Eötvös Loránd Tudományegyetem
Informatikai Kar

Alkalmazott modul: Programozás

10. előadás

Objektumorientált programozás: tervezés és megvalósítás

Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Objektumorientált programozás

Kialakulása

- A procedurális programozási paradigma összetett alkalmazások esetén számos korlátozást tartalmaz:
  - a program nem tagolható kellő mértékben
  - az adatok élettartama nem eléggé testre szabható
  - a feladat módosítása utóhatásokkal rendelkezhet
- Ezek megoldása a *felelősség továbbadása*
  - *programegység*eket alakítunk ki, amely rendelkeznek saját adataikkal és műveleteikkel, ezeket egységbe zárjuk, megvalósításukat elrejtjük
  - a feladat megoldását a programegységek együttműködésével, *kommunikáció*jával valósítjuk meg

Objektumorientált programozás

Objektumok

- *Objektum*nak (*object*) nevezzük a feladat egy adott tárgyköréért felelős programegységet, amely tartalmazza a tárgykör megvalósításához szükséges adatokat, valamint műveleteket
  - az objektum működése során saját adatait manipulálja, műveleteit futtatja és kommunikál a többi objektummal
  - pl.: egy téglalap
    - adatai: szélessége és magassága
    - műveletei: területkiszámítás, méretváltoztatás
  - pl.: egy verem adatszerkezet
    - adatai: elemek tartalmazó tömb és a felhasznált méret
    - műveletei: push, pop, top

Objektumorientált programozás

Objektumok állapotai

- Az objektumok *állapot*tal (state) rendelkeznek, ahol az állapot mezőértékeinek összessége
  - két objektum állapota ugyanaz, ha értékeik megegyeznek (ettől függetlenül az objektumok különbözőek)
  - az állapot valamilyen *esemény* (műveletvégzés, kommunikáció) hatására változhat meg
- A program teljes állapotát a benne lévő objektumok összállapota adja meg
- Az objektumok *életciklus*sal rendelkeznek, létrejönnek (a konstruktorral), működést hajtanak végre (további műveletekkel), majd megsemmisülnek (a destruktorral)

Objektumorientált programozás

Objektum-orientált programok

- *Objektum-orientált*nak nevezzük azt a programot, amelyet egymással kommunikáló objektumok összessége alkot
  - minden adat egy objektumhoz tartozik, és minden művelet egy objektumhoz rendelt tevékenység, nincsenek globális adatok, vagy globális műveletek
  - a program így kellő tagoltságot kap az objektumok mentén
  - az adatok élettartama így összekapcsolható az objektum élettartamával
  - a módosítások általában az objektum belsejében véghezvihetők, ami nem befolyásolja a többi objektumot, így nem szükséges jelentősen átalakítani a programot

Objektumorientált programozás

Objektum-orientált programok

- Az objektum-orientáltság öt alaptényezője:
  - *absztrakció*: az objektum reprezentációs szintjének megválasztása
  - *enkapszuláció*: az adatok és alprogramok egységbe zárása, a belső működés elrejtése
  - *nyílt rekurzió*: az objektum mindig látja saját magát, eléri műveleteit és adatait
  - *öröklődés*: az objektum tulajdonságainak átruházása más objektumokra
  - *polimorfizmus és dinamikus kötés*: a műveletek futási időben történő működéshez kötése

Objektumorientált programozás

Osztályok

- Az objektumok viselkedési mintáját az *osztály* tartalmazza, az osztályból *példányosít*hatjuk az objektumokat
  - tehát az osztály az objektum típusa
- Az osztályban tárolt adatokat *attribútum*oknak, vagy *mező*knek (*field*), az általa elvégezhető műveleteket *metódus*oknak (*method*) nevezzük, együtt ezek az osztály *tag*jai (*member*)
- Az osztály tagjainak szabályozhatjuk a láthatóságát, a kívülről látható részét *interfész*nek, a kívülről nem látható részét *implementáció*nak nevezzük
  - a metódusok megvalósítása az implementáció része, tehát más osztályok számára a működés mindig ismeretlen

Objektumorientált programozás

Osztályok megvalósítása

- Az osztályokat minden nyelv más formában valósítja meg, de az általános jellemzőket megtartja
- A C++ programozási nyelv támogatja az objektumorientált programozást, noha alapvetően procedurális
- A C++ osztály szerkezete:

```
class/struct <osztálynév> {
public/private:
    <típus> <mezőnév>;
    …
    <típus> <metódusnév> ([ <paraméterek> ]) { … }
    …
};
```

