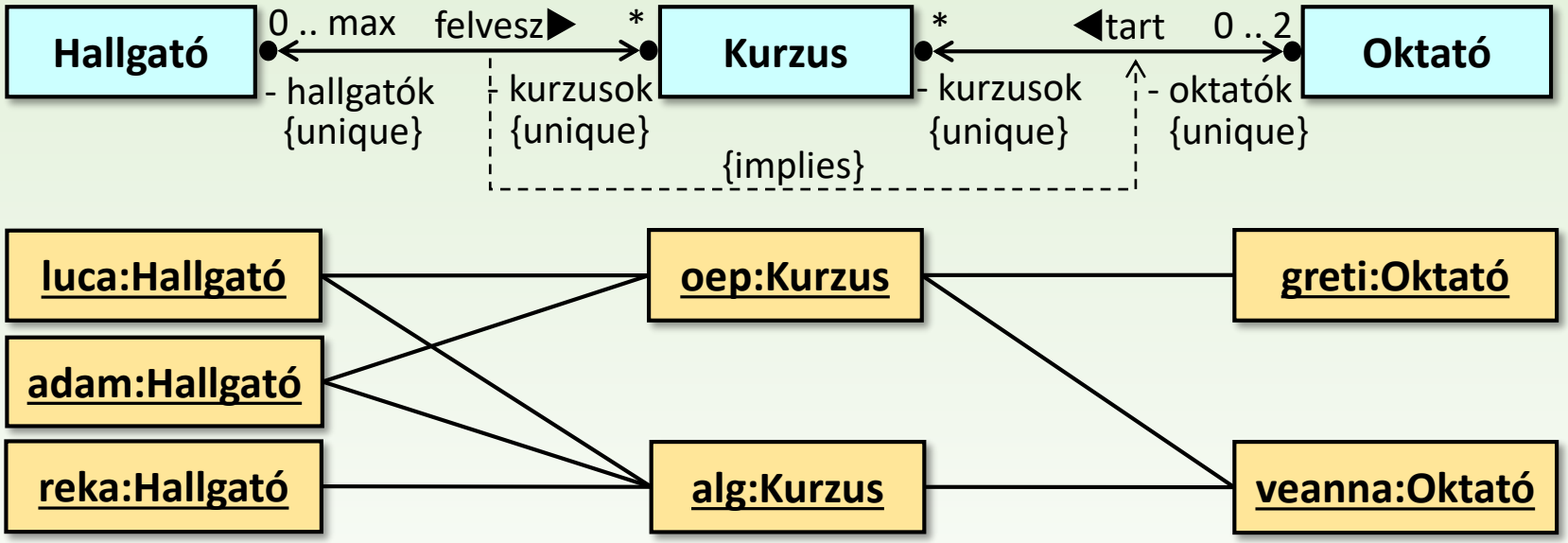


Asszociációk közötti feltételek

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



Példa




```
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);
```

Példa osztályai


```
class Course {
private:
        std::vector<Teacher*> _teachers; // 0 .. 2
        std::vector<Student*> _students; // 0 .. max
public:
};
```

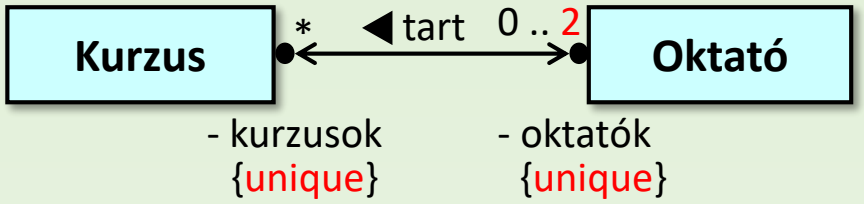
A „felvesz” asszociáció a Kurzus osztálynál többszörös multiplicitást jelez, ami egy gyűjteményre utal. Erre a gyűjteményre a megadott szerepnévvel az ellenkező oldali (Hallgató) objektum adattagja hivatkozik.

