Tervminták I. (Sablonfüggvény, Stratégia, Egyke, Látogató)

Testek térfogata

Lények túlélési versenye

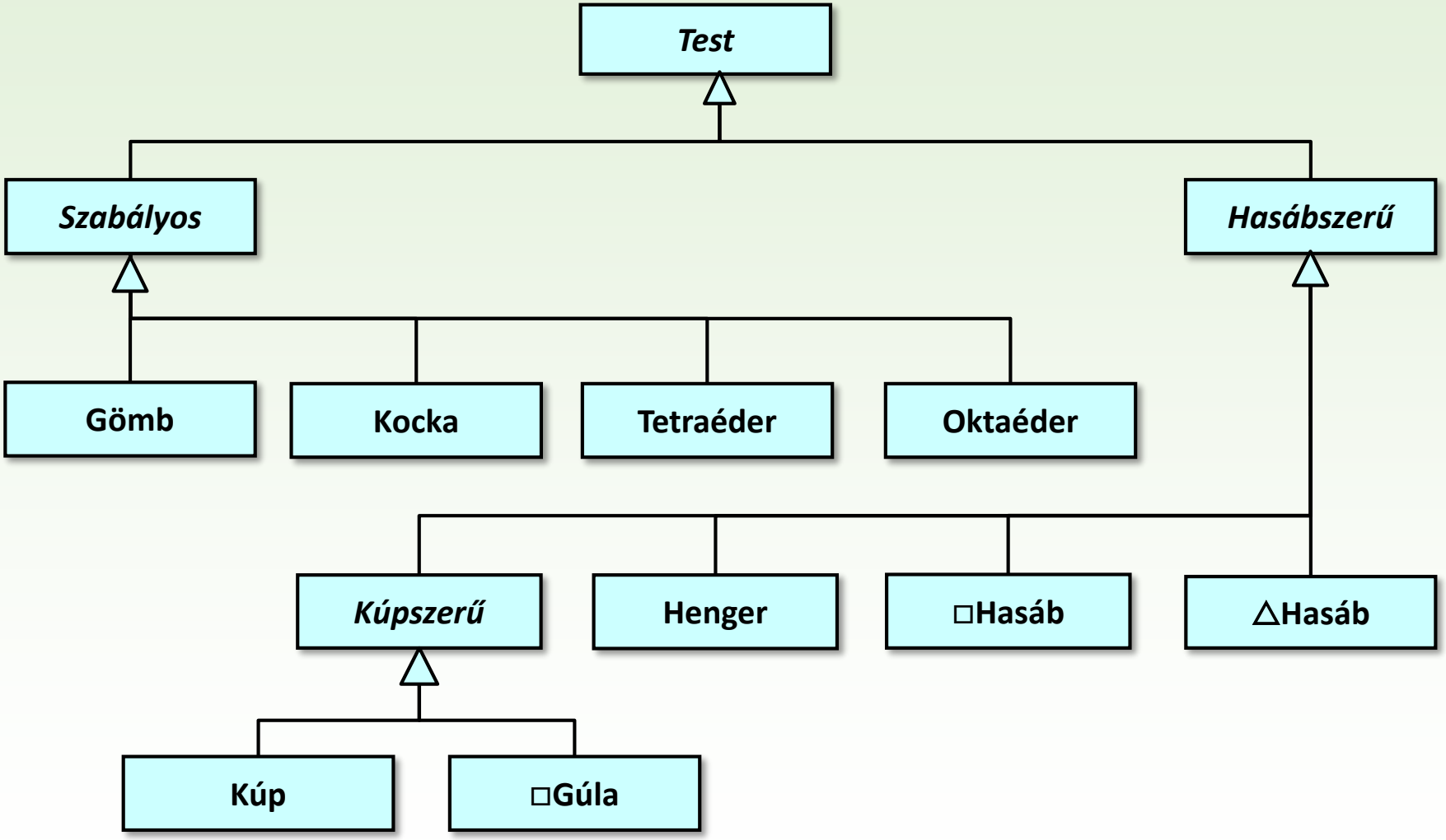
1.Feladat

Készítsünk programot, amellyel különféle testek térfogatát számolhatjuk ki, illetve megadhatjuk azt is, hogy az egyes testfajtákból hány objektumot hoztunk létre!

A lehetséges fajták:

- szabályos testek: gömb, kocka, tetraéder, oktaéder;
- hasáb jellegű testek: henger, négyzet alapú és szabályos háromszög alapú hasáb;
- gúla jellegű testek: kúp, négyzetes gúla.

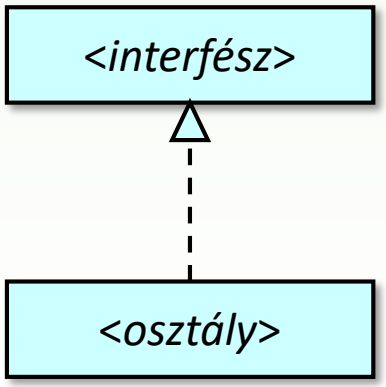
Osztálydiagram



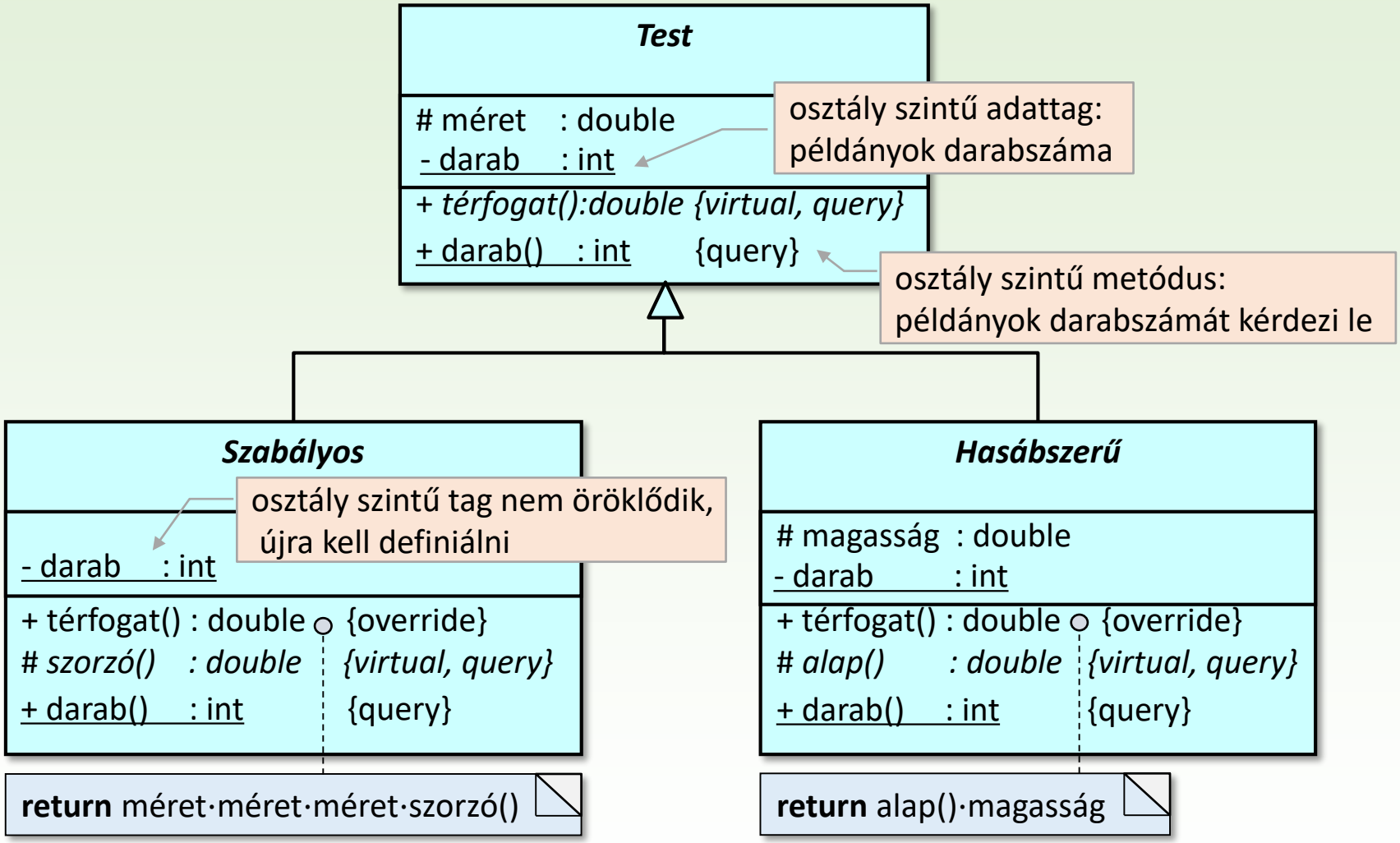
Absztrakt osztály, interfész

- Absztrakt (*abstract*) osztály az, amelyből nem példányosítunk objektumokat, kizárólag ősosztályként szolgálnak a származtatásokhoz.
  - az absztrakt osztály nevét dőlt betűvel kell írni.
- Nyelvi szempontból egy osztály attól lesz absztrakt, hogy
  - konstruktorai nem publikusak, vagy
  - legalább egy metódusa absztrakt, azaz nincs implementálva, és csak a származtatás során írjuk majd felül
    - az absztrakt metódust dőlt betűvel jelöljük

- Interfésznek, azaz tisztán absztrakt (*pure abstract*) osztálynak nevezzük azt az osztályt, amelyiknek egyetlen metódusa sincs implementálva.
- Amikor egy osztály a származtatása során egy interfész minden absztrakt metódusát implementálja, akkor megvalósítja az interfészt.



