Eötvös Loránd Tudományegyetem
Informatikai Kar

Szoftvertechnológia

4. előadás

Objektumorientált tervezés: általánosítás

Giachetta Roberto

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

Objektumorientált tervezés: általánosítás

Példa

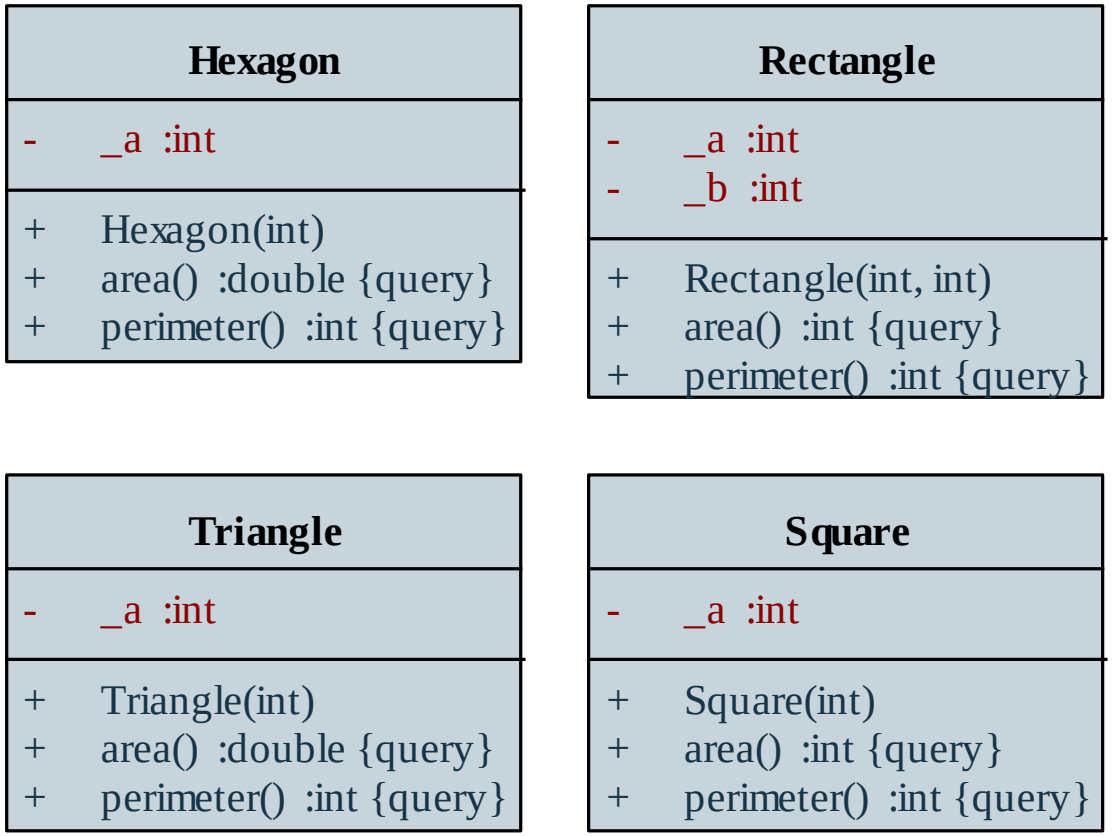
Feladat: Készítsünk egy programot, amelyben különböző geometriai alakzatokat hozhatunk létre (háromszög, négyzet, téglalap, szabályos hatszög), és lekérdezhetjük a területüket, illetve kerületüket.

- a négy alakzatot négy osztály segítségével ábrázoljuk (**`Triangle`**, **`Square`**, **`Rectangle`**, **`Hexagon`**)
- a háromszöget, négyzetet, hatszöget egy egész számmal (**`_a`**), a téglalapot két számmal (**`_a`**, **`_b`**) reprezentáljuk
- mindegyik osztálynak biztosítunk lekérdező műveleteket a területre (**`area`**) és a kerületre (**`perimeter`**)

Objektumorientált tervezés: általánosítás

Példa

Tervezés:



Objektumorientált tervezés: általánosítás

Példa

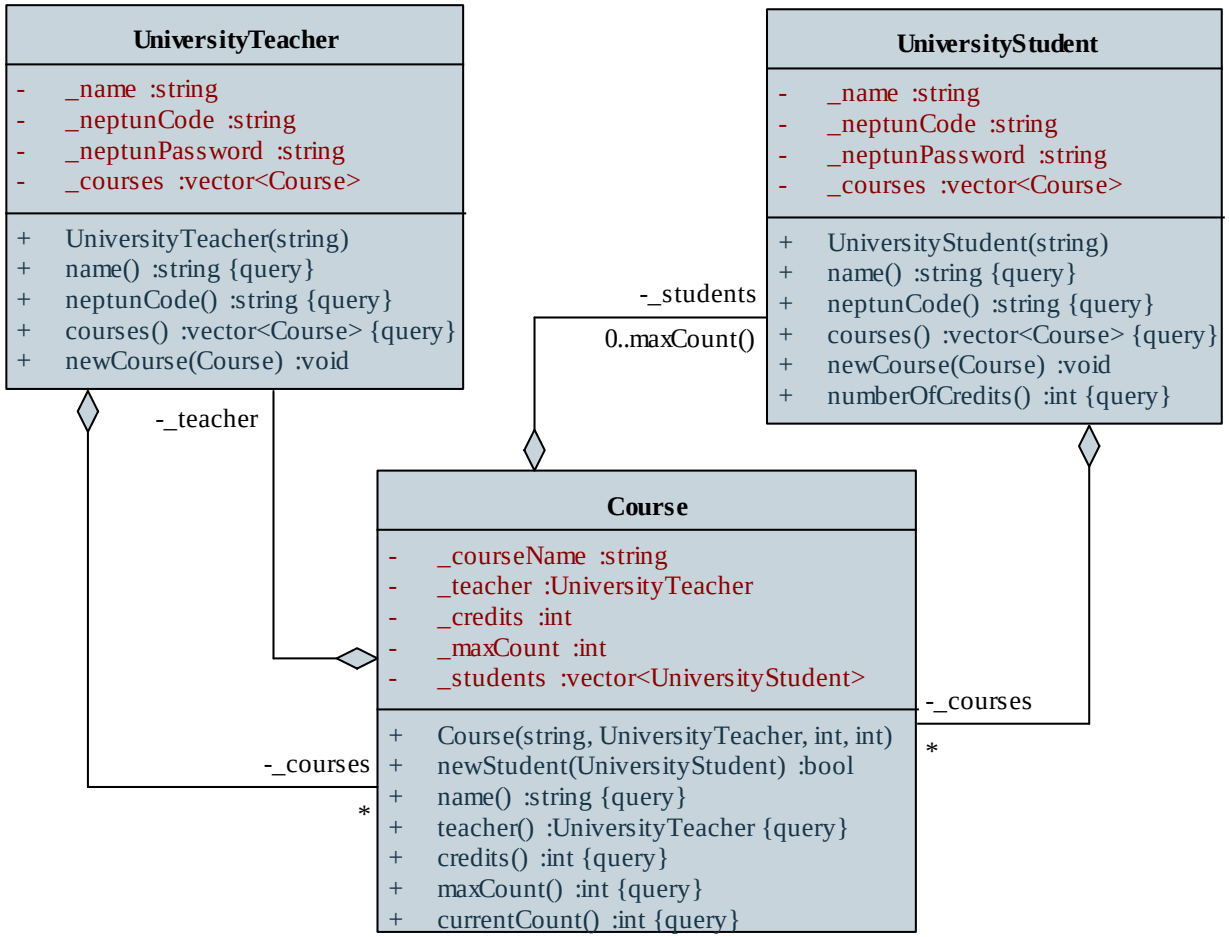
Feladat: Készítsünk egy programot, amelyben egyetemi oktatók, hallgatók és kurzusok adatait tudjuk tárolni.