Objektumorientált tervezés

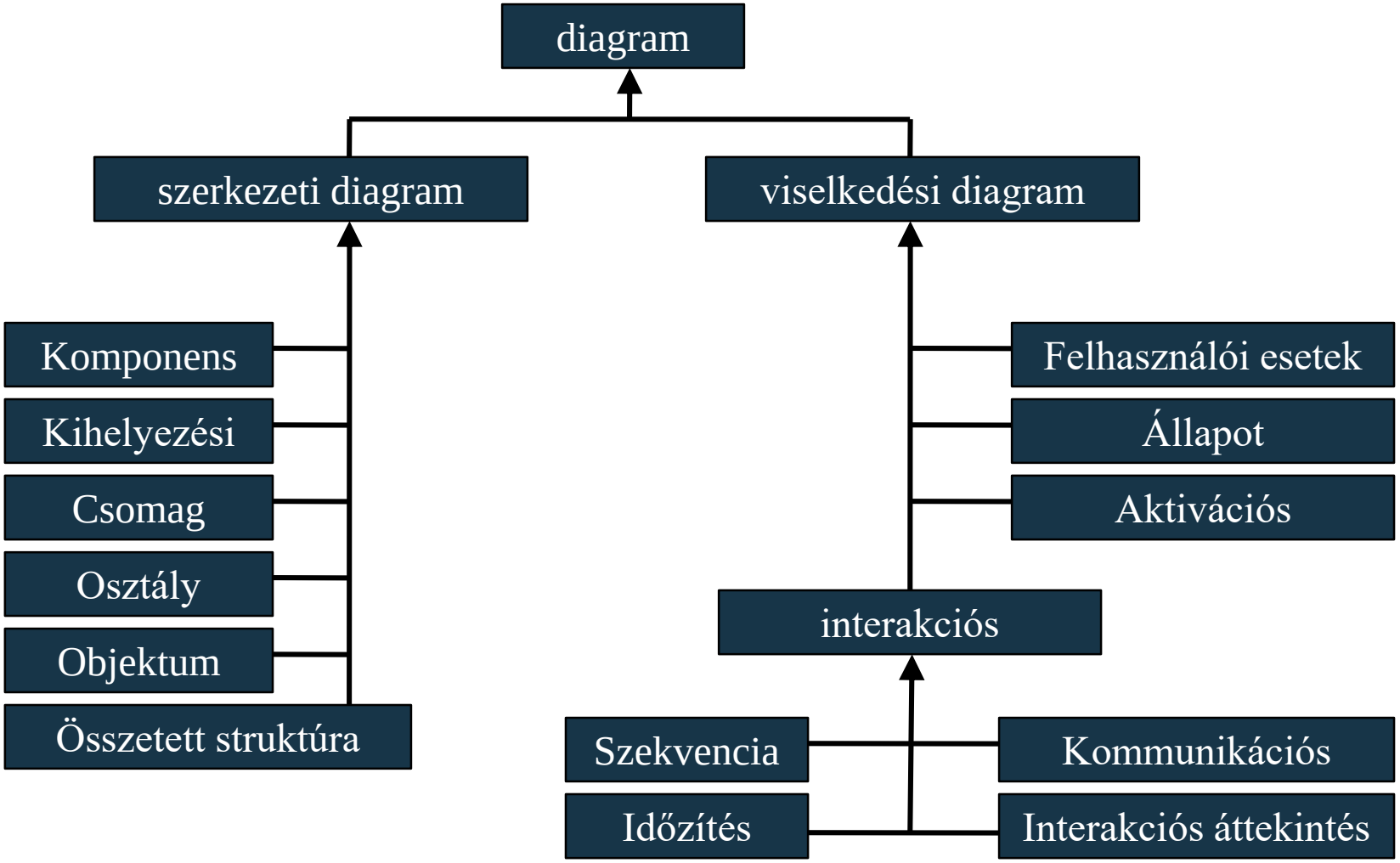
Szoftverek tervezése

- A program szerkezetének és működésének megtervezése az osztályok és objektumok szempontjából történik
  - a *statikus tervezés* során az objektumok, osztályok felépítését és kapcsolatait határozzuk meg
  - a *dinamikus tervezés* során az objektumok, osztályok közötti kommunikáció folyamatát, az állapotváltoztatásokat határozzuk meg
- Az objektumorientált tervezés eszköze a *Unified Modeling Language* (UML), amely egy grafikus programozási nyelv
  - 13 diagramtípust biztosít a program szerkezetének és működésének megtervezésére

- Az UML a szoftverrendszert a következő szempontok szerint tudja jellemezni:
  - *funkcionális modell*: a szoftver funkcionális követelményeit és a felhasználóval való interakciókat adja meg
    - pl.: felhasználói esetek diagramja, kihelyezési diagram
  - *szerkezeti modell*: a program felépítését adja meg, milyen osztályok, objektumok, relációk alkotják a programot
    - pl.: osztálydiagram, objektumdiagram
  - *dinamikus modell*: a program működésének lefolyását, az objektumok együttműködésének módját ábrázolja
    - pl.: állapotdiagram, szekvenciadiagram

Objektumorientált tervezés

Az UML nyelv



Objektumorientált tervezés

Az osztálydiagram

- Az *osztálydiagram* a programban szereplő osztályok szerkezetét, és a közöttük lévő kapcsolatokat definiálja
  - az osztálynak megadjuk a nevét, valamint mezőinek és metódusainak halmazát (típusokkal, paraméterekkel)
  - megadjuk a tagok láthatóságát látható (+), illetve rejtett (–) jelölésekkel

<osztálynév>
- *<mezőnév>* :*<típus>* ...
+ *<metódusnév>*(*<paraméterek>*) :*<típus>* ...

Objektumorientált tervezés

Példa

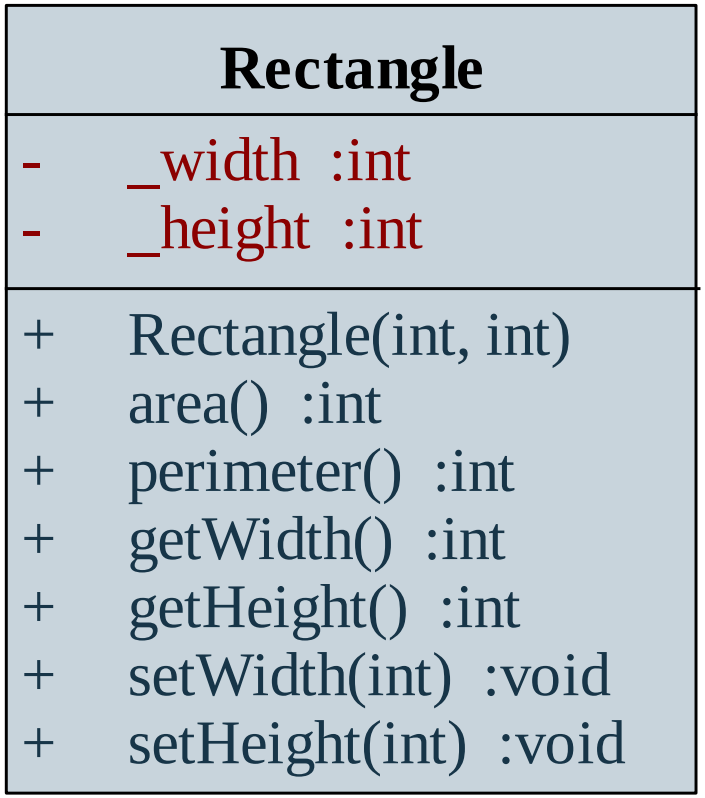
Feladat: Valósítsuk meg a téglalap (**`Rectangle`**) osztályt, amely utólag átméretezhető, és le lehet kérdezni a területét és kerületét.