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

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

A „tart” asszociáció neve egyben egy metódus neve is abban az osztályban (Oktató) ahonnan a hatásiránya indul. A paramétere a hatásirány által mutatott osztálynak (Kurzus) objektuma.

Példa metódusai 1.



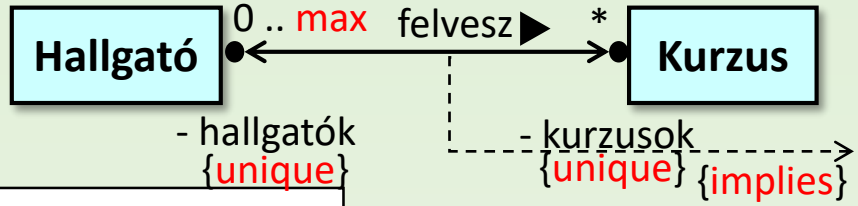

```
class Course {
private:
	std::vector<Teacher*> _teachers; // 0 .. 2
	std::vector<Student*> _students; // 0 .. max
public:
	bool can_lead(Teacher *pt) {
		bool l = false;
		for (Teacher *p : _teachers) {
			if ((l=p==pt)) break;
		}
		if (!l && _teachers.size()<2) {
			_teachers.push_back(pt);
		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

Ez a lineáris keresés azt adja meg, hogy van-e kapcsolat az adott kurzus és az adott oktató között. Ha nincs, akkor a „tart” kapcsolat létesítése mindkét unique feltételt kielégíti.

Itt vizsgáljuk a multiplicitás felső korlátját.

```
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) {
            bool l = false;
            for (Student *p : _students) {
                  if ((l = p==ps)) break;
            }
            if ( !l && _students.size()<max ){
                  _students.push_back(ps);
                  return true;
            }
            else return false;
      }
};
```

ez a lineáris keresés mindkét unique feltételt ellenőrzi

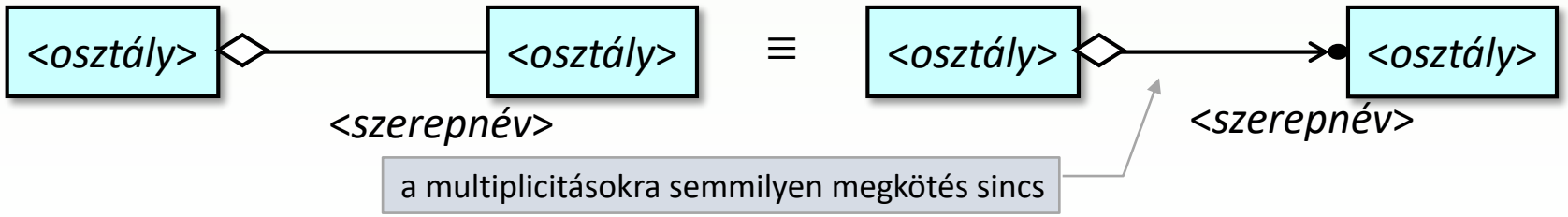
ez vizsgálja az implies feltételt

ez vizsgálja a multiplicitás felső korlátját

