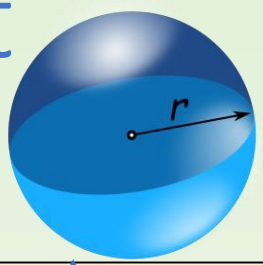
Tervminták I. (Sablonfüggvény, Stratégia, Egyke, Látogató)

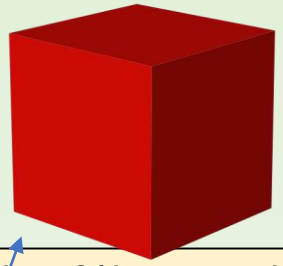
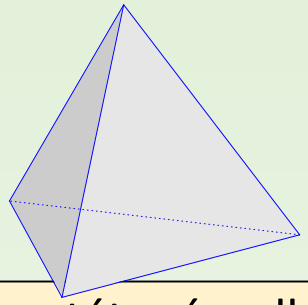
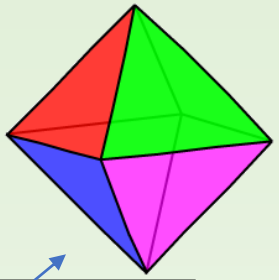
Gregorics Tibor

gt@inf.elte.hu

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

1.Feladat

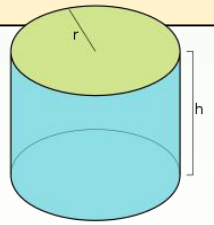


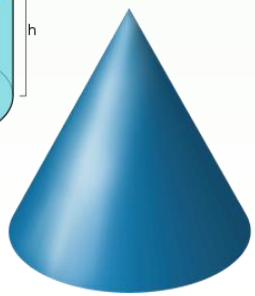
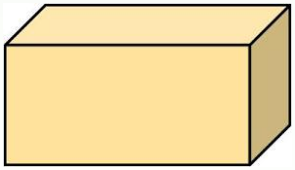
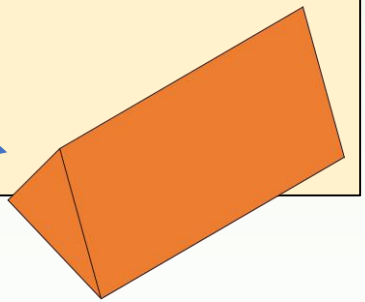
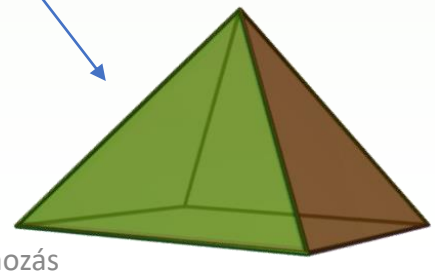




Készítsünk programot, amellyel különféle testek térfogatát számolhatjuk ki, illetve megadhatjuk azt is, hogy az egyes testfajtákból hány objektumot hoztunk létre!

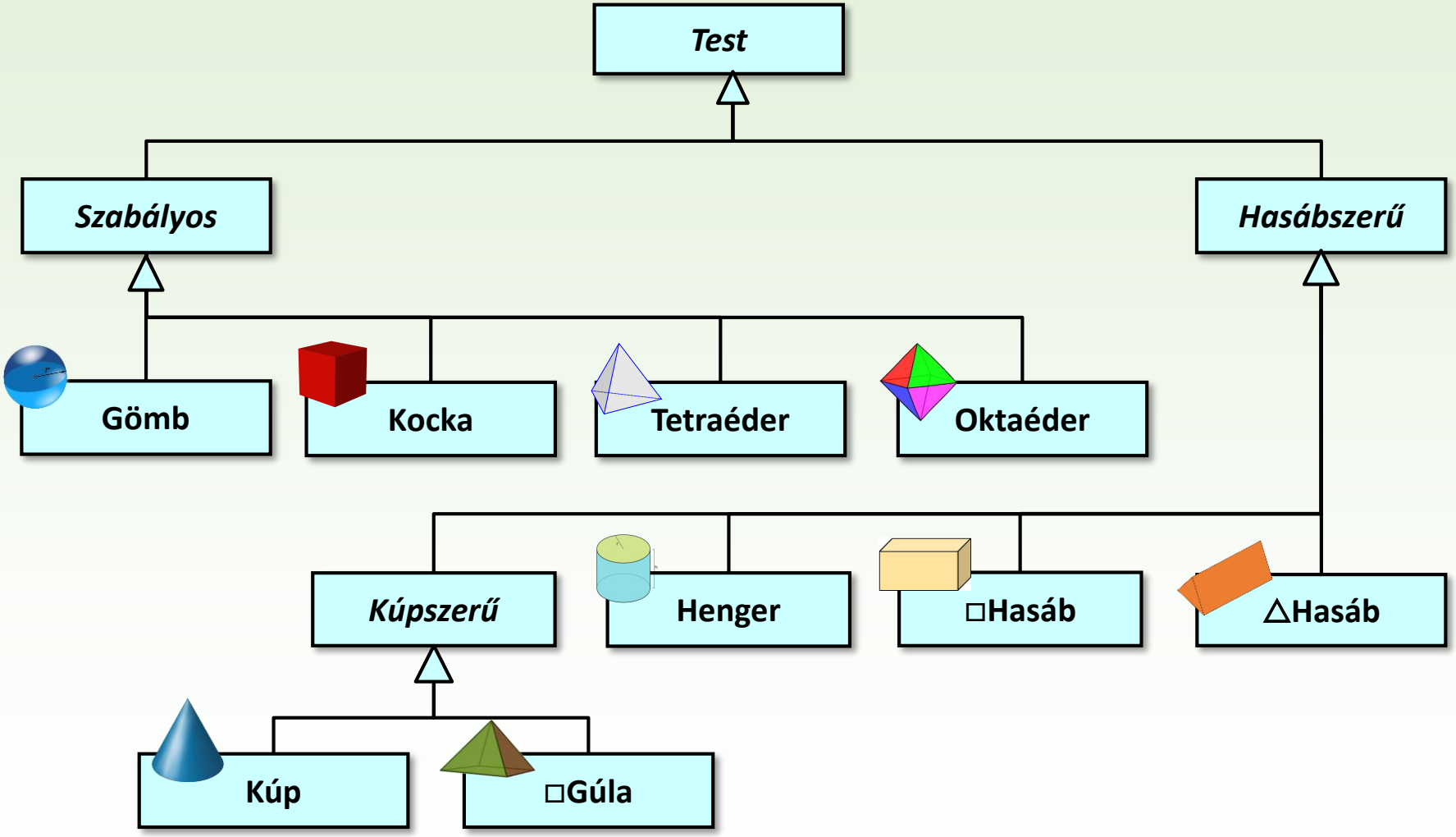
A lehetséges fajták:

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

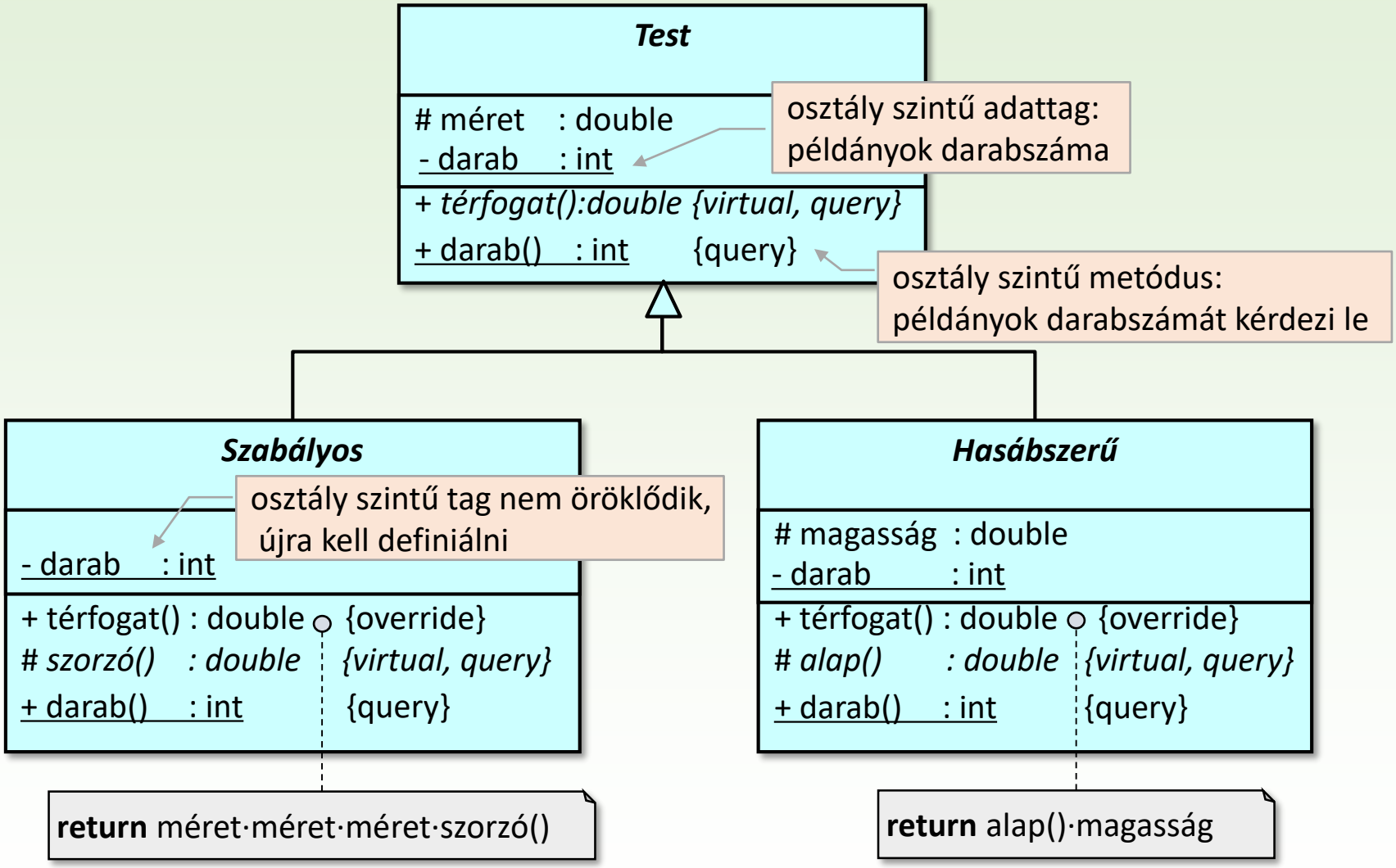


Osztálydiagram



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

absztrakt metódus: ettől az osztály is absztrakt

osztályszintű metódus

védett konstruktor: ettől az osztály is absztrakt

osztályszintű adattag

```
int Shape::_piece = 0;

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

osztályszintű adattag kezdeti értékadása

Szabályos absztrakt test

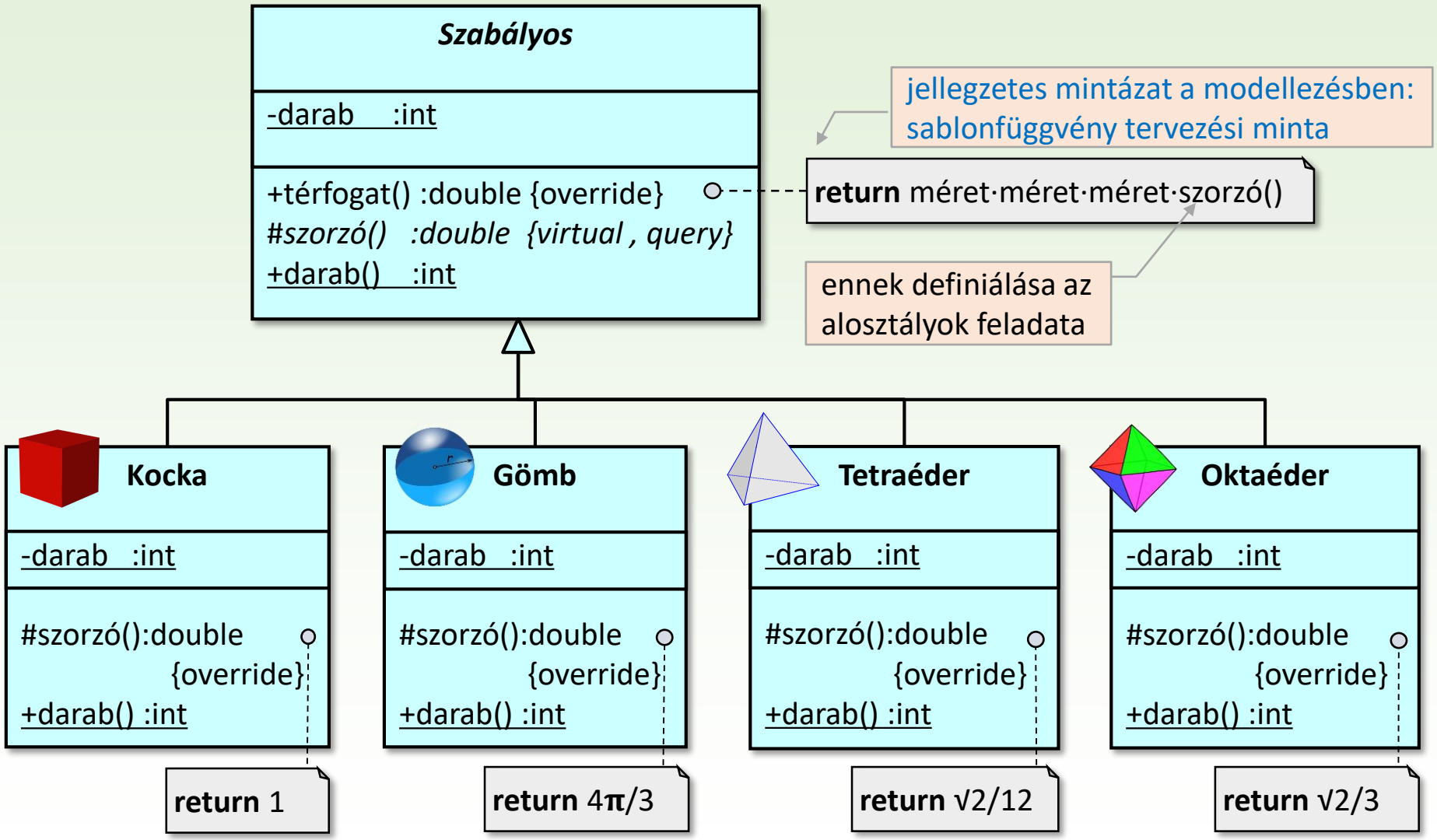
```
class Regular : public Shape{
    public:
        ~Regular();
        double volume() const override;
        static int piece() { return _piece; }
    protected:
        Regular(double size);
        virtual double multiplier() const = 0;
    private:
        static int _piece;
};
```

```
int Regular::_piece = 0;

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

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

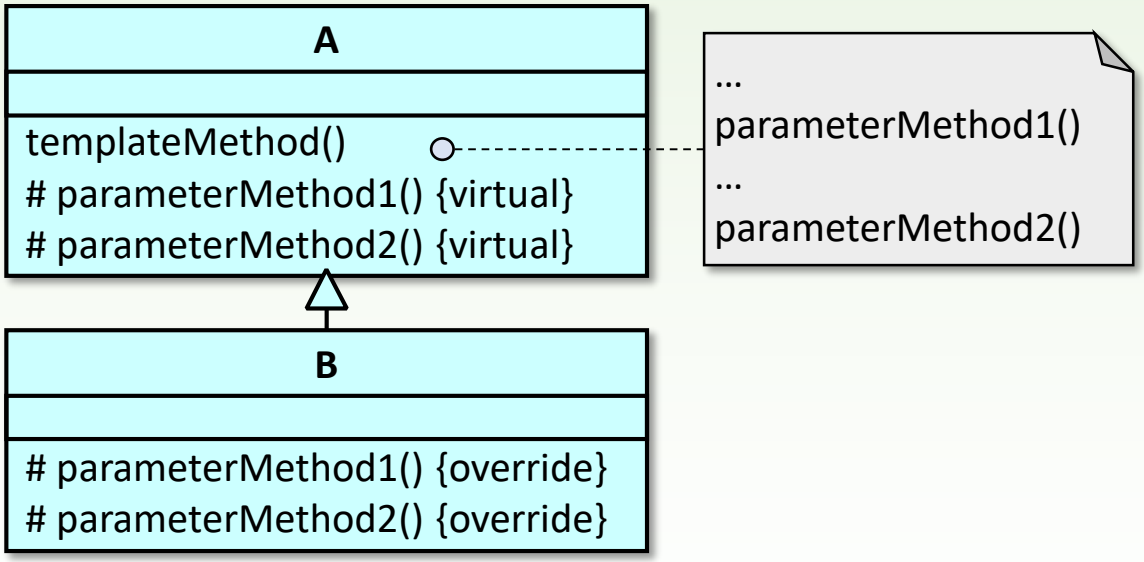
a destruktor automatikusan hívja az ősosztály destruktorát

Szabályos testek



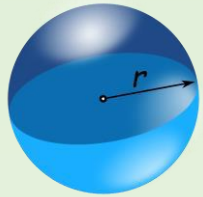
Sablonfüggvény tervezési minta (template method)