- a téglalapnak a feladat alapján elég a méreteit letárolni (**`_height`**, **`_width`**), amelyek egész számok lesznek
- ezeket a későbbiekben lekérdezhetjük (**`getWidth()`**, **`getHeight()`**), vagy felülírhatjuk (**`setWidth(int)`**, **`setHeight(int)`**)
- lehetőséget adunk a terület, illetve került lekérdezésére (**`area()`**, **`perimeter()`**)
- lehetőséget adunk a téglalap létrehozására a méretek alapján (**`Rectangle(int, int)`**)

Objektumorientált tervezés

Példa

Tervezés:



Objektumorientált tervezés

Példa

Megoldás:

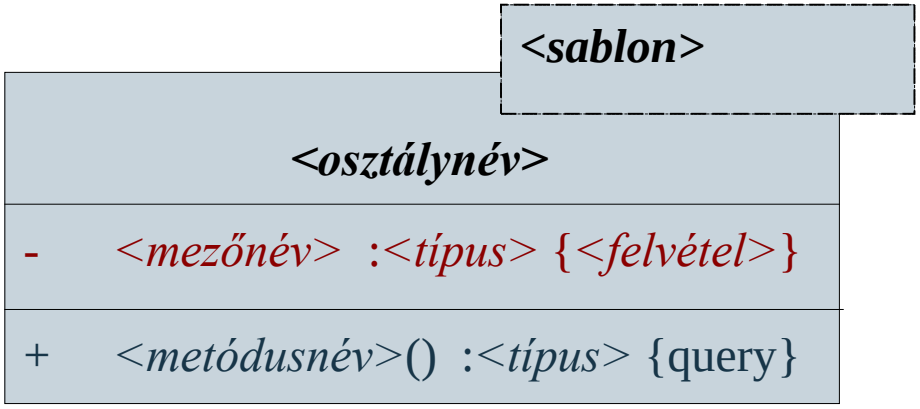
```
class Rectangle // téglalap osztálya
{
private:
    int _width; // szélesség
    int _height; // magasság

public: // látható rész
    Rectangle(int w, int h) { … }
        // 2 paraméteres konstruktor művelet
    int area(); // téglalap területe
    int perimeter(); // téglalap kerülete
    …
};
```

Objektumorientált tervezés

Az osztálydiagram

- Az osztálydiagram számos kiegészítést tartalmazhat
  - feltételeket szabhatunk a mezőkre, metódusokra a `{…}` jelzéssel (pl. a lekérdezést a `{query}` jelzéssel jelölünk)
  - adhatunk az osztálynak sablonparamétert
  - további tulajdonságokat jelölhetjük, illetve csoportba foglalásokat végezhetünk a **<<…>>** jelzéssel



Objektumorientált tervezés

Példa

Feladat: Valósítsuk meg a verem (**`Stack`**) osztályt, ahol tetszőleges számban helyezhetjük a verembe az elemeket.

- a verem műveletei a behelyezés (**`push`**), kivétel (**`pop`**), tetőelem lekérdezés (**`top`**) lesznek, továbbá segédműveletként megjelenik az üresség (**`isEmpty`**), a méret (**`size`**) lekérdezése, valamint a kiürítés (**`clear`**)
- annak érdekében, hogy a verem bármilyen elemet fogadhasson, sablonosan fogalmazzuk meg
- a tetszőleges bővíthetőség érdekében dinamikus tömbbel valósítjuk meg (**`_values`**), amely automatikusan méreteződik (**`resize`** segédművelettel) elem hozzáadásakor, illetve törlésekor