Absztrakt testek



A testek ősosztálya

```
class Shape
{
    public:
        virtual ~Shape();
        virtual double volume() const = 0;
        static int piece() { return _piece; }
    protected:
        Shape(double size);
        double _size;
    private:
        static int _piece;
};
```

virtuális a destruktor

egy absztrakt metódusú osztály absztrakt

osztályszintű metódus

egy védett konstruktorú osztály absztrakt

osztályszintű adattag

```
int Shape::_piece = 0;

Shape::Shape(double size){
    _size = size;
    ++_piece;
}
Shape::~Shape(){
    --_piece;
}
```

osztályszintű adattag kezdeti értékadása

Szabályos absztrakt test

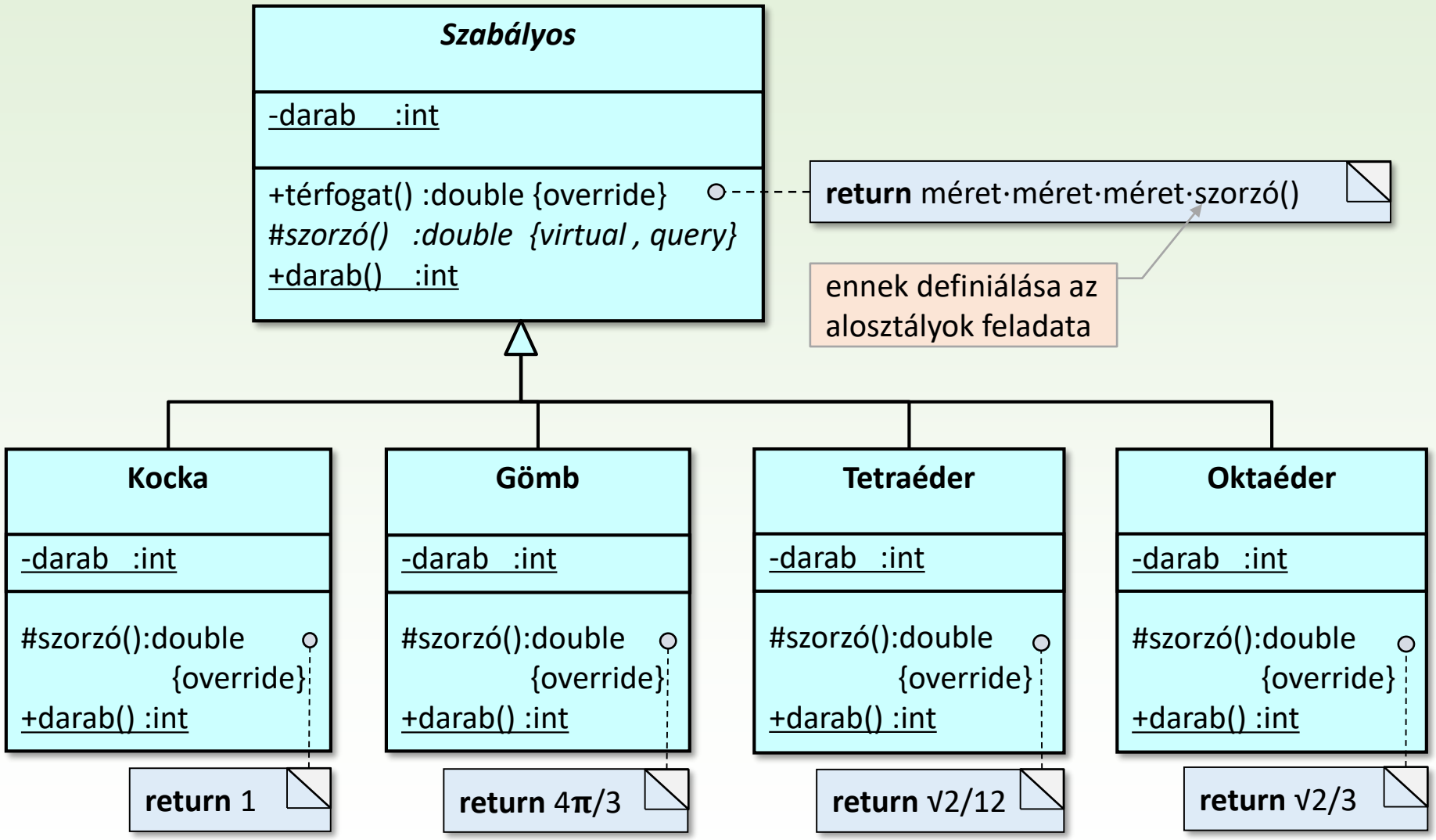
```
class Regular : public Shape{
    public:
        ~Regular();
        double volume() const override;
        static int piece() { return _piece; }
    protected:
        Regular(double size);
        virtual double multiplier() const = 0;
    private:
        static int _piece;
};
```

Enélkül a konstruktor automatikusan hívná az ősosztály üres (paraméter nélküli) konstruktorát, de olyan most nincs.

```
int Regular::_piece = 0;

Regular::Regular(double size) : Shape(size){
    ++_piece;
}
Regular::~Regular(){
    --_piece;
}
double Regular::volume() const {
    return _size * _size * _size * multiplier();
}
```

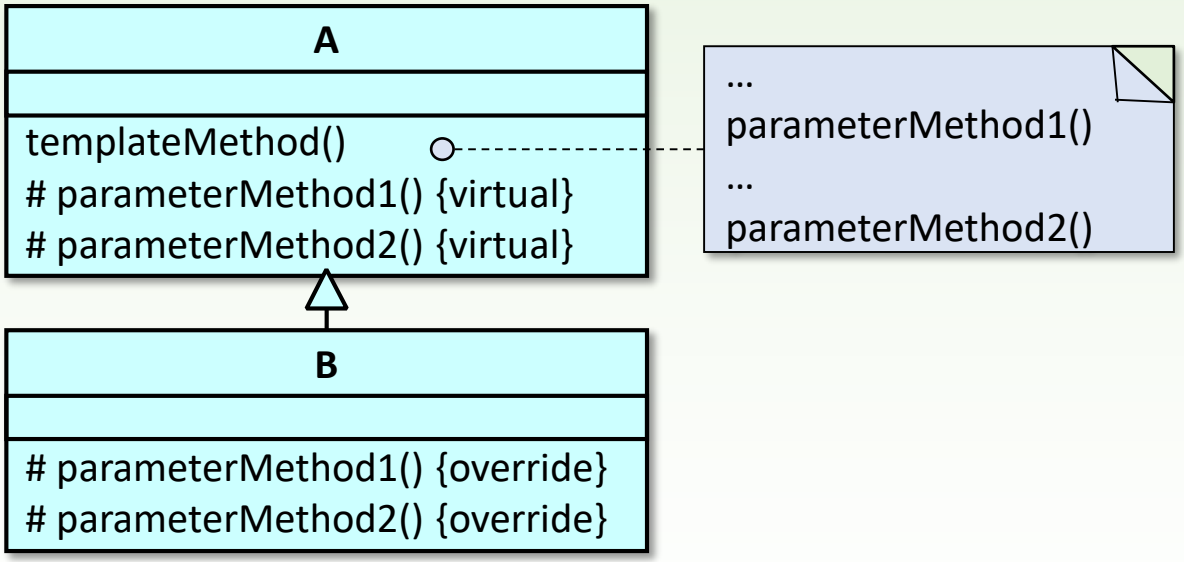
a destruktor automatikusan hívja az ősosztály destruktorát

Szabályos testek



Sablonfüggvény (template method) tervezési minta

- Egy algoritmust egy osztály metódusaként úgy adunk meg, hogy annak egyes lépéseit az algoritmus szerkezetének változtatása nélkül a futási idejű polimorfizmusra támaszkodva meg tudjuk változtatni.



A tervezési minták az objektum-alapú modellezést támogató osztálydiagram minták, amelyek az újrafelhasználhatóság, módosíthatóság, hatékonyság biztosításában játszanak szerepet.

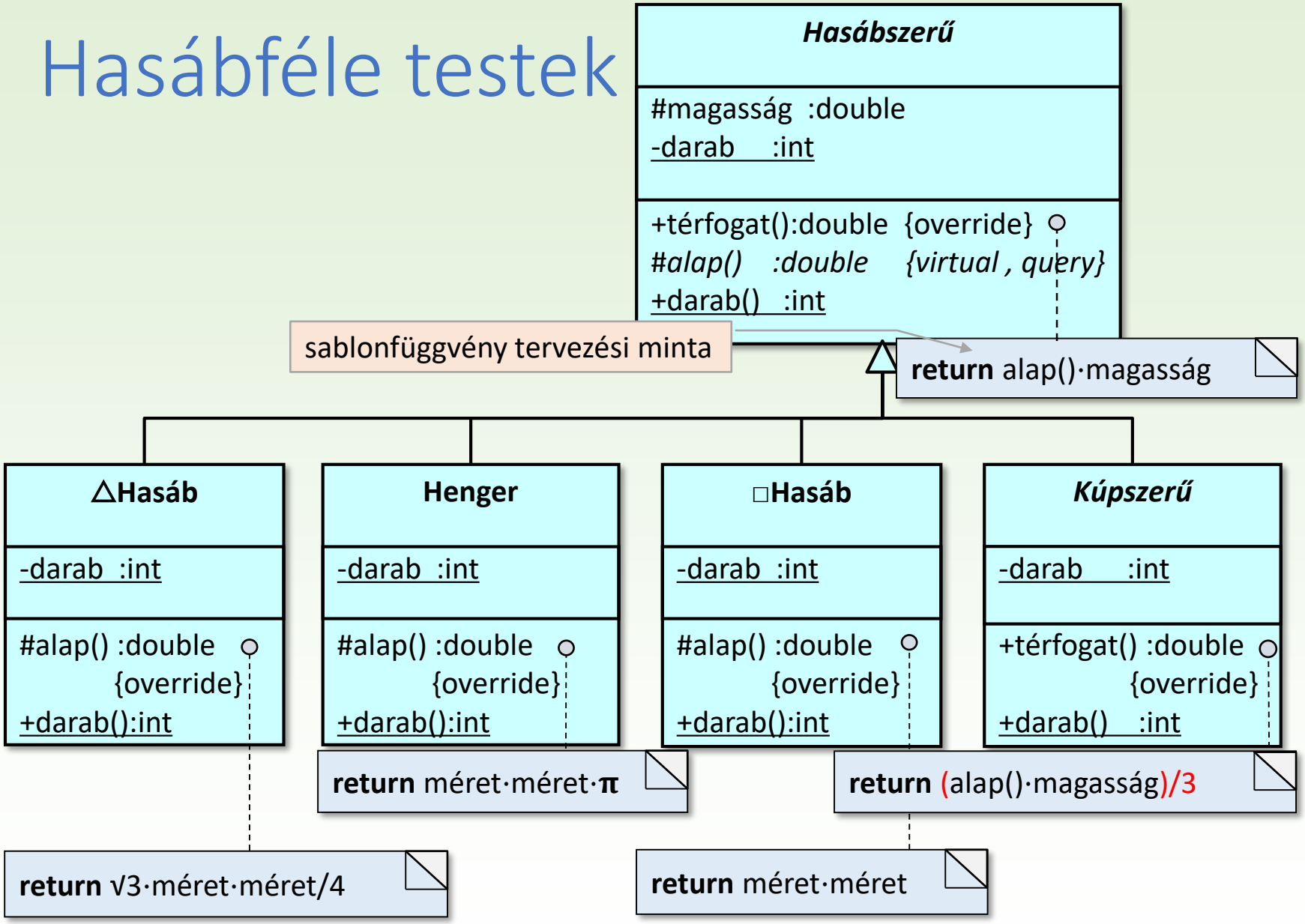
Gömb

```
class Sphere : public Regular
{
    public:
        Sphere(double size);
        ~Sphere();
        static int piece() { return _piece; }
    protected:
        double multiplier() const override { return _multiplier; }
    private:
        constexpr static double _multiplier = (4.0 * 3.14159) / 3.0;
        static int _piece;
};
```