- 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 tervezés: általánosítás

Példa

Tervezés:



Objektumorientált tervezés: általánosítás

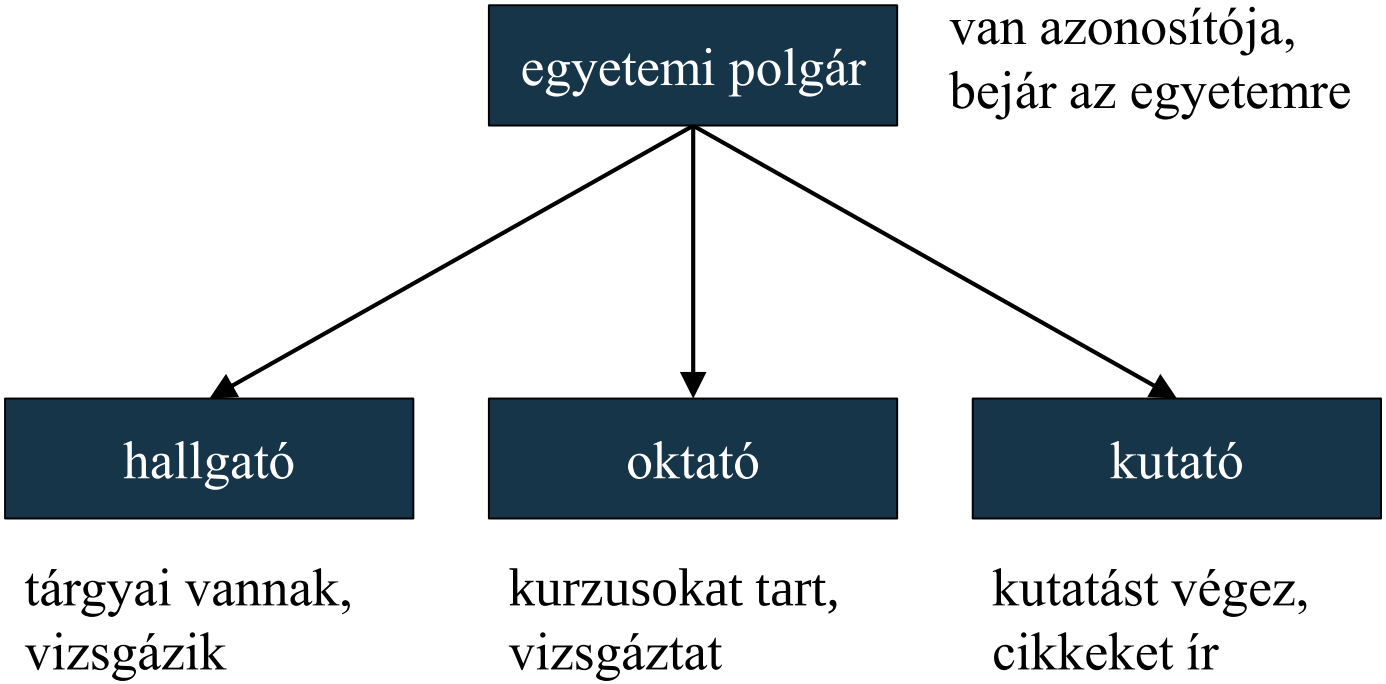
Kódismétlődés

- Az objektum-orientált programokban a különböző osztályok felépítése, viselkedése megegyezhet
  - ez *kódismétlődés*hez vezet, és rontja a kódminőséget
- Hasonlóan procedurális programozás esetén is előfordulhat kódismétlődés, amely alprogramok bevezetésével kiküszöbölhető
  - objektumorientált programok esetén a működés szorosan összekötött az adatokkal
  - így csak együttesen emelhetőek ki, létrehozva ezzel egy új, *általánosabb* osztályt, amelyet össze kell kapcsolnunk a jelenlegi, *speciálisabb* osztállyal

Objektumorientált tervezés: általánosítás

Általánosabb és speciálisabb osztályok

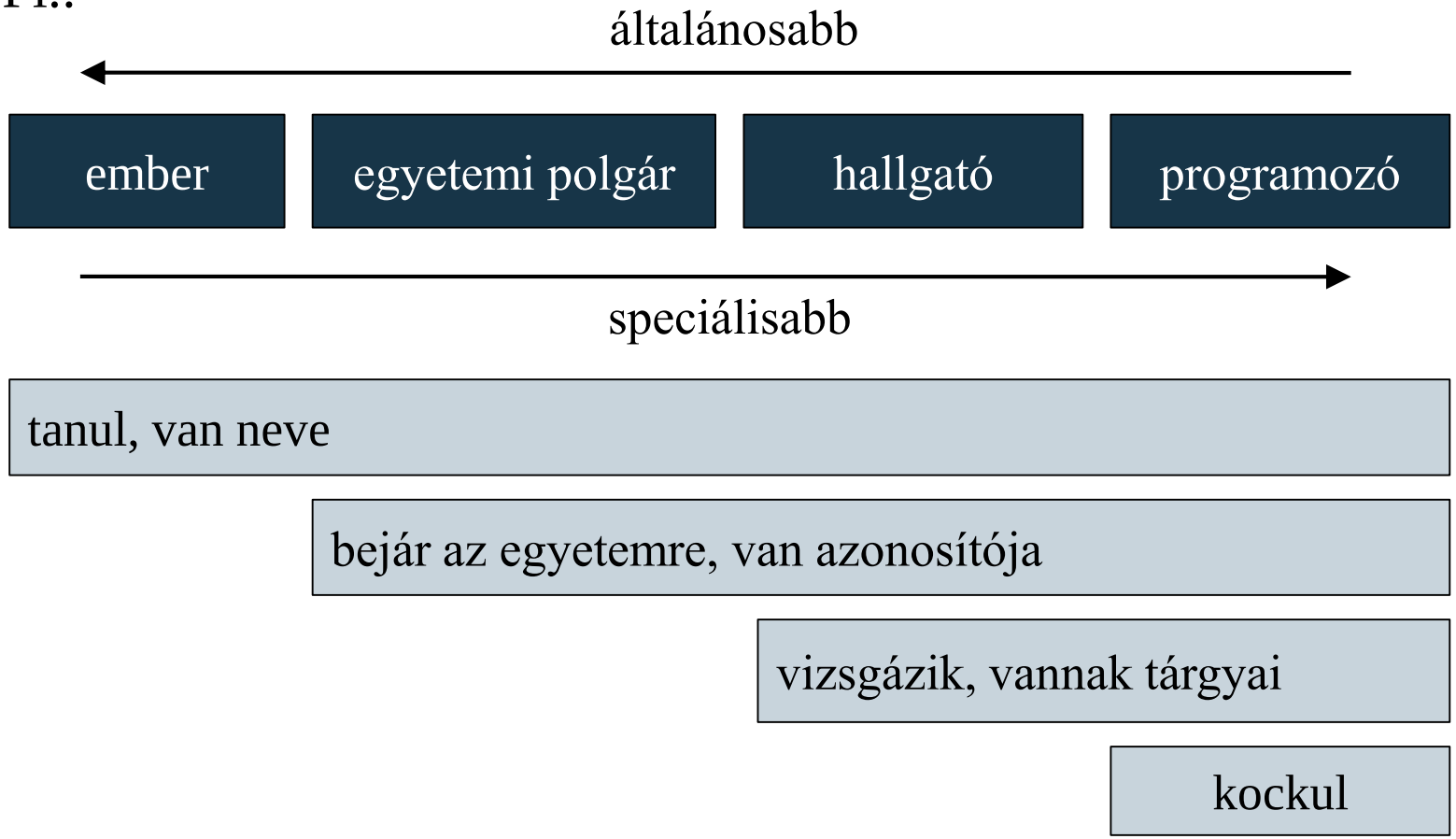
- Pl.:



Objektumorientált tervezés: általánosítás

Általánosabb és speciálisabb osztályok

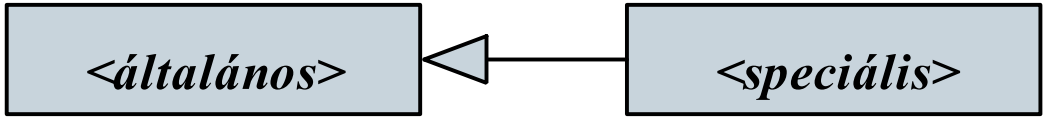
- Pl.:



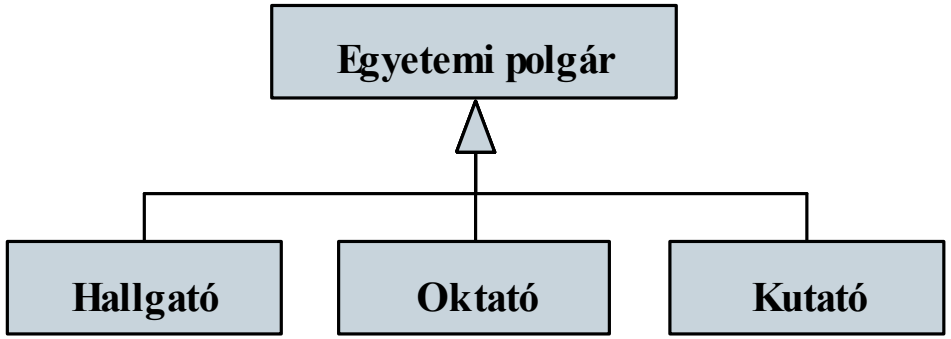
Objektumorientált tervezés: általánosítás

Általánosítás

- Az általánosabb, illetve speciálisabb osztályok között fennálló kapcsolatot nevezzük *általánosítás*nak (*generalization*)



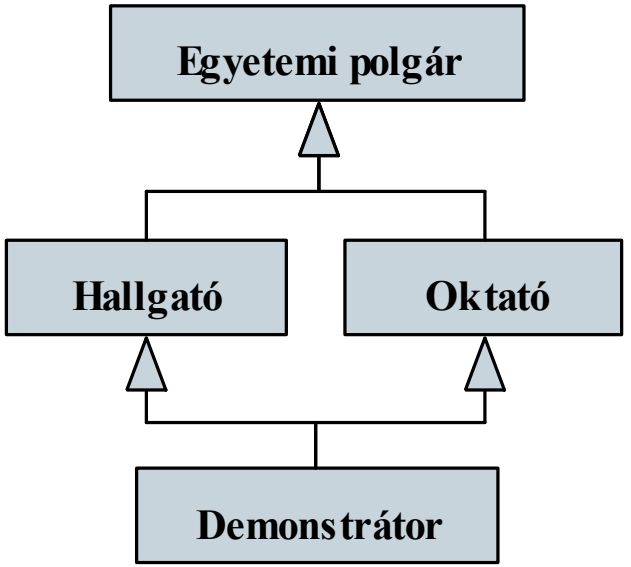
  - a speciális átveszi az általános összes jellemzőjét (tagok, kapcsolatok), amelyeket tetszőlegesen kibővíthet, vagy újrafogalmazhat
  - az ellentétes irányú relációt nevezzük *specializáció*nak (*specialization*)



Objektumorientált tervezés: általánosítás

Általánosítás és specializáció

- Az általános osztályt *ős*nek (*base*, *superclass*), a speciális osztályt *leszármazott*nak (*descendant*, *subclass*) nevezzük
  - ha csak egy szint a különbség, akkor *szülő*nek (*parent*), illetve *gyerek*nek (*child*) nevezzük
  - egy osztálynak lehet több szülője, ekkor többszörös általánosításról beszélünk
  - nem lehet reflexív, vagy ciklikus
  - nincs multiplicitása, elnevezése, szerepei



Objektumorientált tervezés: általánosítás

Öröklődés

- Az általánosítást a programozási nyelvekben az *öröklődés* (*inheritance*) technikájával valósítjuk meg, amely lehet
  - *specifikációs*: csak az általános absztrakt jellemezőit (interfészét) veszi át a speciális
  - *implementációs*: az osztály absztrakt és konkrét jellemzőit (interfészét és implementációját) veszi át a speciális

- Öröklődés C++ nyelven:

```
class <osztálynév> : <láthatóság> <ősosztály> {
   <kiegészítések>
};
```

Objektumorientált tervezés: általánosítás

Öröklődés

- Pl.:

```
class SuperClass // általános osztály
{
public:
    int value; // mező
    SuperClass() { value = 1; } // konstruktor
    void setValue(int v) { value = v; }
    int getValue() { return value; } // metódusok
};

SuperClass super; // osztály példányosítása
cout << super.value; // 1
super.setValue(5);
cout << super.value; // 5
```

Objektumorientált tervezés: általánosítás

Öröklődés

- Pl.:

```
class SubClass : public SuperClass
    // speciális osztály, amely megkapja a value,
    // setValue(int), getValue() tagokat
{
public:
    int otherValue;
    SubClass() {
        value = 2; // használhatjuk az örökölt mezőt
        otherValue = 3;
    }
    void setOtherValue(int v) { otherValue = v; }
    int getOtherValue() { return otherValue; }
};
```

Objektumorientált tervezés: általánosítás

Öröklődés

- Pl.:

```
SubClass sub; // leszármazott példányosítása

// elérhetjük az örökölt tagokat:
cout << sub.value; // 2
sub.setValue(5);
cout << sub.value; // 5

// elérhetjük az új tagokat:
cout << sub.otherValue; // 3
sub.setOtherValue(4);
cout << sub.getOtherValue(); // 4
```

