Eötvös Loránd Tudományegyetem
Informatikai Kar

Szoftvertechnológia

3. előadás

Objektumorientált tervezés: alapismeretek

Giachetta Roberto

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

„Actually I made up the term 'Object-Oriented', and I can tell you I did not have C++ in mind.”

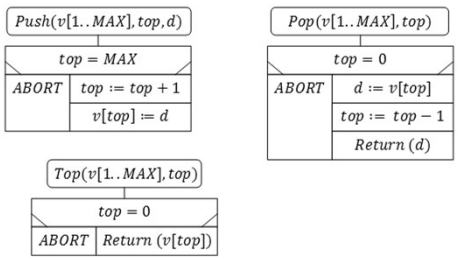
(Alan Kay)

- A procedurális programozási paradigma összetett alkalmazások esetén számos korlátozást tartalmaz:
  - a program nem tagolható kellő mértékben (csupán alprogramok adottak)
  - az adatok élettartama nem eléggé testre szabható (vannak lokális és globális változók)
  - a vezérlés egy helyre (főprogram) összpontosul
  - a feladat módosítása utóhatásokkal rendelkezhet
- Pl. amennyiben módosítjuk egy alprogramban az adatok reprezentációjának módját, az hatással lehet az összes, vele kapcsolatban álló alprogramra

Feladat: Valósítsuk meg a verem (**`Stack`**) adatszerkezetet aritmetikai reprezentáció mellett. Lehessen elemet behelyezni (**`push`**), kivenni (**`pop`**), lekérdezni a tetőelemet (**`top`**), üres-e (**`isEmpty`**)

- a verem absztrakt adattípusa $S = (D^n, \{Push, Pop, Top, IsEmpty\})$, ahol
  - $Push: S \times D \to S$
  - $Pop: S \to S \times D$
  - $Top: S \to D$
  - $IsEmpty: S \to \mathbb{L}$
- aritmetikai reprezentáció esetén egy vektor (**`v`**) és tetőelem index (**`top`**) biztosítja a vermet

- a verem műveletei aritmetikai reprezentációval:



Megvalósítás:

```
void push(vector<int> v, int top, int d) {
   if (top == v.size())
      throw STACK_FULL; // hibajelzés (kivétel)

   top = top + 1; // művelet végrehajtása
   v[top] = d;
}

int top(vector<int> v, int top) {
   if (top == 0)
      throw STACK_FULL;
   return v[top];
}
```

Objektumorientált tervezés: alapismeretek
Procedurális programozás

Megvalósítás:

…

```
int main()
{
   vector<int> v;
   int top = 0; // verem létrehozása (2 lépésben)
   …
   push(v, top, d); // elem behelyezése
   …
   cout << top(v, top); // tetőelem lekérdezése
   …
}
```


Objektumorientált tervezés: alapismeretek
Kialakulása

- Megoldások:
  - 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
  - a *reprezentáció és a belső működés elrejtése*
    - a külvilág (többi programegység) elől elrejtjük a működést, beleértve az adatok kezelésének módját
    - a belső módosítások így nem befolyásolják a kommunikációt

Objektumorientált tervezés: alapismeretek
Az objektum

- *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 tervezés: alapismeretek
Az objektum

- Az objektumok *életciklus*sal rendelkeznek: létrejönnek (a konstruktorral), működést hajtanak végre (további műveletekkel), majd megsemmisülnek (a destruktorral)
- Az objektumok *állapot*tal (state) rendelkeznek, ahol az állapot adatértékeinek 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 összesített állapota adja meg

Objektumorientált tervezés: alapismeretek
Az objektum-orientált program

- *Objektum-orientált*nak nevezzük azt a programot, amelyet egymással kommunikáló objektumok összessége alkot
  - minden adat egy objektumhoz tartozik, és minden algoritmus egy objektumhoz rendelt tevékenység, nincsenek globális adatok, vagy globális algoritmusok
  - a program így kellő tagoltságot kap az objektumok mentén
  - az adatok élettartama 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 tervezés: alapismeretek
Az objektum-orientált program