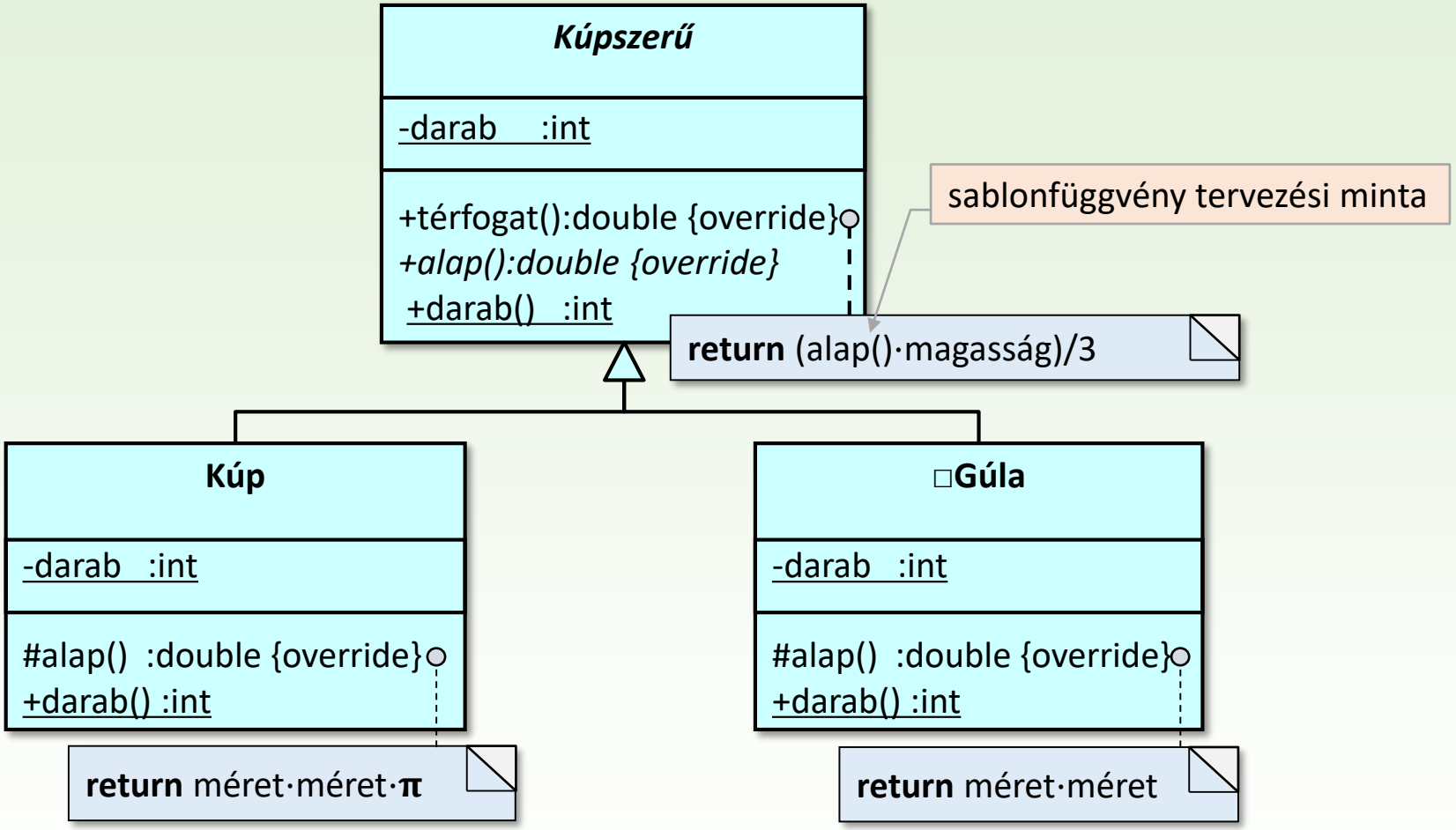
konstans osztályszintű kifejezés definiálása

```
int Sphere::_piece = 0;

Sphere::Sphere(double size) : Regular(size){
    ++_piece;
}
Sphere::~Sphere(){
    --_piece;
}
```

Hasábféle testek



Kúpszerű testek



Kritika a modellről:
Redundancia jelent meg a modellben: ilyen alapterület metódusokat már a hasábnál, hengernél definiáltunk.

Főprogram - populálás

```
#include <iostream>
#include <fstream>
#include <vector>
#include "shapes.h"

using namespace std;
Shape* create(ifstream &inp);
void statistic();

int main()
{
     ifstream inp("shapes.txt");
     if(inp.fail()) { cout << "Wrong file name!\n";  return 1; }

     int shape_number;
     inp >> shape_number;
     vector<Shape*> shapes(shape_number);

     for ( int i = 0; i < shape_number; ++i ){
          shapes[i] = create(inp);
     }
     inp.close();
```

shapes.txt

```
8
Cube 5.0
Cylinder 3.0 8.0
Cylinder 1.0 10.0
Tetrahedron 4.0
SquarePyramid 3.0 10.0
Octahedron 1.0
Cube 2.0
SquarePyramid 2.0 10.0
```

vector<Shape> nem lenne jó, mert
1. a Shape-nek nincs publikus üres konstruktora
2. a Shape absztrakt
3. a tömbbe úgyis a Shape leszármazottjainak referenciáit vagy pointereit kell betenni, ha a futási idejű polimorfizmust használni akarjuk

Test példányosítása

```
Shape* create(ifstream &inp)
{
    Shape *p;
    string type;
    inp >> type;
    double size, height;
    inp >> size;
    if ( type == "Cube" ){
        p = new Cube(size);
    }
    else if ( type == "Sphere" ){
        p = new Sphere(size);
    }
    else if ( type == "Tetrahedron" ){
        p = new Tetrahedron(size);
    }
    else if ( type == "Octahedron" ){
        p = new Octahedron(size);
    }
    else if ( type == "Cylinder" ){
        inp >> height;
        p = new Cylinder(size, height);
    }
    …
}
```

Lehetne a Shape osztályszintű metódusa is, ha látnia kellene a Shape rejtett elemeit.

A származtatás ténye miatt lehet értékül adni egy Shape* típusú változónak egy Cube* pointert

shapes.txt

```
8
Cube 5.0
Cylinder 3.0 8.0
Cylinder 1.0 10.0
Tetrahedron 4.0
SquarePyramid 3.0 10.0
Octahedron 1.0
Cube 2.0
SquarePyramid 2.0 10.0
```

Test példányosítása folyt.

```
Shape* create(ifstream &inp)
{
    …
    else if ( type == "SquarePrism" ){
        inp >> height;
        p = new SquarePrism(size, height);
    }
    else if ( type == "TriangularPrism" ){
        inp >> height;
        p = new TriangularPrism(size, height);
    }
    else if ( type == "Cone" ){
        inp >> height;
        p = new Cone(size, height);
    }
    else if ( type == "SquarePyramid" ){
        inp >> height;
        p = new SquarePyramid(size, height);
    }
    else{
        cout << "Unknown shape" << endl;
    }
    return p;
}
```

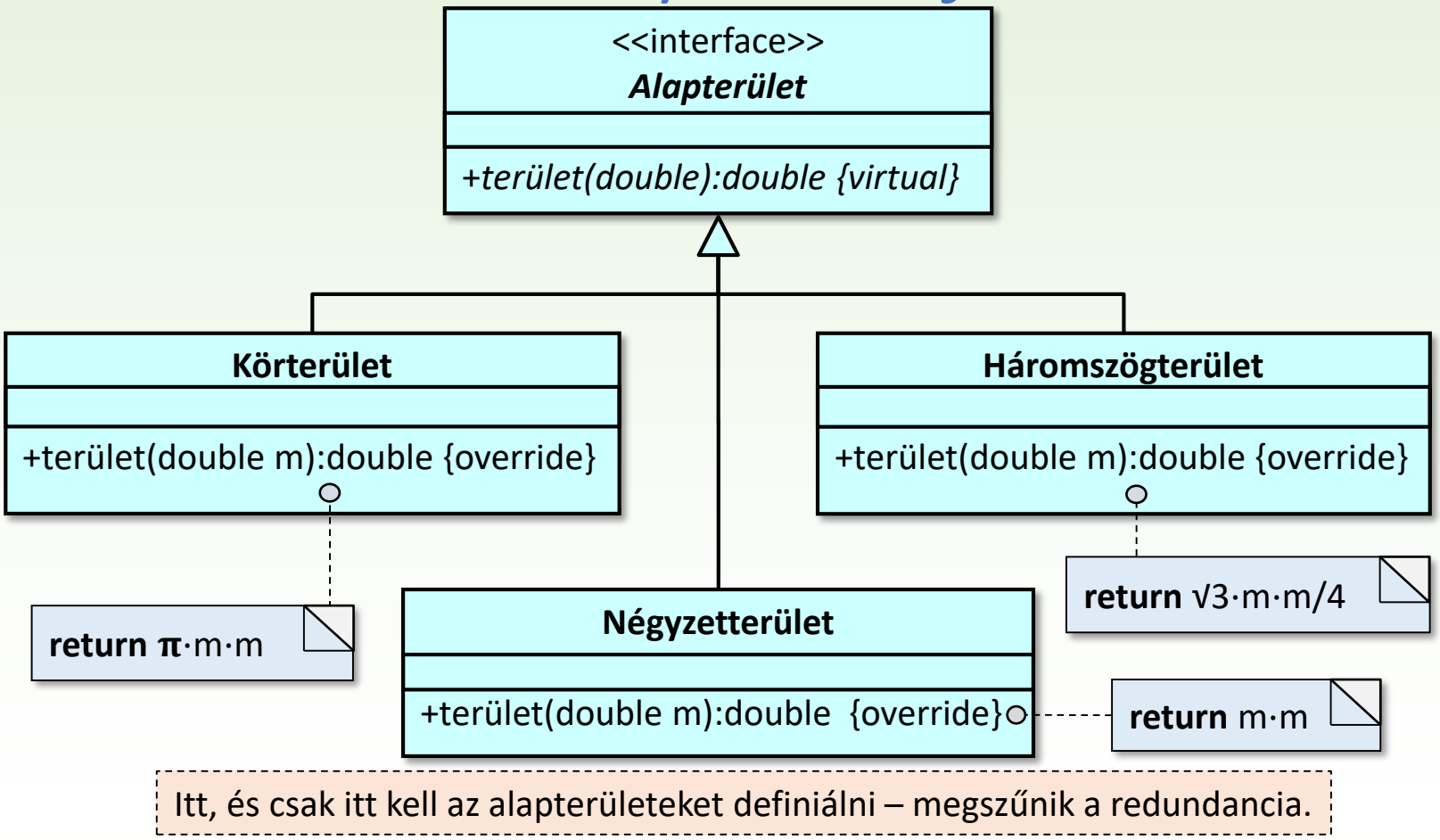
Főprogram folyt.

```
    …
    for ( Shape *p : shapes ){
        cout << p->volume() << endl;
    }

    statistic();

    for ( Shape *p : shapes ) delete p;

    statistic();
}

void statistic(){
    cout << Shape::piece()       << " " << Regular::piece()      << " "
         << Prismatic::piece()   << " " << Conical::piece()      << " "
         << Sphere::piece()      << " " << Cube::piece()         << " "
         << Tetrahedron::piece() << " " << Octahedron::piece()   << " "
         << Cylinder::piece()    << " " << SquarePrism::piece()  << " "
         << TriangularPrism::piece() << " "
         << Cone::piece()        << " " << SquarePyramid::piece()<<
    endl;
}
```

