

Object-oriented programming

Teréz A. Várkonyi

<http://people.inf.elte.hu/treszka>

Procedural vs. Object-oriented programming

- **Procedural programming:** the solution is structured into independent units (subroutines, macros, procedures, functions). The execution is set by the control transfers (calls in case of procedures and functions) between these units.
- **Object-oriented program:** parts of the data needed for the solution and the related activities (called methods) are structured into independent units (called objects). The execution is set by the control transfers (calls or messages) between the methods of the objects.

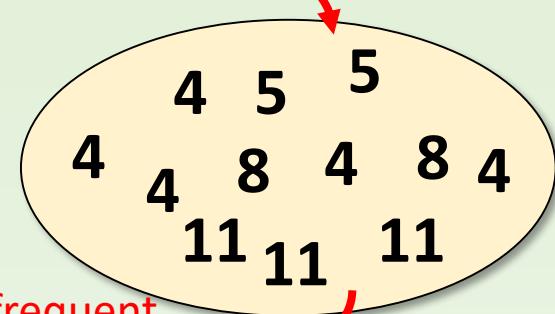
Task

In a series there are natural numbers between **0** and **m**. Which element is the most frequent?

- There are several ways to solve it, e.g.
 - **Procedural solution:** *counting* embedded in *maximum search*,
 - Let's find the maximum occurrence of the numbers between 0 and m ,
 - Let's find the maximum occurrence of the elements of the series.
 - **Object-oriented solution:** we place the elements into a container (collection) where insertion and maximum searching are quick.
 - Bag containing numbers between 0 and m (*set with multiplicity*),
 - Bag containing any natural number

Analysis and planning

insert: \cup



b:Bag

variables of the task and of the program at the same time

$A : m:\mathbb{N}, x:\mathbb{N}^*, \text{elem}:\mathbb{N}$

m 's initial value: m_0
 x 's initial value: x_0

most frequent element

x is not empty,
 x 's elements are between 0 and m

$Pre : m = m_0 \wedge x = x_0 \wedge |x| \geq 1 \wedge \forall i \in [1 .. |x|] : x[i] \in [0 .. m]$

initial value of the variables

$Post : Pre \wedge b:\text{Bag} \wedge b = \bigcup_{i=1}^{|x|} \{x[i]\} \wedge \text{elem} = \text{frequent}(b)$

final value of the variables

executive specification

Pre means that the inputs keep their initial values

Summation:

indexes:

$i \in [m .. n] \sim i \in [1 .. |x|]$

values:

$f(i) \sim \{x[i]\}$

result:

$s \sim b$

operator:

$H, +, 0 \sim \text{Bag}, \cup, \emptyset$

$b := \emptyset$

$i = 1 .. |x|$

$b := b \cup \{x[i]\}$

$\text{elem} := \text{frequent}(b)$

Testing strategies

❑ Black box: test cases based on the specification.

- test cases that violate the precondition
- test cases for the postcondition
- ...

❑ White box: test cases based on the code.

- cover every command
- cover every conditional
- ...

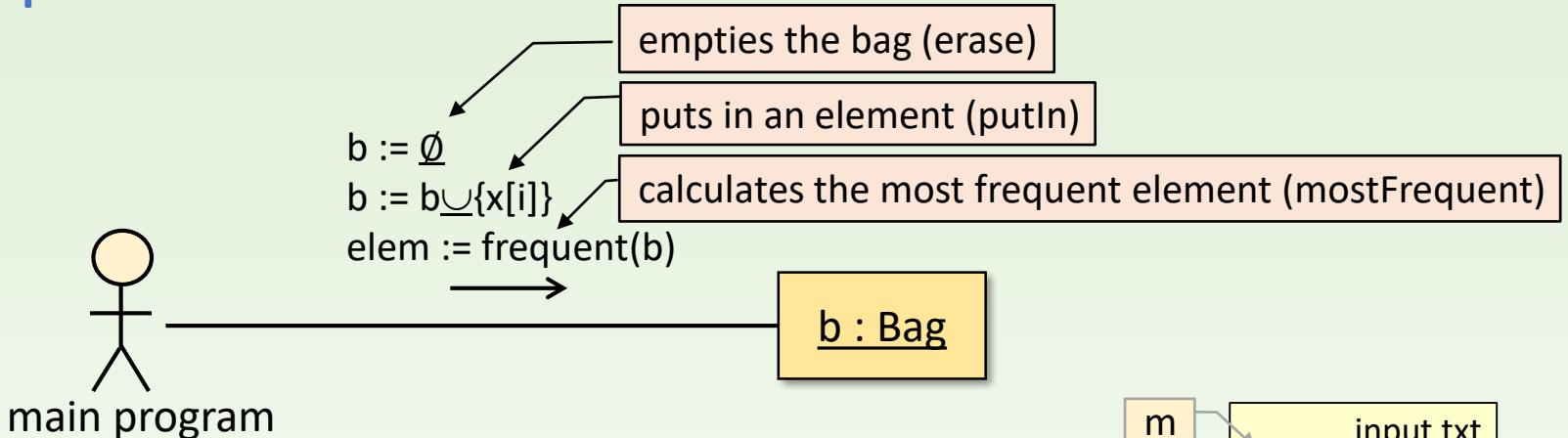
❑ Grey box: test cases based on a predicted algorithm from the executive specification

- If the specification refers to a famous algorithm then it is worthy to check the algorithm's typical test cases.

Test

Test cases of summation		test data (after bagging series x , we are looking for the most frequent element)		
interval [1 .. n]	length: 0	$ x = 0$	$x = < >$	\rightarrow invalid
	length: 1	$ x = 1$	$x = < 2 >$	$\rightarrow e=2$
	length: 4	$ x = 4$	$x = < 3, 5, 5, 1 >$	$\rightarrow e=5$
	beginning	$ x = 2$	$x = < 1, 2 >$	$\rightarrow e=1$
		$ x = 3$	$x = < 1, 1, 2 >$	$\rightarrow e=1$
	end	$ x = 3$	$x = < 1, 2, 2 >$	$\rightarrow e=2$
	middle	$ x = 4$	$x = < 1, 2, 2, 3 >$	$\rightarrow e=2$
algorithm	load	$ x = 10000$	$x = < 2, 2, \dots, 2 >$	$\rightarrow e=2$

Implementation

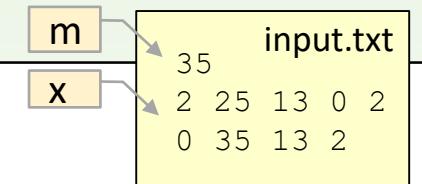


```
int main()
{
    ifstream f( "input.txt" );
    if(f.fail()) { cout << "Wrong file name!\n"; return 1; }

    int m; f >> m;

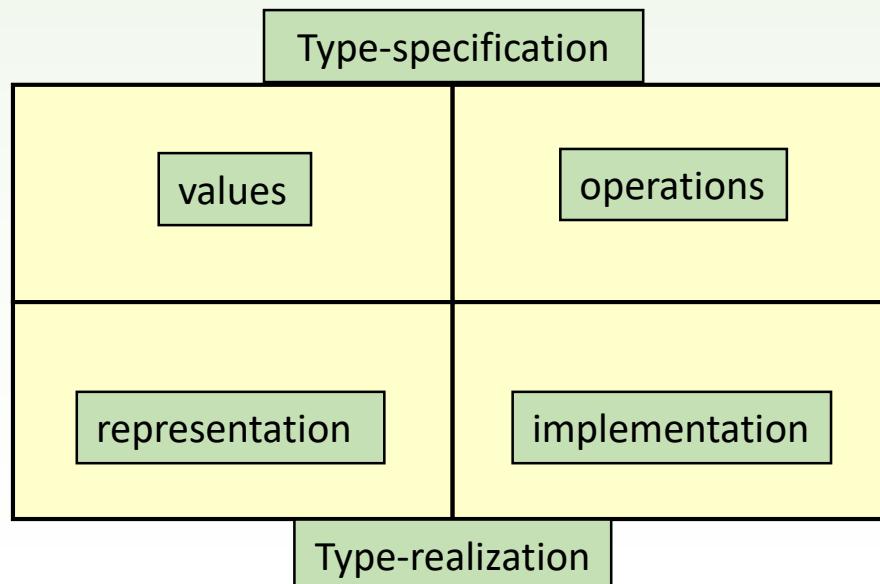
    Bag b(m);
    int e;
    b.erase();
    while(f >> e) { b.putIn(e); }

    cout << "The most frequent number: " << b.mostFrequent() << endl;
    return 0;
}
```