- Az objektum-orientáltság öt alaptényezője:
  - *absztrakció*: az objektum reprezentációs szintjének megválasztása
  - *enkapszuláció*: az adatok és alprogramok egységbe zárása, a belső megvalósítás 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, a viselkedés átdefiniálása

Objektumorientált tervezés: alapismeretek
Az osztály

- Az objektumok viselkedési mintáját az *osztály* tartalmazza, az osztályból *példányosít*hatjuk az objektumokat
  - tehát az osztály az objektum típusa
  - speciális osztályoknak tekinthetőek a
    - *rekordok* (*record*, *structure*), amelyek általában metódusok nélküli, egyszerűsített osztályok, adattárolási céllal
    - *felsorolási típusok* (*enumeration*), amelyek csak értékkel rendelkeznek
- Az osztályban tárolt adatokat *attribútum*oknak, vagy *mező*knek (*field*), az általa elvégezhető műveleteket *metódus*oknak (*method*) nevezzük, ezek alkotják az osztály *tag*jait (*member*)

Objektumorientált tervezés: alapismeretek
Láthatóság

- Az osztály tagjainak szabályozhatjuk a láthatóságát, a kívülről látható (*public*) részét *felület*nek, vagy *interfész*nek, a kívülről rejtett (*private*) részét *implementáció*nak nevezzük
  - a metódusok megvalósítása az implementáció része, tehát más osztályok számára a működés mindig ismeretlen
  - az osztály mezői is az implementáció része, ezért minden mező rejtett (kivéve rekordok esetén)
  - a mezőkhöz hozzáférést *lekérdező* (*getter*), illetve *beállító* (*setter*) műveletekkel engedélyezhetünk
- Az osztályokat minden nyelv más formában valósítja meg, de az általános jellemzőket megtartja

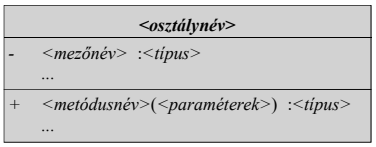
Objektumorientált tervezés: alapismeretek
Osztályok C++-ban

- A C++ programozási nyelv támogatja az objektumorientált programozást, noha alapvetően procedurális
  - a program indítási pontja (`main` függvény) mindig procedurális
- A C++ osztály szerkezete:

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

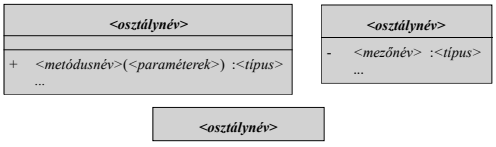

Objektumorientált tervezés: alapismeretek
Az osztálydiagram

- Az *UML osztálydiagram* (*class diagram*) a programban szereplő osztályok szerkezetét, illetve kapcsolatait definiálja
  - az osztálynak megadjuk a nevét, valamint mezőinek és metódusainak halmazát (típusokkal, paraméterekkel)
  - megadjuk a tagok láthatóságát látható (`+`), illetve rejtett (`-`) jelölésekkel



Objektumorientált tervezés: alapismeretek
Az osztálydiagram

- Az osztálydiagram egyszerűsíthető
  - elhagyhatjuk attribútumait, metódusait, vagy minden tagját



  - tagjainak szintaxisát is egyszerűsíthetjük (pl. paraméterek, visszatérési típus elhagyása)
- Felsorolási típusoknál csak a felvehető értékeket adjuk meg (láthatóság nélkül)

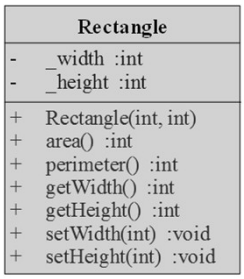
Objektumorientált tervezés: alapismeretek
Az osztálydiagram

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: alapismeretek
Az osztálydiagram