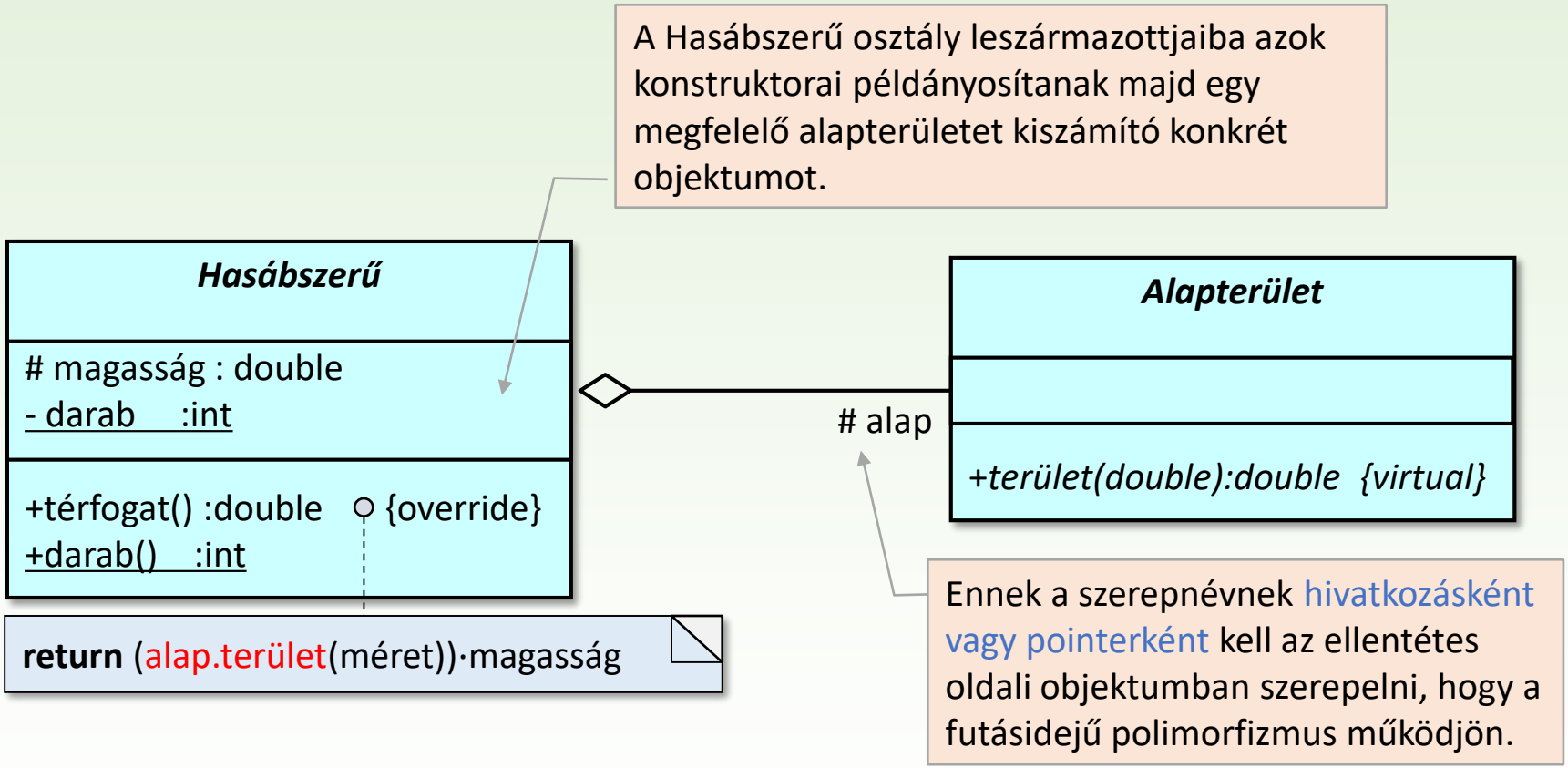
A futási idejű polimorfizmus miatt
a Shape ősosztály virtuális volume() metódusa
helyett a megfelelő test térfogatát számolja.

A futási idejű polimorfizmus miatt
a Test ősosztály virtuális destruktora
helyett a megfelelő test destruktora fut le.

Redundancia felszámolása: alapterületet számoló metódusok helyett objektumok

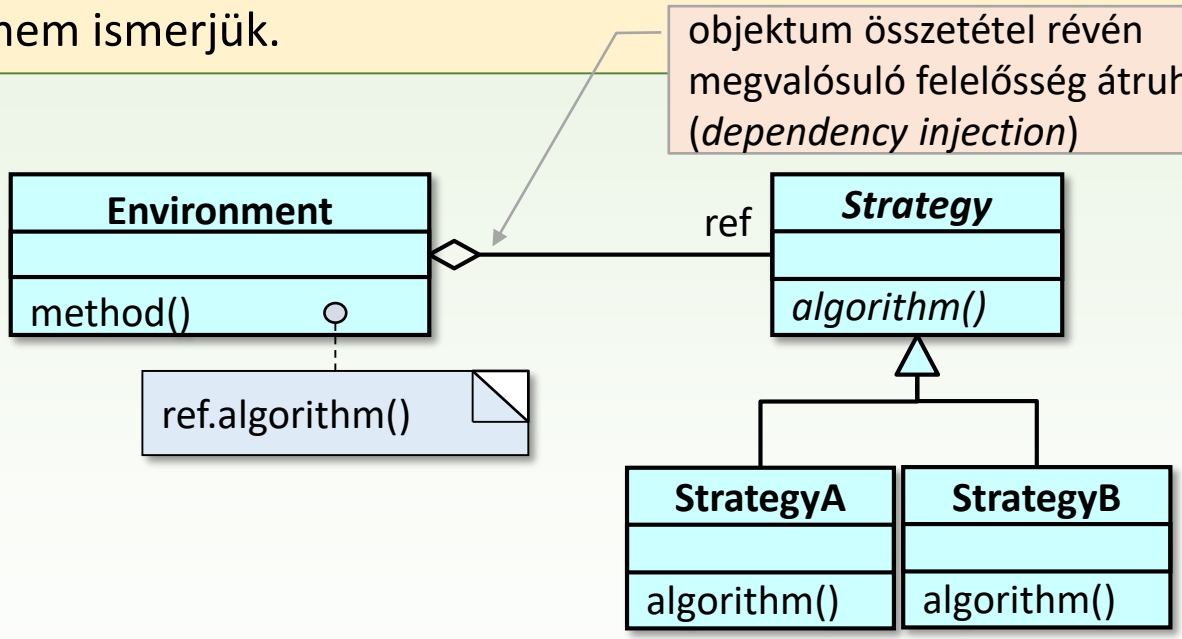


Függőség befecskendezés

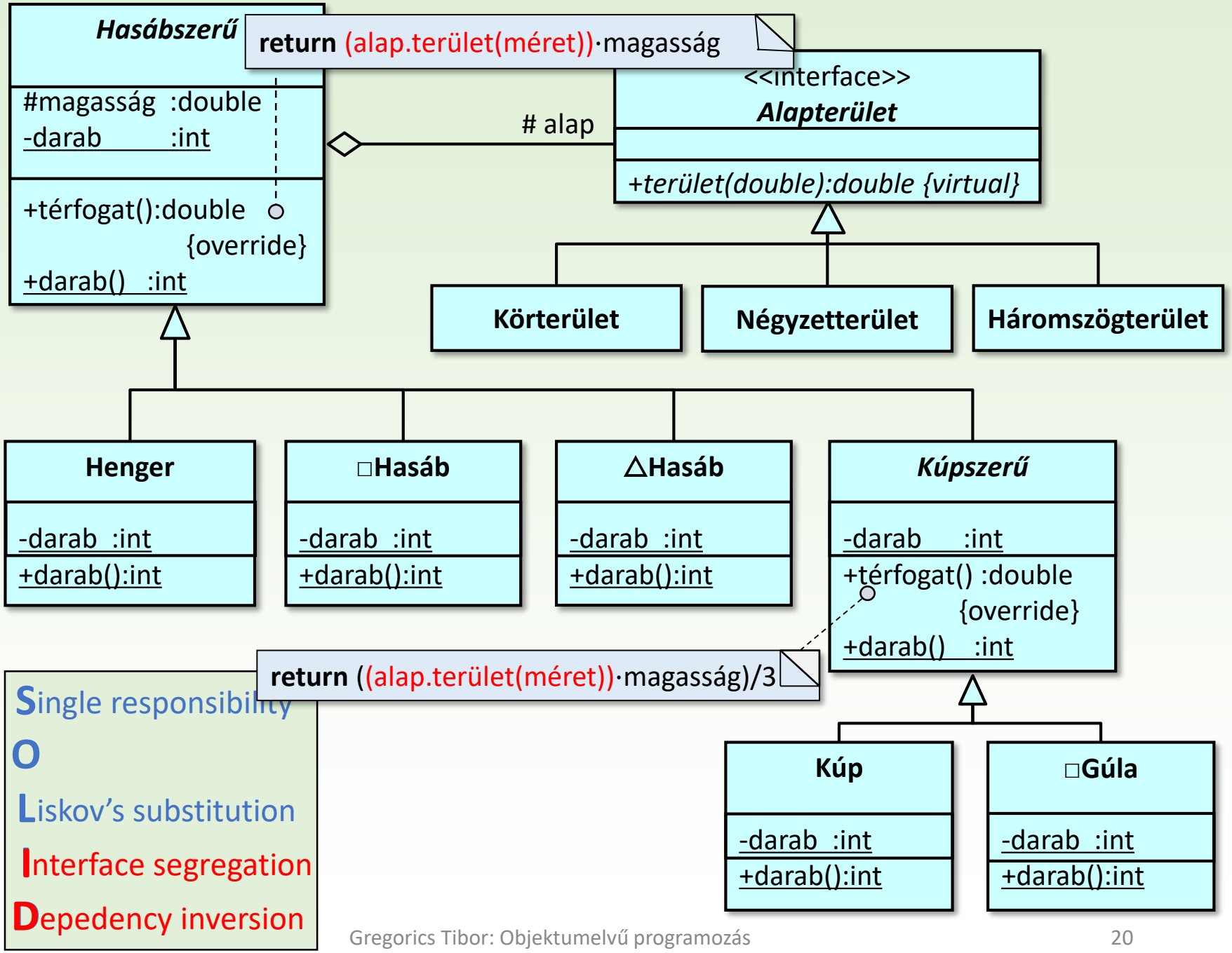


Stratégia (strategy) tervezési minta

- Egy algoritmus-családot definiálunk azért, hogy az egyik algoritmust felhasználhassuk, de hogy melyiket, azt a felhasználás kódjának leírásakor még nem ismerjük.