```
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);
      }
};
```

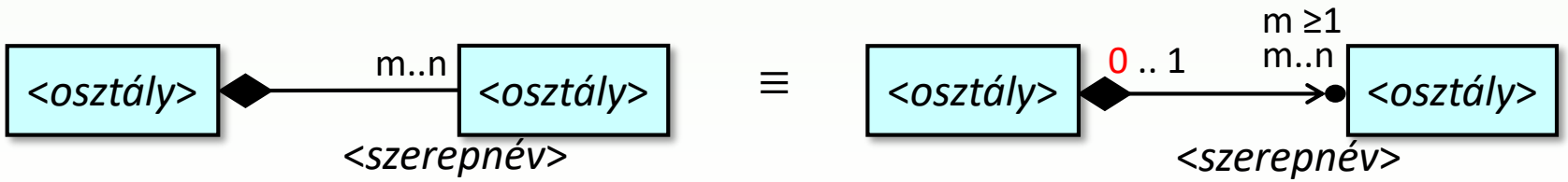
Aggregáció

- Egész-rész kapcsolatot kifejező bináris asszociáció, amely azt írja le, hogy egy objektumnak része, tulajdona egy másik:
  - Ez egy aszimmetrikus, tranzitív, nem reflexív reláció, amely nem alkothat irányított kört (azaz egy objektum még közvetett módon sem lehet önmaga része, tulajdona).
  - A tartalmazó objektumnak nem kell feltétlenül rendelkeznie tartalmazott objektummal, és a tartalmazott objektum egyidejűleg akár több objektumnak is lehet része.
- Megállapodunk továbbá abban, hogy
  - a kapcsolat a tartalmazott osztály irányába navigálható,
  - a tartalmazott osztály szerepneve a tartalmazó osztály tulajdona.

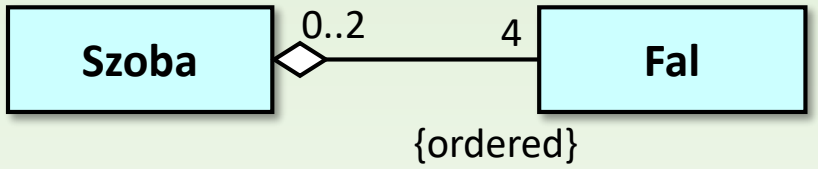


Kompozíció

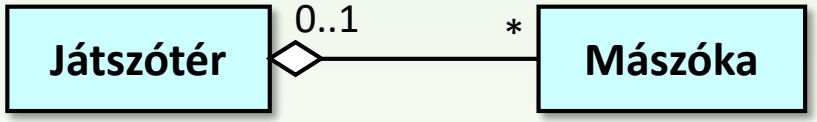
- Speciális aggregáció (aszimmetrikus, tranzitív, nem reflexív reláció, és nem alkothat irányított kört), ahol
  - a tartalmazó objektum nem élhet tartalmazott objektum nélkül,
  - a tartalmazott objektum kizárólag egy objektum része lehet.
- A kompozíció különböző értelmezései további, egyre szigorodó megkötéseket írhatnak elő a tartalmazott objektumra:
  - van tartalmazó objektuma: önmagában nem létezhet