Objektumorientált tervezés: általánosítás

A láthatóság szerepe

- A tagok láthatósága az öröklődés során is szerepet játszik
  - a látható (*public*) tagok elérhetőek lesznek a leszármazottban, a rejtett (*private*) tagok azonban közvetlenül nem
  - ugyanakkor a rejtett tagok is öröklődnek, és közvetetten (örökölt látható műveleteken keresztül) elérhetőek
  - sokszor hasznos, ha a leszármazott osztály közvetlenül elérheti a rejtett tartalmat, ezért használhatunk egy harmadik, védett (*protected*) láthatóságot
    - az osztályban és leszármazottaiban látható, kívül nem
    - az osztálydiagramban **#** jelöli

Objektumorientált tervezés: általánosítás

A láthatóság szerepe

- Pl.:

```
class SuperClass {
private:
    int value; // rejtett mező
public:
    SuperClass() { value = 1; }
    void setValue(int v) { value = v; }
    int getValue() { return value; }
};

SuperClass super;
cout << super.getValue(); // 1
super.setValue(5);
cout << super.getValue(); // 5
```

Objektumorientált tervezés: általánosítás

A láthatóság szerepe

- Pl.:

```
class SubClass : public SuperClass {
   // a value már nem látszódik, de öröklődik
private:
   int otherValue;
public:
   SubClass() {
      setValue(2);
         // nem látja a value mezőt, de
         // közvetetten használhatja
      otherValue = 3;
   }
   …
};
```

Objektumorientált tervezés: általánosítás

A láthatóság szerepe

- Pl.:

```
SubClass sub; // leszármazott példányosítása

// elérhetjük az örökölt tagokat:
cout << sub.getValue(); // 2
sub.setValue(5);
cout << sub.getValue(); // 5

// elérhetjük az új tagokat:
cout << sub.getOtherValue(); // 3
sub.setOtherValue(4);
cout << sub.getOtherValue(); // 4
```

Objektumorientált tervezés: általánosítás

A láthatóság szerepe

- Pl.:

```
class SuperClass {
protected:
    int value; // védett mező
public:
    SuperClass() { value = 1; }
    void setValue(int v) { value = v; }
    int getValue() { return value; }
};

SuperClass sup;
cout << sup.getValue(); // 1
sup.setValue(5);
cout << sup.getValue(); // 5
```

Objektumorientált tervezés: általánosítás

A láthatóság szerepe

- Pl.:

```
class SubClass : public SuperClass {
    // minden elérhető lesz az ősből
private:
    int otherValue;
public:
    SubClass() {
        value = 2; // használhatjuk az örökölt mezőt
        otherValue = 3;
    }
    …
};
```

Objektumorientált tervezés: általánosítás

Öröklődés

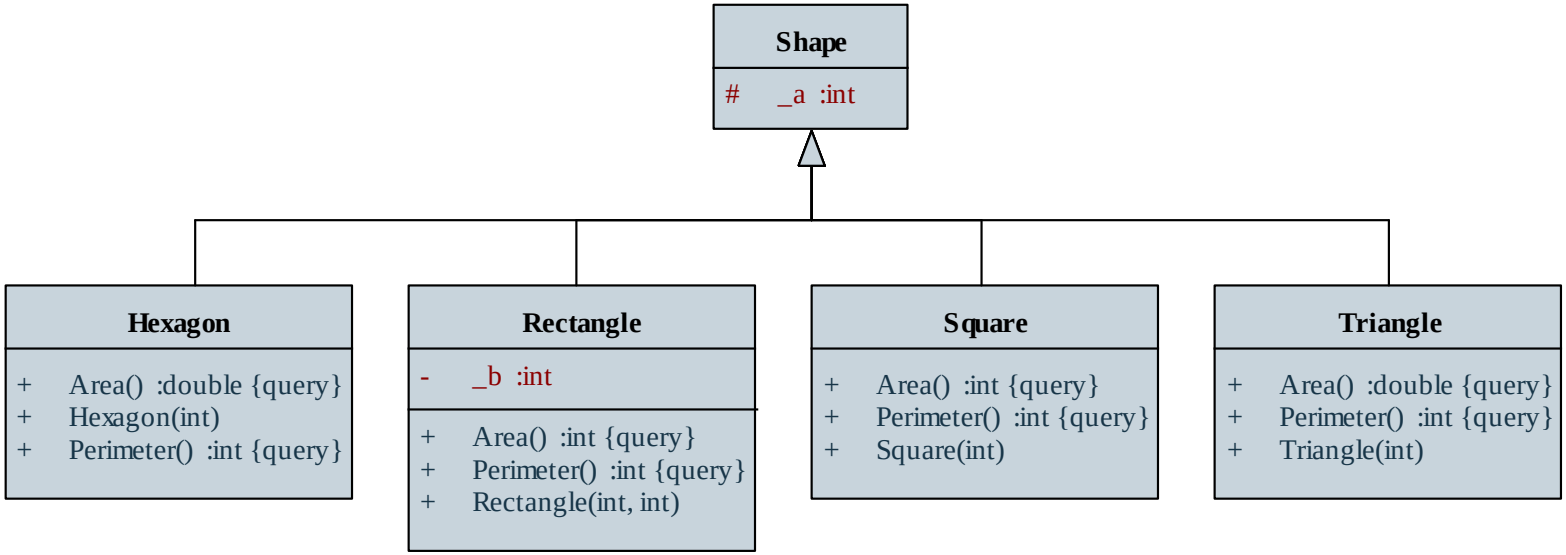
Feladat: Készítsünk egy programot, amelyben különböző geometriai alakzatokat hozhatunk létre (háromszög, négyzet, téglalap, szabályos hatszög), és lekérdezhetjük a területüket, illetve kerületüket.

- javítsunk a korábbi megoldáson öröklődés segítségével
- kiemelünk egy általános alakzat osztályt (**`Shape`**), amelybe helyezzük a közös adatot (**`_a`**), ennek védett (**`protected`**) láthatóságot adunk
- a többi osztályban csak az öröklődést jelezzük, a működés nem változik (továbbra is a konstruktorok állítják be **`_a`** értékét)

Objektumorientált tervezés: általánosítás

Öröklődés

Tervezés:



Objektumorientált tervezés: általánosítás

Öröklődés

Megvalósítás:

```
class Shape {
   protected:
      int _a;
};

class Ractangle : public Shape {
   private:
      int _b;
   public:
      Rectangle(int a, int b) { _a = a; _b = b; }
      …
};
```

Objektumorientált tervezés: általánosítás

Metódusok viselkedése

- Öröklődés során lehetőségünk van a viselkedés újrafogalmazására
  - a leszármazottban az ősével megegyező szintaktikájú metódusok vagy *elrejtik*, vagy *felüldefiniálják* az örökölt metódusokat
  - a leszármazott példányosításakor az ott definiált viselkedés érvényesül
- Öröklődés során a leszármazott osztály példányosításakor az ős is példányosodik
  - a konstruktor is öröklődik, és meghívódik a leszármazott példányosításakor (implicit, vagy explicit)
  - a destruktor is öröklődik, és automatikusan meghívódik a leszármazott megsemmisítésekor

Objektumorientált tervezés: általánosítás

Metódusok viselkedése

- Pl.:

```
class SuperClass {
    …
    int getDefaultValue() { return 1; }
    int computeDoubleDefault()
    {
        return 2 * getDefaultValue();
        // felhasználjuk a másik metódust
    }
};

Superclass super;
cout << sub.computeDoubleDefault(); // kiírja: 2
```