A tervezési minták az objektum-alapú modellezést támogató osztálydiagram minták, amelyek az újrafelhasználhatóság, módosíthatóság, hatékonyság biztosításában játszanak szerepet.


Hasábszerű
#magasság :double
-darab :int
+térfogat():double
{override}
+darab() :int
return (alap.terület(méret))·magasság
alap
<<interface>>
Alapterület
+terület(double):double {virtual}
Körterület
Négyzetterület
Háromszögterület
Henger
-darab :int
+darab():int
□Hasáb
-darab :int
+darab():int
△Hasáb
-darab :int
+darab():int
Kúpszerű
-darab :int
+térfogat() :double
{override}
+darab() :int
return ((alap.terület(méret))·magasság)/3
Kúp
-darab :int
+darab():int
□Gúla
-darab :int
+darab():int
Single responsibility
O
Liskov's substitution
Interface segregation
Depedency inversion

```
Cylinder::Cylinder(…) : Prismatic(…) {
    ++_piece; _basis = new CircleArea();
}
Cylinder::~Cylinder(){
    --_piece; delete _basis;
}
```

```
Cone::Cone(…) : Conical(…){
    ++_piece; _basis = new CircleArea();
}
Cone::~Cone(){
    --_piece; delete _basis;
}
```

```
SquarePrism::SquarePrism(…) : Prismatic(…){
    ++_piece; _basis = new SquareArea();
}
SquarePrism::~SquarePrism(){
    --_piece; delete _basis;
}
```

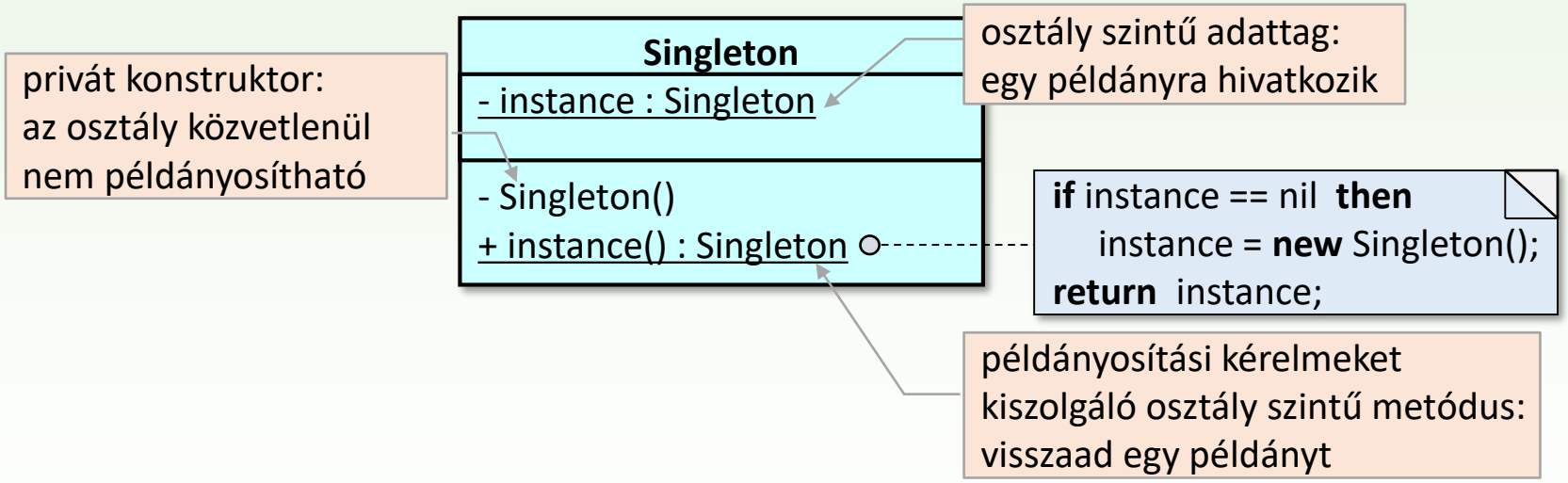
```
SquarePyramid::SquarePyramid(…) : Conical(…){
    ++_piece; _basis = new SquareArea();
}
SquarePyramid::~SquarePyramid(){
    --_piece; delete _basis;
}
```

```
TriangularPrism::TriangularPrism(…) : Prismatic(…){
    ++_piece; _basis = new TriangularArea();
}
TriangularPrism::~TriangularPrism(){
    --_piece; delete _basis;
}
```

Kritika a hatékonyságról:
A kód-redundancia megszűnt, de előállt egy memória pazarlás: például 5 henger és 3 kúp létrehozásához összesen 8 körterület objektumot kell példányosítani, pedig egy is elég lenne, amelyet mindenki használhatna.

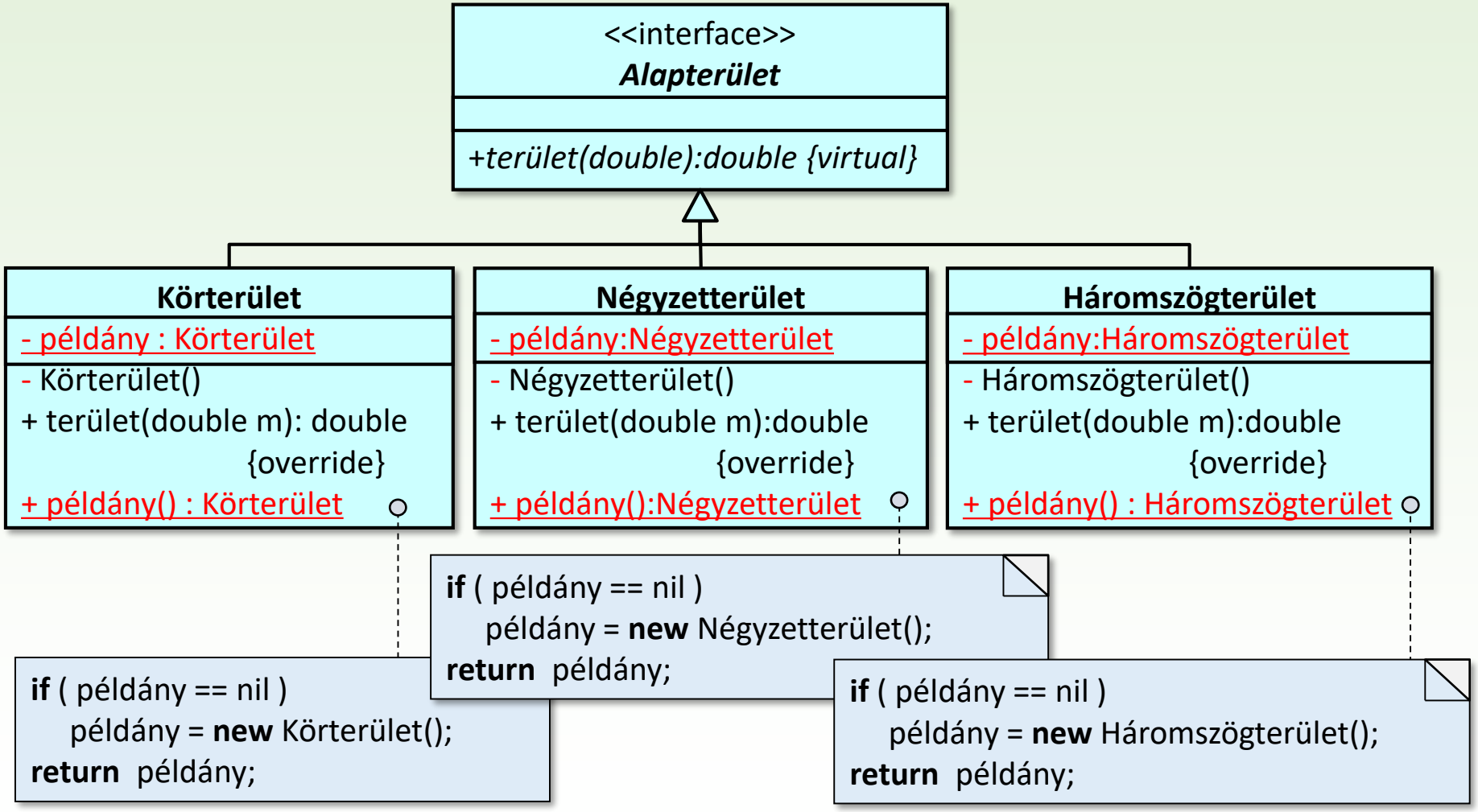
Egyke (singleton) tervezési minta

- Egy osztálynak legfeljebb egy objektumát akarjuk példányosítani függetlenül a példányosítási kérelmek számától.



A tervezési minták az objektum-alapú modellezést támogató osztálydiagram minták, amelyek az újrafelhasználhatóság, módosíthatóság, hatékonyság biztosításában játszanak szerepet.