  - tartalmazó objektuma nem változik: létrehozása és megszüntetése a tartalmazó objektum feladata
  - élettartama azonos a tartalmazó objektuméval: annak konstruktora példányosítja, destruktora törli.



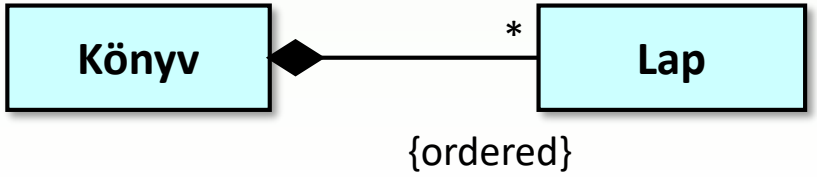
Példák



A falak a szobák részei, de ugyanaz a fal egyszerre két szobához is tartozhat.
Egy fal magában is állhat.

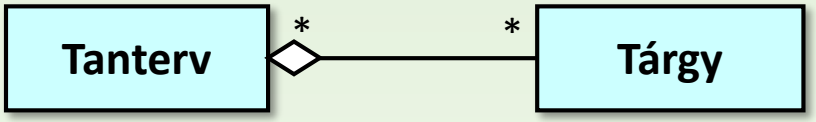


A mászókák a játszótér részei, és bár egyszerre csak egy játszótéré, de mászóka nélkül a játszótér még funkcionálhat. Egy mászókát át lehet vinni másik játszótérre, sőt önmagában is használható.

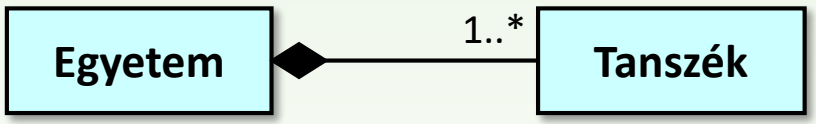


Egy könyv adott számú lapból áll, és a lapjai mindig csak az adott könyvhöz tartozhatnak. Ha egy lap kiesik, a könyv már nem használható jól.

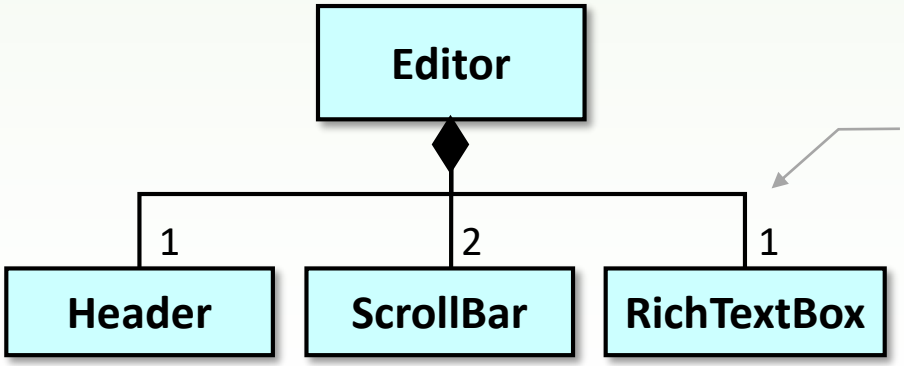
Példák



Egy tantárgy akár több tanterv része is lehet, de 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.

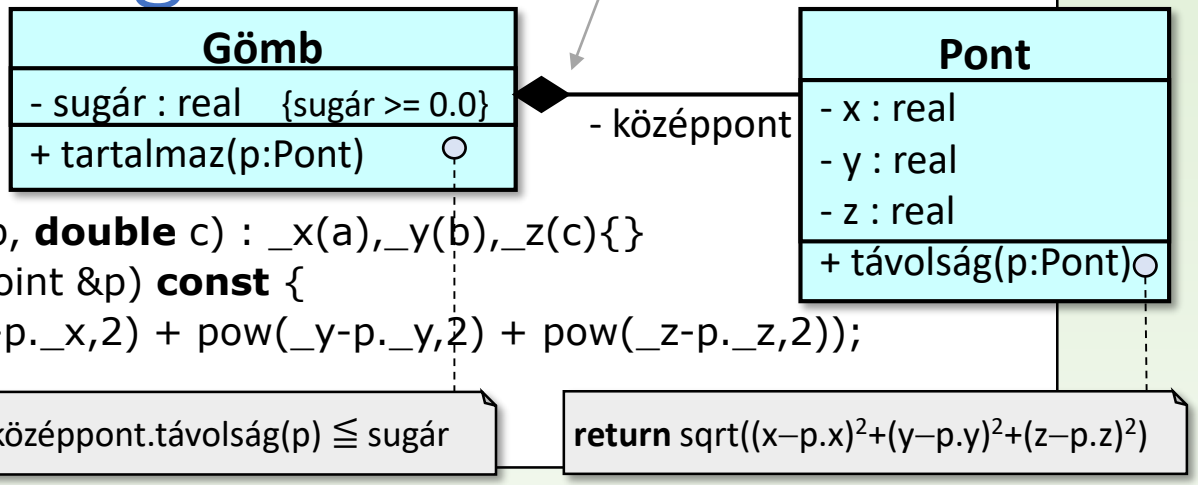


Egy szerkesztő ablak létrehozásakor létrejön annak fejléce, gördítősávjai, és szerkesztő területe is, amelyek megszűnnek az ablak megszűnésekor.

Példa: pont a gömbben

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));
        }
};
```




```
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;
        }
};
```

