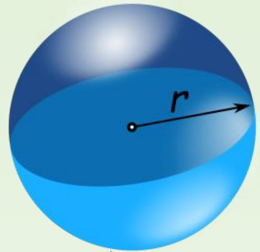
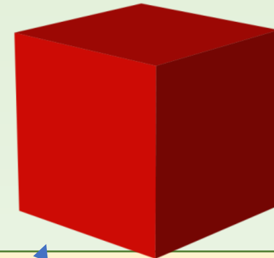
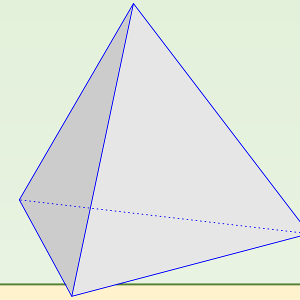
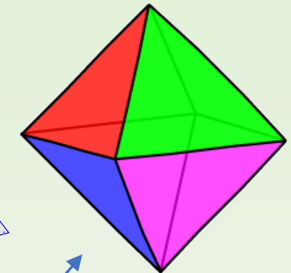
Design patterns I. (Template method, Strategy, Singleton, Visitor)

Volume of geometrical shapes

Competition of creatures

1st task

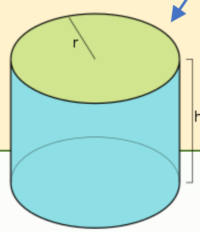


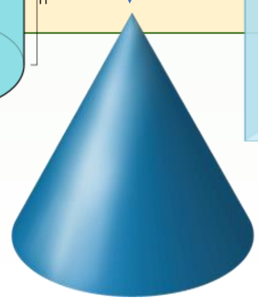
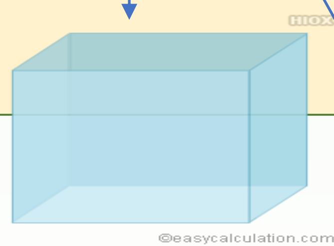




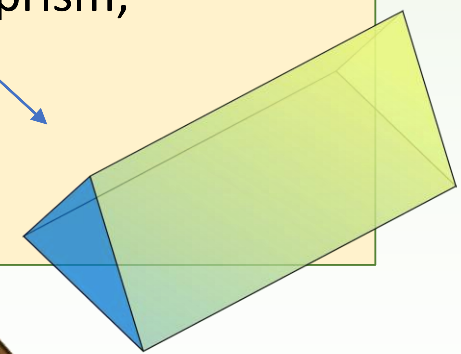
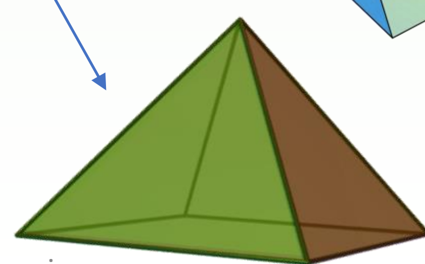
Create a program to calculate the volume of different geometrical shapes and to calculate how many objects of the different types were created.

Possible types:

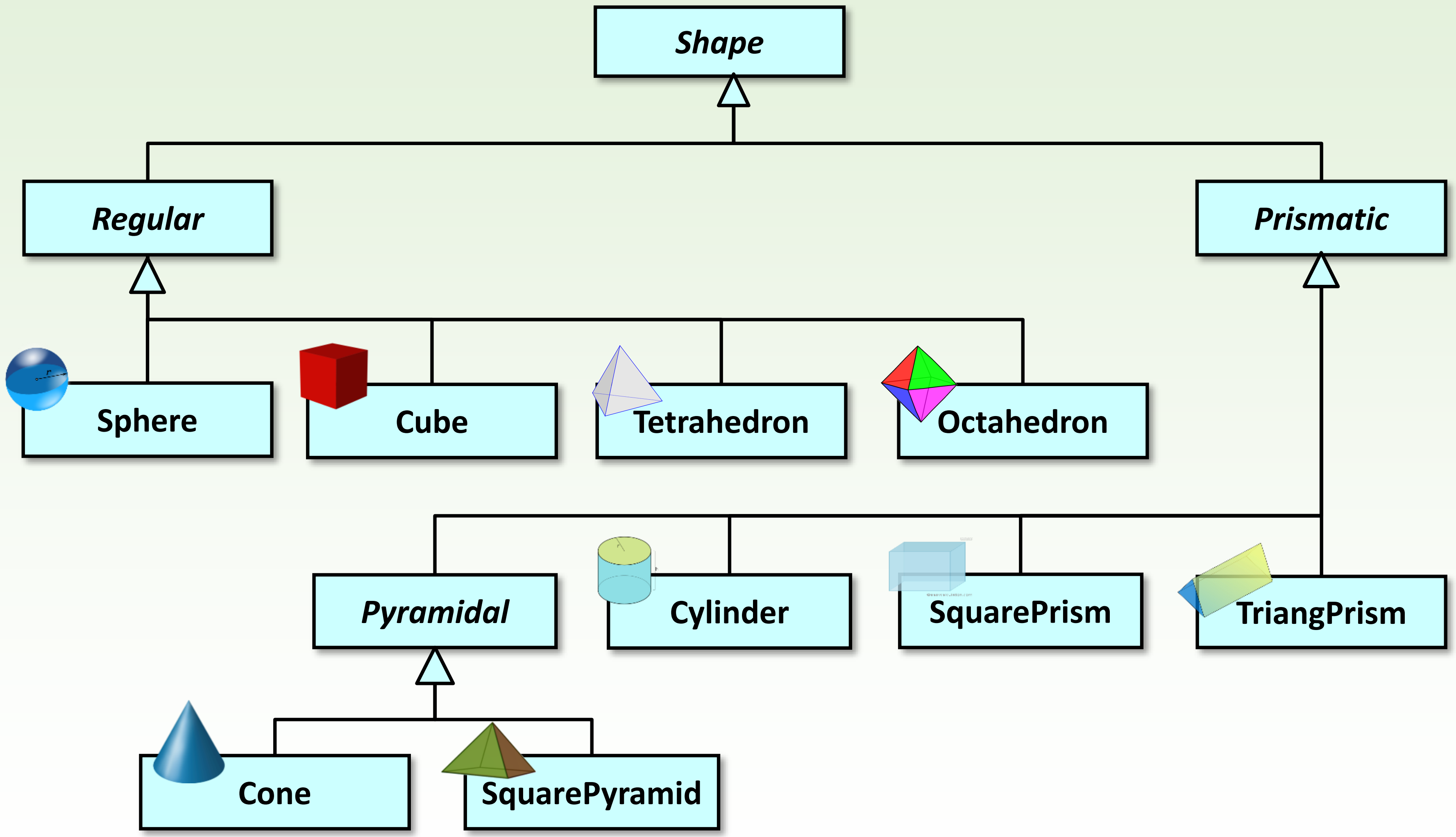
- regular shapes: sphere, cube, tetrahedron, octahedron;
- prismatic shapes: cylinder, square prism and triangular prism;
- pyramidal shapes: cone, square pyramid.