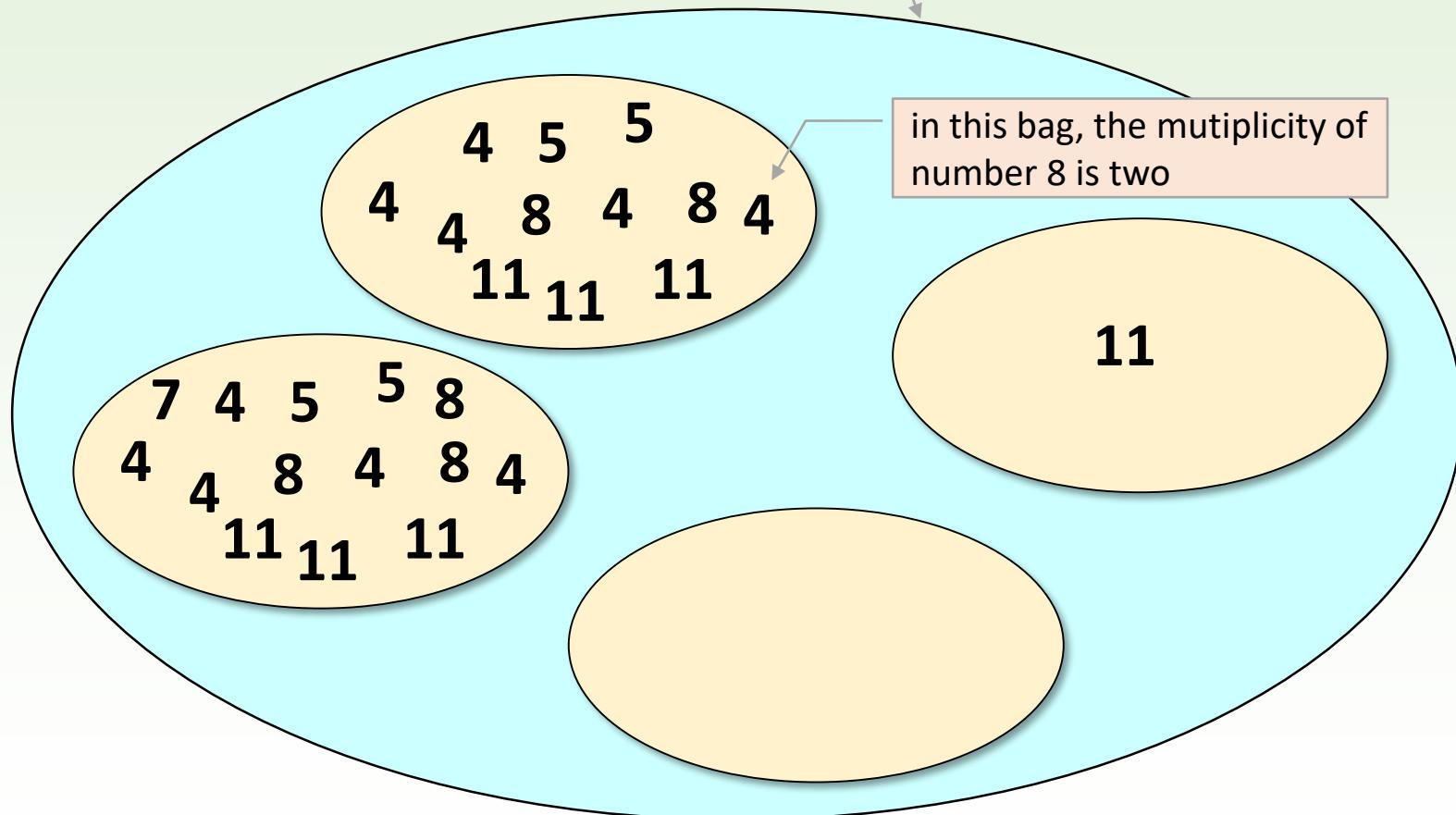
Datatype

- ❑ A datatype is given (**specified**) by its **values** and its **operations**.
- ❑ In certain cases we have to describe how we want to **represent** the values on the computer and how we **implement** the operations. These two are called **type-realization**.



Bag, set of values

The values that a Bag-type variable can have, together form the set of values of a Bag.



Bag, operations

Empty the bag (erase):

 $b := \emptyset$ $b: \text{Bag}$

notation for empty bag

Insert an element into a bag (putIn):

 $b := b \cup \{e\}$ $b: \text{Bag}, e: \mathbb{N}$

notation for „insert”

error if $e \notin [0 .. m]$

error if the
bag is empty

Most frequent element of the bag (mostFrequent) :

 $e := \text{frequent}(b)$ $b: \text{Bag}, e: \mathbb{N}$

operation for giving the
most frequent element

Representation

$b:$ 4 5 5
4 4 8 4 8 4
11 11 11

Here we take advantage of the fact that the elements of the bag are between 0 and m .

0 1 2 3 4 5 6 7 8 9 10 11 ... m

$vec:$ 0 0 0 0 5 2 0 0 2 0 0 3 ... : $\mathbb{N}^{0..m}$

$max:$ 4 : \mathbb{N}

we store separately the most frequent element

type invariant:

$max \in [0..m]$

$\forall i \in [0..m] \quad vec[i] \leq max$

properties of the representative attributes and their relation

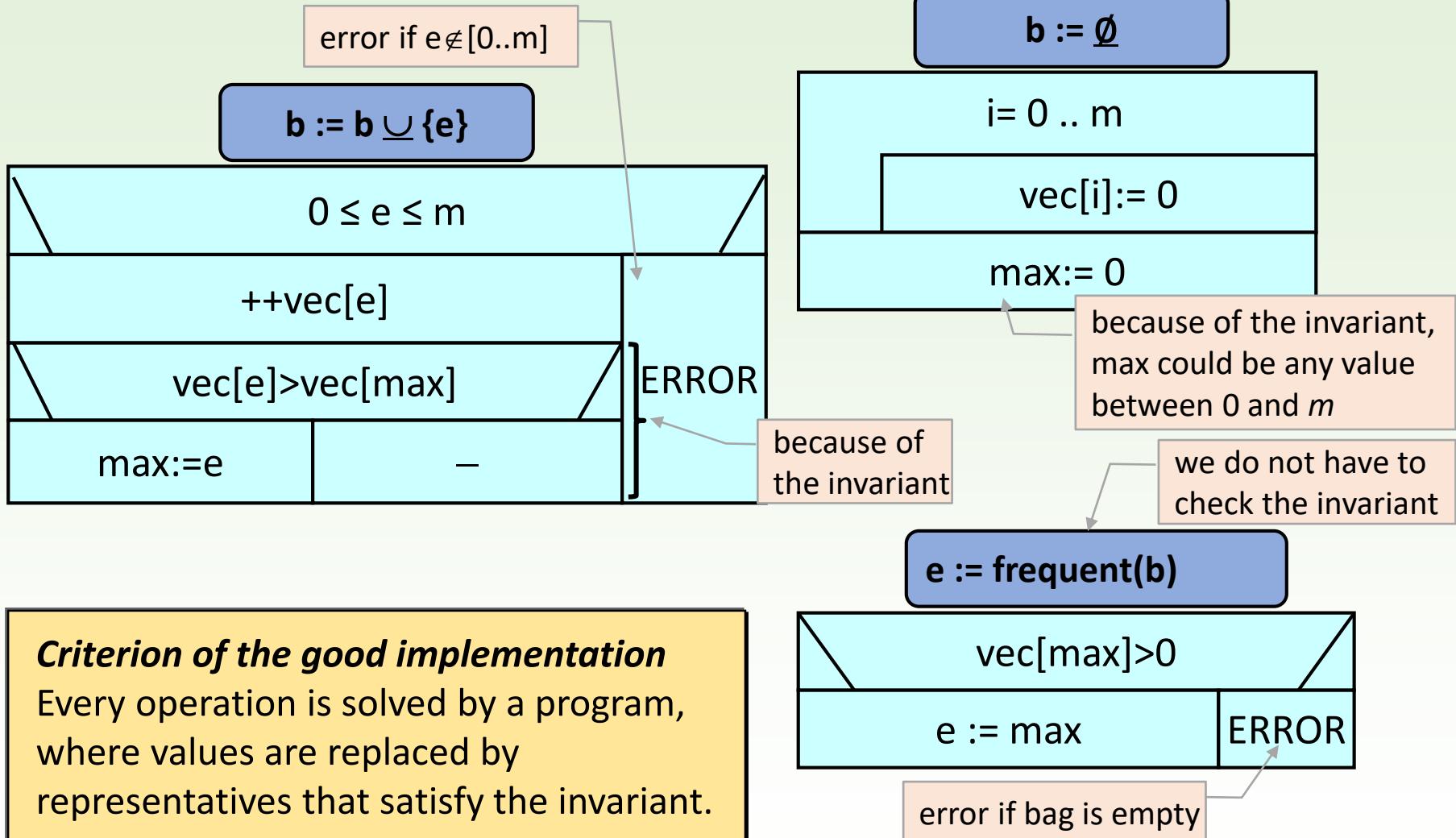
$vec[max] = 0$

means the empty bag

Criterion of the good representation

Every value (bag) has a representative (here a pair of an array and a maximum value) that satisfies the invariant, and every representative corresponds to a value.

Implementation



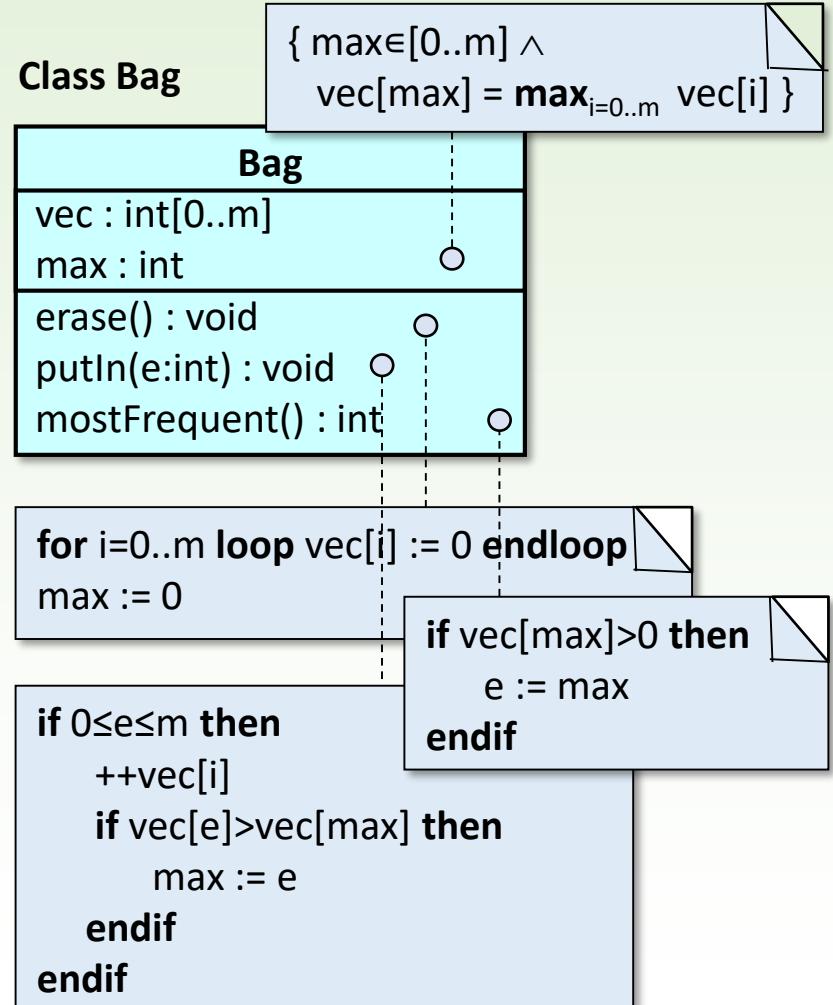
Type and class

- In case of object-oriented planning, type is realized by a class.

Type Bag

Type-specification		
values	bags	operations
representation	$\text{vec : } \mathbb{N}^{0..m}$ $\text{max : } \mathbb{N}$ $\{ \text{max} \in [0..m] \wedge \text{vec}[\text{max}] = \max_{i=0..m} \text{vec}[i] \}$	$b := \emptyset$ $b := b \cup \{e\}$ $e := \text{frequent}(b)$ $b: \text{Bag}, e: \mathbb{N}$
Type-realization		implementation

Class Bag



UML diagram of a class

□ Description of a class needs

- its **name**, **attributes**, **methods** and their visibility, which can be
 - seen from outside: *public* (+)
 - hidden from outside: *private* (-) or *protected* (#)

<class name>

<+|-|#> <attribute name> : <type>

...

<+|-|#> <method name>(<parameters>) : <type>

...

Bag

- vec : int[0..m]

- max : int

+ erase() : void

+ putIn(e:int) : void

+ mostFrequent() : int

Testing of type Bag

Test of Bag

erase():

- empty bag is created: `b.mostFrequent()=0`

putIn():

- put an element into an empty bag
- put an existing element into a non-empty bag
- put a non-existing element into a non-empty bag
- put 0 and m into the bag
- put an illegal element into the bag

Invariant has to be satisfied
every time we put in an element.

mostFrequent(): (like a maximum search)

- in case of empty bag
- in case of bag with one element
- in case of bag with more elements and the most frequent one is obvious
- in case of bag with more elements, the most frequent one is not obvious
- the most frequent element is 0 or m

Integration testing (modules a tested together as a group)

- after `b.erase()`, `b:=b.putIn(e)` is run many times
- after `b.erase()`, `e:=b.mostFrequent()` is run immediately