Tervezés:



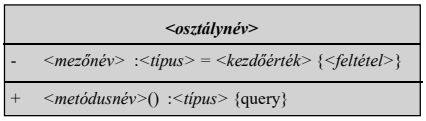
Objektumorientált tervezés: alapismeretek
Az osztálydiagram

Megvalósítás:

```
class Rectangle // téglalap típusa
{
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
    …
};
```

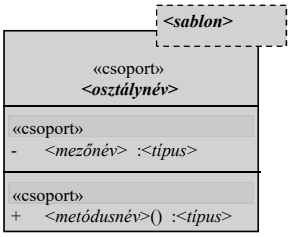

Objektumorientált tervezés: alapismeretek
Az osztálydiagram kiegészítései

- Az osztálydiagram számos kiegészítést tartalmazhat
  - feltételeket (invariánst) szabhatunk a mezőkre, metódusokra a **{…}** jelzéssel
    - speciálisan a lekérdező műveleteket a **{query}** jelzéssel jelölhetjük
  - jelölhetünk kezdőértéket a mezőkre (amelyet a konstruktor állít be)



Objektumorientált tervezés: alapismeretek
Az osztálydiagram kiegészítései

- adhatunk sablont a típusnak
- további tulajdonságokat jelölhetjük, illetve csoportba foglalásokat végezhetünk a **<<…>>** jelzéssel



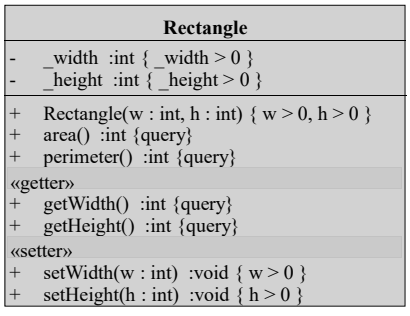
Objektumorientált tervezés: alapismeretek
Az osztálydiagram kiegészítései

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

- a téglalap méretei nem lehetnek negatívak, ezt mind a mezőknél, mint a paramétereknél megadhatjuk feltételként (a paraméternek jelöljük a nevét)
  - az implementációban is biztosítanunk kell a feltételek ellenőrzését
- a terület és kerület műveletek csak lekérdező műveletek, ezt jelölhetjük a tervben és a megvalósításban (**const**)
- a lekérdező, illetve beállító műveleteket külön csoportokba sorolhatjuk

Objektumorientált tervezés: alapismeretek
Az osztálydiagram kiegészítései

Tervezés:



Objektumorientált tervezés: alapismeretek
Az osztálydiagram kiegészítései

Megvalósítás:

```
class Rectangle
{
   …
public:
   Rectangle(double w, double h)
   {
      if (w < 0) … // feltétel ellenőrzése
      …
   }
   double area() const; // konstans művelet
   double perimeter() const;
   …
};
```

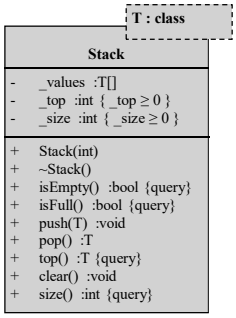

Objektumorientált tervezés: alapismeretek
Az osztálydiagram kiegészítései

Feladat: Valósítsuk meg a verem (**`Stack`**) adatszerkezetet aritmetikai reprezentáció mellett. Lehessen elemet behelyezni (**`push`**), kivenni (**`pop`**), kitörölni a teljes vermet (**`clear`**), lekérdezni a tetőelemet (**`top`**), üres-e (**`isEmpty`**), illetve tele van-e a verem (**`isFull`**), valamint mi az aktuális mérete (**`size`**).