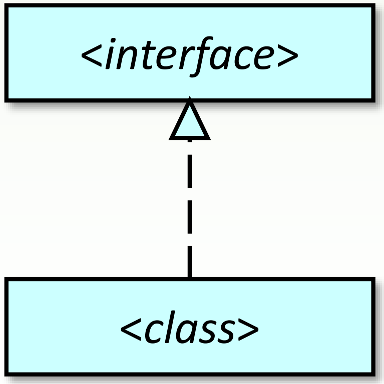
Class diagram



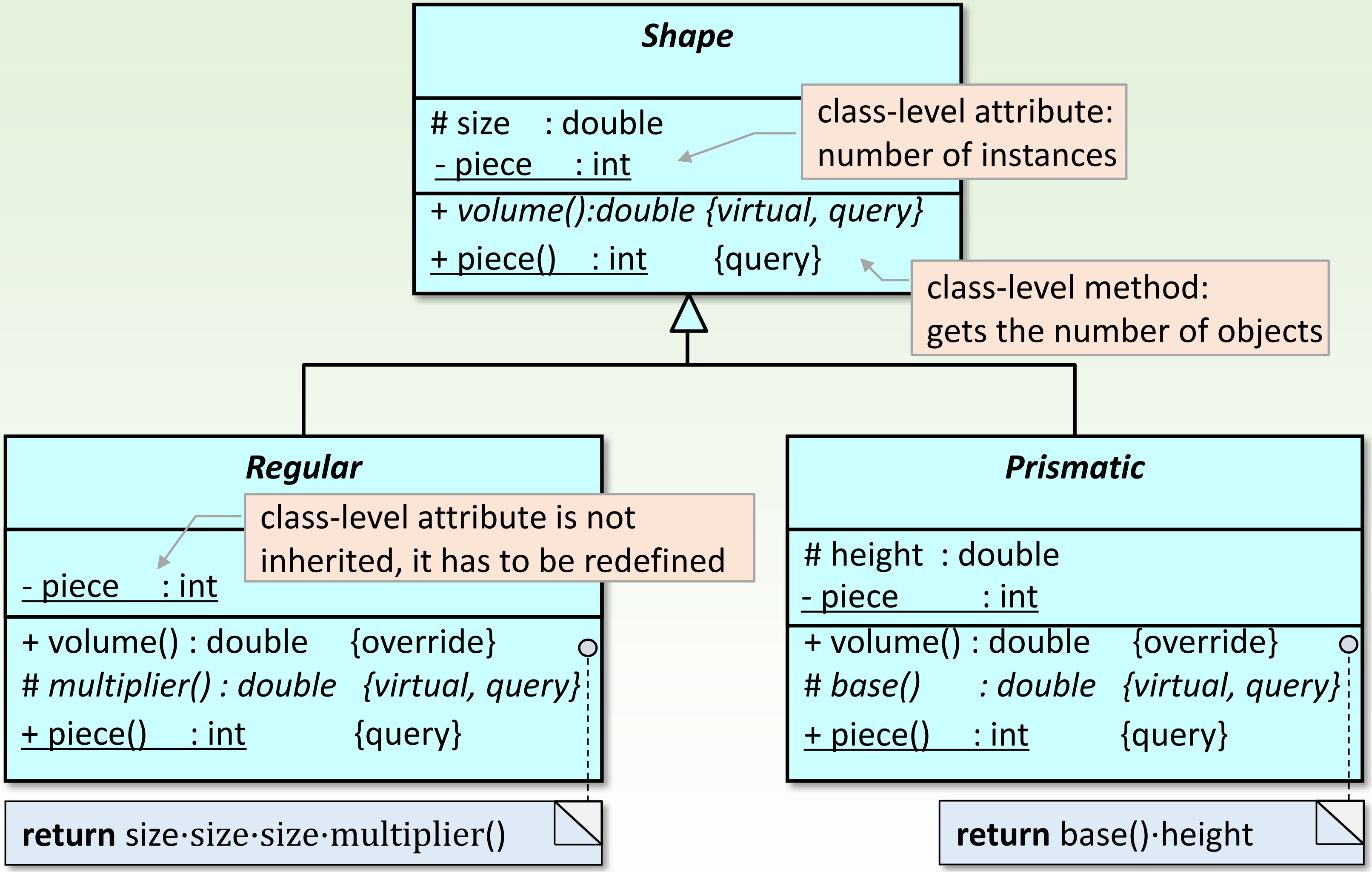
Abstract class, interface

- Abstract class is never instantiated, it is used only as a base class for inheritance.
  - name of the abstract class is italic.
- A class is abstract if
  - its constructors are not public, or
  - at least one method is abstract (it is not implemented and it is overridden in a child)
    - name of the abstract method is also italic

- *Pure abstract* classes are called interfaces, none of their methods are implemented.
- When a class implements all of the abstract methods of an interface, it realizes the interface.



Abstract shapes



Base class of shapes

```
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;
};
```

virtual destructor

class with abstract method is abstract

class-level method

class with protected constructor is abstract

class-level attribute

```
int Shape::_piece = 0;

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

initial value of a class-level attribute

Abstract class of regular shapes

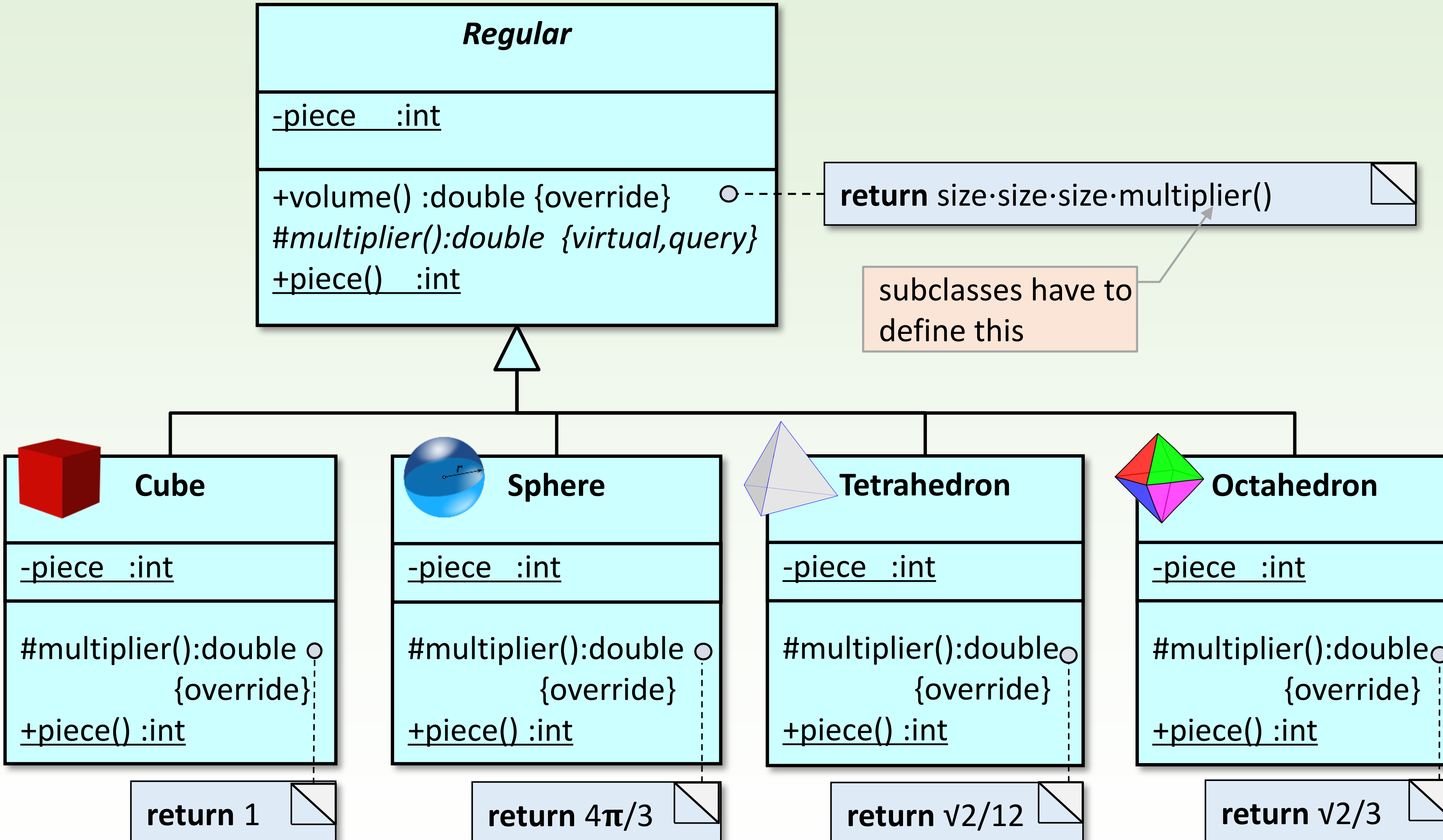
```
class Regular : public Shape{
    public:
        ~Regular();
        double volume() const override;
        static int piece() { return _piece; }
    protected:
        Regular(double size);
        virtual double multiplier() const = 0;
    private:
        static int _piece;
};
```

Without this, the constructor would automatically call the empty constructor of the base class which does not exist.

```
int Regular::_piece = 0;

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

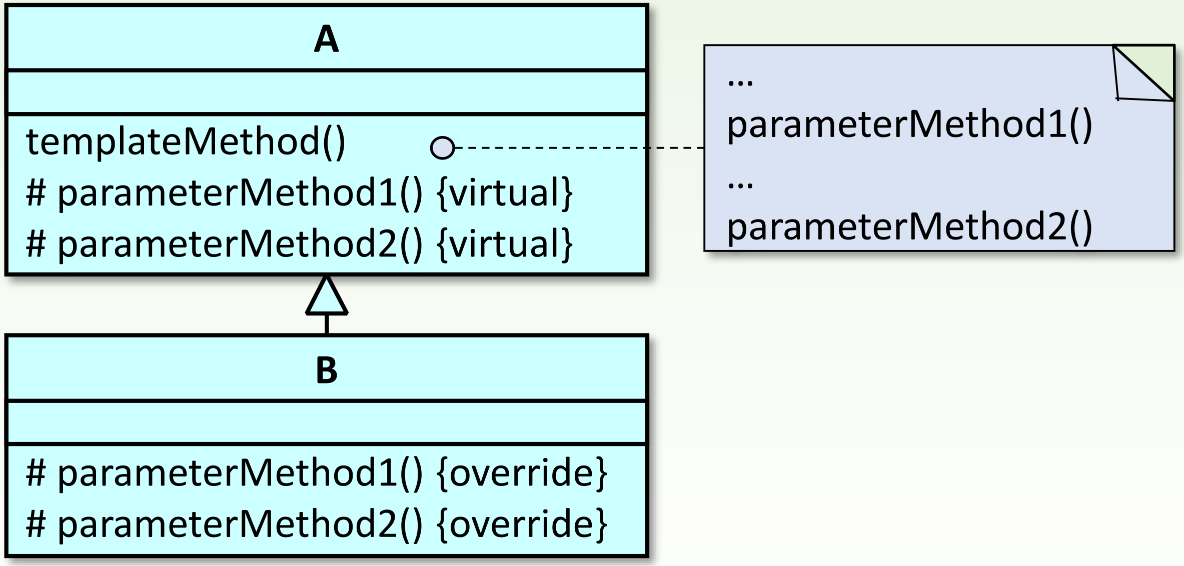
the destructor automatically calls the destructor of the base class

Regular shapes



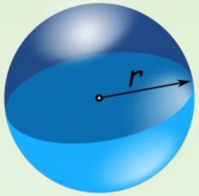
Template method design pattern

- In a class, the algorithm of a method is given by other protected submethods. The submethods can be overridden in the children, but the structure of the main method does not change.