Objektumorientált tervezés

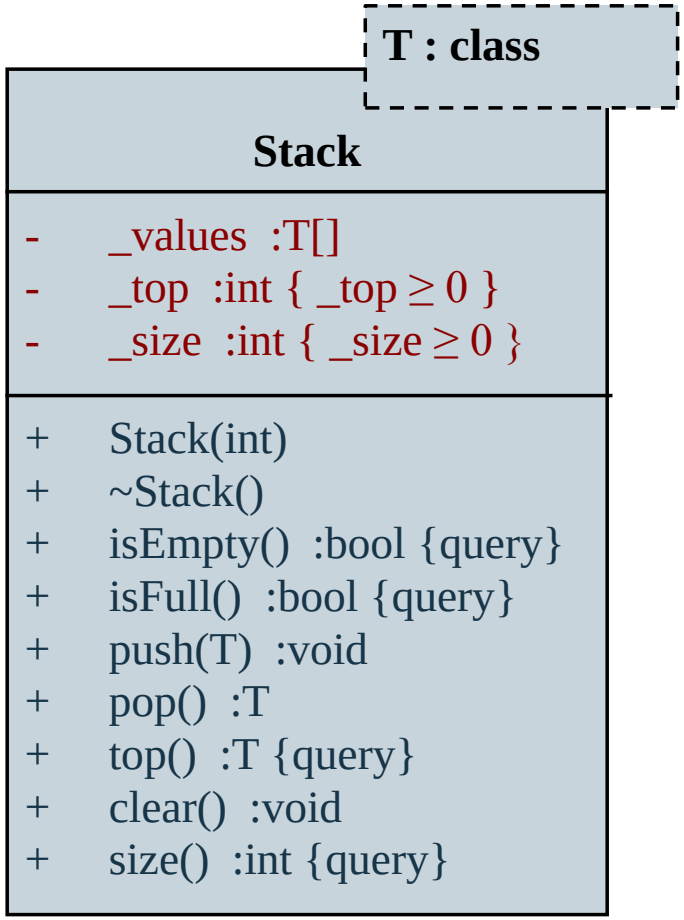
Példa

- eltároljuk a teljes és a felhasznált méretet (`_size`, `_top`), amely nem lehet negatív
- a dinamikus reprezentáció miatt megvalósítjuk a konstruktor mellett a destruktort, másoló konstruktort, valamint értékadás operátort
- a mezőket, valamint a átméretező segédműveletet elrejtjük
- a lekérdező műveleteket (`top`, `isEmpty`, `size`) konstanssá alakítjuk, így konstans példányra is meghívhatóak lesznek

Objektumorientált tervezés

Példa

Tervezés:



Objektumorientált tervezés

Példa

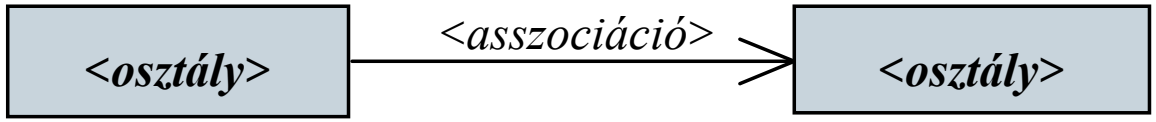
Megoldás:

```
template <class T>
class Stack { // verem osztály
private:
    T* _values;
    int _top;
    int _size;
    void resize(int newSize);
public:
    Stack();
    Stack(const Stack<T>& other);
    ~Stack();
    bool isEmpty() const;
    …
```

Objektumorientált tervezés

Objektumok közötti kapcsolatok

- Az alkalmazásokban rendszerint több osztály szerepel, amelyek objektumai kommunikálhatnak egymással, ezért az osztályok között relációkat állíthatunk fel
- az *asszociáció* (*association*) egy kommunikációs kapcsolat, ahol egy objektum üzenetet küldhet egy (vagy több) további objektumnak



  - a kommunikáció lehet irányított, irányítatlan (kétirányú), reflexív (saját osztály másik objektumára)
  - az osztályoknak lehet *multiplicitása*, ami a relációbeli számosságukra utal

Objektumorientált tervezés

Példa

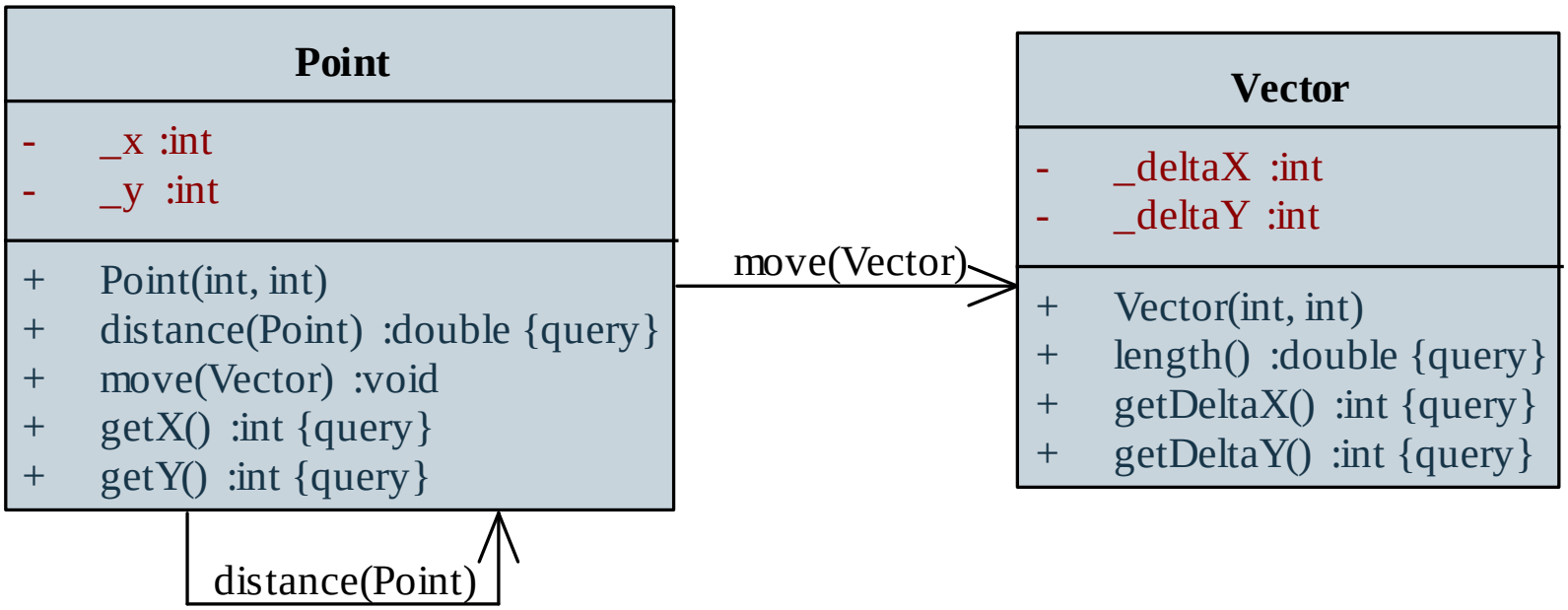
Feladat: Valósítsuk meg a 2D koordinátarendszerben ábrázolható pontokat (`Point`), valamint vektorokat (`Vector`). A pontok eltolhatóak vektorral, illetve megadható két pont távolsága. A vektornak ismert a hossza.