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



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

Gömb

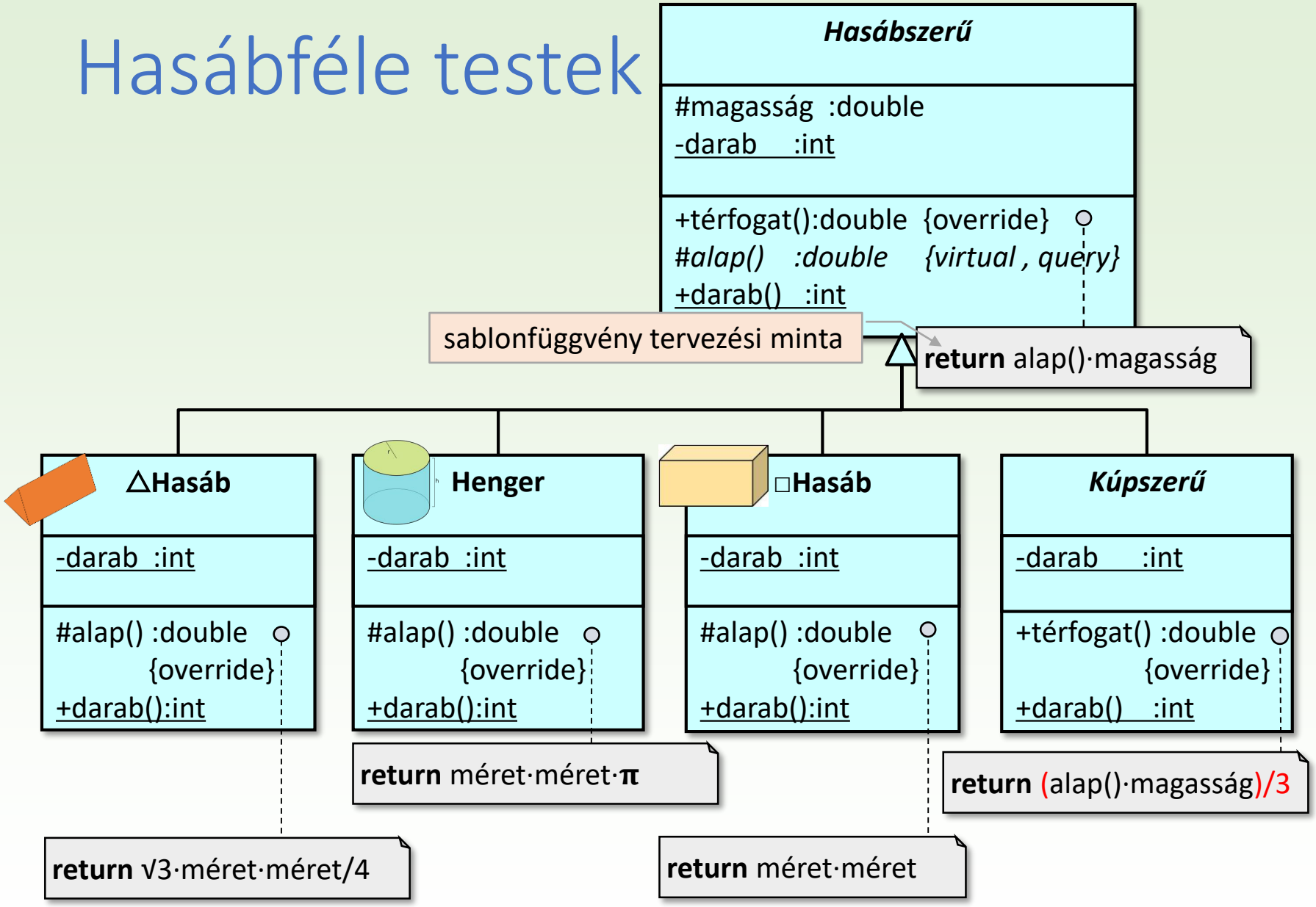



```
class Sphere : public Regular
{
	public:
		Sphere(double size);
		~Sphere();
		static int piece() { return _piece; }
	protected:
		double multiplier() const override { return _multiplier; }
	private:
		constexpr static double _multiplier = (4.0 * 3.14159) / 3.0;
		static int _piece;
};
```

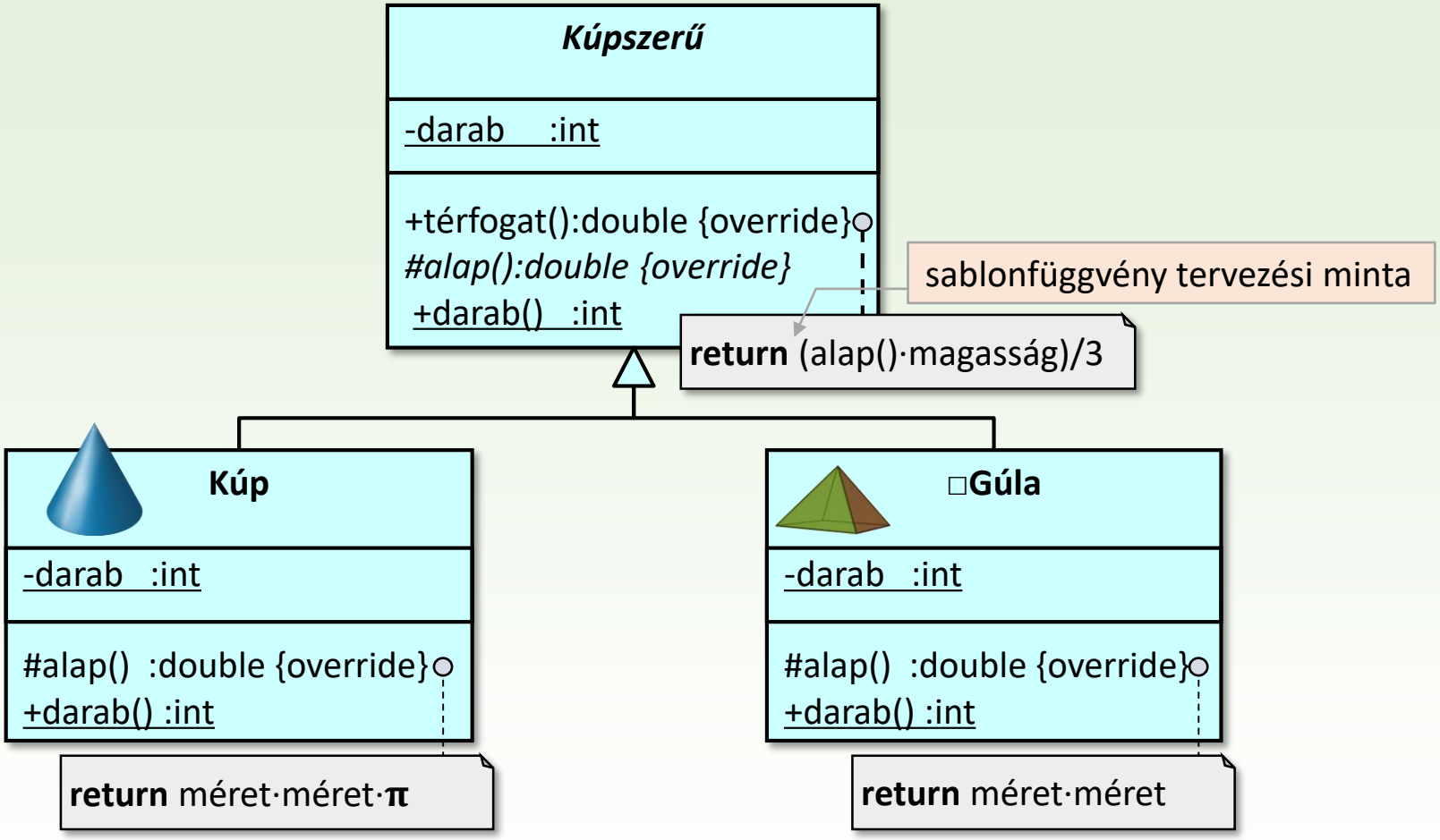
konstans osztályszintű kifejezés definiálása

```
int Sphere::_piece = 0;

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

Hasábféle testek



Kúpszerű testek



Kritika a modellről: redundancia (kódismétlődés) jelent meg a modellben. Ugyanolyan alapterület kiszámolására több osztályban is született metódus: például a kör területét a kúpnál is, a hengernél is definiáltuk; négyzet területét a négyzetalapú hasábnál is, a négyzetalapú gúlánál is megadtuk.

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 absztrakt
2. a Shape-nek nincs üres konstruktora
3. a tömbbe a Shape leszármazottjainak referenciáit vagy pointereit kell betenni, hogy a futási idejű polimorfizmust használni tudjuk

Test példányosítása

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

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

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

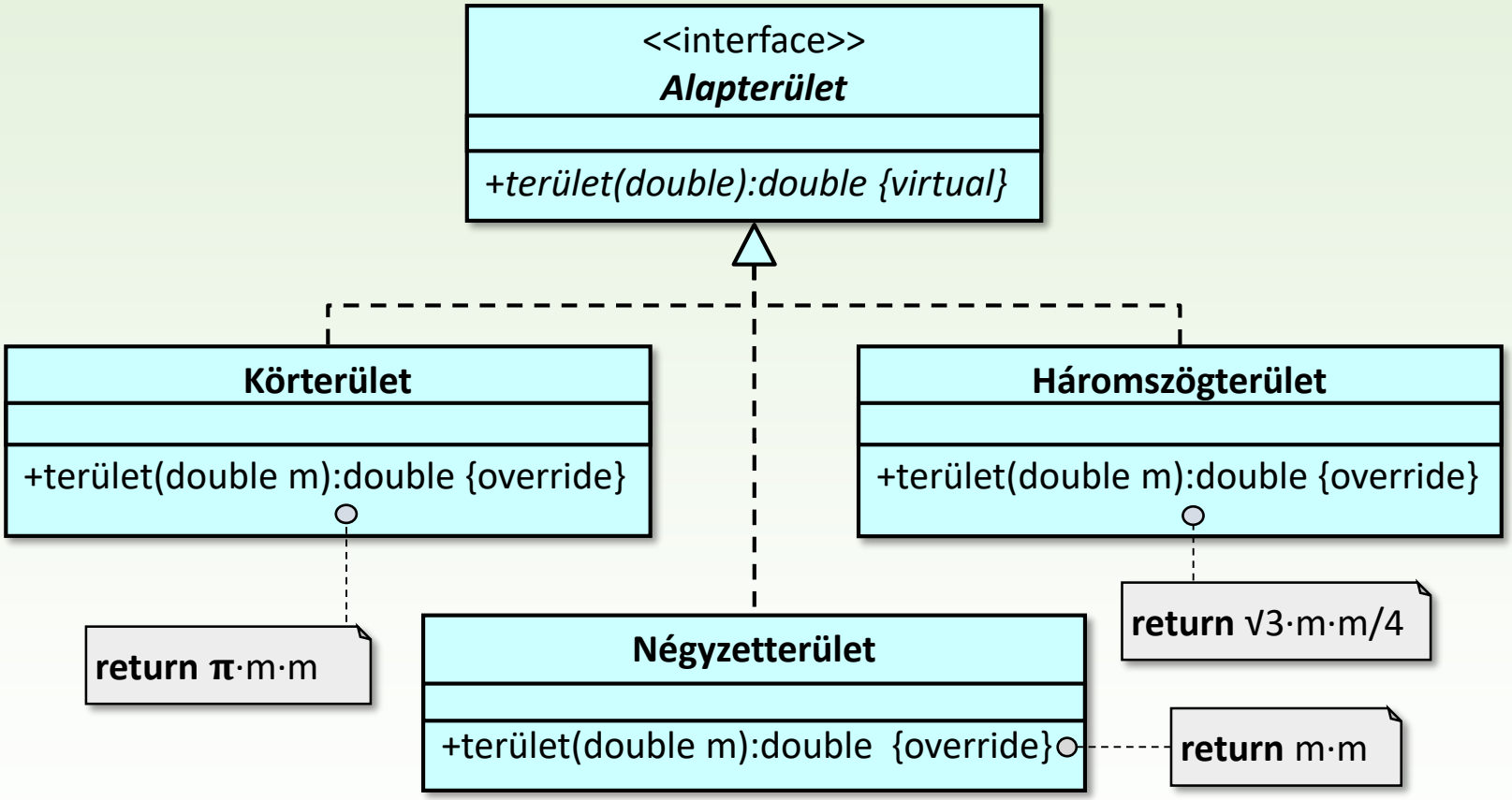
Főprogram folyt.