Design patterns are class diagram patterns that help object-oriented modeling. They play a significant role in reusability, modifiability, and ensuring efficiency.

Sphere

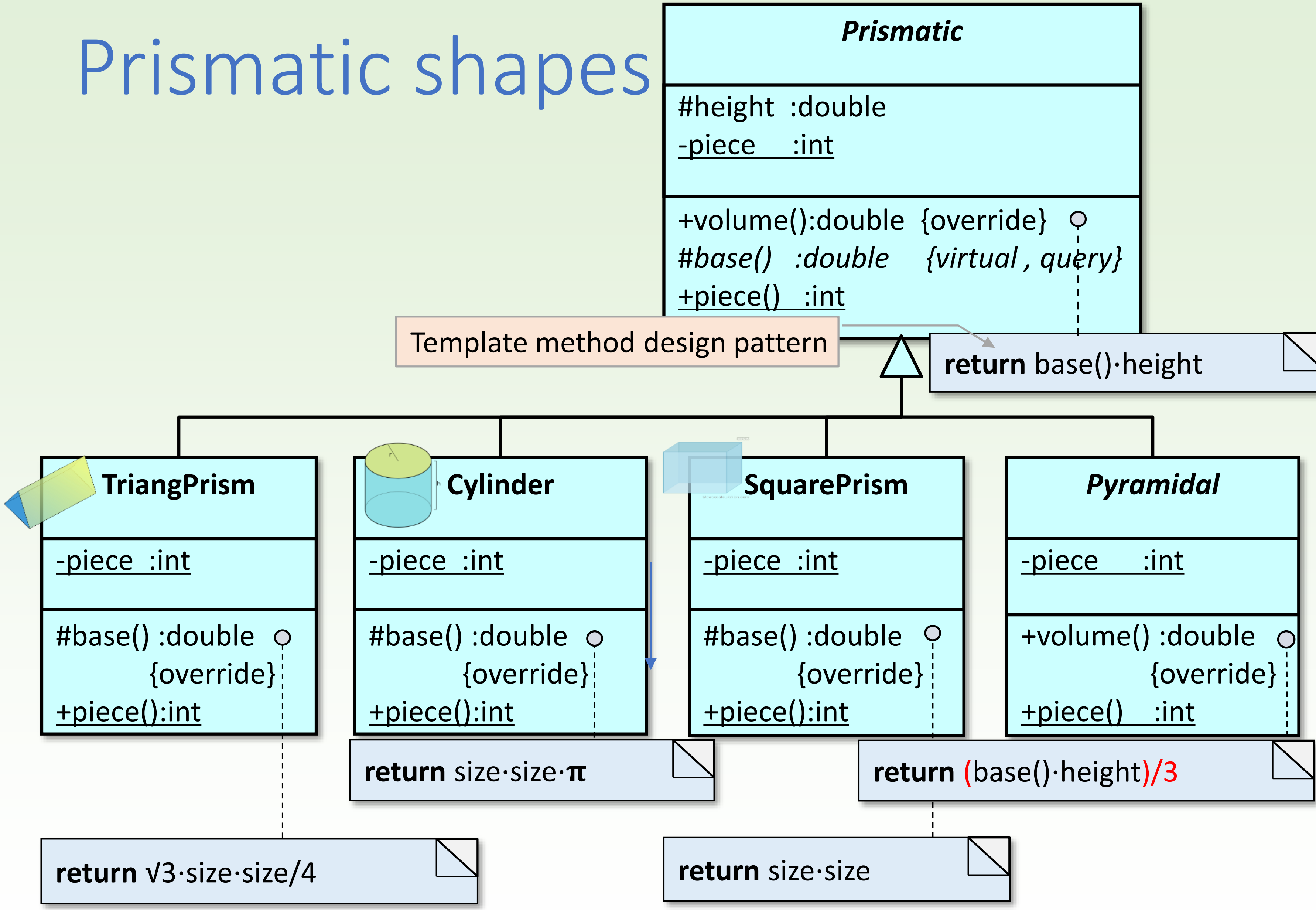



```
class Sphere : public Regular
{
    public:
        Sphere(double size);
        ~Sphere();
        static int piece() { return _piece; }
    protected:
        double multiplier() const override { return _multiplier; }
    private:
        constexpr static double _multiplier = (4.0 * 3.14159) /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

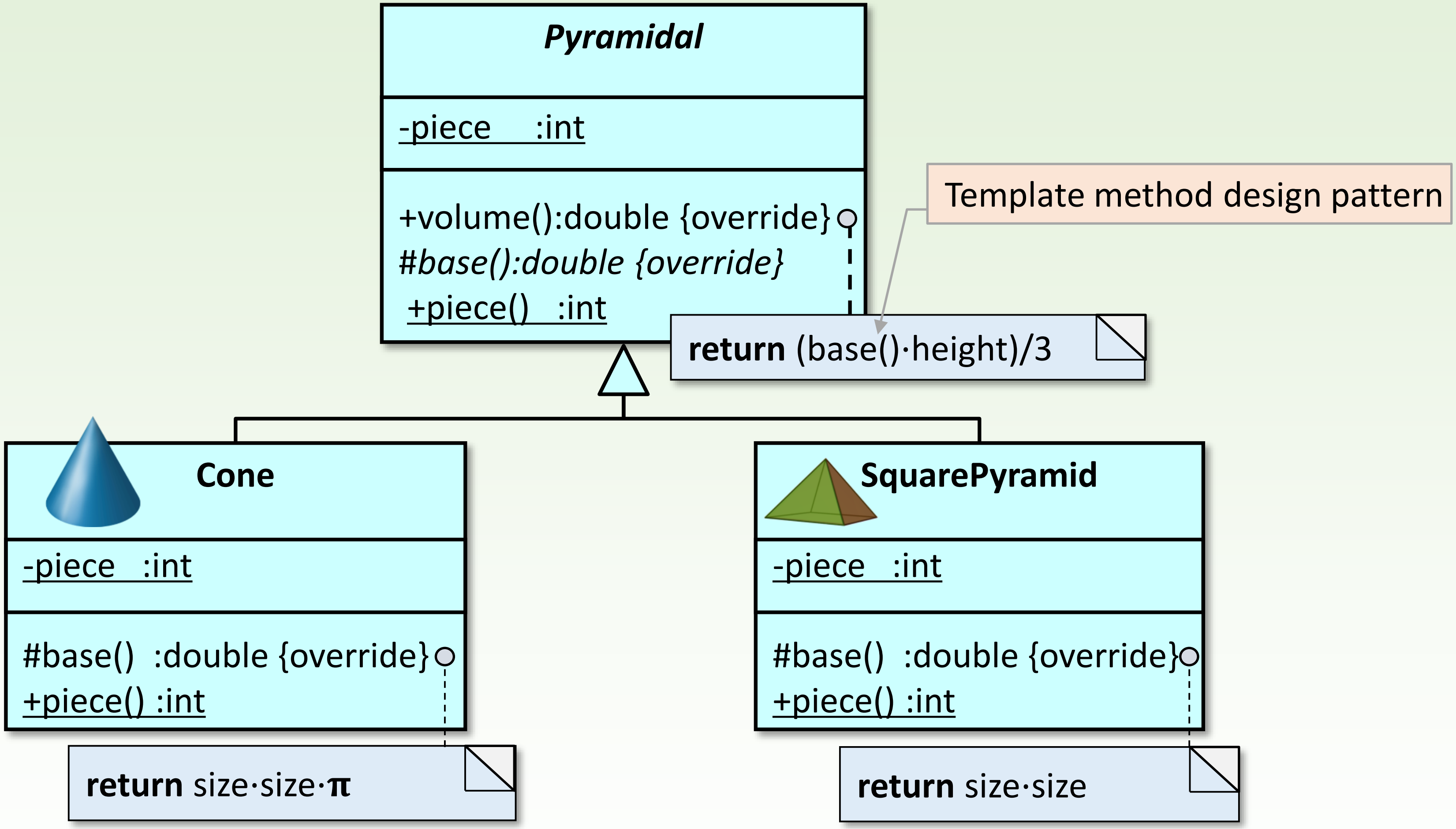
constant class-level expression

```
int Sphere::_piece = 0;

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

Prismatic shapes



Pyramidal shapes



Critique of the model

Redundancy has occurred: the area of a square is calculated in both SquarePyramid and SquarePrism; the area of a circle is calculated in both Cone and Cylinder.