- a pont ábrázolható a vízszintes és függőleges értékkel (`_x`, `_y`), amelyek lekérdezhetőek (`getX()`, `getY()`) lekérdezhető a távolsága egy másik ponthoz képest (`distance(Point)`), valamint eltolható egy vektorral (`move(Vector)`)
- a vektor ábrázolható a vízszintes és függőleges eltéréssel (`_deltaX`, `_deltaY`), amelyek lekérdezhetőek (`getDeltaX()`, `getDeltaY()`), ahogy a hossza (`length()`) is

Objektumorientált tervezés

Példa

Tervezés:



Megoldás (`point.hpp`*):*

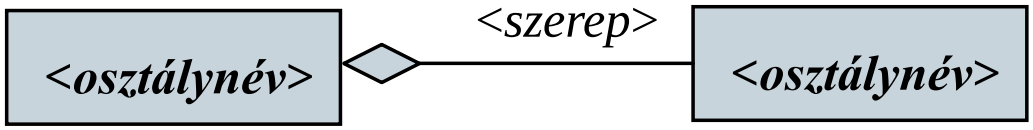
```
#include <cmath>
#include "vector.hpp"

class Point { // 2D pont osztály
public:
   Point(int x = 0, int y = 0) { _x = x; _y = y; }
   double distance(Point p) const { … }
   void move(Vector v) {
      _x += v.getDeltaX();
      _x += v.getDeltaY();
   }
   int getX() const { return _x; }
   …
```

Objektumorientált tervezés

Objektumok közötti kapcsolatok

- Az *aggregáció* (*aggregation*) egy speciális asszociáció, amely az objektumok laza egymáshoz rendelését fejezi ki

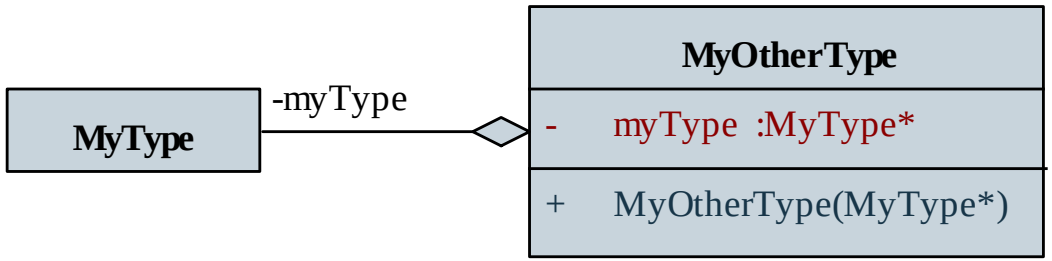


  - egy tartalmazási, rész/egész kapcsolatot jelent, amely állandó a két objektum között
  - a részt vevő objektumok életpályája különbözik, egymástól függetlenül is léteznek
  - az aggregált osztálynak lehet *szerepe*, ami a relációbeli minőségükre utal, ezt az adott végponton jelöljük (ez is megjelölhető láthatósággal)

Objektumorientált tervezés

Objektumok közötti kapcsolatok

- aggregációs kapcsolat megvalósításakor az aggregált osztály egy hivatkozását (mutatóját, vagy referenciáját) tároljuk el, mivel így
  - nem kötjük össze a két objektum életpályáját (egyik megsemmisülése nem vonja maga után a másik megsemmisülését), így egymástól függetlenül kezelhetőek
  - egy sekély másolatát kezeljük a megadott objektumnak
- pl.:



Objektumorientált tervezés

Objektumok közötti kapcsolatok

- pl.:

```
class SomeClass { … }; // aggregált osztály
…
class OtherClass
{
private:
   SomeClass* someField; // egy mutatót tárolunk
public:
   OtherClass(SomeClass* s) : someField(s) { }
};
…
SomeClass sc;
OtherClass oc(&sc);
   // csak sekély másolatot készítünk
```

Objektumorientált tervezés

Példa

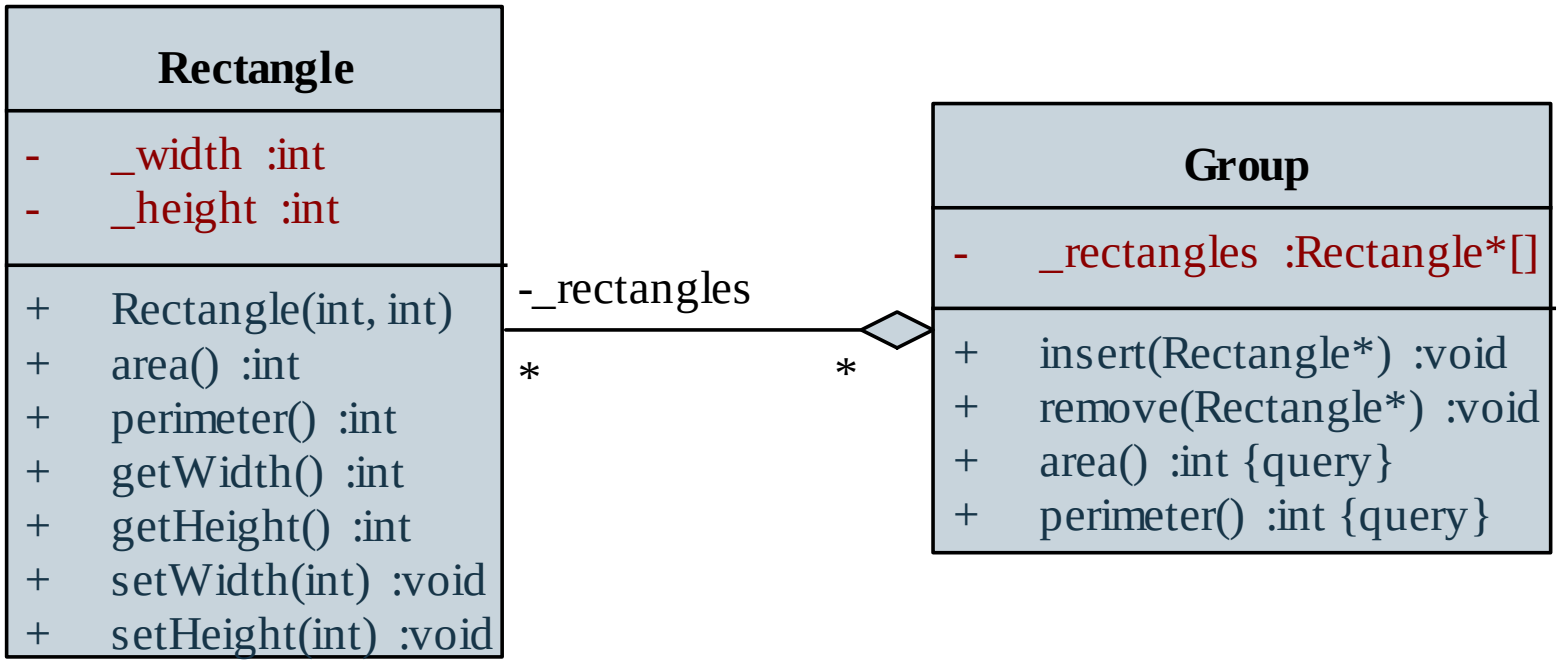
Feladat: Valósítsuk meg a csoportosítható téglalapokat. A téglalapot (**`Rectangle`**) tetszőleges csoportba (**`Group`**) helyezhetjük (**`insert(Rectangle)`**), illetve kivehetjük belőle (**`remove(Rectangle)`**). Lehetőségünk van a csoportban lévő téglalapok összterületét (**`area()`**), illetve összkerületét (**`perimeter()`**) lekérdezni.