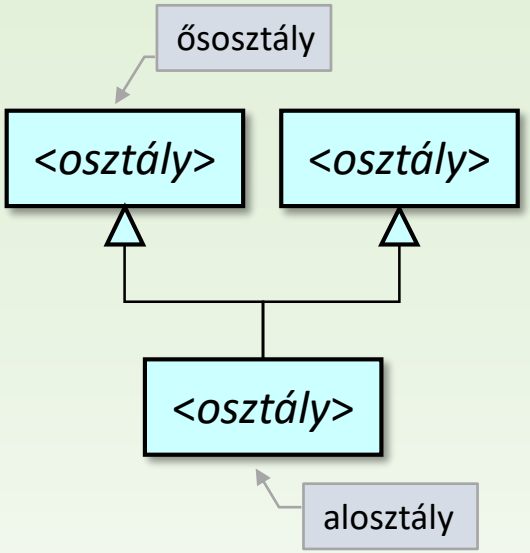
tartalmazás

A konstruktor másolja le a c pontot az automatikusan létrejövő középpontnak, amely ezután külön objektum, és 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;
```

Származtatás, öröklődés

- Ha egy objektum más objektumokra hasonlít, azokkal megegyező adattagjai és metódusai vannak), akkor osztálya a vele hasonló objektumok osztályainak mintájára adható meg, azaz belőlük származtatható, ami által örökli azok tulajdonságait, amelyeket módosíthat, és ki is egészíthet.



- A modellezés során kétféle okból végezhetünk 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-orientáltság legtöbbet emlegetett ismérve az öröklés: osztályok származtathatóak már meglévő osztályokból.
Ősosztály változójának értékül adható az alosztályának objektuma.

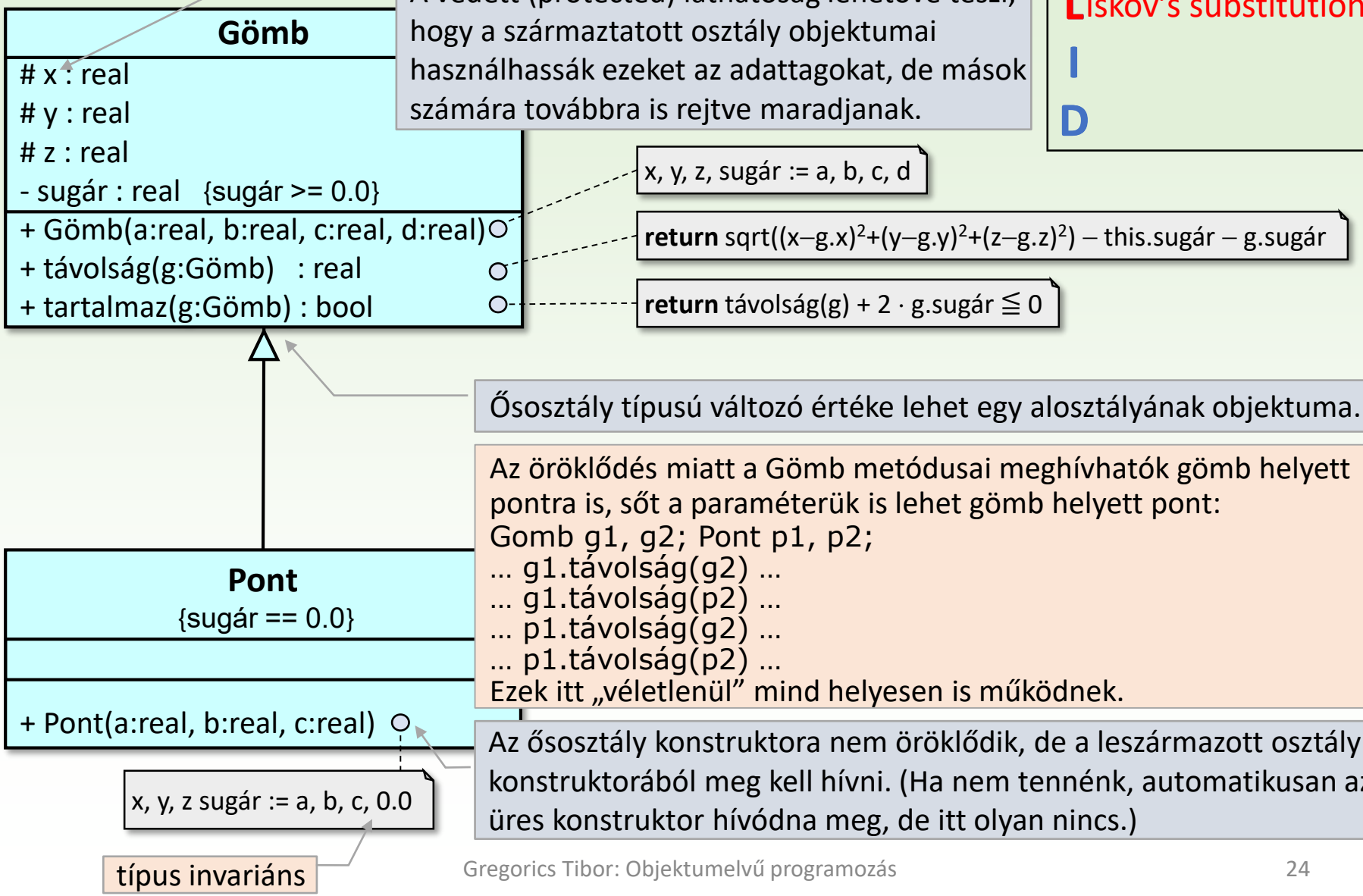
Származtatás és láthatóság

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

Példa: gömbből pont

Single responsibility
O
Liskov's substitution
I
D

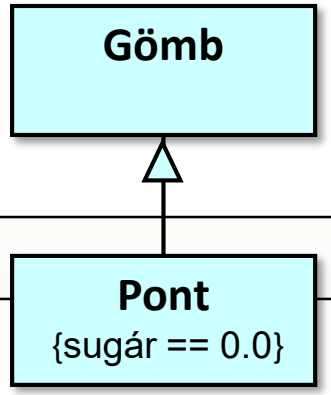
A védett (protected) láthatóság lehetővé teszi, hogy a származtatott osztály objektumai használhassák ezeket az adattagokat, de mások számára továbbra is rejtve maradjanak.



C++ : gömbből pont