Main program - population

```cpp
#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> is not good, as

1. Shape does not have empty constructor
2. Shape is abstract
3. the vector has to contain the reference or the pointer of the children of Shape, otherwise runtime polymorphism cannot be applied.

Instantiation of a shape

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

This could be class-level method of Shape if it needs to access the hidden part of class Shape.

A Cube* pointer can be assigned to a Shape* variable because of inheritance.

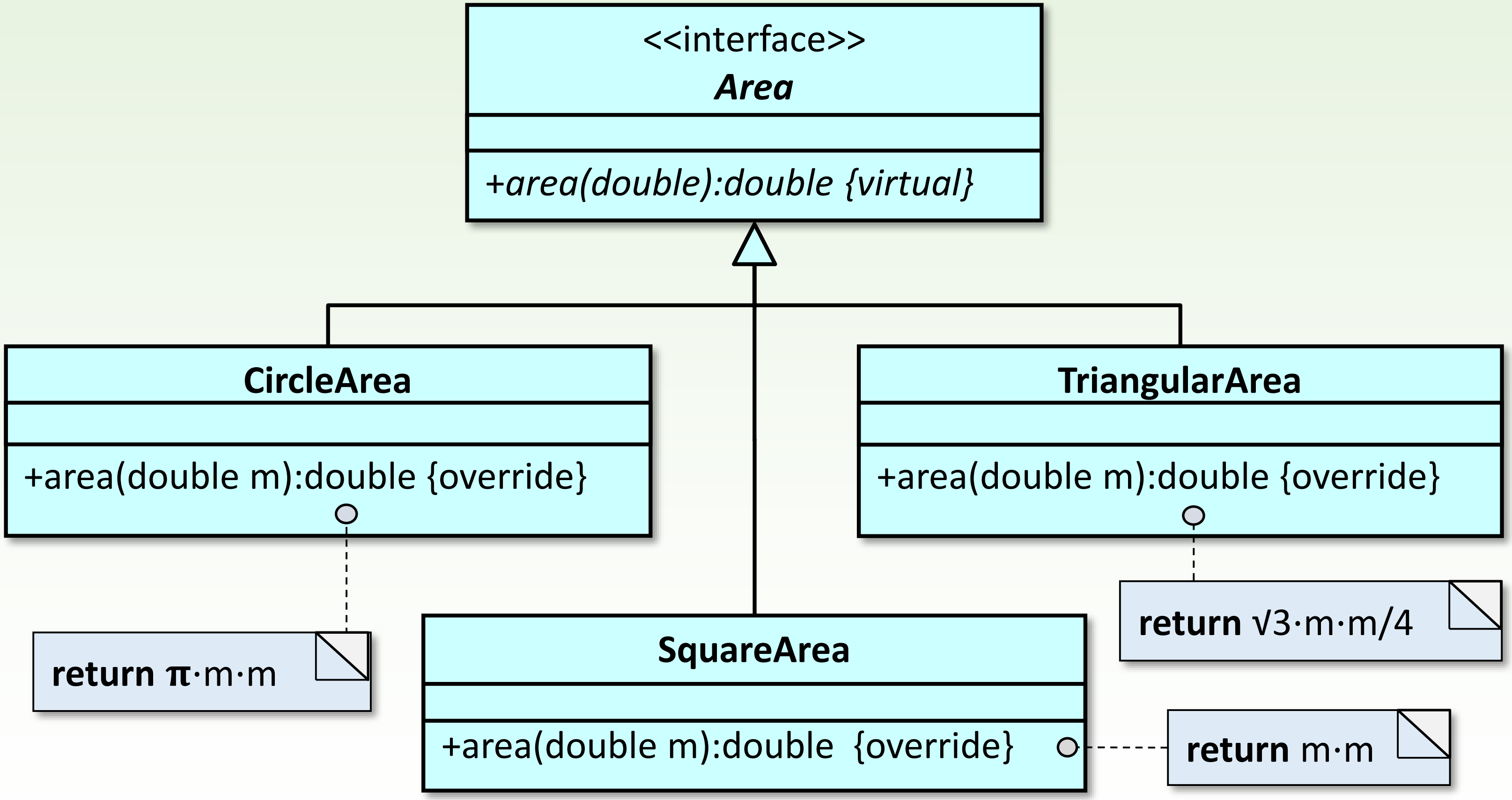
Main program - continue

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

    print_statistics();

    for ( Shape *p : shapes ) delete p;

    print_statistics();
}

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

This calculates the volume of the proper shape, instead of the virtual volume() of the base class Shape, because of runtime polymorphism.

The destructor of the proper shape runs instead of the virtual destructor of base class Shape, because of runtime polymorphism.

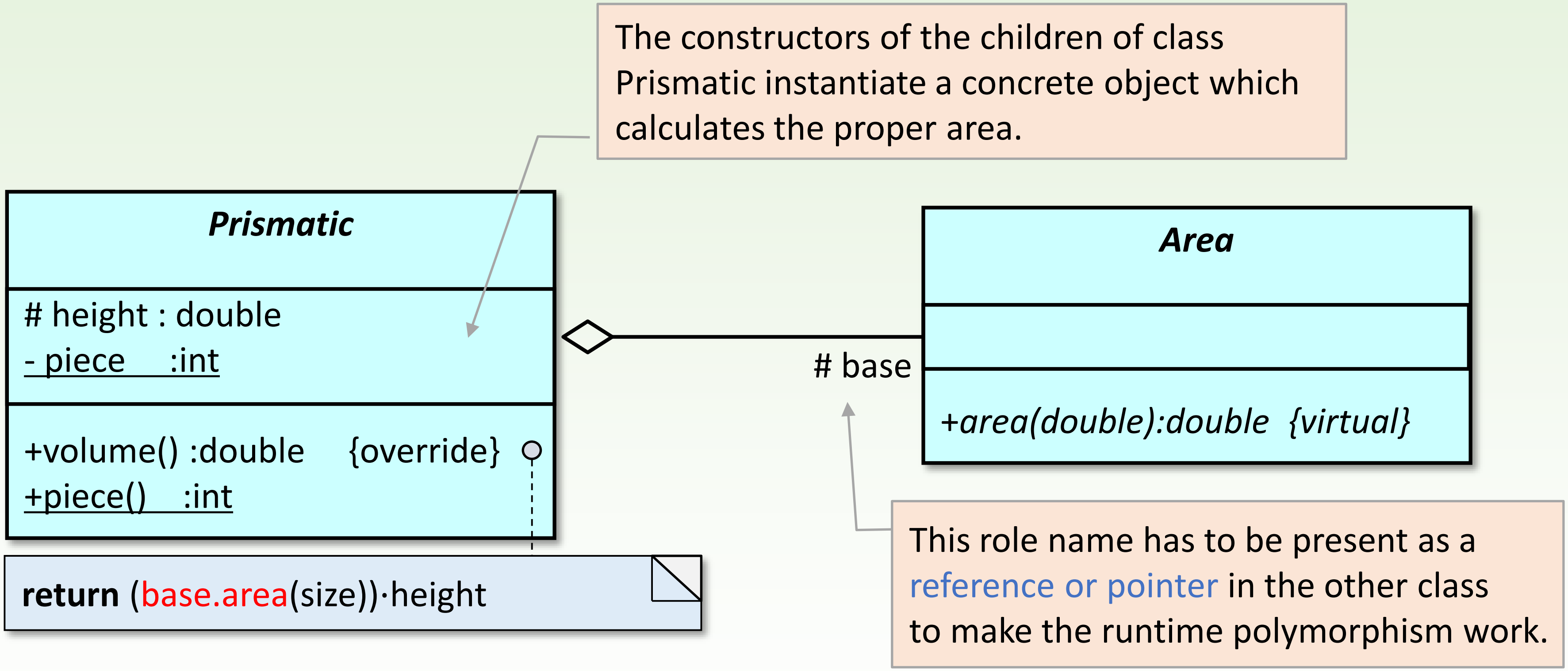
Redundancy elimination: objects instead of methods for area calculus



The area has to be defined here (and just here) – redundancy is eliminated.

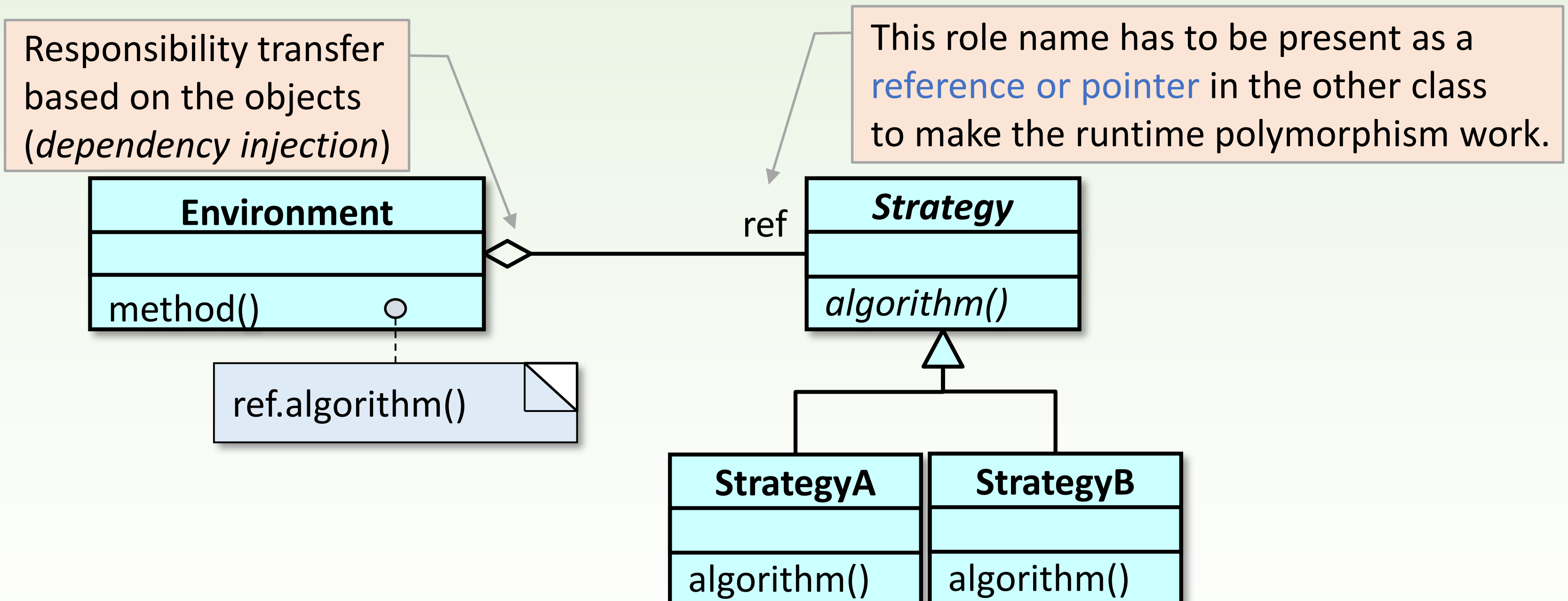
Dependency injection

- *Dependency injection* makes the behaviour (operation of methods) of an object dependent on a code written in another class.

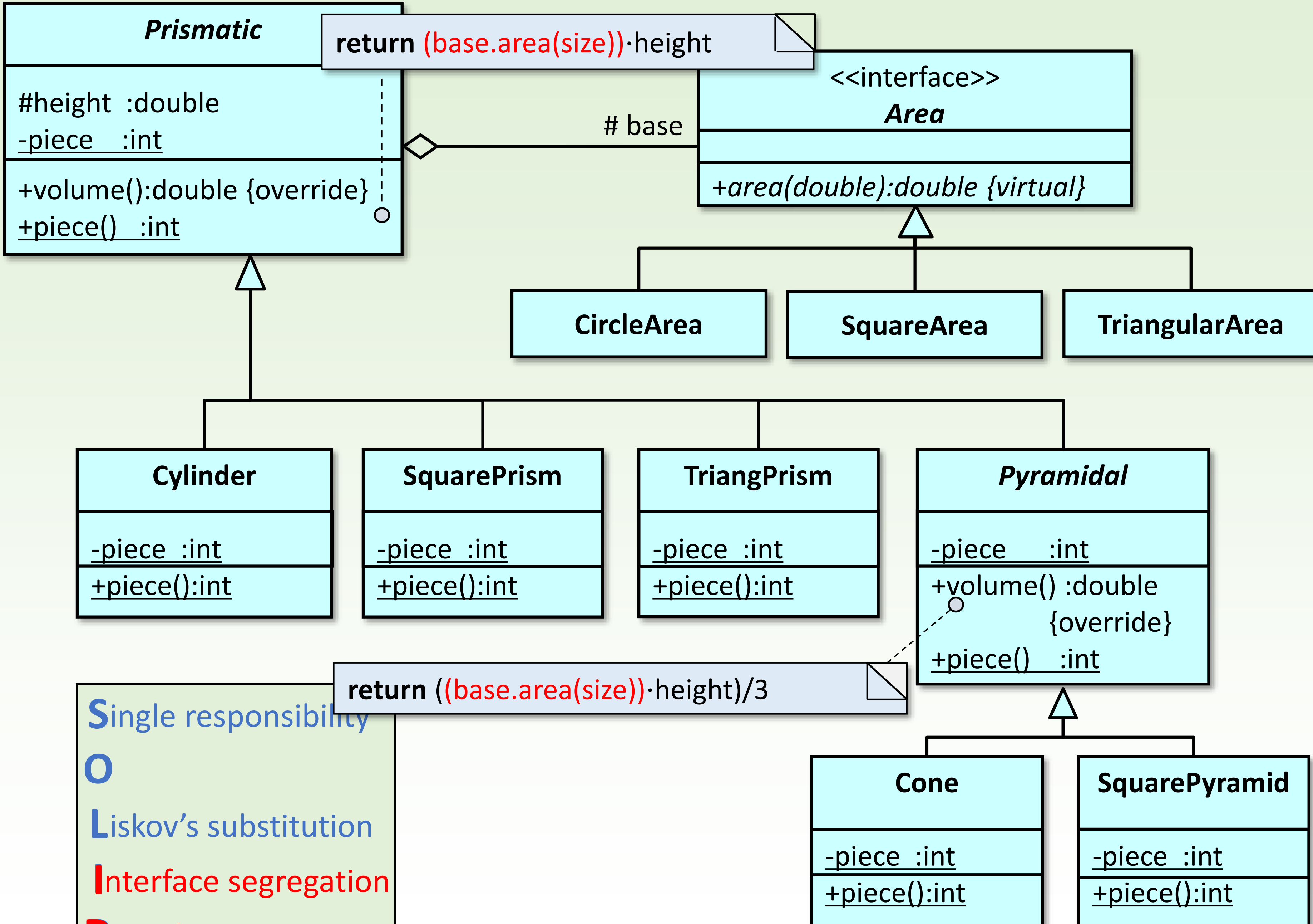


Strategy design pattern

- An algorithm-family is defined. During coding, we don't know yet which algorithm we are going to use.



Design patterns are class diagram patterns that help object-oriented modeling. They play a significant role in reusability, modifiability, and ensuring efficiency.



Single responsibility
O
Liskov's substitution
Interface segregation
Depedency inversion

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

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

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

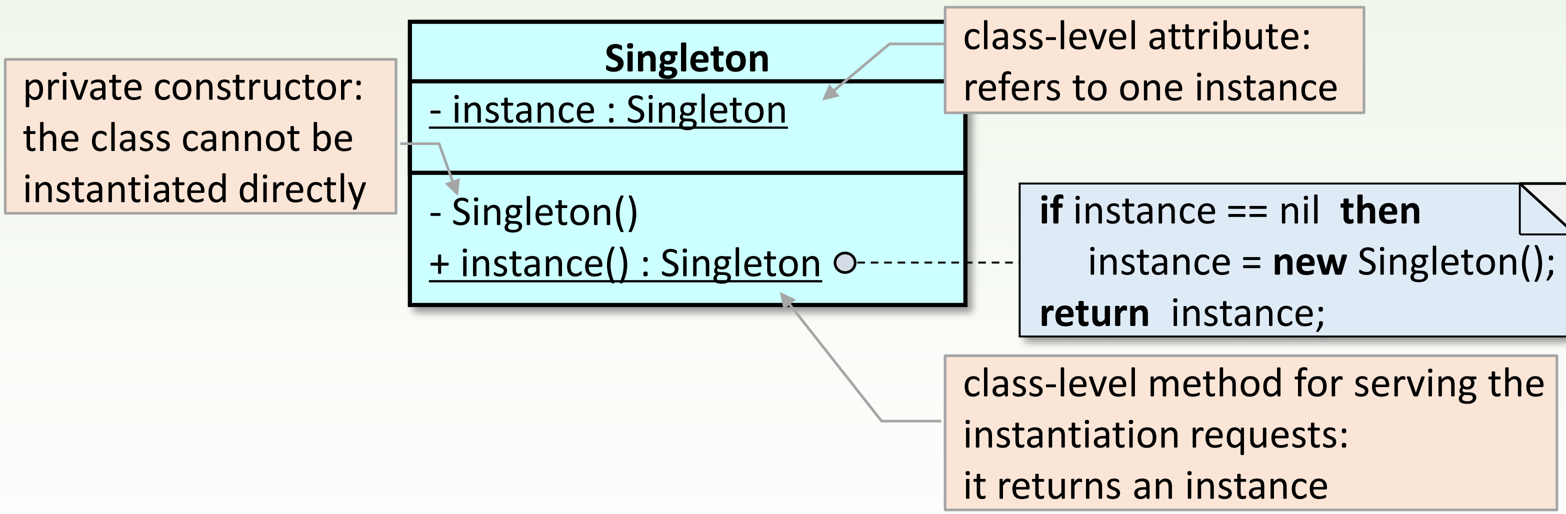
```
SquarePyramid::SquarePyramid(…) : Pyramidal(…){