Test cases for Bag

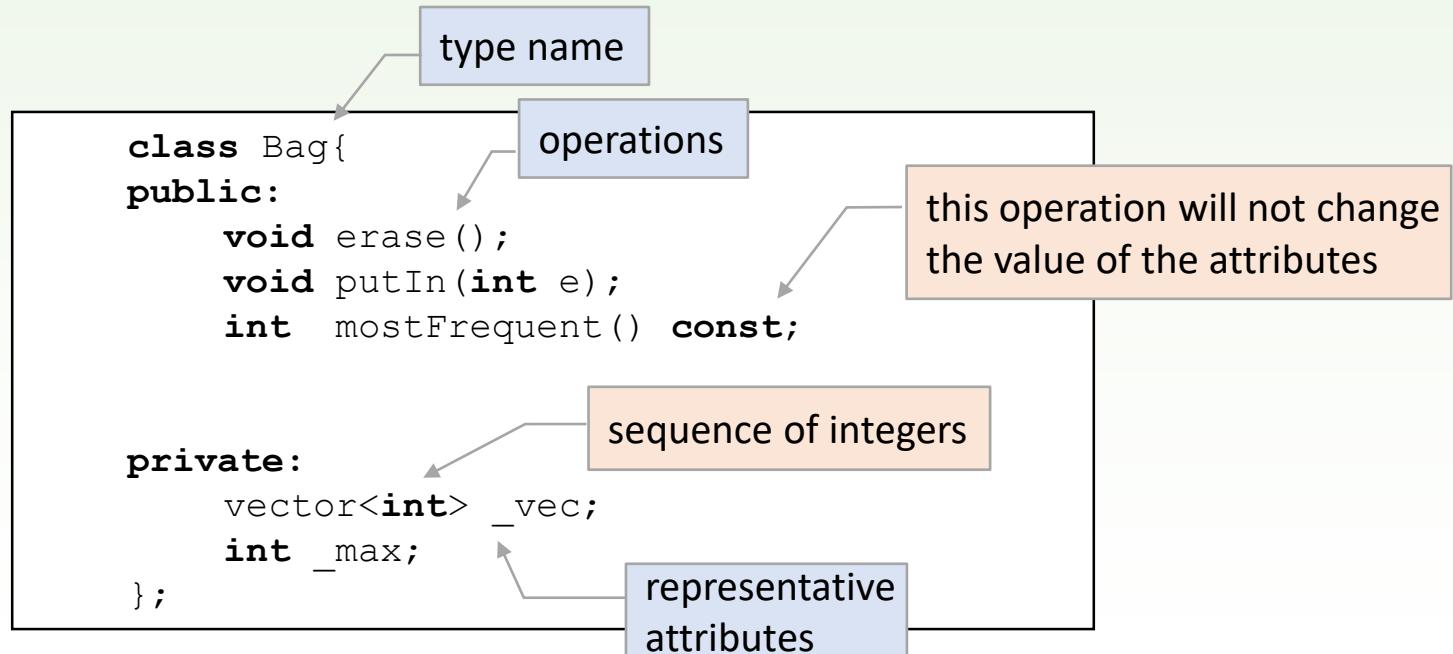
mostFrequent()	Test data		
empty bag	m = 1	vec = < 0, 0 >	→ error
one element	m = 1	vec = < 1, 0 >	→ 0
obvious maximum	m = 1	vec = < 2, 1 >	→ 0
more maxima	m = 1	vec = < 2, 2 >	→ 0 ∨ 1
max is 0	m = 1	vec = < 1, 0 >	→ 0
max is m	m = 1	vec = < 0, 1 >	→ 1

erase()	Test data		
create empty bag	m = 0	→ vec = < 0 >	max = 0
	m = 1	→ vec = < 0, 0 >	max = 0 ∨ 1
	m = 4	→ vec = < 0, 0, 0, 0, 0 >	max = 0

putIn()	Test data					
first	m = 1	vec = < 0, 0 >	putIn(0)	→ vec = < 1, 0 >	max = 0	
new	m = 1	vec = < 1, 0 >	putIn(1)	→ vec = < 1, 1 >	max = 0	
existing	m = 1	vec = < 1, 1 >	putIn(1)	→ vec = < 1, 2 >	max = 1	
0	m = 1	vec = < 1, 1 >	putIn(0)	→ vec = < 2, 1 >	max = 0	
m	m = 1	vec = < 1, 1 >	putIn(1)	→ vec = < 1, 2 >	max = 1	
illegal	m = 1	vec = < 1, 2 >	putIn(2)	→ error		

Class from type

- In object-oriented languages, **custom types** are realized by **classes**.
 - the **name** of the class has to match the name of the type
 - the attributes of the representation are the **attributes** of the class (with name and type)
 - the operations are the **methods** of the class



Basic terms

- ❑ Object is responsible for a part of the task, that contains the corresponding data and operations.

vec and max

erase, putIn, mostFrequent

e.g. the bag

- ❑ Class gives the sample structure and behaviour of an object

- enumerates the object's attributes (name and type)
- gives the methods that can be called on the object

vec is an array, max is an integer

- ❑ Class is like a data type of the object: object is created based on its class, it is instantiated.

- ❑ Based on a class, more objects can be instantiated.

*erase() : void,
putIn(int) : void,
mostFrequent() : int*

1

Important criterion of object-orientation is encapsulation: the data and its manipulation needed for a domain are given as a unit, separately from the other parts of the program.

Visibility

```
class Bag{  
public:  
    void erase();  
    void putIn(int e);  
    int mostFrequent() const;  
private:  
    vector<int> _vec;  
    int _max;  
};  
  
int main()  
{  
    Bag b;  
    b.erase();  
    b.putIn(5);  
    int a = b.mostFrequent();  
  
    b._vec[5]++;  
}
```

Methods of an object are always called on the object itself (which is a special parameter of the method). In the body of the method we can access the attributes and we can call other methods of the object.

Outside the class the private attributes are unavailable.

Important criterion of object-orientation is **data hiding**: the visibility of the encapsulated elements are restricted. (Usually, attributes are hidden, they are only accessible through public methods.)

Constructor

When an object is created ([instantiated](#)), a special method called [constructor](#) is called, that runs the constructor of the attributes.

```
class Bag {  
public:  
    Bag (int m);  
    ...  
private:  
    vector<int> _vec;  
    int _max;  
};
```

Every object, as long as other constructor is not set, has an empty constructor, that does not need any parameters. It just calls the empty constructor of the attributes. In our case, declaration *Bag b* is meaningful, it creates a vector of length 0 and an integer. Unfortunately, it is not enough for us.

A constructor is needed where the length of the array can be set. Command *Bag b(21)* would instantiate the bag with an array of length 22.

it belongs to class Bag

the constructor resizes the vector of length 0 to length of $m+1$ and runs method *erase()*.

```
Bag::Bag (int m) { _vec.resize(m+1); erase(); }
```

it does the job of *erase()*, too

```
Bag::Bag (int m) { _vec.resize(m+1, 0); _max = 0; }
```

not empty constructor of the vector and the int

```
Bag::Bag (int m) : _vec(m+1, 0), _max(0) { }
```

Class of Bag in details

The diagram shows a C++ code snippet for a `Bag` class. The code includes a `#pragma once`, an `#include <vector>`, and the class definition with `public:` and `private:` sections. Annotations include a box for "header files and guards" pointing to the `#pragma` and `#include` lines, a box for "error cases:" and "enum is a new type that has only values" pointing to the `enum Errors` declaration, and a box at the bottom pointing to the `using namespace std;` implication. A blue box labeled "bag.h" is at the bottom right.

```
#pragma once
#include <vector>

class Bag {
public:
    enum Errors{EmptyBag, WrongInput};

    Bag(int m);
    void erase();
    void putIn(int e);
    int mostFrequent() const;

private:
    std::vector<int> _vec;
    int _max;
};


```

header files and guards

error cases:
`enum` is a new type that has only values

In the header files *using namespace std* is not common.
We have to indicate that the definition of vector is in `namespace std`

bag.h

Methods of class Bag

Class definition of *Bag* is needed

```
#include "bag.h"

Bag::Bag(int m) : _vec(m+1, 0), _max(0) { }

void Bag::erase() {
    for(unsigned int i=0; i<_vec.size(); ++i) _vec[i] = 0;
    _max = 0;
}

void Bag::putIn(int e) {
    if( e<0 || e>=int(_vec.size()) ) throw WrongInput;
    if( ++_vec[e] > _vec[_max] ) _max = e;
}

int Bag::mostFrequent() const {
    if( 0 ==_vec[_max] ) throw EmptyBag;
    return _max;
}
```

length of `_vec`

cast to int

throws an exception: indicates the error but does not handle it

throws an exception

belongs to class Bag

bag.cpp

Main program

```
#include <iostream>
#include <fstream>
#include "bag.h"
using namespace std;

int main()
{
    ifstream f( "input.txt" );
    if(f.fail()) { cout << "Wrong file name!\n"; return 1; }
    int m; f >> m;
    if(m<0) {
        cout << "Upper limit of natural numbers cannot be negative!\n";
        return 1;
    }
    try{
        Bag b(m);
        int e;
        while(f >> e) { b.putIn(e); }
        cout << "The most frequented element: " << b.mostFrequent() << endl;
    }catch(Bag::Errors ex){
        if(ex==Bag::WrongInput){ cout << "Illegal integer!\n"; }
        else if(ex==Bag::EmptyBag){ cout << "No input no maximum!\n"; }
    }
    return 0;
}
```