```
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;
```

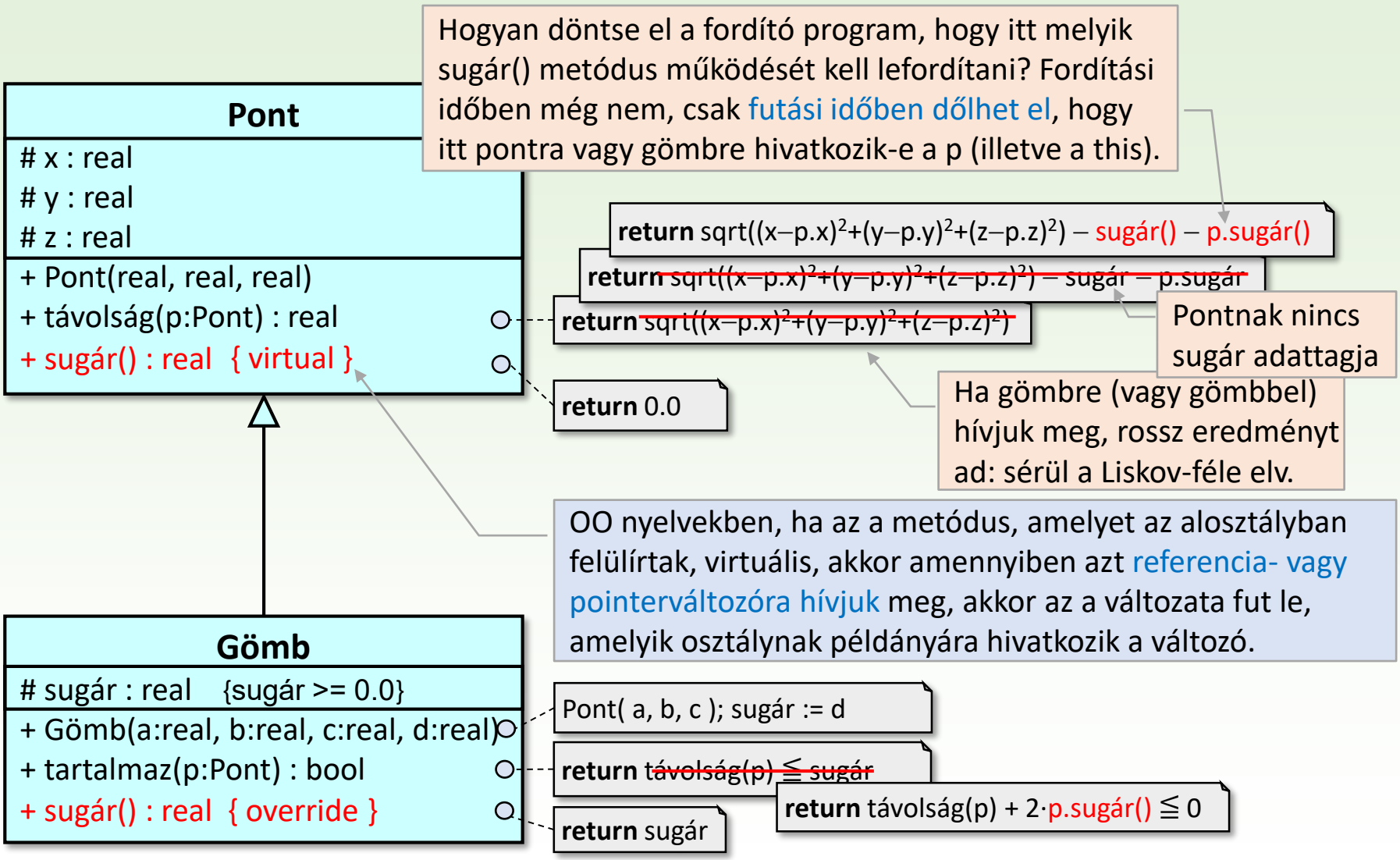
```
class Sphere{
protected:
        double _x, _y, _z;
private:
        double _radius;
public:
        enum Errors{ILLEGAL_RADIUS};
        Sphere(double a, double b, double c, double d = 0.0):
                _x(a),_y(b),_z(c),_radius(d){ if (_radius<0.0) throw ILLEGAL_RADIUS; }
        double distance(const 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