    ++_piece; _base = new SquareArea();
}
SquarePyramid::~SquarePyramid(){
    --_piece; delete _base;
}
```

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

Critique:
Redundancy in the code is eliminated, but there is a waste of memory: for creating 5 cylinders and 3 cones, 8 SquareArea objects are instantiated, though one would be enough which could be used by all of them.

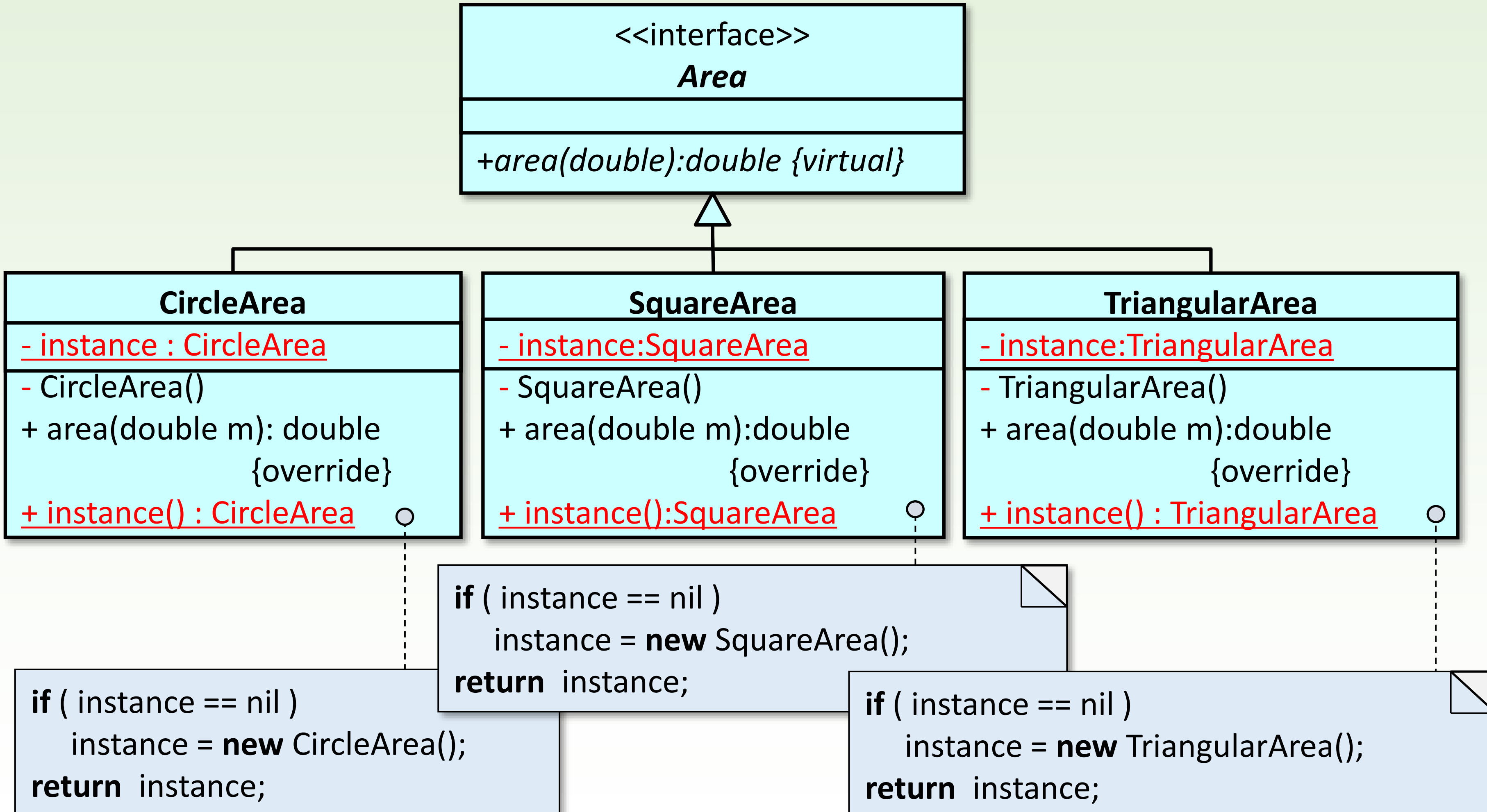
Singleton design pattern

- The class is instantiated only once, irrespectively of the number of instantiation requests.



Design patterns are class diagram patterns that help object-oriented modeling. They play a significant role in reusability, modifiability, and ensuring efficiency.

One object is enough to calculate the area of the base



SquareArea

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

pointer pointing to nowhere

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

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

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

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

instead of _base = **new** CircleArea()

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

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

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

2nd task

Create a program to model survival competition of creatures.

The creatures may belong to 3 species (greenfinch, dune beetle, squelchy). Every creature has a name (string) and health (natural number). The creatures (one after the other) pass a racetrack which consists of fields with different types of ground (sand, grass, marsh). When a creature passes on a ground, it may transmute it while its health changes. If the health of the creature falls to zero or less, it dies. Give the name of the creatures who survive the competition.

- ***Greenfinch***: *its health increases by one on grass, decreases by two on sand and by one on marsh. It transmutes marsh to grass.*
- ***Dune beetle***: *its health decreases by two on grass, by four on marsh, and increases by three on sand. It transmutes marsh to grass and grass to sand.*