```
    ...
    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;
}
```

Ez a kifejezés a futási idejű polimorfizmus miatt a Shape ősosztály virtuális volume() metódusa helyett a megfelelő test térfogatát kiszámoló volume() metódust hívja meg.

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

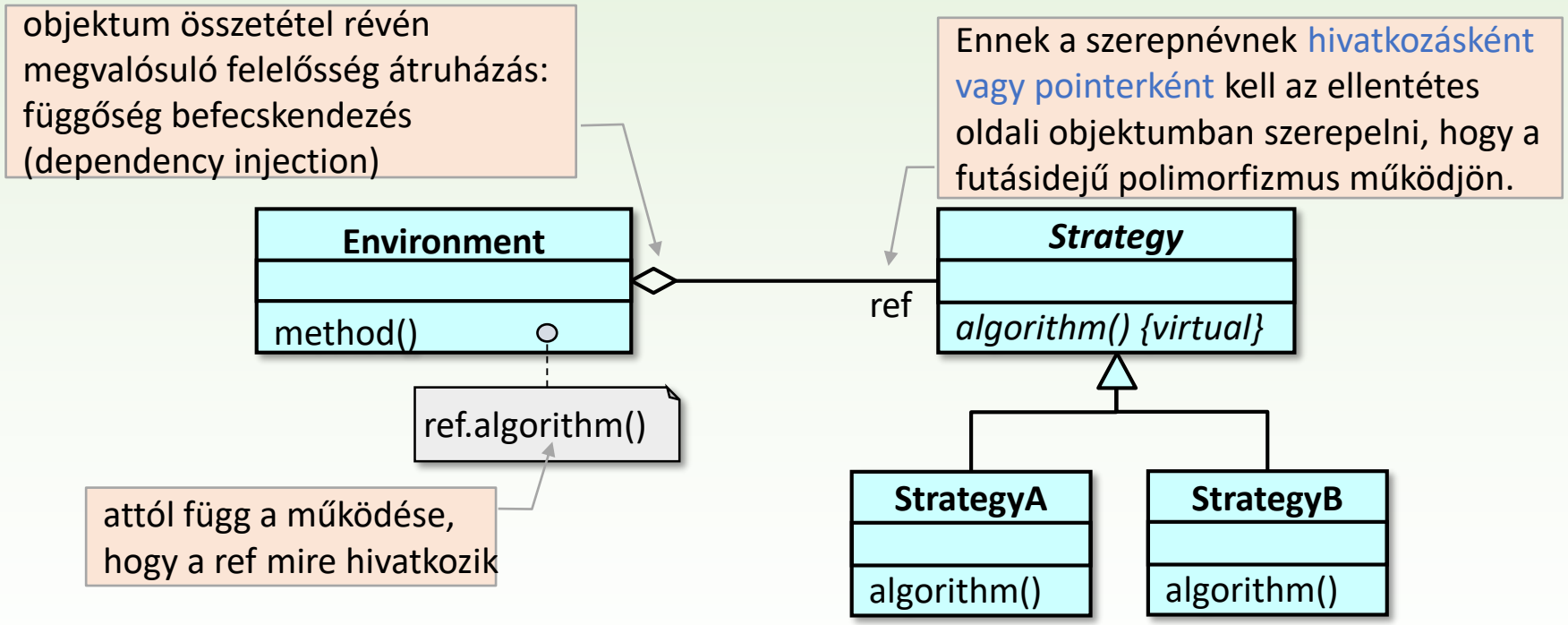
Alapterületet számoló objektumok



Itt, és csak itt kell az alapterületeket definiálni – megszűnik a redundancia.

Stratégia (strategy) tervezési minta

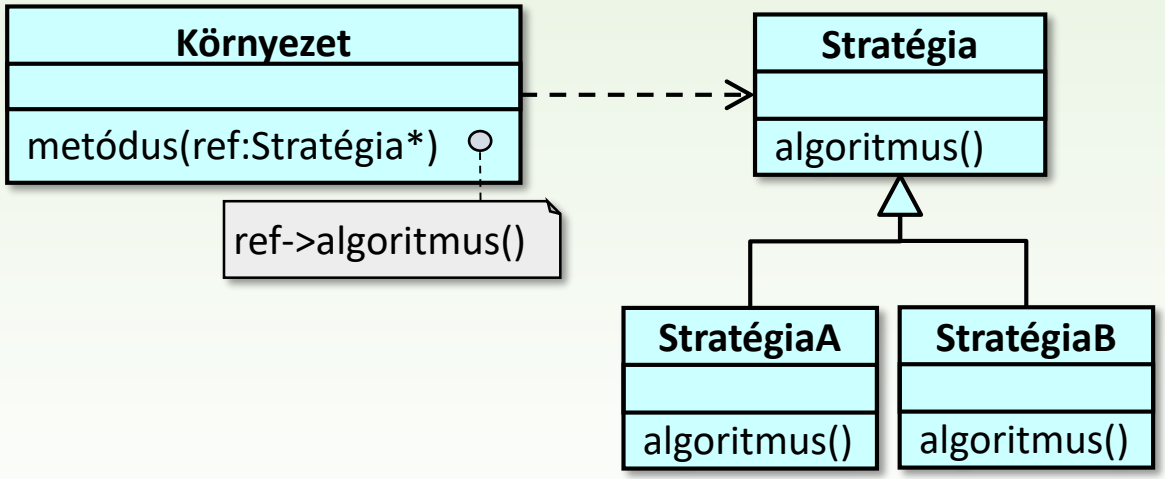
- Egy algoritmus-családot definiálunk úgy, hogy annak algoritmusait egy másik osztály objektumainak metódusai használhassák, de hogy melyiket, azt a metódus nem tartalmazza, hanem a metódus objektumától függjön.



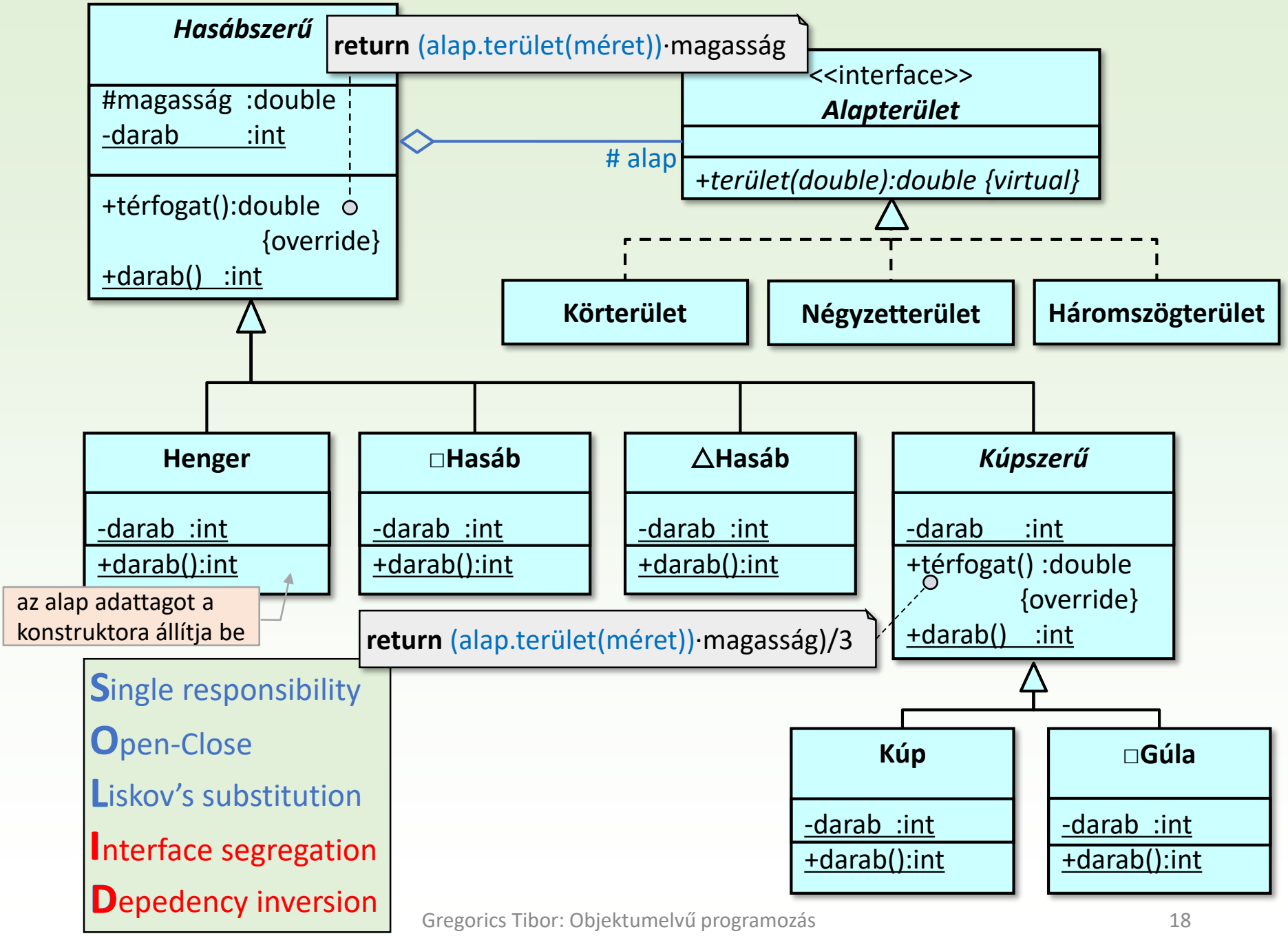
A tervezési minták az objektum-alapú modellezést támogató osztálydiagram minták, amelyek az újrafelhasználhatóság, rugalmas módosíthatóság, hatékonyság biztosításában játszanak szerepet.

Stratégia tervezési minta 2.

- A stratégia minta gyenge változata: egy algoritmus-családot definiálunk úgy, hogy ne kelljen fordítási időben még megadni azt, hogy ezek közül melyik algoritmust akarjuk majd használni.



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
<<interface>>
Alapterület
+terület(double):double {virtual}
alap
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
az alap adattagot a konstruktora állítja be
return (alap.terület(méret))·magasság)/3
Single responsibility
Open-Close
Liskov's substitution
Interface segregation
Depedency inversion
Kúp
-darab :int
+darab():int
□Gúla
-darab :int
+darab():int