- a téglalap megvalósítása változatlan marad, a téglalapokat aggregációval rendeljük a csoporthoz
- egy téglalap több csoportban is szerepelhet, illetve egy csoportban tetszőleges sok téglalap lehet
- a csoportban felvesszük a téglalapok tömbjét (**`_rectangles`**)

Objektumorientált tervezés

Példa

Tervezés:



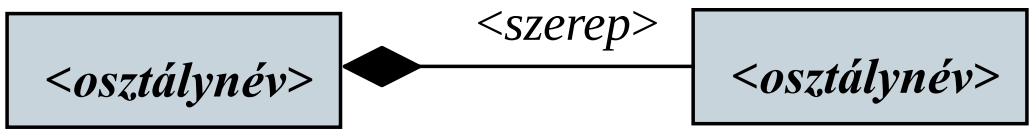
Objektumorientált tervezés

Példa

Megoldás (`group.hpp`*):*

```
class Group {
private:
   vector<Rectangle*> _rectangles;
      // vector-t választunk a megvalósításban, és
      // csak hivatkozásokat kezelünk
public:
   void insert(Rectangle* rec);
      // csak hivatkozásokat veszünk át
   void remove(Rectangle* rec);
   double area() const;
   double perimeter() const;
};
```

- A *kompozíció* (*composition*) egy speciális asszociáció, amely az objektumok szoros egymáshoz rendelését fejezi ki



  - fizikai tartalmazást jelent, így nem lehet reflexív, vagy ciklikus
  - a tartalmazott objektum életpályáját a tartalmazó felügyeli
    - a tartalmazó objektum megsemmisülésekor a tartalmazott is megsemmisül

Objektumorientált tervezés

Példa

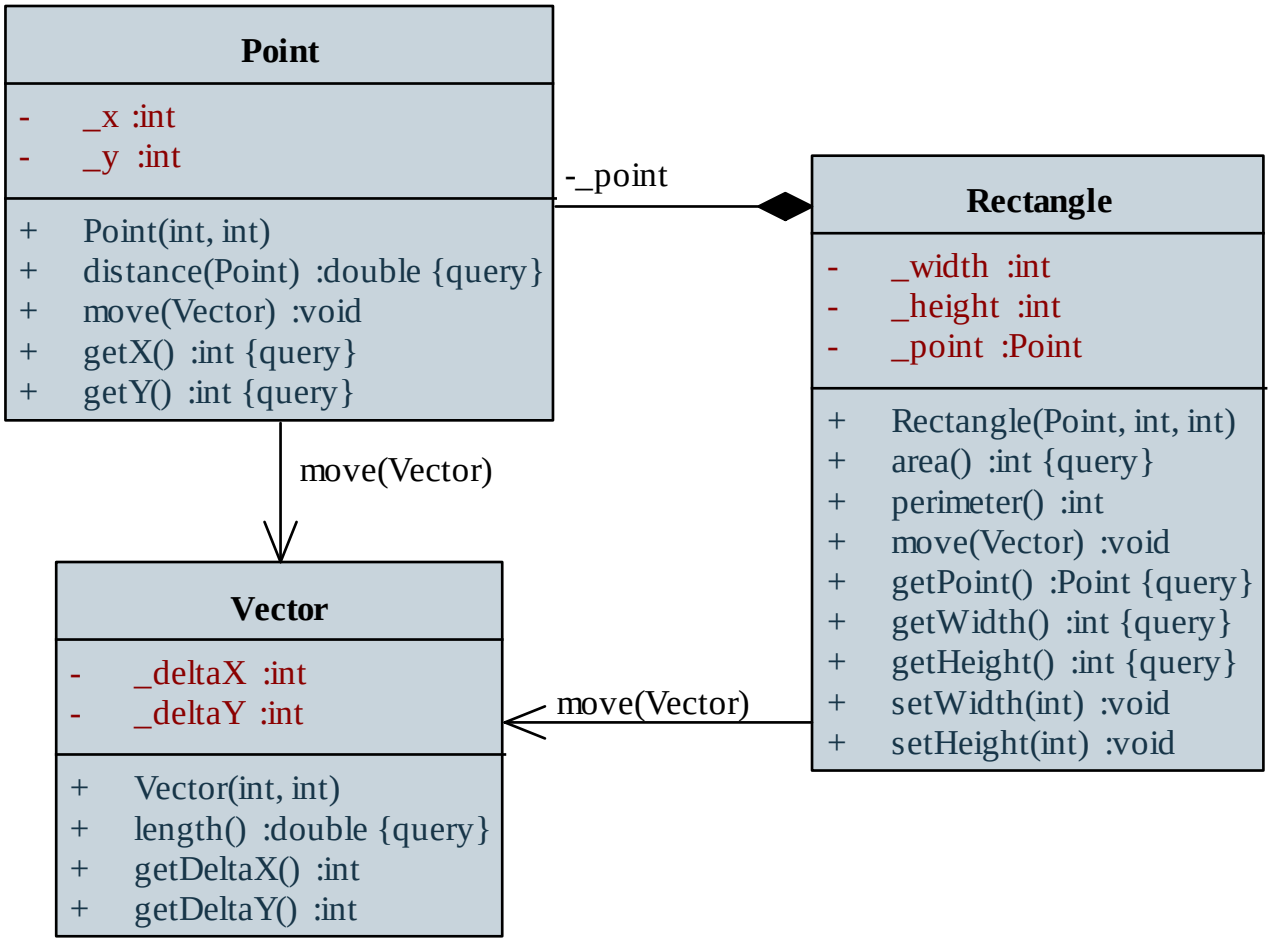
Feladat: Valósítsuk meg a 2D koordinátarendszerben ábrázolható téglalap (`Rectangle`) osztályt, amely párhuzamos/merőleges a koordinátatengelyre, lekérdezhető a területe, átméretezhető, illetve eltolható egy megadott vektorral.