Class definition of *Bag* is needed

exception filter

catches an exception and handles it

main.cpp

Automatic test

```
#include <iostream>
#include <fstream>
#include "bag.h"
using namespace std;

#define CATCH_CONFIG_MAIN
#include "catch.hpp"

TEST_CASE("empty file", "[sum]") {
    ifstream f( "input1.txt" );
    int m; f >> m;
    Bag b(m); b.erase();
    int e;
    while(f >> e){ b.putIn(e); }
    CHECK_THROWS(b.mostFrequent());
}

TEST_CASE("one element in the file", "[sum]") {
    ifstream f( "input2.txt" );
    int m; f >> m;
    Bag b(m); b.erase();
    int e;
    while(f >> e){ b.putIn(e); }

    CHECK(b.mostFrequent() == 2);
}

...
```

summation
length if interval: 0
 $f = \langle 15 \rangle$ ($m = 15$)
 $b = \{ \}$ \rightarrow no frequent element
throw `Bag::EmptyBag`

summation
length if interval: 1
 $f = \langle 15, 2 \rangle$ ($m = 15$)
 $b = \{ 2(1) \}$ \rightarrow `mostFrequent = 2`

Automatic test

```
TEST_CASE("create an empty bag", "[bag]")
{
```

```
    int m = 0;
    Bag b(m);
    vector<int> v = { 0 };
    CHECK(v == b.getArray());
```

bag()

create an empty bag:

m = 0 → **_vec = < 0 >**

```
TEST_CASE("new element into empty bag", "[putIn]")
{
```

```
    Bag b(1);
    b.putIn(0);
    vector<int> v = { 1, 0 };
    CHECK(v == b.getArray());
```

getArray() may come
in handy for the testing

putIn()

new element into empty bag:

m = 1, e = 0 → **_vec = < 1, 0 >**

```
}
```

```
...
```

```
class Bag {
private:
    std::vector<int> _vec;
    int _max;
public:
    ...
    const std::vector<int>& getArray() const { return _vec; }
};
```

Not a nice solution, it is like “littering” in the code. It would be more elegant to create a class `Bag_Test` that inherits all the properties of class `Bag`. Then, it could be extended with method `getArray()`. The test cases could use class `Bag_Test` instead of `Bag`.

Instantiation of objects

Double meaning of object

In modeling

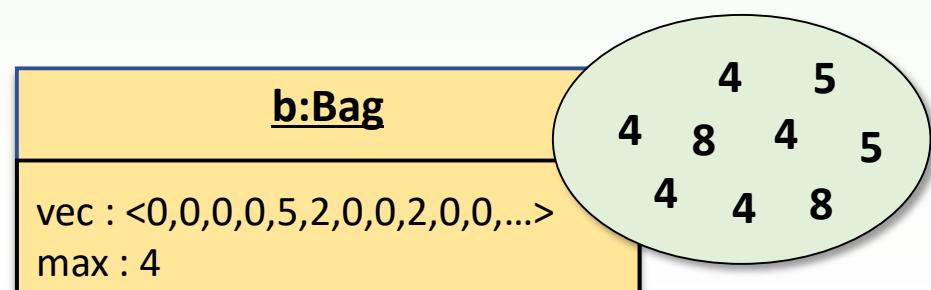
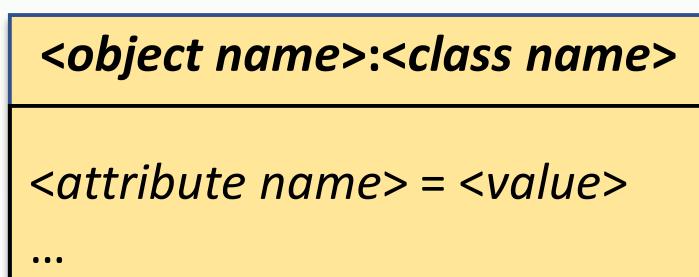
- ❑ Object is a **unique** part of the task to be solved.
- ❑ Object is responsible for a part of the data: it hides them and handles (reads and modifies) them exclusively through its methods.
- ❑ An object is created and later destroyed.

In programming languages

- ❑ Object is the **memory allocation**, where its data is stored.
- ❑ The memory area of the object is only accessable through its methods (except if the data is public).
- ❑ The memory allocation (instantiation) of the object is done by its constructor, the destruction is by its destructor.

Notation for object in UML

- ❑ An object (the unit responsible for part of the task) is specified by its
 - **class**, the set of objects with the same attributes and methods. It describes
 - the **data** (properties, attributes, fields) of the objects in name-type pairs
 - the **methods** (operations, member functions) that can be called on the object (to manipulate the attributes of the object).
 - **name** (non obligatory),
 - **state** (set by all the attribute name-value pairs).



Task

Let us create program that fills an array with polygons. Then it moves all of them along the same vector. Finally, the program calculates the center of the moved polygons. The coordinates of the vertices and the centers should be integers.

Objects:

- **polygons**
- **planar points**, vertices and centers
- **array of polygons**

Single responsibility

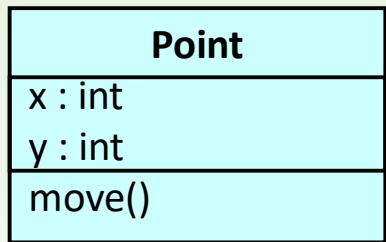
O
L
I
D

Responsibility of the objects:

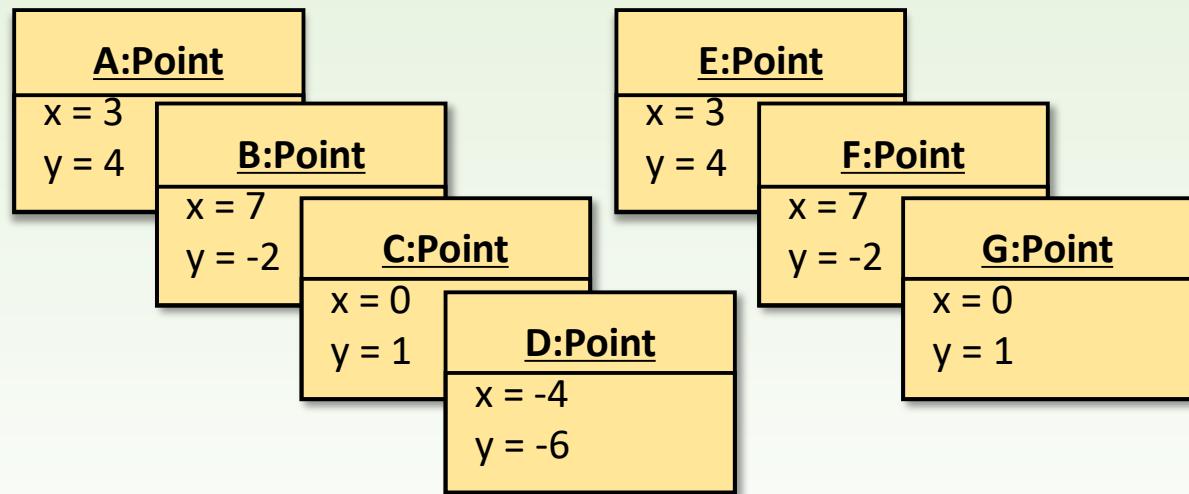
- **polygons:** **move**, **center calculation**, **get** and **set** 1. the **number of vertices** or 2. a **given vertex**
- **planar points:** **move**, **get** and **set** its coordinates
- **array:** **get** the element at a **given index**, **get** its **size**

Objects of the task

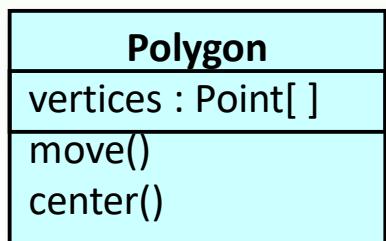
Class Point:



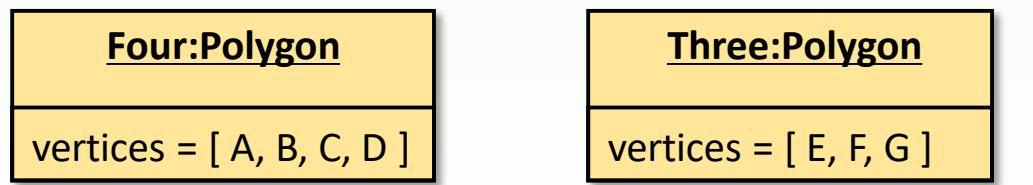
Point objects:



Class polygon:



Polygon objects:



Level of detail of a class description

- Class description evolves gradually during modeling, some details might be missing on a certain level.
 - There are no attributes and/or methods.
(For enumerations, already only values are given.)
 - There are no attribute types, method return types, method parameters.
 - There are no notations for visibility.