```
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;
}
```

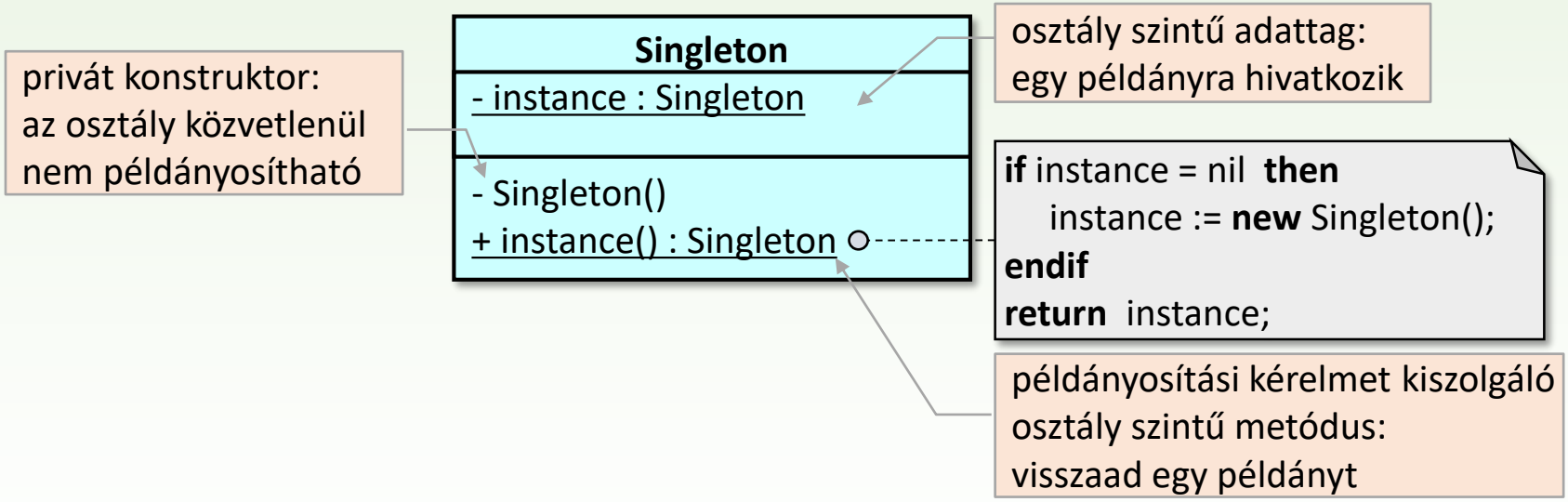
```
SquarePyramid::SquarePyramid(…) : Conical(…){
        ++_piece; _basis = new SquareArea();
}
SquarePyramid::~SquarePyramid(){
        --_piece; delete _basis;
}
```

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

Kritika a hatékonyságról:
A kód-redundancia megszűnt, de előállt egy memória-pazarlási probléma:
például 5 henger és 3 kúp példányosításakor összesen 8 körterület objektumot hozunk létre, pedig egy is elég lenne. Legyenek egykék a terület-objektumok.

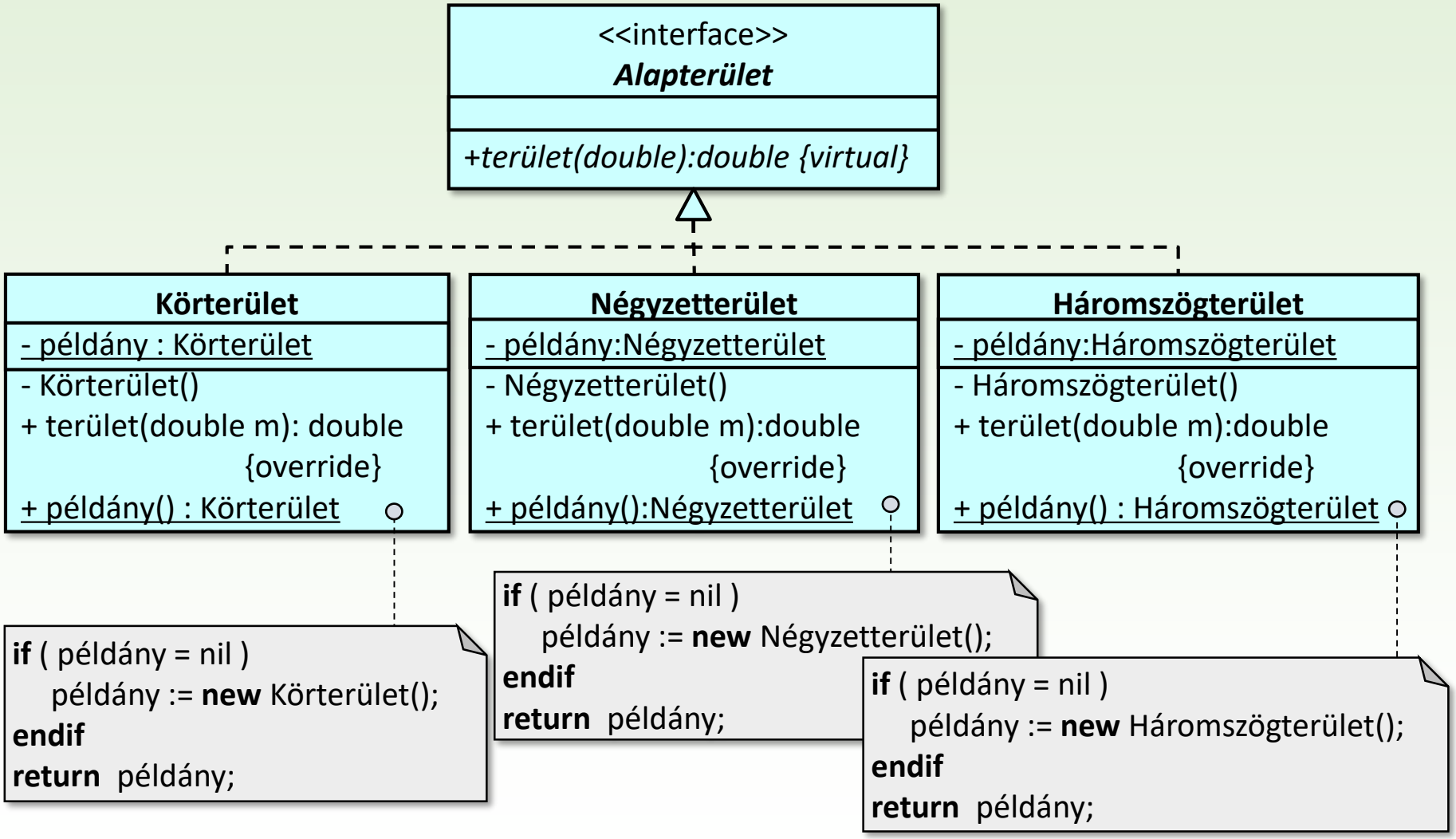
Egyke (singleton) tervezési minta

- 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, rugalmas módosíthatóság, hatékonyság biztosításában játszanak szerepet.

Egyke alapterület objektumok