- a vermet sablonos segítségével valósítjuk meg, így eltároljuk az adott típusú elemek tömbjét (**`_values`**), valamint az elemek számát (**`_top`**)
- a konstruktorban megadhatjuk a verem maximális méretét, így a megvalósításban dinamikus tömböt használunk
- gondoskodnunk kell a törlésről is destruktor segítségével

Objektumorientált tervezés: alapismeretek
Az osztálydiagram kiegészítései

Tervezés:



Objektumorientált tervezés: alapismeretek
Az osztálydiagram kiegészítései

Megvalósítás:

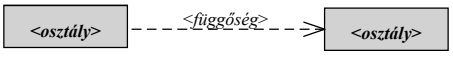
```
template <class T> // elemtípus sablonja
class Stack { // verem típus
private:
   T* _values;
   int _top;
   int _size;
public:
   Stack(int max); // konstruktor
   …
   bool push(T value); // elem behelyezése
   T pop(); // elem kivétele
   …
};
```


Objektumorientált tervezés: alapismeretek
Kapcsolatok

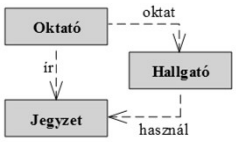
- A programot általában több osztály alkotja, az osztályok, illetve objektumok között pedig kapcsolatokat építhetünk fel, úgymint
  - *függőség*: szemantikai kapcsolat
  - *asszociáció* (*társítás*): szerkezeti kapcsolat (csak objektumok között)
    - *aggregáció* (*hivatkozás*): laza összekapcsolás
    - *kompozíció* (*tartalmazás*): szoros összekapcsolás
  - *általánosítás* (*generalizáció*): általános és speciális kapcsolata (csak osztályok között)
    - *megvalósítás*: szemantikai kapcsolat a fogalom és megvalósítója között (csak interfész és osztály között)

Objektumorientált tervezés: alapismeretek
Függőség

- A *függőség* (*dependency*) a legáltalánosabb kapcsolat, amelyben egy objektum (osztály) igénybe vehet funkcionalitást egy másik objektumtól (osztálytól)



- a függőség kialakítható metódushívással, példányosítással, hivatkozással (pl. visszatérési értékként, paraméterként)
- a függő osztály felületének megváltoztatása hatással van a másik osztály működésére



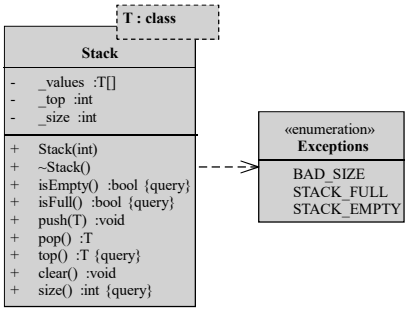
Objektumorientált tervezés: alapismeretek
Függőség

Feladat: Valósítsuk meg a verem (**`Stack`**) adatszerkezetet aritmetikai reprezentáció mellett. Lehessen elemet behelyezni (**`push`**), kivenni (**`pop`**), kitörölni a teljes vermet (**`clear`**), lekérdezni a tetőelemet (**`top`**), üres-e (**`isEmpty`**), illetve tele van-e a verem (**`isFull`**), valamint mi az aktuális mérete (**`size`**).

- amennyiben nem megfelelő az állapot a művelet végrehajtására (pl. üres, vagy tele), akkor azt jelezzük kivétellel
- a kivételeket felsorolási típussal adjuk meg, két kivétel az üres, illetve tele verem (**`STACK_FULL`**, **`STACK_EMPTY`**), illetve egy további a hibás veremméret (**`BAD_SIZE`**)

Objektumorientált tervezés: alapismeretek
Függőség

Tervezés:



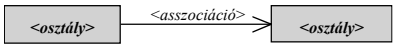
Objektumorientált tervezés: alapismeretek
Függőség

Megvalósítás:

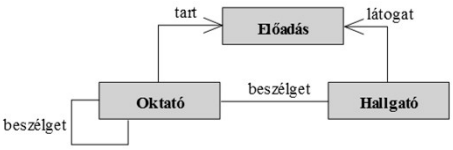
```
template <class T> // elemtípus sablonja
class Stack { // verem típus
   …
public:
   enum Exceptions { BAD_SIZE, STACK_FULL, … };
      // kivételek felsorolási típusa

   Stack(int size) {
      if (size <= 0) // ellenőrizzük a paramétert
         throw BAD_SIZE;
      …
   }
   …
};
```


Objektumorientált tervezés: alapismeretek
Asszociáció

- Az *asszociáció* (*association*) egy kommunikációs kapcsolat, ahol egy objektum üzenetet küldhet egy (vagy több) további objektumnak

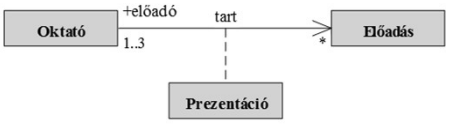


- a kommunikáció lehet irányított, irányítatlan (kétirányú), reflexív (saját osztály másik objektumára)



Objektumorientált tervezés: alapismeretek
Asszociáció

- az asszociációnak lévő osztályoknak lehet
  - *szerepe*, ami a relációbeli minőségükre utal, ezt az adott végponton jelöljük (ez is megjelölhető láthatósággal)
  - *multiplicitása*, ami a relációbeli számosságukra utal (lehet rögzített, tetszőleges érték, vagy intervallum)
- a relációnak lehetnek további tulajdonságai, amelyeket függőségként csatolhatunk



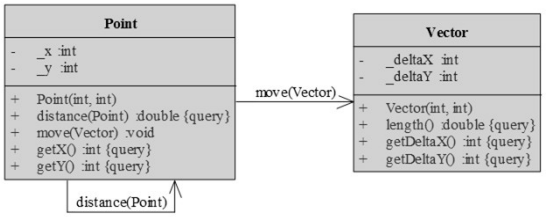
Objektumorientált tervezés: alapismeretek
Asszociáció

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: alapismeretek
Asszociáció

Tervezés:



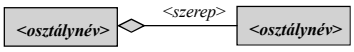
Objektumorientált tervezés: alapismeretek
Asszociáció

Megvalósítás:

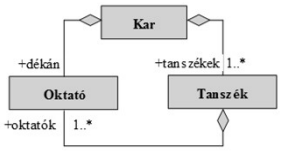
```
class Point {
public:
   Point(int x = 0, int y = 0) _x(x), _y(y) { }
   double distance(Point p) const {
      return sqrt(pow(abs(_x - p._x), 2)
                  + pow(abs(_y - p._y), 2));
   }
   void move(Vector v) {
      _x += v.getDeltaX();
      _y += v.getDeltaY();
   }
   ...
};
```


Objektumorientált tervezés: alapismeretek
Aggregáció

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



  - egy tartalmazási, rész/egész kapcsolatot jelent, állandó kapcsolat a két objektum között
  - a részt vevő objektumok életpályája különbözik, egymástól függetlenül is léteznek



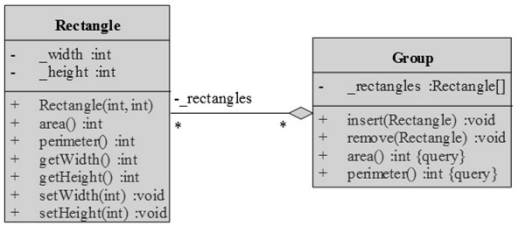
Objektumorientált tervezés: alapismeretek
Aggregáció

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 (**`_rectangle`**)

Objektumorientált tervezés: alapismeretek
Aggregáció

Tervezés:



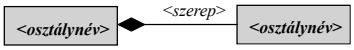
Objektumorientált tervezés: alapismeretek
Aggregáció

Megvalósítás:

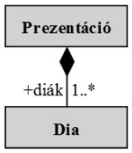
```
class Group {
private:
   vector<const Rectangle&> _rectangles;
      // vector-t választunk a megvalósításban, és
      // csak hivatkozásokat kezelünk
public:
   void insert(const Rectangle& rec);
      // csak hivatkozásokat veszünk át
   void remove(const Rectangle& rec);
   double area() const;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

Objektumorientált tervezés: alapismeretek
Kompozíció

- 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



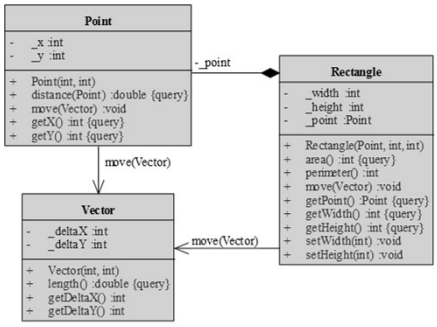
Objektumorientált tervezés: alapismeretek
Kompozíció

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: alapismeretek
Kompozíció

Tervezés:



Objektumorientált tervezés: alapismeretek
Kompozíció

Megvalósítás:

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

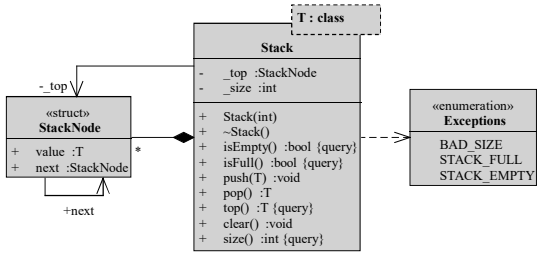

Objektumorientált tervezés: alapismeretek
Függőség

Feladat: Valósítsuk meg a verem (**`Stack`**) adatszerkezetet láncolt reprezentáció mellett. Lehessen elemet behelyezni (**`push`**), kivenni (**`pop`**), kitörölni a teljes vermet (**`clear`**), lekérdezni a tetőelemet (**`top`**), üres-e (**`isEmpty`**), illetve tele van-e a verem (**`isFull`**), valamint mi az aktuális mérete (**`size`**).

- a megvalósításhoz szükséges egy verem elem rekord (**`StackNode`**), amely tartalmazza az adatot (**`value`**), és hivatkozik a rákövetkező elemre (**`next`**)
- a veremben elég a tetőelem mutatóját (**`_top`**), és a méretet (**`_size`**) eltárolnunk
- amennyiben nem megfelelő az állapot a művelet végrehajtására (pl. üres, vagy tele), akkor azt jelezzük kivétellel

Objektumorientált tervezés: alapismeretek
Függőség

Tervezés:



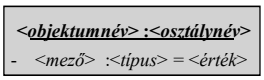
Objektumorientált tervezés: alapismeretek
Kompozíció

Megvalósítás:

```
template <class T>
struct StackNode { // verem elem rekordja
   T value
   StackNode* next;
}

template <class T>
class Stack { // verem osztálya
…
private:
   StackNode<T>* _top;
   int _size;
};
```

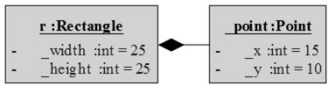

Objektumorientált tervezés: alapismeretek
Az objektumdiagram

- Az *UML objektumdiagram* (*object diagram*) a programban szereplő objektumokat, és kapcsolataikat ábrázolja
  - az objektumnak megadjuk nevét, osztályát, valamint mezőinek értékeit
  - ennek megfelelően az objektumdiagram mindig egy adott állapotban ábrázolja a rendszert, és tetszőlegesen sok lehet belőle
  - az objektumdiagram mindig megfelel az osztálydiagramnak



Objektumorientált tervezés: alapismeretek
Az objektumdiagram

Feladat: Példányosítsunk egy téglalapot (`r`), amely a 10, 15 koordinátákban helyezkedik el, és 25 széles, illetve magas.



Feladat: Példányosítsunk két téglalapot (20 szélesség, 30 magasság, valamint 10 szélesség, 10 magasság), amelyeket egy csoportba helyezünk.