- a téglalapot elhelyezzük a koordinátarendszerben, amihez el kell tárolnunk legalább egy koordinátáját, legyen ez a bal alsó sarok (`_point`)
- ehhez szükséges a pont (`Point`) típusa, amit egybekötünk a téglalap élettartamával, azaz kompozícióval helyezzük a téglalapba
- a téglalap eltolásához (`move()`) igazából a bal alsó koordinátát kell eltolnunk

Objektumorientált tervezés

Példa

Tervezés:



Példa

Megoldás (`rectangle.hpp`*):*

```
#include "vector.hpp"
#include "point.hpp"

class Rectangle {
public:
   Rectangle(Point p, int w, int h) { … }
   void move(Vector v) { _point.move(v); }
   …
private:
   Point _point;
   …
};
```

Objektumorientált programozás

Érték inicializálás

- Amennyiben egy osztály mezőként szerepel egy másik osztályban, még a konstruktor lefutása előtt inicializálni kell
  - amennyiben van paraméter nélküli konstruktora, ez automatikusan megtörténik
  - amennyiben csak paraméteres konstruktora van, az inicializálást manuálisan (a paraméterek megadásával) kell elvégezni, a konstruktor törzse előtt, kettősponttal megadva a megfelelő paraméterezést:

```
<osztálynév>(<típus> <paraméter>) :
    <mezőnév>(<kezdőérték>),
    <mezőnév>(<kezdőérték>), … {
  … // konstruktor törzse
}
```

Objektumorientált programozás

Érték inicializálás

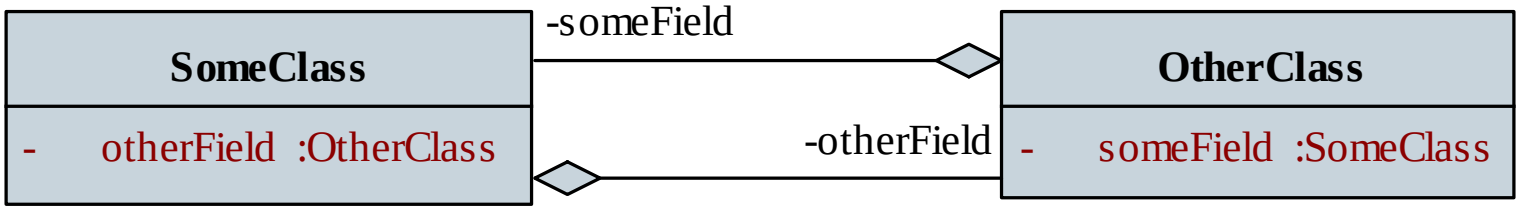
- Pl.:

```
class Rectangle { // téglalap osztály
   …
   Rectangle(Point p, int x, int y);
};
…
Rectangle:: Rectangle(Point p, int x, int y)
    : _point(p) // inicializálás
{
    _width = w > 0 ? w : 0;
    _height = h > 0 ? h : 0;
       // további kezdőértékek beállítása
}
```

Objektumorientált programozás

Ciklikus aggregáció

- Az osztályok közötti kapcsolatok könnyen ciklikussá válhatnak
  - pl. két osztály kölcsönösen hivatkozhat egymásra, azaz aggregálhatja a másik osztály egy példányát



- Ez a megvalósításban azt eredményezné, hogy mindkét osztálynak ismernie kell a másikat
  - egy fájlon belül mindkettőnek meg kell előznie a másikat
  - külön fájlok esetén mindegyikbe kell hivatkozni (másolni) a másikat (`#include`)

Objektumorientált programozás

Osztálydeklarációk

- A ciklikus relációk feloldásáért osztálydeklarációkat kell alkalmaznunk, amely során csak az osztály nevét adjuk meg
  - ezzel jelöljük, hogy a megadott osztály létezik, de nem fedjük fel az interfészét (pl. konstruktor), így csak hivatkozást (referencia, mutató) adhatunk rá
  - pl. (**someclass.hpp**):

```
class OtherClass; // osztály deklarációja

class SomeClass {
    …
    OtherClass* otherField;
    // a deklarált osztály felhasználható lesz
};
```