```
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 különféle lények túlélési versenyét modellezhetjük.

A lények három faj (zöldikék, buckabogarak, tocsogók) valamelyikéhez tartoznak. Van nevük, ismert a fajuk, és az aktuális életerejük (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 fajtájától (és az adott tereptől is) függően átalakítja a terepet, miközben változik az életereje. Ha az életereje elfogy, a lény elpusztul. Adjuk meg a pálya végéig eljutó, 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.*

Hogyan alakít át egy lény egy terepet?

átalakít : Lény×Terep ⟶ Lény×Terep

feltéve, hogy a lény él

zöldikék	életerő változás	terepváltozás
homok	-2	-
fű	+1	-
mocsár	-1	fű

buckabogarak	életerő változás	terepváltozás
homok	+3	-
fű	-2	homok
mocsár	-4	fű

tocsogók	életerő változás	terepváltozás
homok	-5	-
fű	-2	mocsár
mocsár	+6	-

Megoldási terv

Miután a pályán áthaladó lények sorban egymás után megváltoznak (és közben a pálya is átalakul), kiválogatjuk a túlélő lények neveit.

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

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

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

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

a pálya az *i*-dik lény áthaladása előtt

Kiválogatás: túlélő lények kiválogatása

lények[i].él()

Szimuláció: a megváltozik a lények tömbjének összes eleme, és a pálya újra és újra lecserélődik

t:enor(E) ~ i = 1 .. n (lények felsorolása)

szimuláció = dupla összegzés

összefűzés és sorozatos lecserélés :

f(e) ~ áthalad(lények[i], pálya)

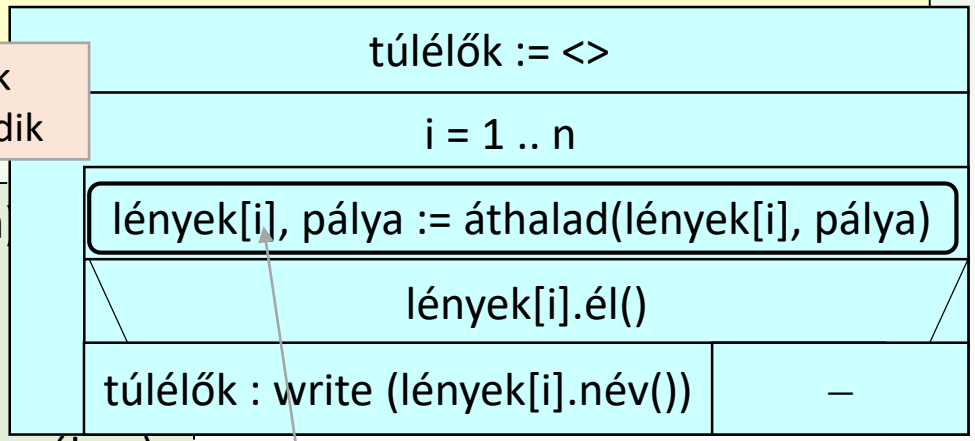
H,+,0 ~ Lény^n×Terep^m, ($\bigoplus$, $\ominus$), (<>, pálya$_0$)

kiválogatás = összegzés

f(e) ~ <lények[i].név()> ha lények[i].él()

H,+,0 ~ String*, $\bigoplus$, <>

régi $\ominus$ új ::= új



a lények tömbjét nem összefűzzük, hanem a megfelelő indexű elemét felülírjuk

Egy lény áthaladása

Az *i*-dik lény az aktuális pálya terepein halad át (amíg él), és minden lépése megváltoztathatja a terepet, miközben a lény maga is átalakul.

A : pálya: Terep^m, lény: Lény

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

Uf: $\forall j \in [1..m]$: (lény$_j$, pálya[j]) = átalakít(lény$_{j-1}$, pálya'[j]) $\wedge$ lény = lény$_m$

Áthaladás: a pálya összeállítása egy lény által sorban átalakított terepekből, mialatt a lény állapota sorozatosan lecserélődik

a lény (lények[i]) a j-dik terep átalakítása után

az új pálya j-dik terepe (pálya$_i$[j]) az átalakítás után

lény, pálya[j] := átalakít(lény, pálya[j]) a pálya tömb összefűzésekor a megfelelő indexű elemét írjuk felül

lény, pálya := áthalad(lény, pálya)

j := 1

lény.él() $\wedge$ j ≤ m

lény.átalakít(pálya[j])

j := j+1

áthaladás = feltételig tartó dupla összegzés (sorozatos lecserélés és összefűzés) :

t:enor(E) ~ j = 1 .. m amíg lény.él()

H,+,0 ~ Lény×Terep^m, (⊖, ⊕), (lény$_0$, <>)

f(e) ~ átalakít(lény, pálya[j])

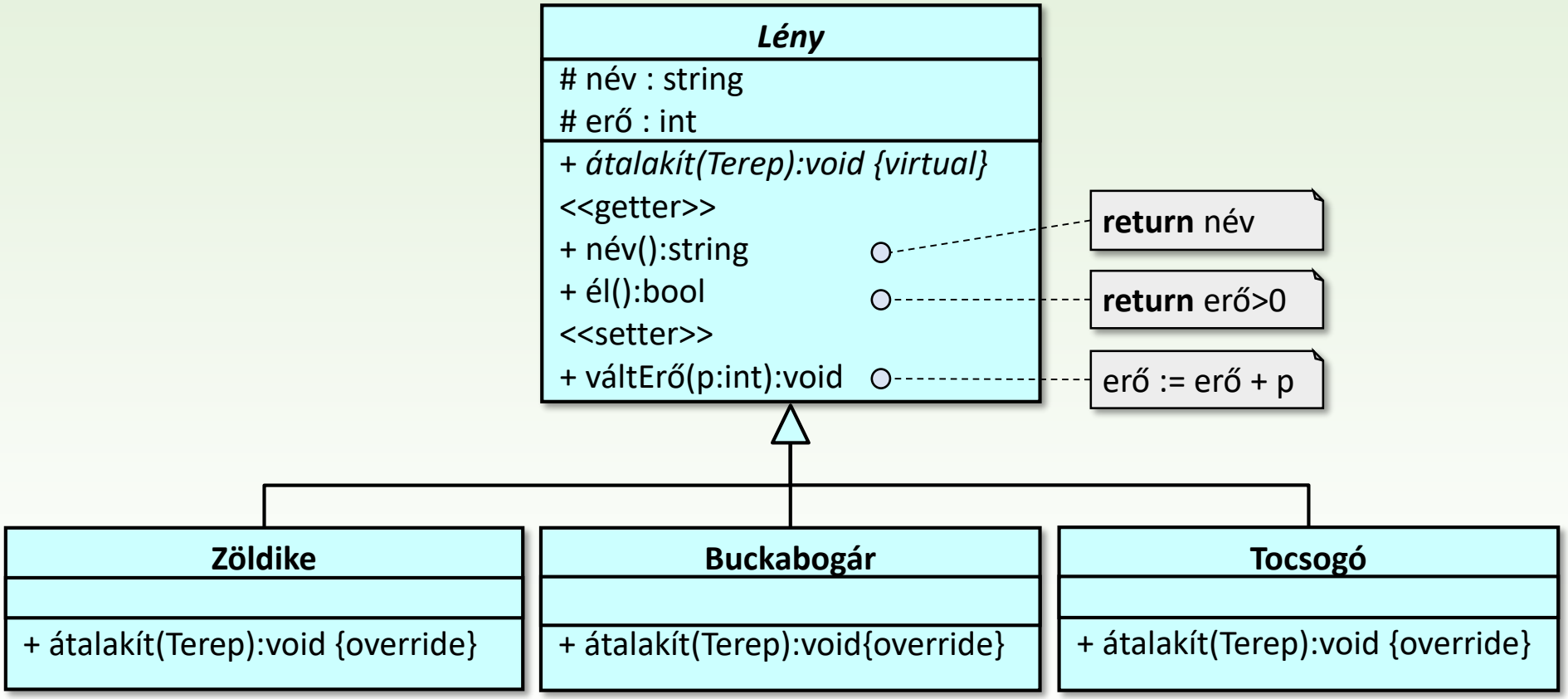
Főprogram

```
// populating
…
// competition
for ( int i=0; i < n; ++i ){
        for ( int j=0; creatures[i]->alive() && j < m; ++j ){
                creatures[i]->transmute(courts[j]);
        }
        if (creatures[i]->alive() ) cout << creatures[i]->name() << endl;
}

// destroying
…
```