- ***Squelchy***: *its health decreases by two on grass, by five on sand, and increases by six on marsh. It transmutes grass to marsh.*

Plan of the solution

a track before the *i*th creature: the *i*-1th state of the track

A : track: $Ground^m$, creatures: $Creature^n$, alive: String*

Pre : $creatures = creatures_0 \wedge track = track_0$

Post : $\forall i \in [1..n]: (creatures[i], track_i) = transmute(creatures_0[i], track_{i-1})$

$\wedge\ track = track_n$

$\wedge\ alive = \bigoplus_{\substack{i=1..n \\ creatures[i].alive()}} < creatures[i].name()>$

a track after the *i*th creature: the *i*th state of the track

Double summation (one conditional):

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

$f(e)_1$ ~ transmute(creatures[i], track)

$f(e)_2$ ~ <creatures[i].name()>

$cond(e)_2$ ~ creatures[i].alive()

H,+,0 ~ $Creature^n \times Ground^m$, $(\oplus, \ominus)$, $(<>, track_0)$

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

old $\ominus$ new ::= new

alive := <>

i = 1 .. n

creatures[i], track := transmute(creatures[i], track)

creatures[i].alive()

alive : write (creatures[i].name())	–

One creatures passes

*i*th creature goes through the fields of $track_{i-1}$ (as long as it lives), and every step transmutes the current ground while the creature itself changes, too.

$track_{i-1}$

state of creature[i] before the competition

A : track: $Ground^m$, creature: Creature

Pre : track = track' ∧ creature = $creature_0$

state of creature[i] after the *j-1*th field

Post: $\forall j \in [1..m]$: ($creature_j$, track[j]) = transmute($creature_{j-1}$, track'[j])

∧ creature = $creature_m$

$track_i$[j]

if ¬ $creature_j$.alive() then skip

state of creature[i] after the competition

creature, track[j] := transmute(creature, track[j])

creature, track := transmute(creature, track)

j := 1

creature.alive() ∧ j ≤ m

creature.transmute(track[j])

j := j+1

<u>Double summation as long as a condition</u>:

t:enor(E) ~ j = 1 .. m

as long as creature.alive()

f(e) ~ transmute(creature, track[j])

H,+,0 ~ Creature ×$Ground^m$, (⊜, ⊕),

($creature_0$, <>)

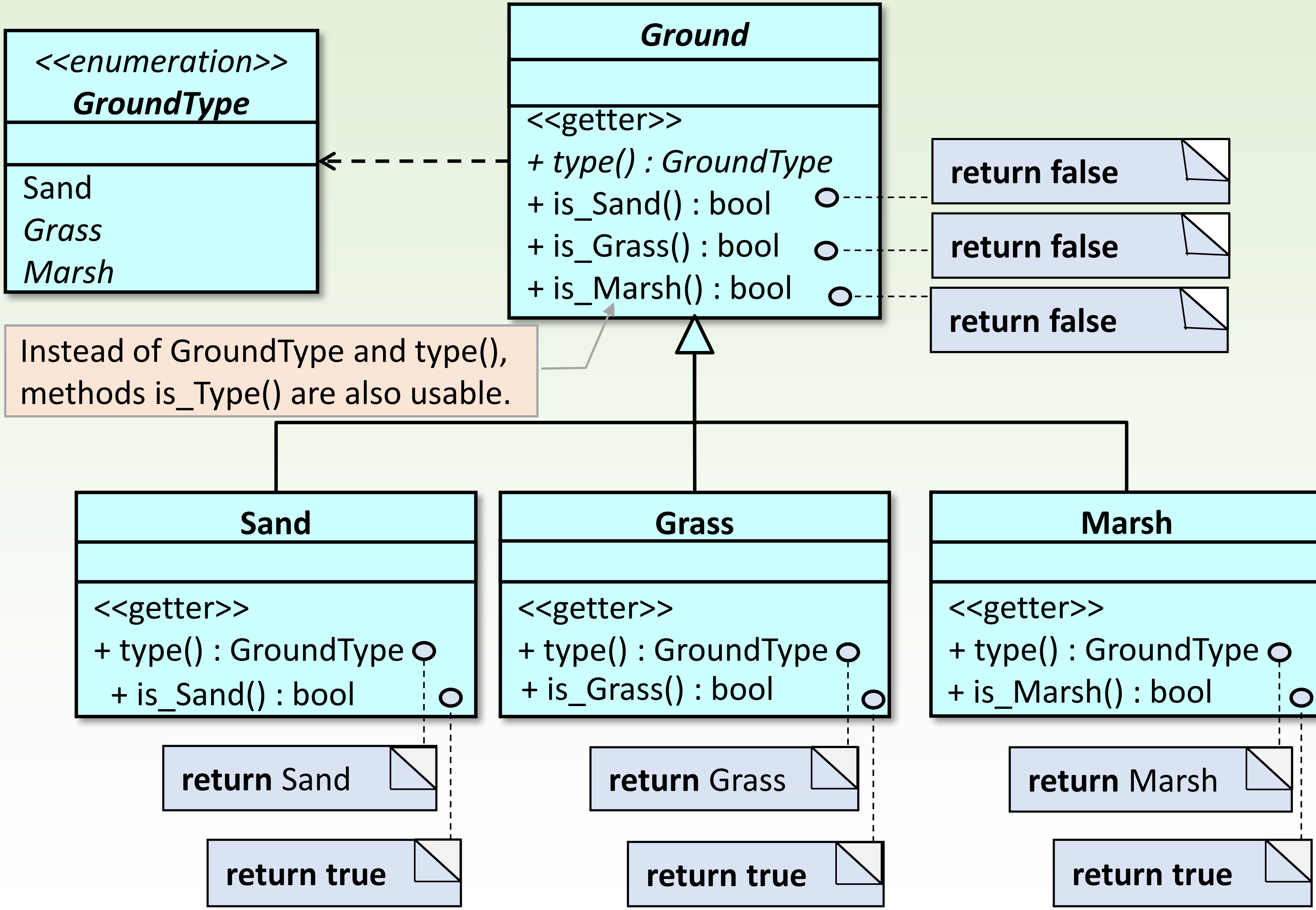
Main program

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

// Destruction
…
```