Egy adott alapterület kiszámolásához elég csak egyetlen objektum



Négyzet terület

```
class SquareArea : public Area
{
    public:
        double area(double m) const override {
            return m * m;
        }
        static SquareArea *instance();
    private:
        static SquareArea *_instance;
        SquareArea () {}
};
```

sehová sem mutató pointer

```
SquareArea* SquareArea::_instance = nullptr;

SquareArea* SquareArea::instance()
{
    if ( _instance == nullptr ) _instance = new SquareArea();
    return _instance;
}
```

```
Cylinder::Cylinder(…) : Prismatic(…){
    ++_piece; _basis = CircleArea::instance();
}
Cylinder::~Cylinder(){
    --_piece;
}
```

```
Cone::Cone(…) : Conical(…){
    ++_piece; _basis = CircleArea::instance();
}
Cone::~Cone(){
    --_piece;
}
```

_basis = **new** CircleArea() helyett

```
SquarePrism::SquarePrism(…) : Prismatic(…){
    ++_piece; _basis = SquareArea::instance();
}
SquarePrism::~SquarePrism(){
    --_piece;
}
```

```
SquarePyramid::SquarePyramid(…) : Conical(…){
    ++_piece; _basis = SquareArea::instance();
}
SquarePyramid::~SquarePyramid(){
    --_piece;
}
```

```
TriangularPrism ::TriangularPrism(…) : Prismatic(…){
    ++_piece; _basis = TriangularArea::instance();
}
TriangularPrism::~TriangularPrism(){
    --_piece;
}
```

2. Feladat

Készítsünk programot, amellyel lények túlélési versenyét modellezhetjük.

A lények három faj (zöldike, buckabogár, tocsogó) valamelyikéhez tartoznak. Minden lénynek van neve (sztring), ismert a faja, és az aktuális életereje (egész szám). A versenyen induló lények sorban egymás után egy olyan pályán haladnak végig, ahol három féle (homokos, füves, mocsaras) terep váltakozik. Amikor egy lény keresztül halad egy terepen, akkor a lény és a terep fajtájától függően átalakíthatja a terepet, miközben változik az életereje. Ha az életereje nulla vagy annál kevesebb értékű lesz, a lény elpusztul. Adjuk meg a pályán végig jutó, azaz életben maradt lények neveit!

- ***Zöldike**: füvön az életereje eggyel nő, homokon kettővel csökken, mocsárban eggyel csökken; a mocsaras terepet fűvé alakítja, a másik két terep fajtát nem változtatja meg.*
- ***Buckabogár**: füvön az ereje kettővel csökken, homokon hárommal nő, mocsárban néggyel csökken; a füvet homokká, a mocsarat fűvé alakítja, de a homokot nem változtatja meg.*
- ***Tocsogó**: füvön az életereje kettővel, homokon öttel csökken, mocsárban hattal nő; a füvet mocsárrá alakítja, a másik két fajta terepet nem változtatja meg.*

Megoldási terv

A : pálya: Terep^m, lény: Lény^n, túlélők: String*

Ef : lény = lény$_0$ $\wedge$ pálya = pálya$_0$

Uf : $\forall i \in [1..n]$: (lény[i], pálya$_i$) = áthalad(lény$_0$[i], pálya$_{i-1}$)

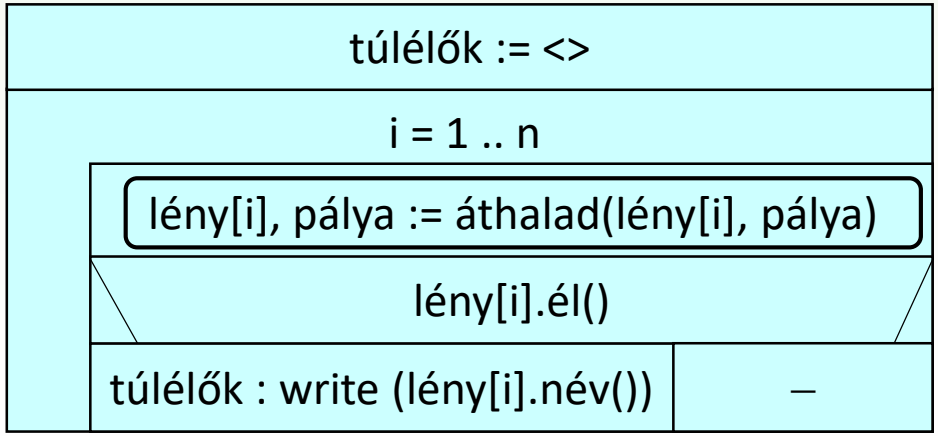
$\wedge$ pálya = pálya$_n$

$\wedge$ túlélők = $\bigoplus_{i=1..n}$ <lény[i].név()>

lény[i].él()

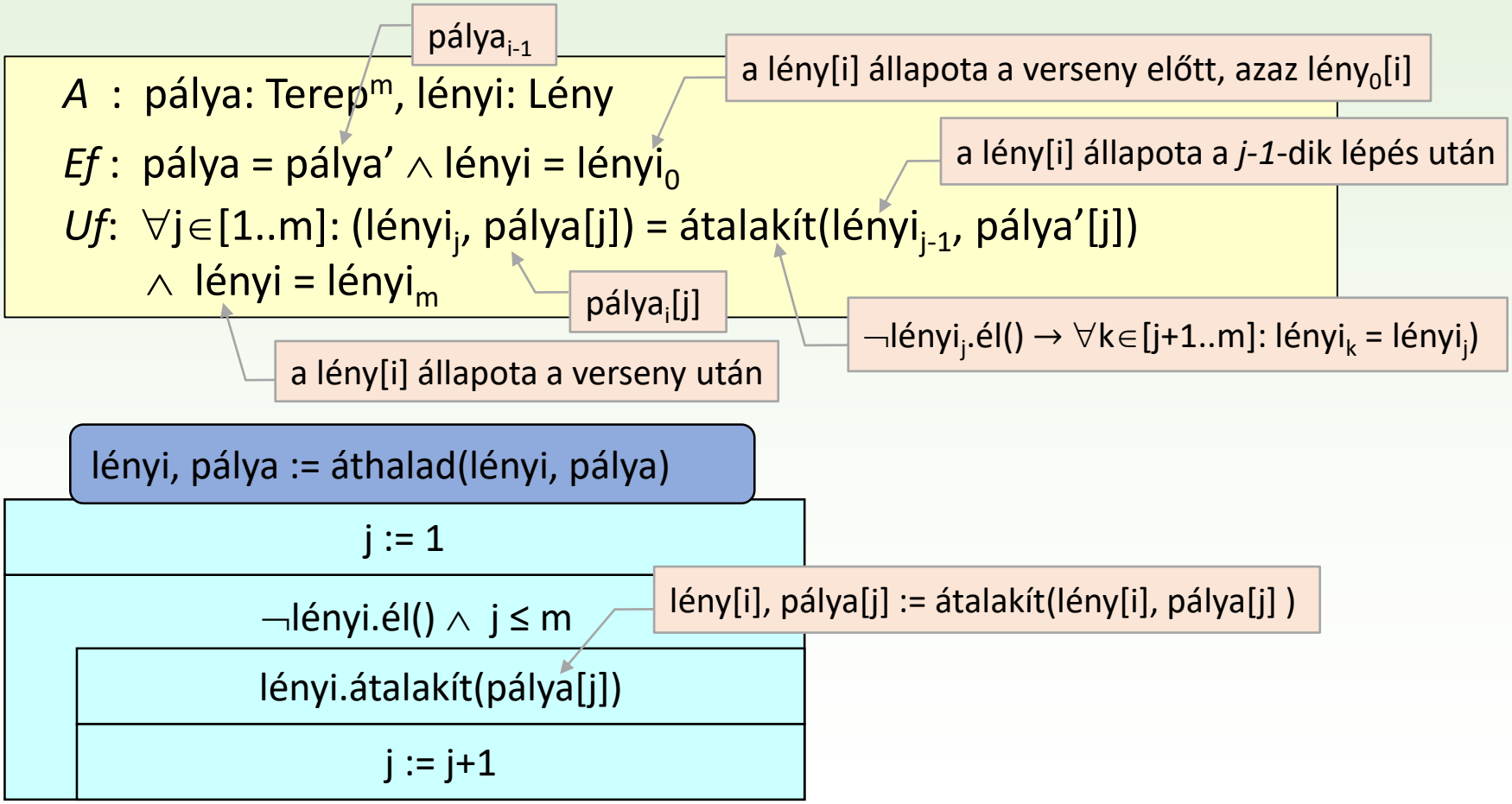
a pálya az *i*-dik lény áthaladása előtt: azaz a pálya *i-1*-dik állapota

a pálya az *i*-dik lény áthaladása után: azaz a pálya *i*-dik állapota



Egy lény áthaladása

Az *i*-dik lény a pálya$_{i-1}$ mezőin egyesével lépked (amíg él), és minden lépése megváltoztathatja az adott terepet, miközben a lény maga is átalakul.



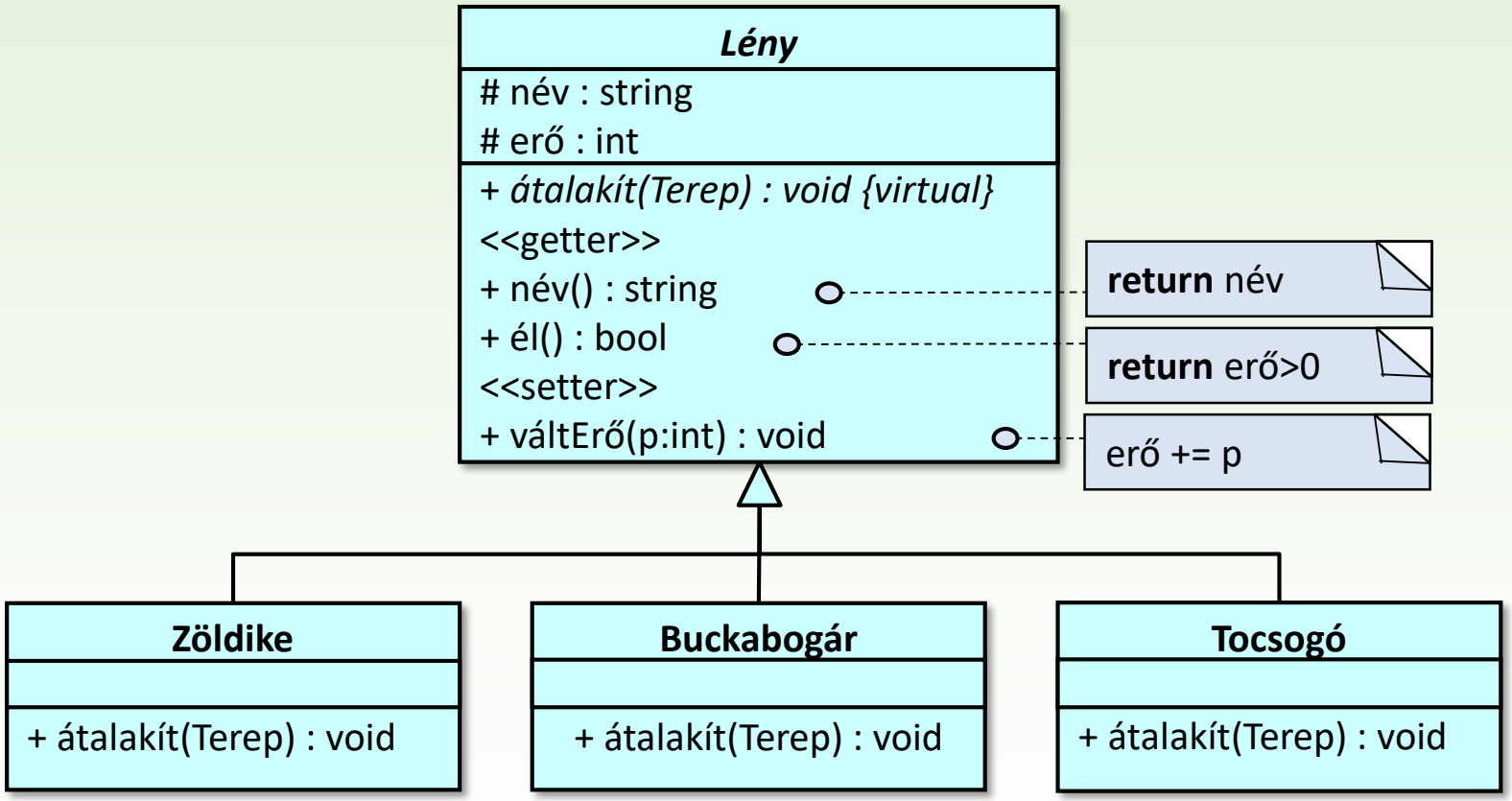
Főprogram

```
// Competition
for( int i=0; i < n; ++i ){
     for( int j=0; creature[i]->alive() && j < m; ++j ){
          creature[i]->transmute(court[j]);
     }
     if (creature[i]->alive() ) cout << creature[i]->name() << endl;
}
```