main.cpp

Itt jól jönne a futási idejű polimorfizmus, azaz hogy a creatures[i] aktuális típusától függjön a transmute() működése.

Lények származtatása



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;
        }
    }
}
```

creature.cpp

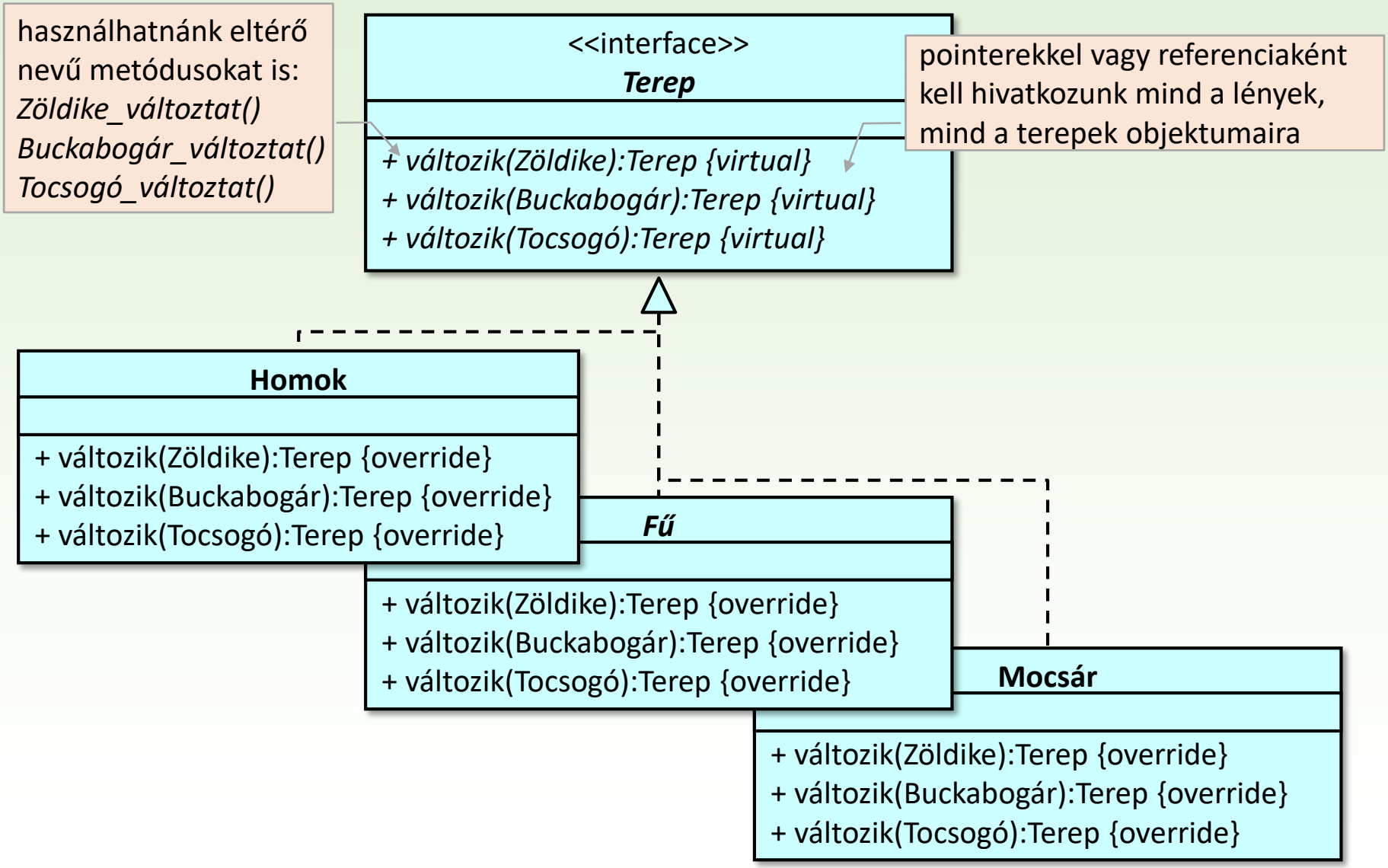
Kritika a kódról:

- nem rugalmas: sérül az Open/Close elv
  - újabb terepfajta bevezetése esetén az összes elágazást módosítani kell, azaz meglévő kódon kell változtatni
- rosszul olvasható:
  - a terepeket azonosító egész számok nem „beszédes” jelölések

bevezethetnénk az alábbi típust:
enum Ground { sand, grass, marsh },
hogy kiváltsuk a 0, 1, 2 használatát,
de nem ezt fogjuk csinálni.

Terepek származtatása

használhatnánk eltérő nevű metódusokat is:
Zöldike_változtat()
Buckabogár_változtat()
Tocsogó_változtat()



pointerekkel vagy referenciaként kell hivatkozunk mind a lények, mind a terepek objektumaira

Terep és lény változása

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

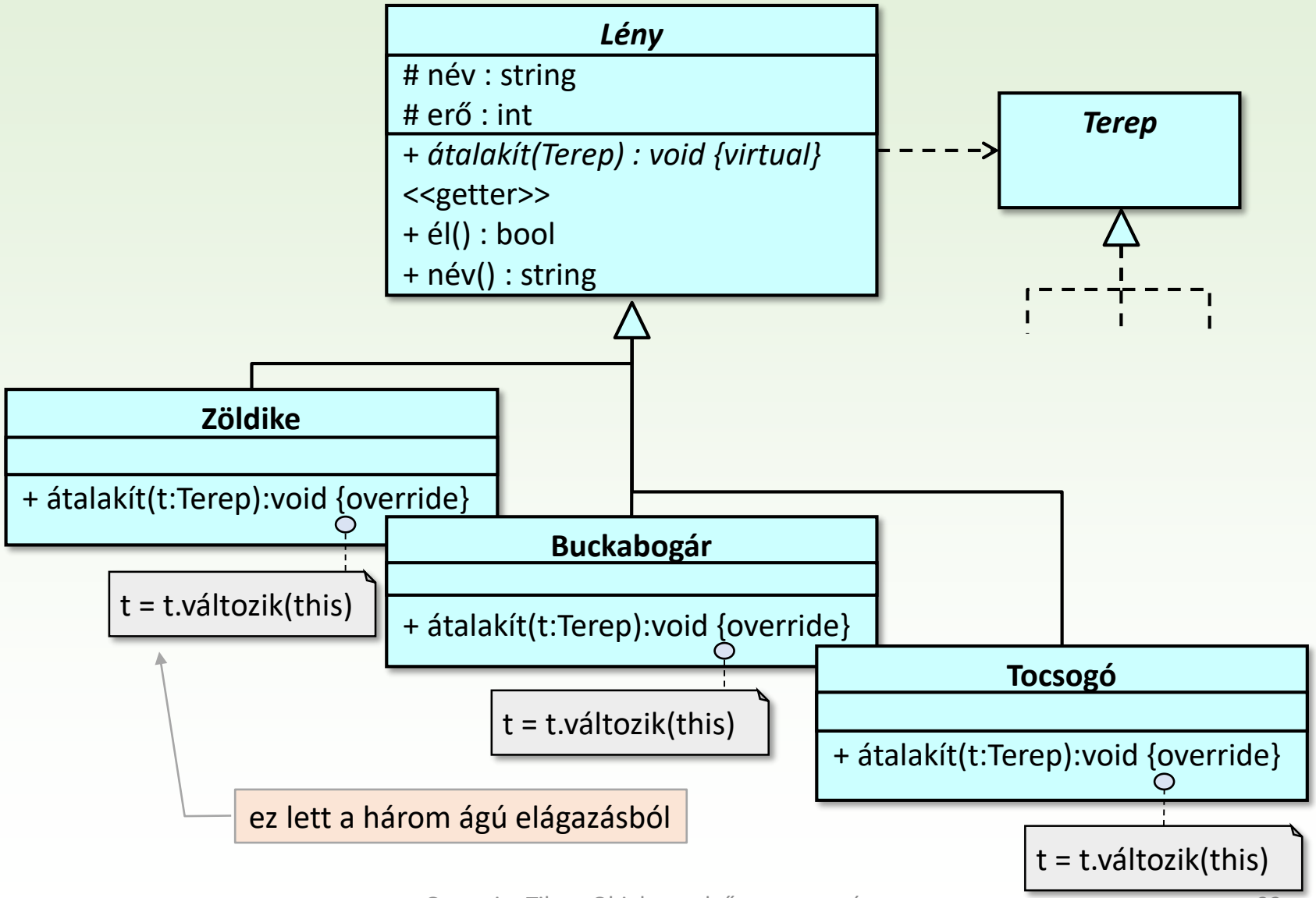
```
Ground* Grass::change(Greenfinch *p){
   p->changePower(1);  return this;
}
Ground* Grass::change(DuneBeetle *p){
   p->changePower(-2); return new Sand;
}
Ground* Grass::change(Squelchy *p){
   p->changePower(-2); return new Marsh;
}
```

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

ground.cpp

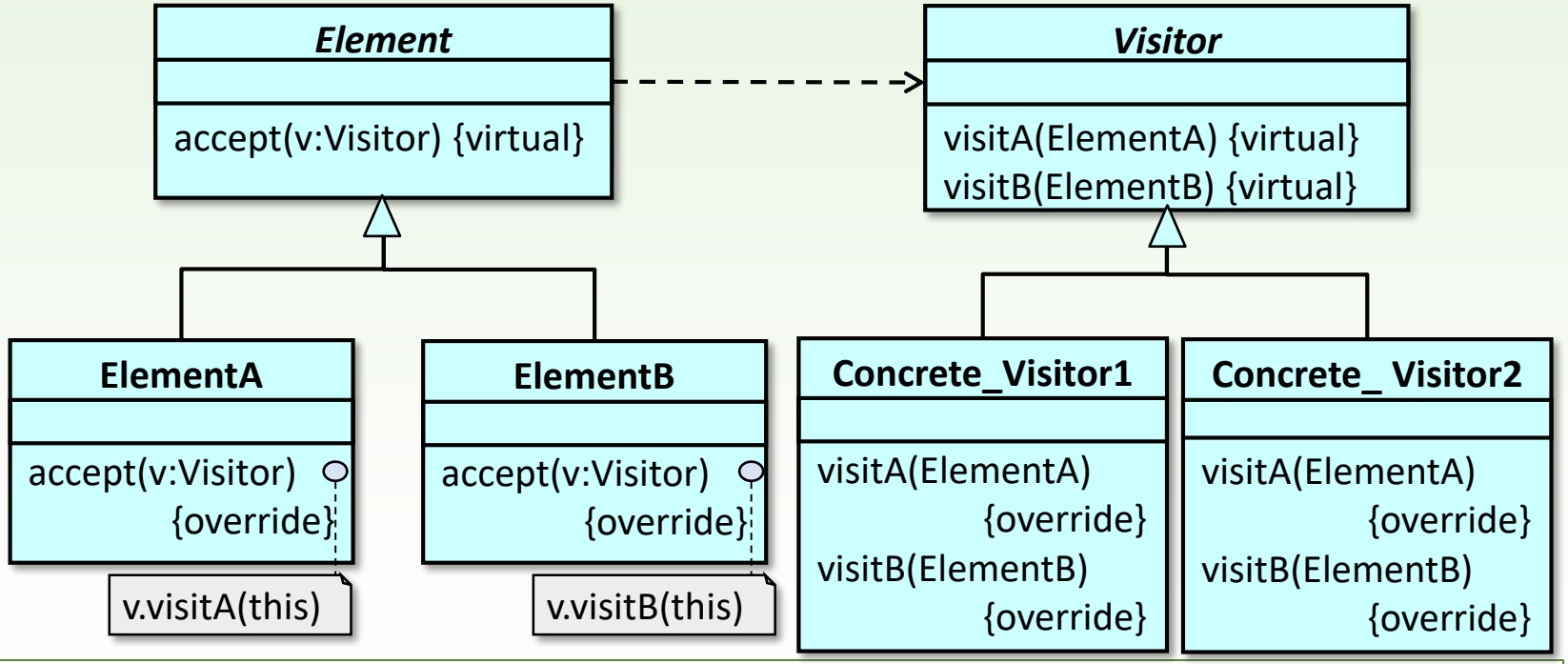
Itt összesen 9 darab, lénytől és tereptől függő change() metódus van.
Újabb terepfajta esetén új osztályt kell 3 új change() metódussal definiálni.
Újabb lényfajta esetén minden osztályba 1-1 újabb change() metódust kell betenni.
De meglevő kódon nem kell változtatni.

Lények osztálydiagramja újra



Látogató (visitor) tervezési minta

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



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

Terepek, mint látogatók