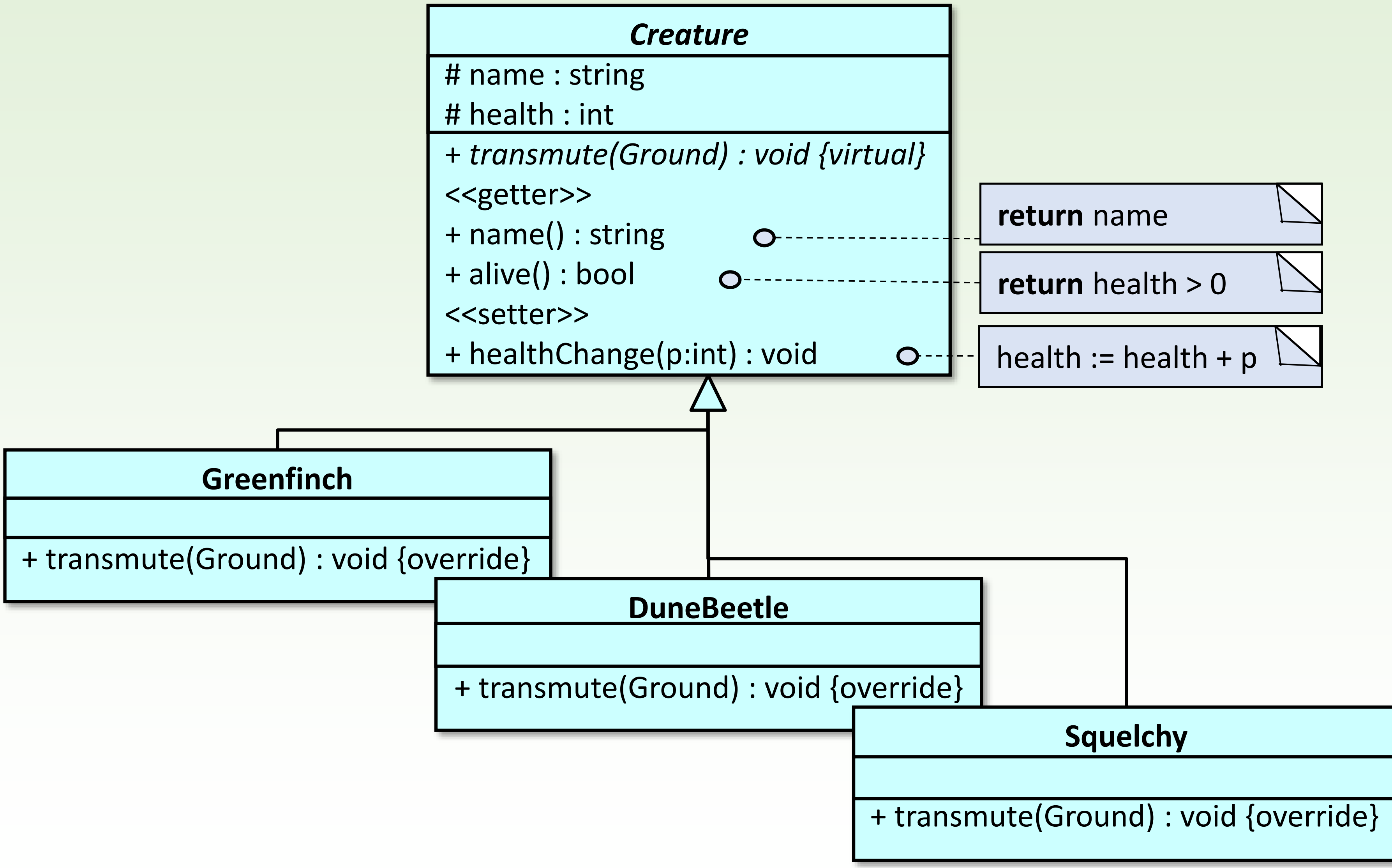
main.cpp

Here, runtime polymorphism would be handy as transmute() and alive() would depend on the actual type of creature[i].

Recognizing the subtypes



Inheritance of creatures



Creatures

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

creature.h

type of the ground is integer

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

creature.h

Interaction of creatures and ground types

greenfinch	health change	ground change
sand	-2	-
grass	+1	-
marsh	-1	grass

dune beetle	health change	ground change
sand	+3	-
grass	-2	sand
marsh	-4	grass

squelchy	health change	ground change
sand	-5	-
grass	-2	marsh
marsh	+6	-

Method transmute()

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

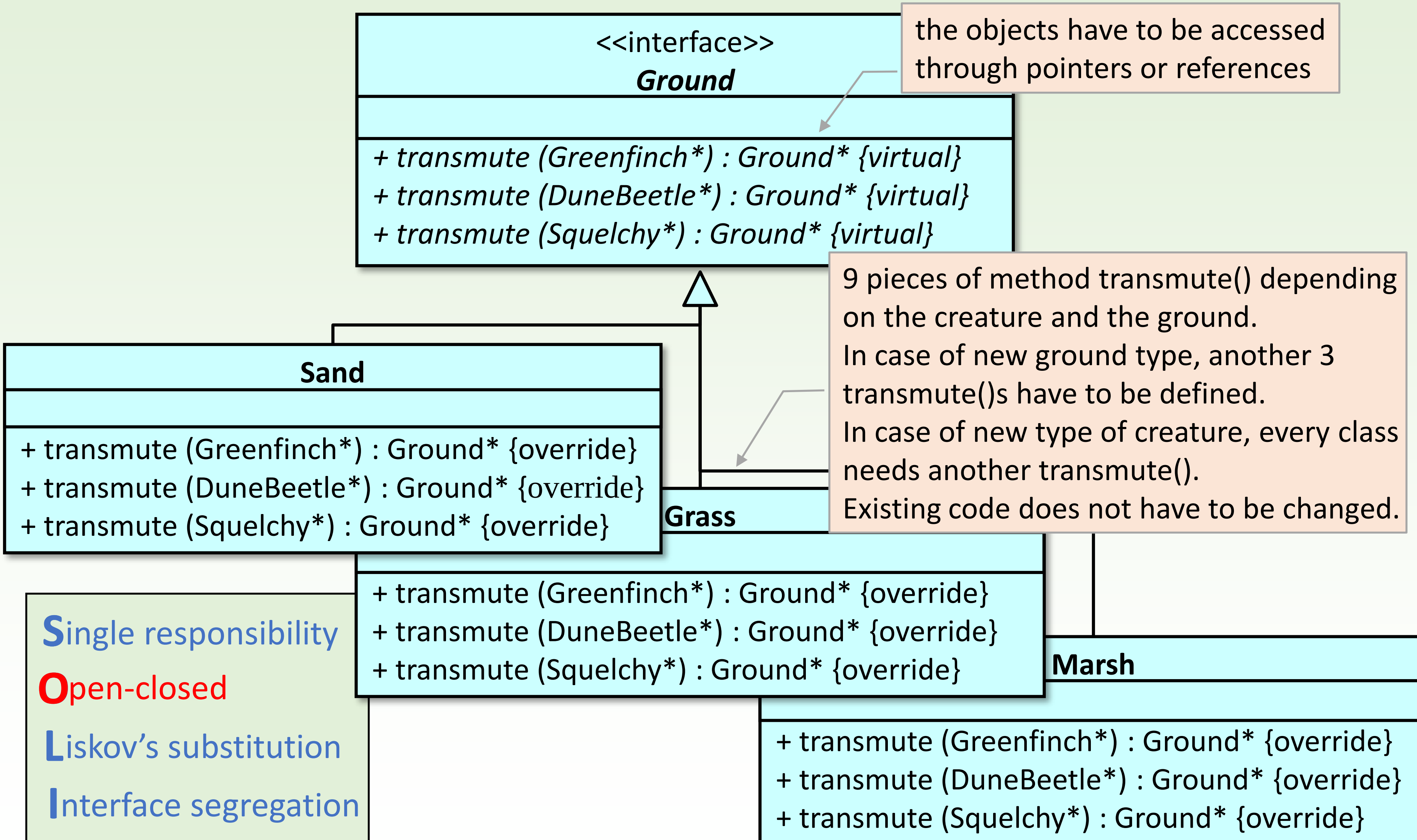
```
void DuneBeetle::transmute(int &ground)  {
     if (alive() ){
          switch(ground){
              case 0: _health +=3; break;
              case 1: _health -=2; ground = 0; break;
              case 2: _health -=4; ground = 1; break;
          }
```