Objektumorientált tervezés: általánosítás

Metódusok viselkedése

```
class SubClass : public SuperClass {
   …
   int getDefaultValue() { return 2; }
      // elrejtő metódus
};

SubClass sub;
cout << sub.computeDoubleDefault();
    // kiírja: 4, mivel getDefaultValue() == 2
```

Objektumorientált tervezés: általánosítás

Metódusok viselkedése

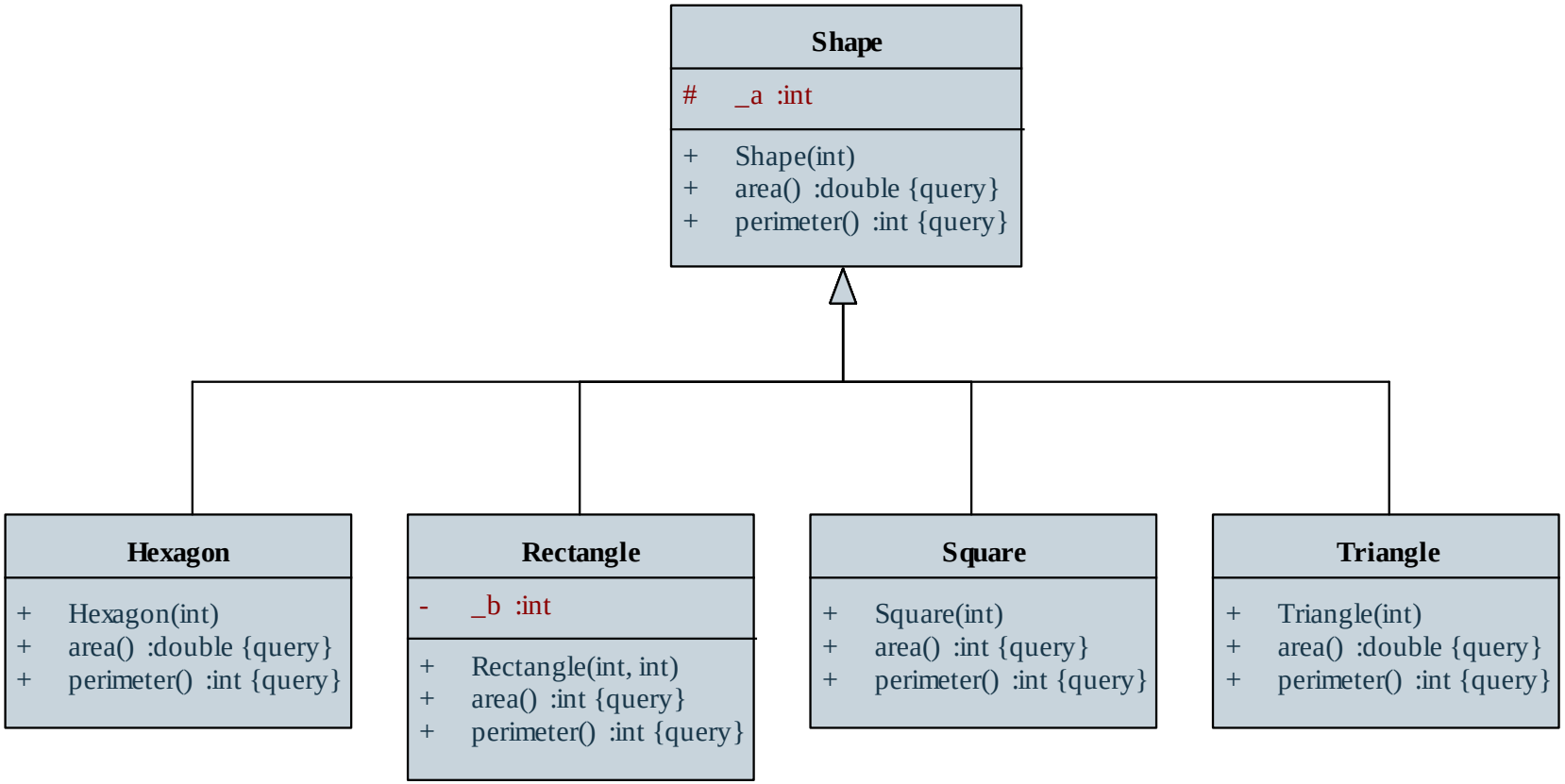
Feladat: Készítsünk egy programot, amelyben különböző geometriai alakzatokat hozhatunk létre (háromszög, négyzet, téglalap, szabályos hatszög), és lekérdezhetjük a területüket, illetve kerületüket.

- az ősben létrehozzuk a terület (**`area`**), illetve kerület (**`perimeter`**) lekérdezés metódusait, amelyeket felüldefiniálunk a leszármazottakban, így jobban kifejezzük, hogy minden alakzat rendelkezik ilyen művelettel
- bővíthetjük az ős konstruktorának feladatkörét a közös adat (**`_a`**) inicializálásával, így minden osztály a saját mezőinek inicializálását végzi

Objektumorientált tervezés: általánosítás

Metódusok viselkedése

Tervezés:



Objektumorientált tervezés: általánosítás

Metódusok viselkedése

Megvalósítás:

```
class Shape {
   protected:
      int _a;
   public:
      Shape(int a) { _a = a; }
      double area() const { return 0; }
      double perimeter() const { return 0; }
        // alapértelmezett viselkedés, ami
        // öröklődik
};
```

Objektumorientált tervezés: általánosítás

Metódusok viselkedése

Megvalósítás:

```
class Rectangle : public Shape {
   private:
      int _b;
   public:
      Rectangle(int a, int b) : Shape(a) {
         _b = b;
      }
      double area() const { return _a * _b; }
      double perimeter() const {
         return 2 * (_a + _b);
      }
         // elrejtő viselkedés
};
```

Objektumorientált tervezés: általánosítás

Polimorfizmus

- Mivel a leszármazott példányosításakor egyúttal az ősből is létrehozunk egy példányt, a keletkezett objektum példánya lesz mindkét típusnak
  - a leszármazott objektum bárhova behelyettesíthető lesz, ahol az ős egy példányát használjuk
- Ezt a jelenséget (*altípusos*) *polimorfizmus*nak (*polymorphism*, *subtyping*), vagy *többalakúság*nak nevezzük
  - pl. a `SubClass sub;` utasítással egyúttal a `SuperClass` példányát is elkészítjük
  - öröklődés nélkül is fennállhat dinamikus típusrendszerű programozási nyelvekben (ez a *strukturális polimorfizmus*)

Objektumorientált tervezés: általánosítás

Polimorfizmus

- A programozási nyelvek az objektumokat általában dinamikusan kezelik (referencián, vagy mutatón keresztül)
  - pl.:

```
SuperClass* super = new SuperClass();
cout << super->getValue(); // 1
```

- A polimorfizmus lehetővé teszi, hogy a dinamikusan létrehozott objektumra hivatkozzunk az ősosztály segítségével
  - pl.:

```
SuperClass* super = new SubClass();
   // a mutató az ősosztályé, de ténylegesen
   // a leszármazott objektummal dolgozunk
cout << super->getValue(); // 2
```