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

Példa: pontból gömb



C++ : pontból gömb

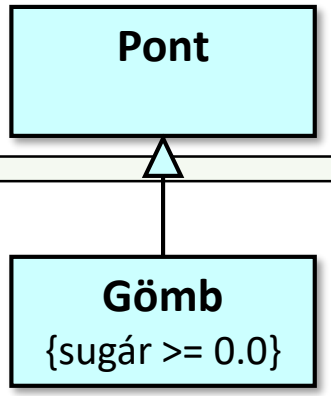
```
Point  p(0,0,0);
Sphere g(1,1,1,1);

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

Polimorfizmus és dinamikus kötés

- Ha egy ősosztály metódusát a leszármazott osztályban felülírjuk (override), akkor ugyanaz a metódus több alakkal fog rendelkezni (polimorf).
- Mivel egy ősosztály típusú változónak mindig értékül adható az alosztályának egy példánya, ezért csak futási időben derülhet ki, hogy a változó az ősosztály egy példányára vagy alosztályának egy példányára hivatkozik. (késői vagy dinamikus kötés).
- Ha egy objektumra hivatkozó referencia vagy pointer változóra meghívjuk az ősosztály egy polimorf virtuális metódusát, akkor az, hogy a metódus melyik változata fut majd le, attól függ, hogy a változó éppen melyik osztály objektumára hivatkozik (futási idejű polimorfizmus).

6

Az objektum-orientált nyelvek ismérve a futási idejű polimorfizmus és az ezzel párban járó dinamikus kötés.