```
class Ground{
public:
   virtual Ground* change(Greenfinch *p) = 0;
   virtual Ground* change(DuneBeetle *p) = 0;
   virtual Ground* change(Squelchy   *p) = 0;
   virtual ~Ground(){}
};
```

ground.h

```
class Sand : public Ground{
public:
   Ground* change(Greenfinch *p) override;
   Ground* change(DuneBeetle *p) override;
   Ground* change(Squelchy   *p) override;
};
class Grass : public Ground{
public:
   Ground* change(Greenfinch *p) override;
   Ground* change(DuneBeetle *p) override;
   Ground* change(Squelchy   *p) override;
};
class Marsh : public Ground{
public:
   Ground* change(Greenfinch *p) override;
   Ground* change(DuneBeetle *p) override;
   Ground* change(Squelchy   *p) override;
};
```

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

Kritika a hatékonyságról:

- a court = court->change(**this**) végrehajtása memória szivárgást okoz, mert azt a terep-objektumot, amelyre a court az értékadás előtt hivatkozik, nem szabadítjuk fel
- a change() mindig új terep-objektumot hoz létre, így ugyanazon fajtájú terep-objektumból sok azonos lesz, pedig elég lenne mindegyikből egy-egy

Legyenek a Terep osztályok egykék

```
class Grass : public Ground{
private:
    static Grass* _instance;
    Grass(){}
public:
    static Grass* instance();
    Ground* change(Greenfinch *p) override;
    Ground* change(DuneBeetle *p) override;
    Ground* change(Squelchy   *p) override;
    static void destroy();
};
```

ground.h

```
Grass* Grass::_instance = nullptr;
void Grass::instance() {
      if ( nullptr ==_instance )
          _instance = new Sand();
      return _instance;
}
void Grass::destroy() {
      if ( nullptr!=_instance ) delete _instance;
}
Ground* Grass::change(Greenfinch *p){
   p->changePower(1);   return this;
}
Ground* Grass::change(DuneBeetle *p){
   p->changePower(-2); return Sand::instance();
}
Ground* Grass::change(Squelchy *p){
   p->changePower(-2); return Marsh::instance();
}
```

osztály szintű (static) törlő metódus

ground.cpp

Feladat felpopulálása

```
// populating
    ifstream f("input.txt");
    if (f.fail()) { cout << "Wrong file name!\n"; return 1;}
    int n; f >> n;
    vector<Creature*> creatures(n);
    for ( int i=0; i<n; ++i ){
        char ch; string name; int p;
        f >> ch >> name >> p;
        switch(ch){
            case 'G' : creatures[i] = new Greenfinch(name, p);  break;
            case 'D' : creatures[i] = new DuneBeetle(name, p); break;
            case 'S' : creatures[i] = new Squelchy(name, p);     break;
        }
    }
    int m; f >> m;
    vector<Ground*> courts(m);
    for ( int j=0; j<m; ++j ) {
        int k; f >> k;
        switch(k){
            case 0 : courts[j] = Sand::instance();  break;
            case 1 : courts[j] = Grass::instance(); break;
            case 2 : courts[j] = Marsh::instance(); break;
        }
    }
```

main.cpp

input.txt

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

```
// destroying creatures
    for (int i=0; i<n; ++i) delete creatures[i];
// destroying creatures
    Sand::destroy();
    Grass::destroy();
    Marsh::destroy();
```

Csomagok

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

class Creature { … };
class Greenfinch : public Creature { …
      void transmute(Ground* &court) override
            {court = court->change(this);}
};
class DuneBeetle : public Creature { … };
class Squelchy   : public Creature { … };
```

creature.h

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

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

Itt a #include "creature.h" körkörös include hivatkozást okozna. Szerencsére elég jelezni, hogy vannak (lesznek) ilyen osztályok.

ground.h

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

ground.cpp