Objektumorientált tervezés: általánosítás

Polimorfizmus

- A dinamikusan létrehozott objektumot így két típussal rendelkeznek:
  - a hivatkozás osztálya az objektum *statikus típusa*, ezt értelmezi a fordítóprogram, ennek megfelelő tagokat hívhatunk meg
  - a példányosított osztály a változó *dinamikus típusa*, futás közben az annak megfelelő viselkedést végzi

- Pl.:

```
SuperClass* super1 = new SubClass();
   // super1 statikus típusa SuperClass,
   // dinamikus típusa SubClass

SuperClass* super2 = new SuperClass();
   // itt egyeznek a típusok
```

Objektumorientált tervezés: általánosítás

Polimorfizmus

- A dinamikus típus futás közben változtatható, mivel az ős típusú hivatkozásra tetszőleges leszármazott példányosítható
  - pl.:

```
SuperClass* super = new SuperClass();
cout << super->getValue(); // 1
delete super;
super = new SubClass();
cout << super->getValue(); // 2
```

  - ugyanakkor a statikus típusra korlátozott az elérhető tagok köre, pl.:

```
cout << super->getOtherValue();
   // fordítási hiba, a statikus típusnak nincs
   // getOtherValue() művelete
```

Objektumorientált tervezés: általánosítás

Polimorfizmus

- A polimorfizmus azt is lehetővé teszi, hogy egy gyűjteményben különböző típusú elemeket tároljunk
  - a gyűjtemény elemtípusa az ős hivatkozása lesz, és az elemek dinamikus típusát tetszőlegesen váltogathatjuk

- Pl.:

```
SuperClass* array[3];
array[0] = new SuperClass();
array[1] = new SubClass();
array[2] = new SubClass();

for (int i = 0; i < 3; i++)
   cout << array[i]->getValue(); // 1 2 2
```

Objektumorientált tervezés: általánosítás

Dinamikus kötés

- Dinamikus példányosítást használva a program a dinamikus típusnak megfelelő viselkedést rendeli hozzá az objektumhoz futási idő alatt, ezt *dinamikus kötés*nek (*dynamic binding*) nevezzük
  - amennyiben *felüldefiniálunk* (*override*) egy műveletet, a dinamikus típusnak megfelelő végrehajtás fog lefutni
  - ehhez azonban a műveletnek engedélyeznie kell a felüldefiniálást, ekkor beszélünk *virtuális* (*virtual*) műveletről
  - a nem virtuális műveletek a *lezárt*, vagy *véglegesített* (*sealed*) műveletek, ezeket csak elrejteni lehet, és ekkor a statikus típus szerint fog végrehajtódni a művelet

Objektumorientált tervezés: általánosítás

Dinamikus kötés

- Pl.:

```
class SuperClass {
protected:
    int value;
public:
    SuperClass() { value = 1; }
    void setValue(int v) { value = v; }

    virtual void getValue() { return value; }
        // virtuális metódus
    void getDefaultValue() { return 1; }
        // véglegesített metódus
};
```

Objektumorientált tervezés: általánosítás

Dinamikus kötés

```
class SubClass : public SuperClass {
   …
   void getValue() { return otherValue; }
      // felüldefináló metódus
   void getDefaultValue() { return 2; }
      // elrejtő metódus
};
…
SuperClass *sup = new SuperClass();
cout << sup->getValue(); // 1
cout << sup->getDefaultValue(); // 1
sup = new SubClass();
cout << sup->getValue(); // 3
cout << sup->getDefaultValue(); // 1
```

Objektumorientált tervezés: általánosítás

Dinamikus kötés

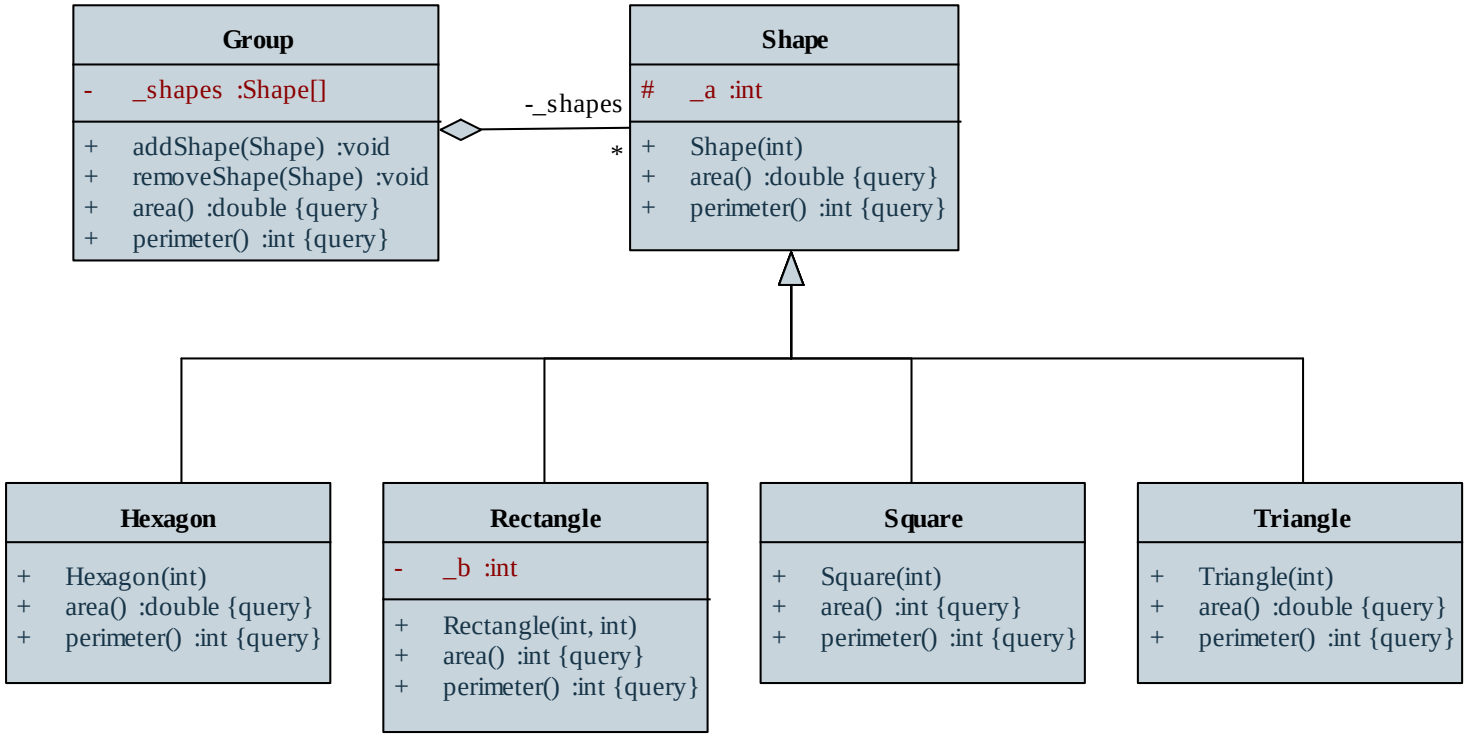
Feladat: Készítsünk egy programot, amelyben különböző geometriai alakzatokat hozhatunk létre (háromszög, négyzet, téglalap, szabályos hatszög), és lekérdezhetjük a területüket, illetve kerületüket. Az alakzatokat csoportosíthatjuk is.