<class name>

<class name>

<method name>()

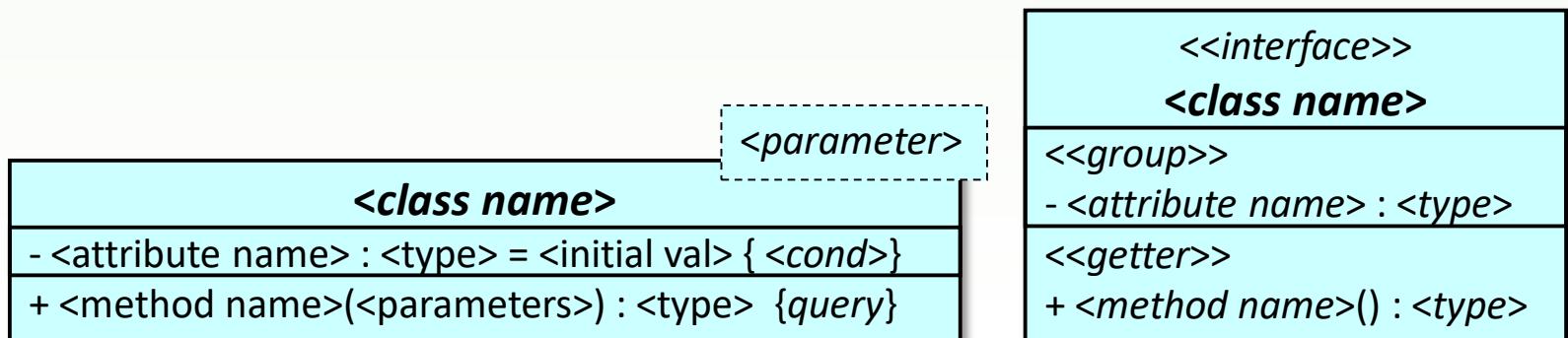
<class name>

- <attribute name> : <type name>

+ <method name>(<parameters>) : <type name>

Extensions for class-description

- ❑ Constraints for the attributes and methods between {...}
(E.g. a method that does not modify the attributes is noted by {query})
- ❑ Initial value for the attributes (set by the constructor).
- ❑ Parameter for a class.
- ❑ Stereotype for the class between <<...>> (e.g. <<interface>> ,
<<enumeration>>), or for a group of attributes or methods (for example
<<getter>> , <<setter>>).
 - Public getters and setters are to retrieve and modify the hidden attributes
in a supervised way.



Levels of modeling

Level of analysis

Point
x : int
y : int
move()

Planning decisions:

- attributes can be public, but let us create setters, too
- move should not modify the original point, it should create a new one:
 $c = a.\text{move}(b)$

Level of planning

Point
+ x : int
+ y : int
+ setPoint(x:int, y:int):void
+ move(mp:Point) : Point

```
Point c;
c.x = x + mp.x;
c.y = y + mp.y;
return c;
```

```
return Point( x + mp.x, y + mp.y)
```

```
return Point( x / f, y / f)
```

Level of implementation

Point
+ x : int
+ y : int
+Point(int,int)
+Point()
+ setPoint(x:int, y:int) :void
+move(mp:Point) : Point {query}
+operator+(mp:Point): Point {query}
+operator/(f:int) : Point {query}

instead of
 $c = a.\text{move}(b)$,
we can use:
 $c = a + b$

new operation for
the center calculus

C++ code for class Point

```
class Point
{
public:
    Point(int x = 0, int y = 0): _x(x), _y(y) { }

    void setPoint(int x, int y) { _x = x; _y = y; }

    Point move(const Point &mp) const
        { return Point(_x + mp._x, _y + mp._y); }

    Point operator+(const Point &mp) const
        { return Point(_x + mp._x, _y + mp._y); }

    Point operator/(int f) const
        { return Point(_x / f, _y / f); }

public:
    int _x, _y;
};
```

default values of the parameters:

```
Point a;  
Point b(3);  
Point c(-4, 8);
```

initializes the attributes

query

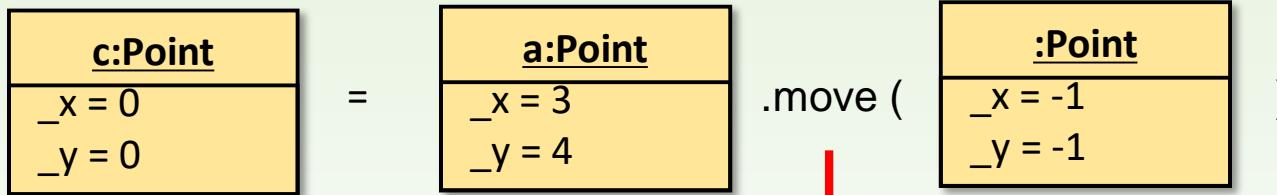
overrides the operator

“inline” definition

Instantiation of Point objects

```
Point a(3,4);  
Point c;  
c = a.move(Point(-1,-1));
```

before calling move() :



after calling move() :



Every object has an assignment operator and a copy constructor.

```
Point move(const Point &mp) const {  
    return Point(_x + mp._x, _y + mp._y);  
}
```

Default pointer of an object

```
class Point{  
public:  
    Point(int x=0, int y=0) : _x(x), _y(y) {}  
  
    void setPoint(int x, int y) { _x = x; _y = y; }  
    void setPoint(int x, int y) { this->_x = x; this->_y = y; }  
    ...  
  
public:  
    int _x, _y;  
};
```

this is a default pointer variable, pointing to the object ('s memory address) on which setPoint() was called:

p.setPoint(3, -2)

3

Another criterion of object orientation is **open recursion**: the object can always see itself and access its methods and attributes.

Planning of class Polygon

Level of analysis

Polygon
vertices : Point[]
move() center()

Planning decisions:

- private attribute
- 2 getter, 1 setter
- write to console
- move modifies the vertices

Level of planning

Polygon
<pre>- vertices : Point[]</pre> <pre>+ numOfSides() : int {query}</pre> <pre>+ vertex(i : int) : Point {query}</pre> <pre>+ setVertex(i:int, p:Point) : void</pre> <pre>+ write() : void {query}</pre> <pre>+ move(mp:Point) : void</pre> <pre>+ center() : Point {query}</pre> <pre>for i=1 .. vertices.size() loop vertices[i] = vertices[i] + mp endloop</pre>

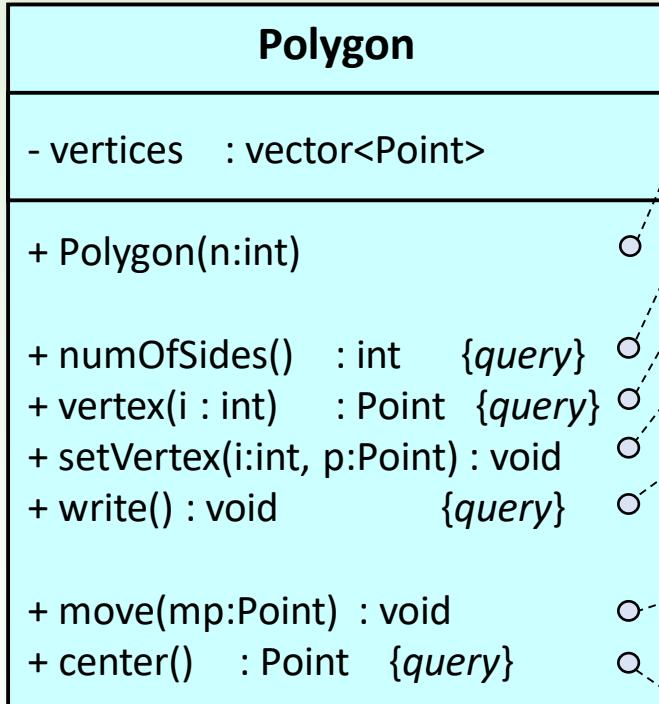
```
Point cp;
for i=1 .. vertices.size() loop
    cp := cp + vertices[i]
endloop
return cp / numOfSides();
```