Osztálydeklarációk

- ahhoz, hogy a megvalósítást is elérjük, a teljes osztályt be kell töltenünk (`#include`), ám ezt elég csak a törzsfájlban megtennünk, így elkerülve a ciklikus hivatkozást
- pl. (`someclass.cpp`):

```
#include "someclass.hpp"
#include "otherclass.hpp"
// itt már használható a teljes osztálydefiníció

SomeClass::SomeClass()
{
   otherField = new OtherClass(…);
   // már ismert az interfész, és benne a
   // konstruktor
}
```

Objektumorientált programozás

Példa

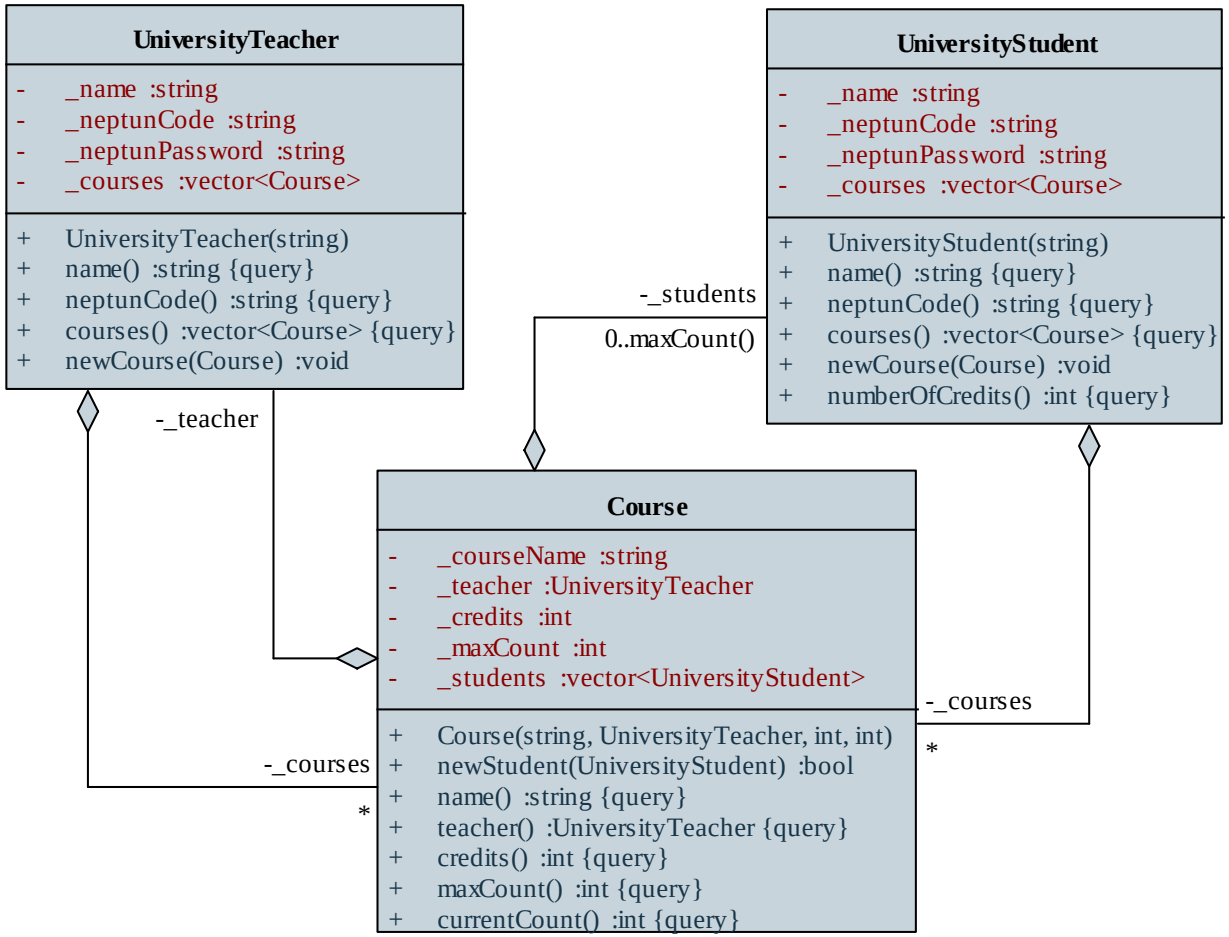
Feladat: Készítsünk egy programot, amelyben egyetemi oktatók, hallgatók és kurzusok adatait tudjuk tárolni.

- a hallgató (`UniversityStudent`) és az oktató (`UniversityTeacher`) rendelkezik névvel, Neptun kóddal és jelszóval, valamint kurzusokkal, a hallgató ezen felül kreditekkel
- a kurzus (`Course`) rendelkezik névvel, oktatóval, hallgatókkal, kreditszámmal és maximális létszámmal
- a kurzus létrehozásakor megkapja az oktatót, amely szintén felveszi azt saját kurzusai közé, míg a hallgató felveheti a kurzust, amennyiben még van szabad hely (ekkor a kurzus megjelenik a hallgatónál, és a hallgató is a kurzusnál

Objektumorientált programozás

Példa

Tervezés:



Példa

Megoldás (`universityteacher.hpp`*):*

```
#include <vector>
#include <string>

class Course;
   // osztályok deklarálása a ciklikus hivatkozás
   // elkerülése végett

class UniversityTeacher // oktató osztály
{
   …
   std::vector<Course*> _courses;
   // mutatókat tárolunk
   …
};
```

Példa

Megoldás (`universitystudent.cpp`*):*

```
…
void UniversityStudent::newCourse(Course* c){
   if (c->newStudent(this))
      // ha a kurzus felveszi a hallgatót
      _courses.push_back(c);
         // csak akkor veheti fel a hallgató a
         // kurzust
}
…
```