- az ősben a terület (**`area`**), illetve kerület (**`perimeter`**) lekérdezés metódusait virtuálissá változtatjuk, így már felüldefiniáljuk őket a leszármazottban
- létrehozzuk az alakzatok csoportját (**`Group`**), amelybe behelyezzük az alakzatok gyűjteményét
- lekérdezhetjük a csoportba lévő elemek összterületét és összkerületét

Objektumorientált tervezés: általánosítás

Dinamikus kötés

Tervezés:



Objektumorientált tervezés: általánosítás

Dinamikus kötés

Megvalósítás:

```
class Shape {
   protected:
      int _a;
   public:
      Shape(int a) { _a = a; }
      virtual double area() const { return 0; }
      virtual double perimeter() const {
         return 0;
      }
        // alapértelmezett viselkedés, ami
        // öröklődik, de felüldefiniálható
};
```

Objektumorientált tervezés: általánosítás

Dinamikus kötés

Megvalósítás:

```
class Group {
   private:
      vector<Shape*> _shapes;
   public:
      …
      double area() const {
         double sum = 0;
         for (int i = 0; i < _shapes.size(); i++)
            sum += _shapes[i]->area();
            // a megfelelő metódus fut le
         return sum;
      }
};
```

Objektumorientált tervezés: általánosítás

Absztrakt osztályok

- Amennyiben egy ősosztály olyan általános viselkedéssel rendelkezik, amelyet konkrétan nem tudunk alkalmazni, vagy általánosságban nem tudunk jól definiálni, akkor megtilthatjuk az osztály példányosítását

- A nem példányosítható osztályt *absztrakt osztály*nak (*abstract class*) nevezzük
  - csak statikus típusként szerepelhetnek
  - absztrakt osztályban létrehozható olyan művelet, amelynek nincs megvalósítása, csak szintaxisa, ezek az *absztrakt*, vagy *tisztán virtuális* műveletek
  - a leszármazottak *megvalósítják* (*realize*) az absztrakt műveletet (vagy szintén absztrakt osztályok lesznek)

Objektumorientált tervezés: általánosítás

Absztrakt osztályok

- absztrakt osztály létrehozható a konstruktor elrejtésével, vagy absztrakt művelet definiálásával
- a diagramban dőlt betűvel jelöljük őket

- Pl.:

```
class SuperClass { // absztrakt osztály
  …
  virtual void getValue() =0; // absztrakt metódus
};

SuperClass *super = new SubClass();
cout << super.getValue(); // 3
super = new SuperClass(); // fordítási hiba
```

Objektumorientált tervezés: általánosítás

Absztrakt osztályok

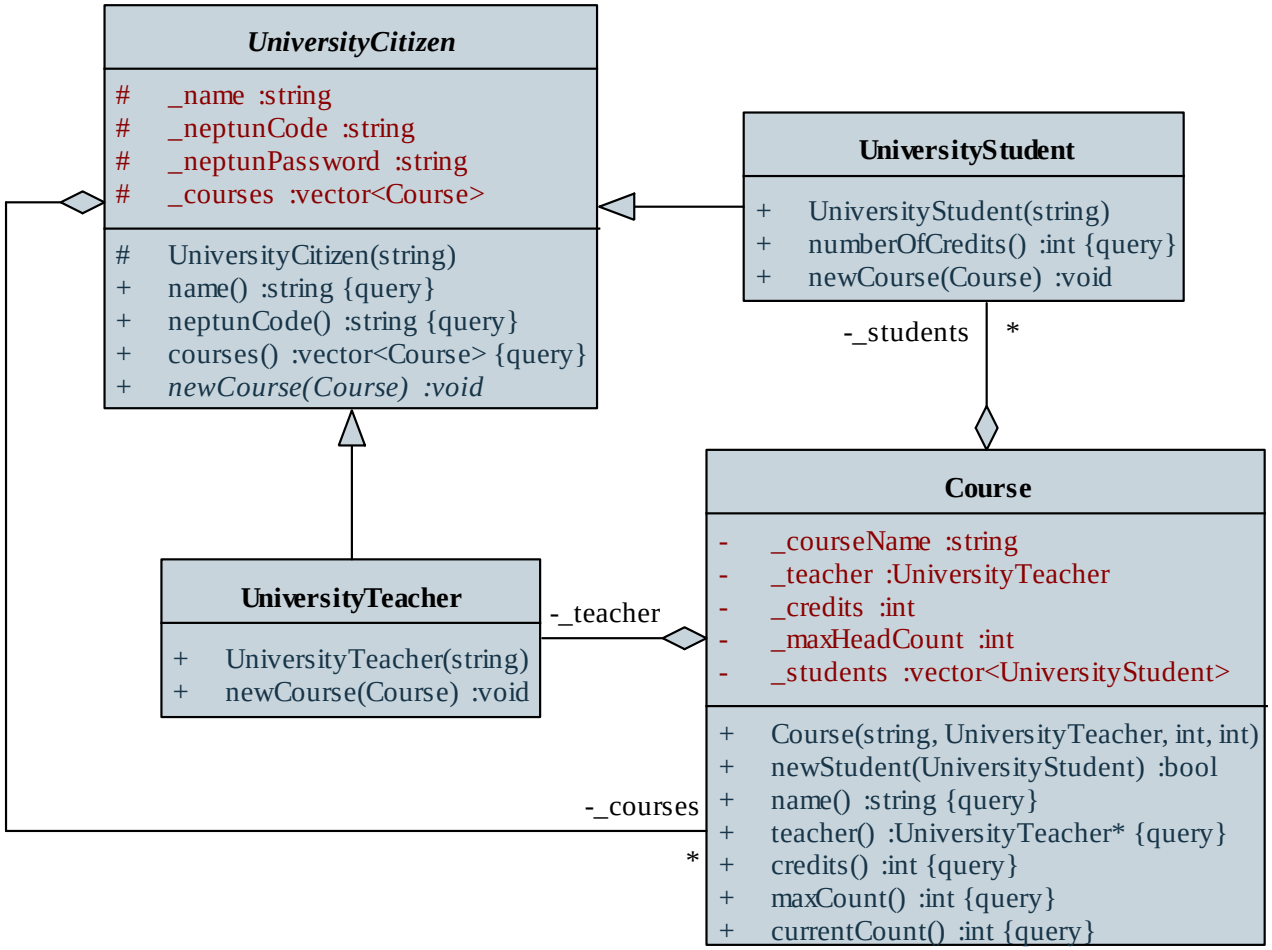
Feladat: Készítsünk egy programot, amelyben egyetemi oktatók, hallgatók és kurzusok adatait tudjuk tárolni.