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

Single responsibility
Open-closed
Liskov's substitution
Interface segregation
Depedency inversion

Critiques:
- difficult to read:
  for the ground, numbers are not meaningful
- not flexible:
  in case of a new ground type, the conditionals have to be modified, existing code has to be modified

Inheritance of grounds

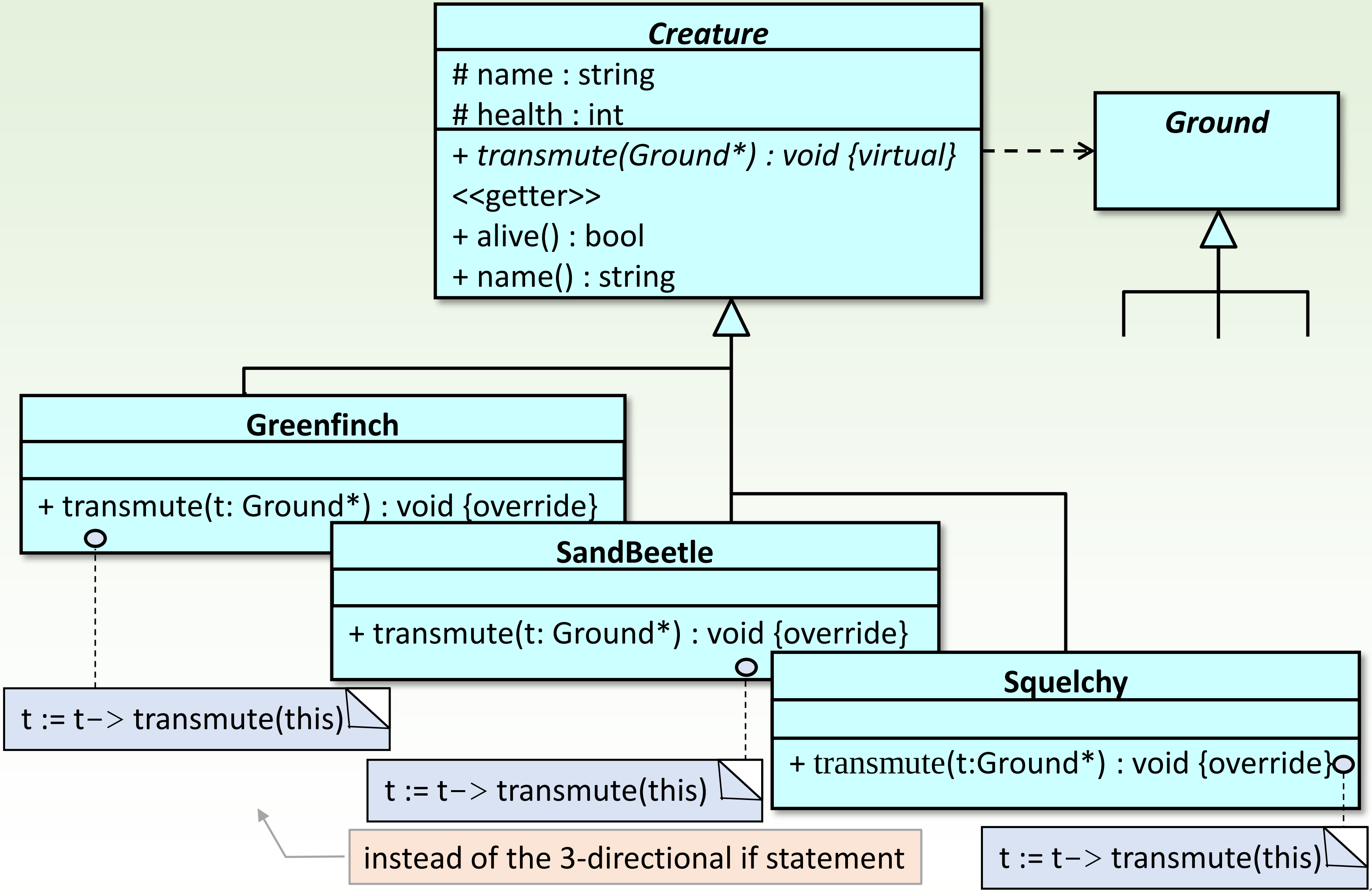


the objects have to be accessed through pointers or references

9 pieces of method transmute() depending on the creature and the ground.
In case of new ground type, another 3 transmute()s have to be defined.
In case of new type of creature, every class needs another transmute().
Existing code does not have to be changed.

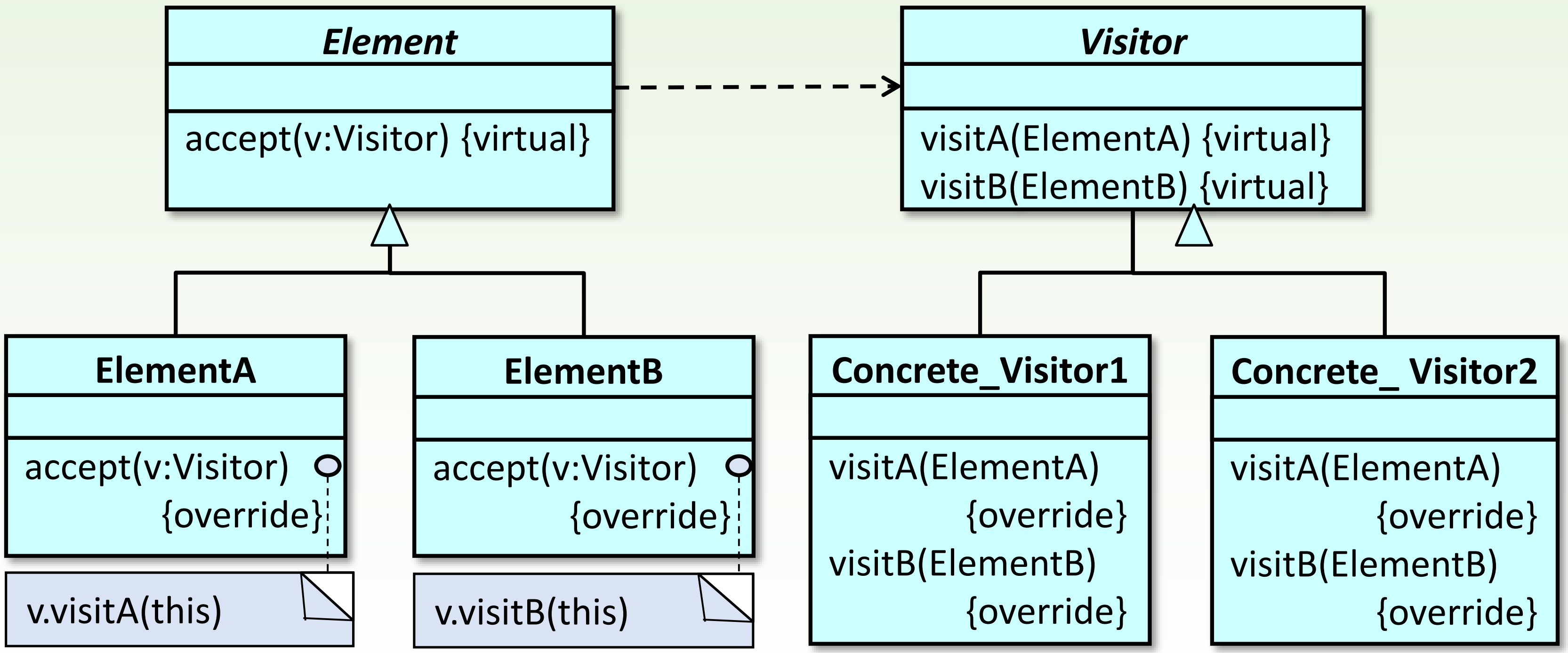
Single responsibility
Open-closed
Liskov's substitution
Interface segregation
Depedency inversion

Class diagram of creatures (again)



Visitor design pattern

- We use it when a method depends on which object of a collection is given as a parameter, and we do not wish to use a conditional that depends on the number of types of objects in the collection.



Design patterns are class diagram patterns that help object-oriented modeling. They play a significant role in reusability, modifiability, and ensuring efficiency.

Creatures with visitors

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

creature.h

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

Methods depending on the creature and the ground

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

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

Critique:
Several ground objects of the same type (e.g. **new** Grass is called many times). One sand, grass, and marsh object is enough.
In addition,
ground=ground -> transmute(this)
causes memory leaking.

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

ground.cpp

Singleton Grounds

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

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

instance of
new Sand
new Grass
new Marsh

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

ground.cpp

Population

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
```

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

instead of
new Sand
new Grass
new Marsh

main.cpp

Packages

#include "creature.h" would cause circular includes. It is enough to indicate that the classes exist.

```
#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