Level of implementation

Polygon
<pre>- vertices : vector<Point></pre> <pre>+ Polygon(n:int)</pre> <pre>+ numOfSides() : int {query}</pre> <pre>+ vertex(i : int) : Point {query}</pre> <pre>+ setVertex(i:int, p:Point) : void</pre> <pre>+ write() : void {query}</pre> <pre>+ move(mp:Point) : void</pre> <pre>+ center() : Point {query}</pre>

Polygon class

Level of implementation



Forall (foreach) is a loop that traverses items in a collection, but cannot modify them.

```
if n < 3 then error endif  
vertices.resize(n)
```

```
return vertices.size()
```

```
if i < 0 || i >= numOfSides() then error endif  
return vertices[i]
```

```
if i < 0 || i >= numOfSides() then error endif  
vertices[i] := p
```

```
forall vertex in vertices loop  
    write(vertex.x); write(vertex.y)  
endloop
```

```
for i=0 .. vertices.size()-1 loop  
    vertices[i] = vertices[i] + mp  
endloop
```

```
Point cp;  
forall vertex in vertices loop  
    cp := cp + vertex  
endloop  
return cp / numOfSides()
```

checks errors

Code of class Polygon

```
class Polygon
{
public:
    enum Errors{FEW_VERTICES, INDEX_OVERLOADING};

    Polygon(int n) : _vertices(n) {
        if (n < 3) throw FEW_VERTICES;
    }
    unsigned int sides() const { return _vertices.size(); }

    Point vertex(unsigned int i) const {
        if (i < 0 || i >=sides()) throw INDEX_OVERLOADING;
        return _vertices[i];
    }
    void setVertex(unsigned int i, const Point &p) {
        if (i < 0 || i >=sides()) throw INDEX_OVERLOADING;
        _vertices[i] = p;
    }
    void write() const;
    void move(const Point &mp);
    Point center() const;
private:
    std::vector<Point> _vertices;
};
```

error cases

Subscript operator

```
class Polygon
{
public:
    ...
    Point vertex(unsigned int i) const { ... }

    Point Polygon::operator[](unsigned int i) const {
        if (i < 0 || i >= sides()) throw INDEX_OVERLOADING;
        return _vertices[i];
    }

    void setVertex(unsigned int i, const Point &p) { ... }

    Point& Polygon::operator[](unsigned int i) {
        if (i < 0 || i >= sides()) throw INDEX_OVERLOADING;
        return _vertices[i];
    }
    ...
};
```

Polygon t(3);
Point p = t.vertex(1); // can be replaced by the following:
Point p = t[1];

Polygon t(3);
t.setVertex(1, Point(2,2)); // can be replaced by the following:
t[1] = Point(2, 2);

Rest of the methods of Polygon

```
void Polygon::move(const Point &mp)
{
    for(unsigned int i=0; i<_vertices.size(); ++i) {
        _vertices[i] += mp;
    }
}

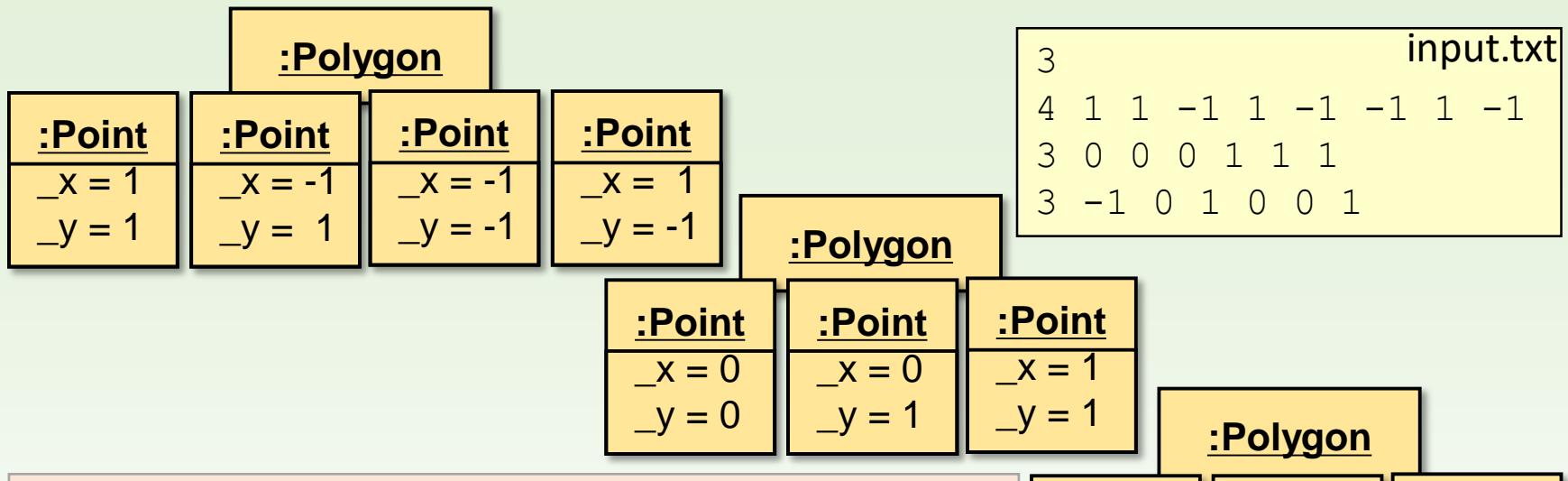
Point Polygon::center() const
{
    Point center;
    for(Point pvertex : _vertices) {
        center += pvertex;
    }
    return center / sides();
}

void Polygon::write() const
{
    cout << "<";
    for( Point pvertex : _vertices ) {
        cout << "(" << pvertex._x
              << "," << pvertex._y << ")";
    }
    cout << ">\n";
}
```

forall (foreach) loop instead of the following:

```
for(unsigned int i=0;
     i<_vertices.size();
     ++i)
{
    center += _vertices[i];
}
```