main.cpp

Ahhoz, hogy ilyen egyszerűen írhassuk le, jól jönne a futási idejű polimorfizmus, azaz hogy a creature[i] aktuális típusától függjön a transmute() működése.

Lények származtatása



Lények

```
class Creature{
protected:
     int _power;
     std::string _name;
     Creature (const std::string &str, int e = 0)
     :_name(str), _power(e) {}
public:
     std::string name() const { return _name; }
     bool alive() const { return _power > 0; }
     void changePower(int e) { _power += e; }
     virtual void transmute(int &court) = 0;
     virtual ~Creature () {}
};
```

creature.h

a terep típusa: integer

```
class Greenfinch : public Creature {
public:
     Greenfinch(const std::string &str, int e = 0) : Creature(str, e){}
     void transmute(int &court) override;
};
class DuneBeetle : public Creature {
public:
     DuneBeetle(const std::string &str, int e = 0) : Creature(str, e){}
     void transmute(int &court) override;
};
class Squelchy : public Creature {
public:
     Squelchy(const std::string &str, int e = 0) : Creature(str, e){}
     void transmute(int &court) override;
};
```

creature.h

A lények és a terepek kölcsönhatása

zöldikék	életerő változás	terepváltozás
homokban	-2	-
fűben	+1	-
mocsárban	-1	fű

buckabogarak	életerő változás	terepváltozás
homokban	+3	-
fűben	-2	homok
mocsárban	-4	fű

tocsogók	életerő változás	terepváltozás
homokban	-5	-
fűben	-2	mocsár
mocsárban	+6	-

Lények átalakít() metódusai

```
void Greenfinch::transmute(int &court) {
    if ( alive() ){
        switch(court){
            case 0: _power-=2; break;
            case 1: _power+=1; break;
            case 2: _power-=1; court = 1; break;
        }
    }
}
```

```
void DuneBeetle::transmute(int &court) {
    if (alive() ){
        switch(court){
            case 0: _power+=3; break;
            case 1: _power-=2; court = 0; break;
            case 2: _power-=4; court = 1; break;
        }
    }
}
```

```
void Squelchy::transmute(int &court) {
    if (alive() ){
        switch(court){
            case 0: _power-=5; break;
            case 1: _power-=2; court = 2; break;
            case 2: _power+=6; break;
        }
    }
}
```

Single responsibility

Open-closed

Liskov's substitution

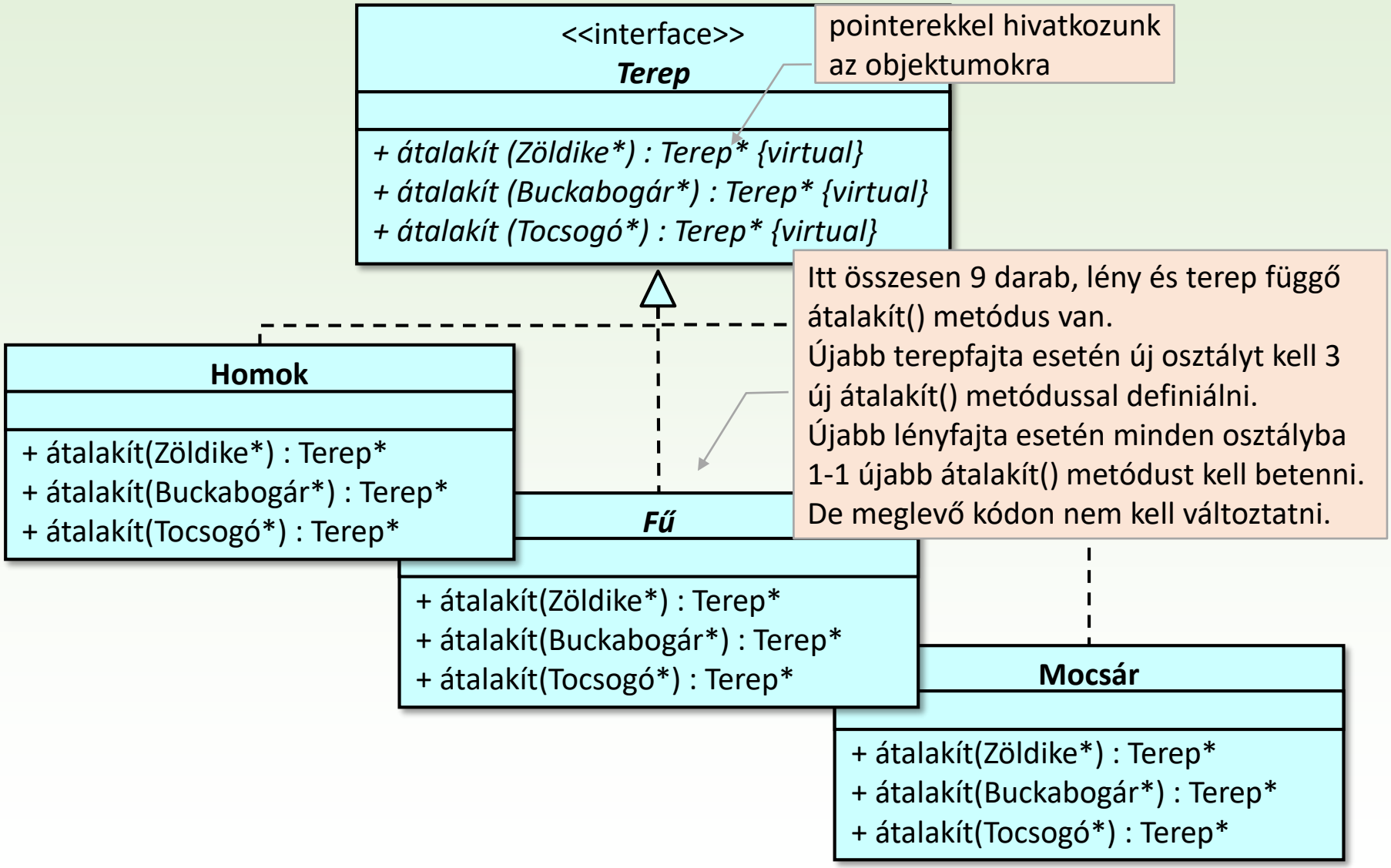
Interface segregation

Depedency inversion

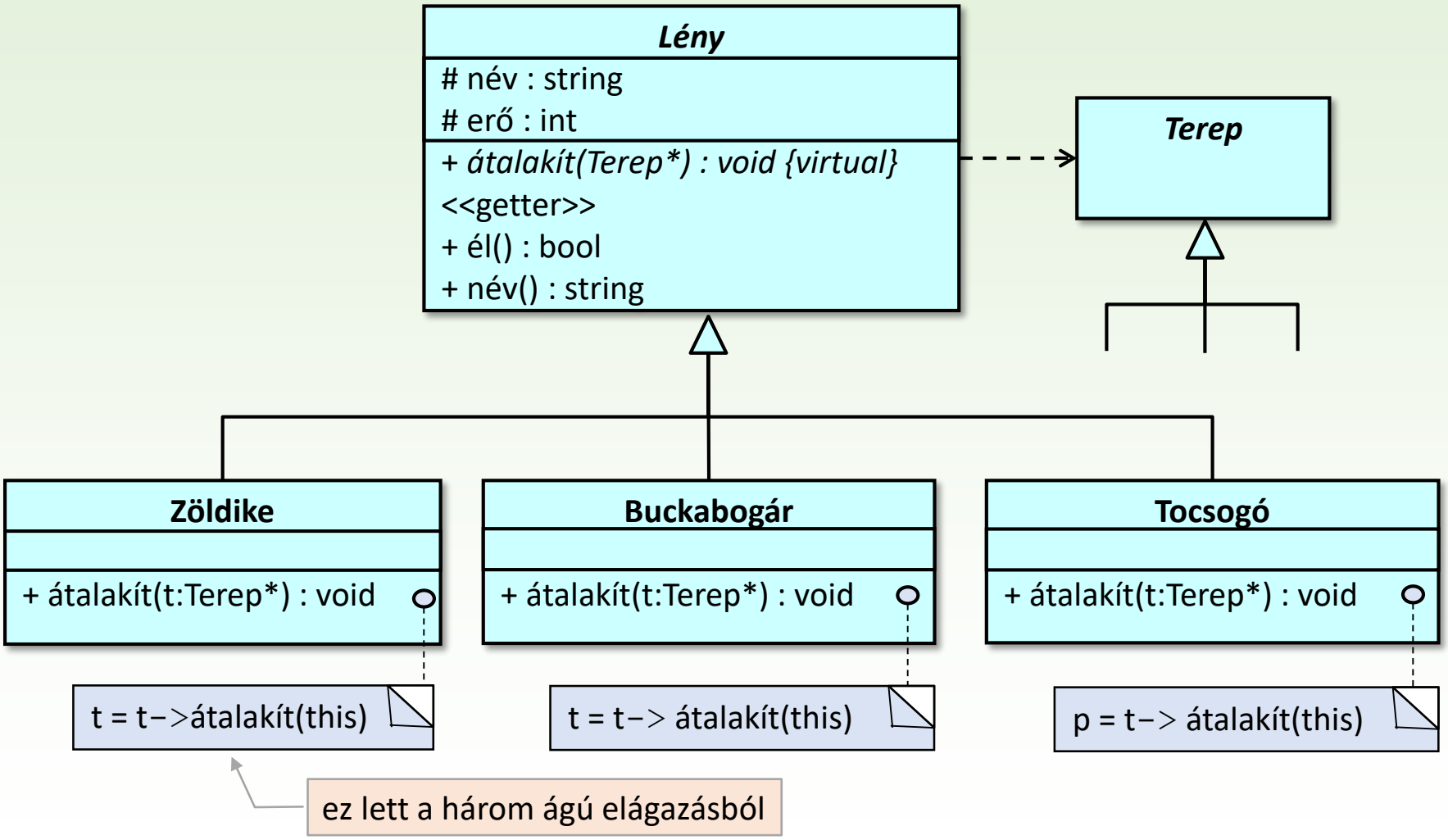
Kritika a kódról:

- rosszul olvasható:
  - a terepeket azonosító egész számok nem „beszédes" jelölések
- nem rugalmas:
  - újabb terepfajta bevezetése esetén az elágazásokat módosítani kell, azaz meglévő kódon kell változtatni

Terepek származtatása

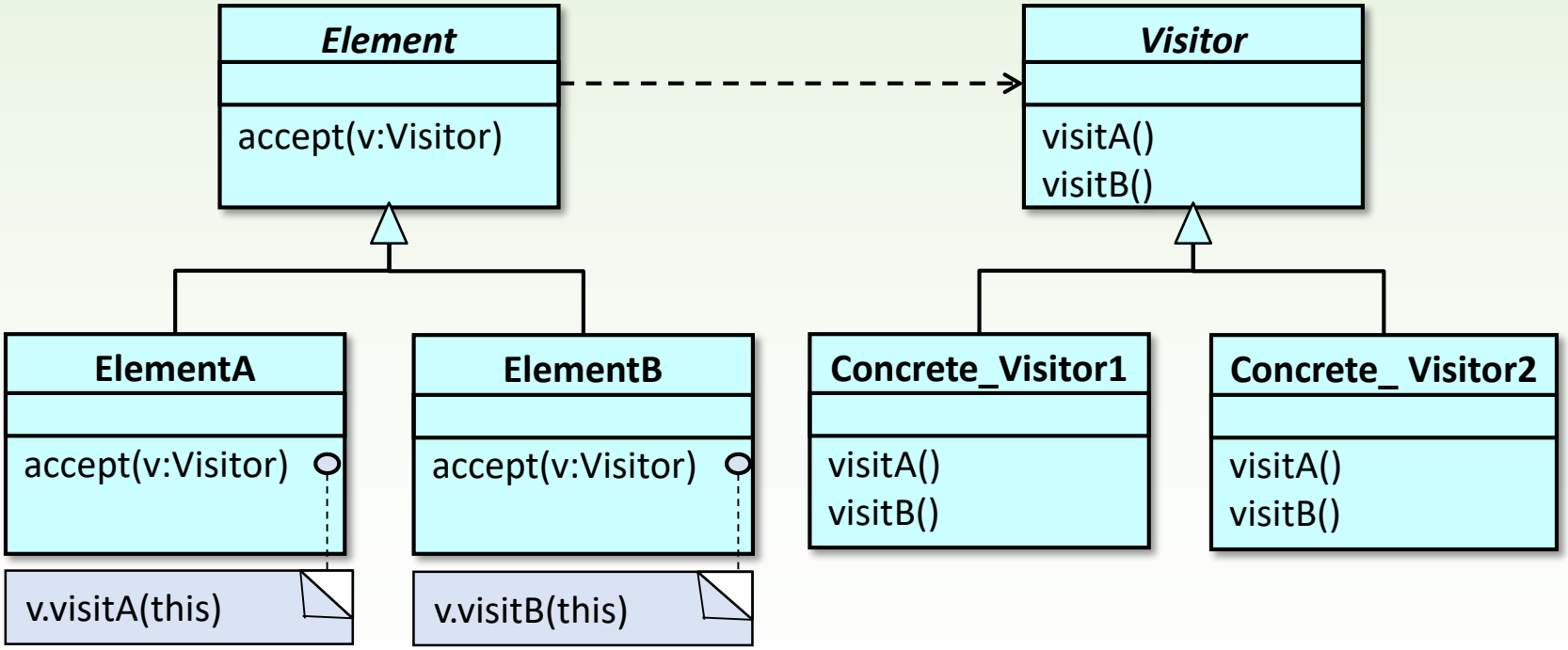


Lények osztálydiagramja újra



Látogató (visitor) tervezési minta

- Amikor egy metódus működése attól függ, hogy egy objektum-készlet melyik objektumát kapja meg paraméterként, de nem akarunk a készlet objektumainak számától függő elágazást használni metódus leírásához.



A tervezési minták az objektum-alapú modellezést támogató osztálydiagram minták, amelyek az újrafelhasználhatóság, módosíthatóság, hatékonyság biztosításában játszanak szerepet.

Lények látogatókkal

```
class Creature{
protected:
     int _power;
     std::string _name;
     Creature (const std::string &str, int e = 0)
     :_name(str), _power(e) {}
public:
     std::string name() const { return _name; }
     bool alive() const { return _power > 0; }
     void changePower(int e) { _power += e; }
     virtual void transmute(Ground* &court) = 0;
     virtual ~Creature () {}
};
```

creature.h

```
class Greenfinch : public Creature {
public:
     Greenfinch(const std::string &str, int e = 0) : Creature(str, e){}
     void transmute(Ground* &court) override {court = court->transmute(this);}
};
class DuneBeetle : public Creature {
public:
     DuneBeetle(const std::string &str, int e = 0) : Creature(str, e){}
     void transmute(Ground* &court) override {court = court->transmute(this);}
};
class Squelchy : public Creature {
public:
     Squelchy(const std::string &str, int e = 0) : Creature(str, e){}
     void transmute(Ground* &court) override {court = court->transmute(this);}
};
```

Terep és lény függő metódusok

```
Ground* Sand::transmute(Greenfinch *p){
    p->changePower(-2);   return this;
}
Ground* Sand::transmute(DuneBeetle *p){
    p->changePower(3);    return this;
}
Ground* Sand::transmute(Squelchy *p){
    p->changePower(-5);   return this;
}
```

```
Ground* Grass::transmute(Greenfinch *p){
    p->changePower(1);    return this;
}
Ground* Grass::transmute(DuneBeetle *p){
    p->changePower(-2);   return new Sand;
}
Ground* Grass::transmute(Squelchy *p){
    p->changePower(-2);   return new Marsh;
}
```

```
Ground* Marsh::transmute(Greenfinch *p){
    p->changePower(-1);   return new Grass;
}
Ground* Marsh::transmute(DuneBeetle *p){
    p->changePower(-4);   return new Grass;
}
Ground* Marsh::transmute(Squelchy *p){
    p->changePower(6);    return this;
}
```

ground.cpp

Kritika a hatékonyságról:
Ugyanazon terep-objektumból (lásd hányszor hívódna meg itt a **new** Grass) nagyon sok jöhet létre. Elég lenne egyetlen homok-, mocsár-, és fű objektum.

Terep osztályok, mint egykék

```
Ground* Sand::transmute(Greenfinch *p){
    p->changePower(-2);   return this;
}
Ground* Sand::transmute(DuneBeetle *p){
    p->changePower(3);    return this;
}
Ground* Sand::transmute(Squelchy *p){
    p->changePower(-5);   return this;
}
```

```
Ground* Grass::transmute(Greenfinch *p){
    p->changePower(1);    return this;
}
Ground* Grass::transmute(DuneBeetle *p){
    p->changePower(-2);   return Sand::instance();
}
Ground* Grass::transmute(Squelchy *p){
    p->changePower(-2);   return Marsh::instance();
}
```

new Sand
new Grass
new Marsh helyett

```
Ground* Marsh::transmute(Greenfinch *p){
    p->changePower(-1);   return Grass::instance();
}
Ground* Marsh::transmute(DuneBeetle *p){
    p->changePower(-4);   return Grass::instance();
}
Ground* Marsh::transmute(Squelchy *p){
    p->changePower(6);    return this;
}
```

ground.cpp

Feladat felpopulálása

```
4
S plash 20
G greenish 10
D bug 15
S sponge 20
10
0 2 1 0 2 0 1 0 1 2
```
input.txt

```
ifstream f("input.txt");
if(f.fail()) { cout << "Wrong file name!\n"; return 1;}

// populating creatures
int n; f >> n;
vector<Creature*> creature(n);
for( int i=0; i<n; ++i ){
     char ch; string nev; int p;
     f >> ch >> nev >> p;
     switch(ch){
          case 'G' : creature[i] = new Greenfinch(nev, p); break;
          case 'D' : creature[i] = new DuneBeetle(nev, p); break;
          case 'S' : creature[i] = new Squelchy(nev, p);   break;
     }
}
// populating courts
int m; f >> m;
vector<Ground*> court(m);
for( int j=0; j<m; ++j ) {
     int k; f >> k;
     switch(k){
          case 0 : court[j] = Sand::instance();  break;
          case 1 : court[j] = Grass::instance(); break;
          case 2 : court[j] = Marsh::instance(); break;
     }
}
```

new Sand
new Grass
new Marsh helyett

main.cpp

Csomagok

#include "creature.h" körkörös include hivatkozást okozna. Itt azonban elég csak jelezni, hogy vannak ilyen osztályok

```
#pragma once
#include "ground.h"

class Creature { … };

class Greenfinch : public Creature { … };
class DuneBeetle : public Creature { … };
class Squelchy   : public Creature { … };
```

creature.h

```
#pragma once
class Greenfinch;
class DuneBeetle;
class Squelchy;

class Ground{
public:
    virtual Ground* transmuteGreenfinch(Greenfinch *g) = 0;
    virtual Ground* transmuteDuneBeetle(DuneBeetle *d) = 0;
    virtual Ground* transmuteSquelchy(Squelchy     *s) = 0;
};
class Sand  : public Ground { … }
class Grass : public Ground { … }
class Marsh : public Ground { … }
```

ground.h

```
#include "ground.h"
#include "creature.h"

    …
```

ground.cpp