- a hallgató (**`UniversityStudent`**) és az oktató (**`UniversityTeacher`**) közös tagjait kiemeljük az egyetemi polgár (**`UniversityCitizen`**) absztrakt ősosztályba
- a leszármazottak definiálják a **`newCourse(…)`** műveletet külön-külön, ezért az ősben tisztán absztrakt lesz, továbbá a hallgatónál megjelenik a kreditek lekérdezése (**`numberOfCredits()`**)
- a változtatás a kurzus (**`Course`**) osztályra nincs hatással

Objektumorientált tervezés: általánosítás

Absztrakt osztályok

Tervezés:



Objektumorientált tervezés: általánosítás

Absztrakt osztályok

Megvalósítás:

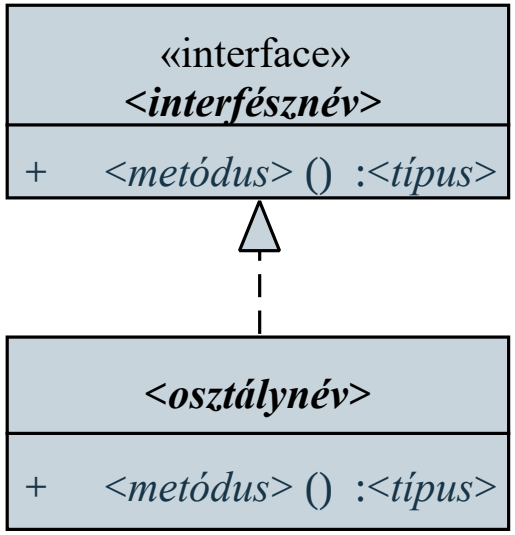
```
class UniversityCitizen { // absztrakt osztály
   public:
      virtual void newCourse(Course& course) = 0;
         // absztrakt művelet
   …
};

class UniversityStudent : public UniversityCitizen
{
   public:
      void newCourse(const Course& course) { … }
   …
};
```

Objektumorientált tervezés: általánosítás

Interfészek

- Amennyiben csak a felületét akarjuk az osztályoknak definiálni, lehetőségünk van *interfészek* (*interface*) létrehozására
  - egy olyan absztrakt osztályt, amely csak publikus absztrakt műveletekből áll
  - célja a különböző feladatkörök elválasztása, a többszörös öröklődés megkönnyítése
  - a diagramban az **`<<interface>>`** csoportot használjuk, és elhagyjuk a dőlt betűket



- Amennyiben interfészt specializálunk osztályba, azzal *megvalósítjuk* (*realize*, *implement*) az interfészt

Objektumorientált tervezés: általánosítás

Interfészek

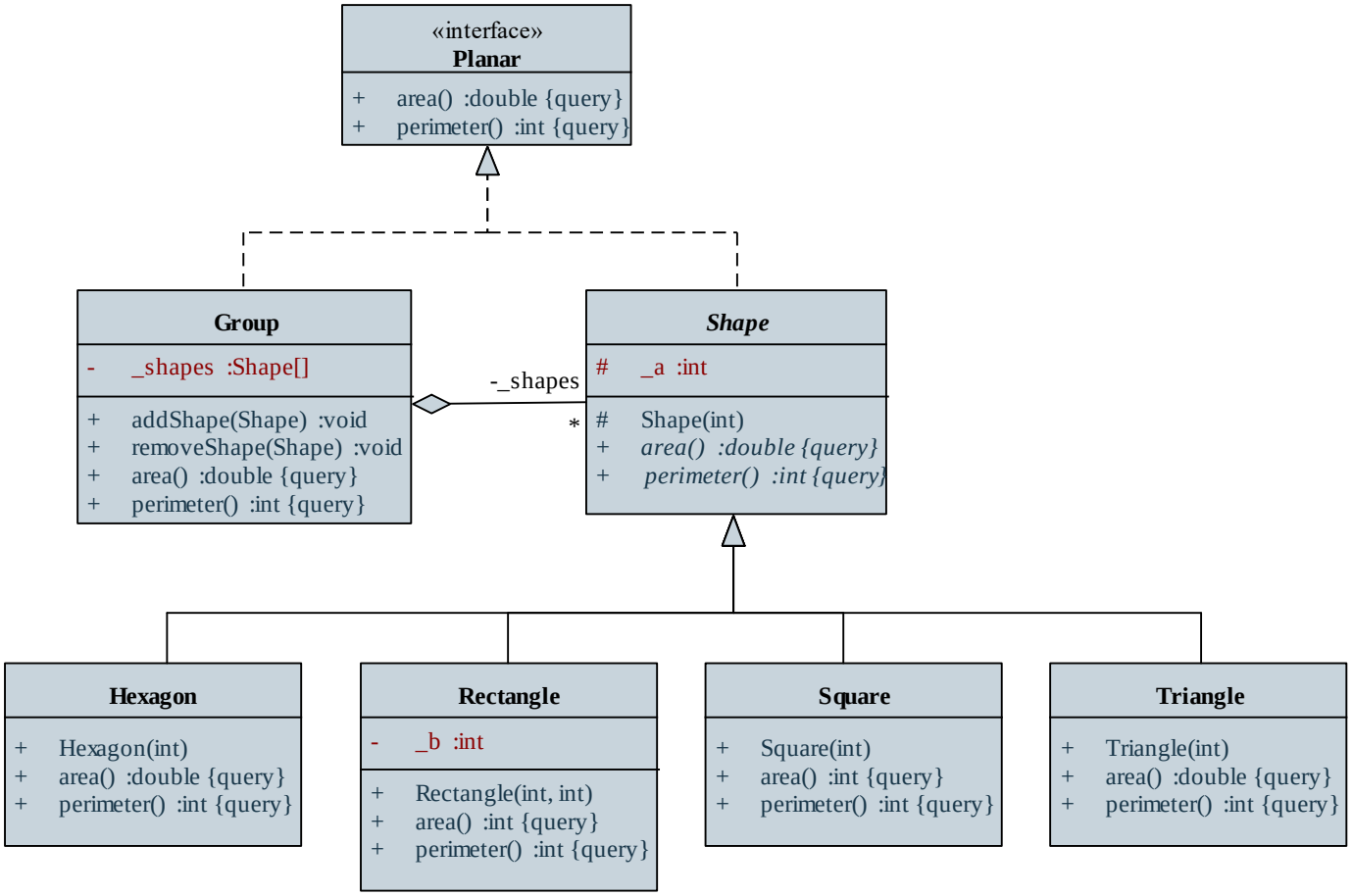
Feladat: Készítsünk egy programot, amelyben különböző geometriai alakzatokat hozhatunk létre (háromszög, négyzet, téglalap, szabályos hatszög), és lekérdezhetjük a területüket, illetve kerületüket. Az alakzatokat csoportosíthatjuk is.

- az egységes kezelés érdekében létrehozhatjuk a síkbeli elemek (`Planar`) interfészét, amely definiálja, hogy minden síkbeli elemnek van területe és kerülete
- a síkbeli elemet megvalósítja a csoport és az alakzat is
- az alakzat lehet absztrakt osztály (a konstruktorát védetté tehetjük)

Objektumorientált tervezés: általánosítás

Interfészek

Tervezés:



Objektumorientált tervezés: általánosítás

Interfészek

Megvalósítás:

```
class Planar { // interfész
   public:
     virtual double area() const = 0;
     virtual double perimeter() const = 0;
        // csak absztrakt műveleteket tartalmaz
};

class Shape : public Planar {
   protected:
      int _a;
      Shape(int a) { _a = a; }
};
```