Population of the task



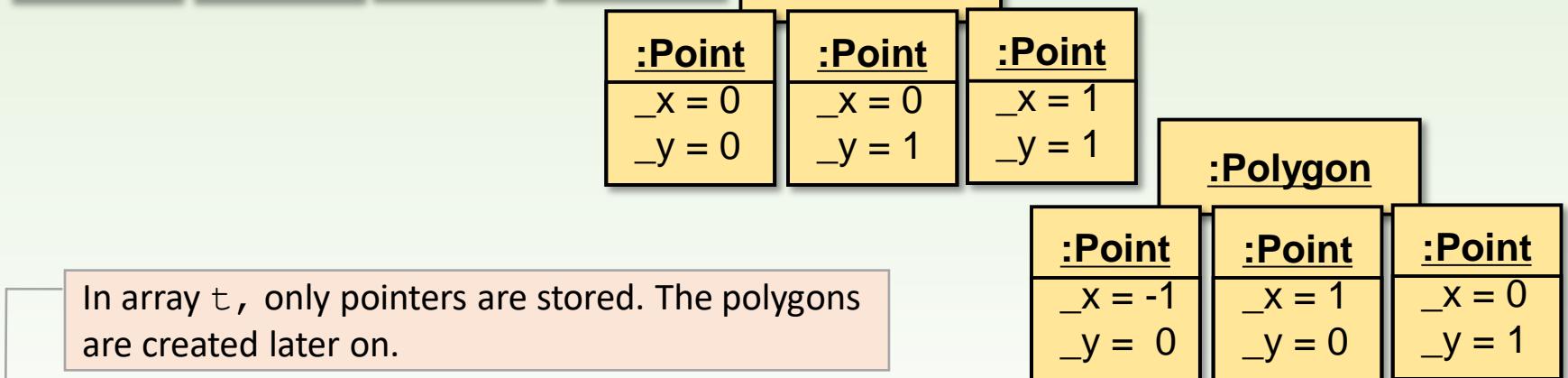
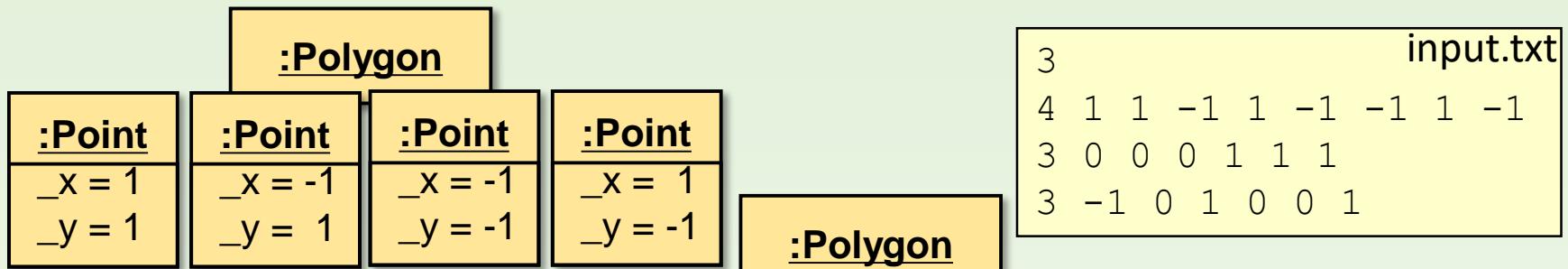
Not good, because there is no empty `Polygon()` constructor that would create the polygons at the instantiation of the vector. It is needed or pointers should be stored in the vector and polygons would be instantiated dynamically later on.

```
cout << "file name: "; string fn; cin>>fn;
ifstream inp(fn.c_str());
if(inp.fail()) { cout << "Wrong file name!\n"; exit(1); }

int n; inp >> n;
vector<Polygon> t(n);
for( unsigned int i=0; i<n; ++i ) t[i] = set(inp);
```

it would set the i^{th} polygon based on the next line in the file

Population of the task again



```
cout << "file name: "; string fn; cin>>fn;
ifstream inp(fn.c_str());
if(inp.fail()) { cout << "Wrong file name!\n"; exit(1); }

unsigned int n; inp >> n;
vector<Polygon*> t(n);
for( unsigned int i=0; i<n; ++i ) t[i] = create(inp);
```

Polygon is created by dynamic memory allocation based on the next line in the file.

```
for( Polygon* p : t ) delete p;
```

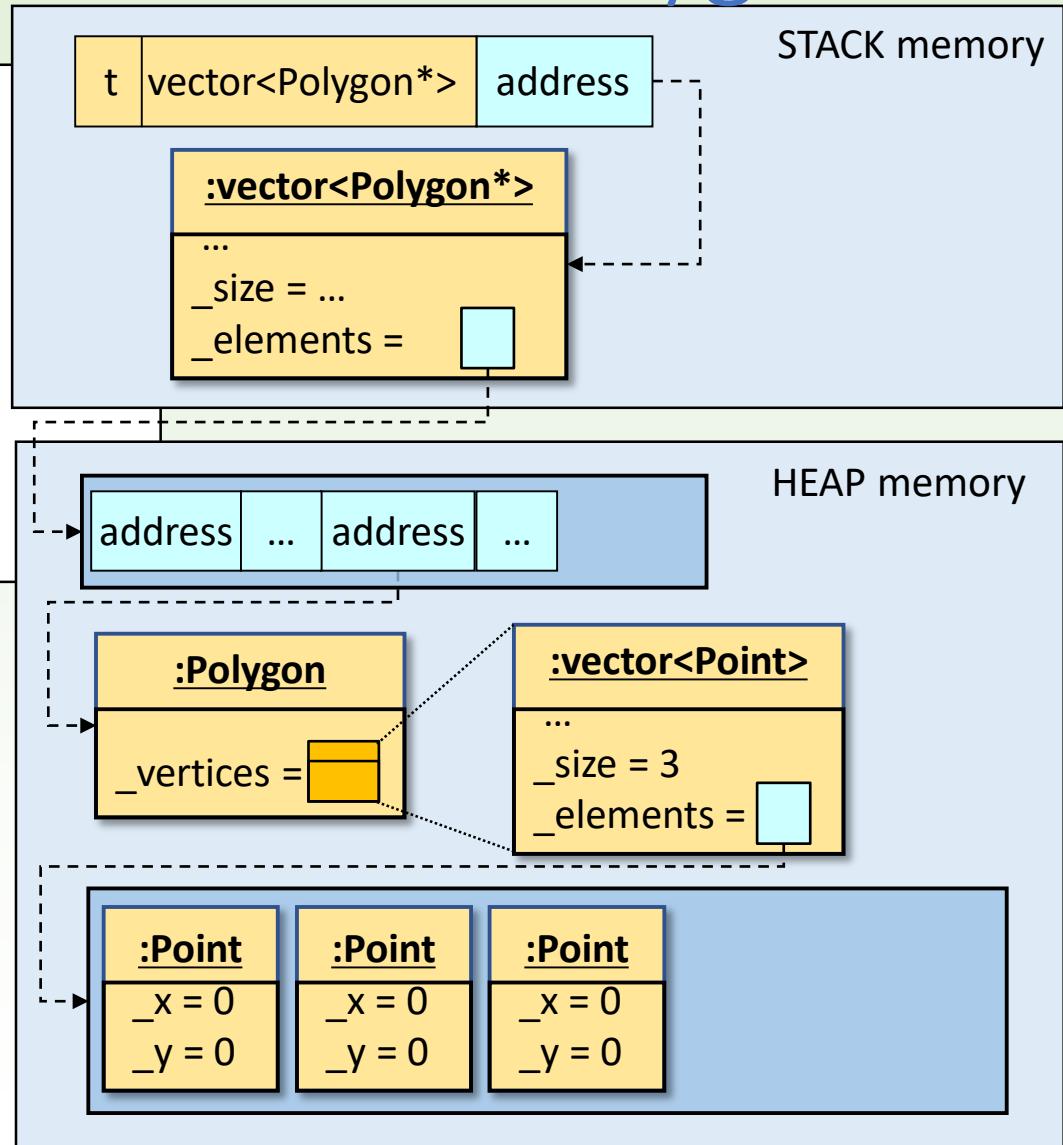
At the end, polygons, the address of which are stored in vector `t`, have to be destroyed.

Dynamic instantiation of Polygon

```
class Polygon {  
public:  
    Polygon(int n) : _vertices(n)  
    {  
        if (n < 3) throw  
            FEW_VERTICES;  
    }  
    ...  
private:  
    vector<Point> _vertices;  
};
```

```
t[i] = new Polygon(3);
```

```
int c = t[i]->center();
```



Creating a Polygon

3	input.txt
4 1 1 -1 1 -1 -1 1 -1	
3 0 0 0 1 1 1	
3 -1 0 1 0 0 1	

```
Polygon* create(ifstream &inp) {
    Polygon *p;
    try{
        int sides;
        inp >> sides;
        p = new Polygon(sides);
        for(int i=0; i < sides; ++i){
            inp >> (*p)[i]._x >> (*p)[i]._y;
        }
    } catch(Polygon::Errors e){
        if(e==Polygon::FEW_VERTICES) cout << " ... ";
    }
    return p;
}
```

int x, y;
inp >> x >> y;
p->_vertices[i].setPoint(x, y)
would be more telling, but as _vertices
is private and create() does not belong to
class Polygon, it is not accessible.

Factory method

```
Polygon* Polygon::create(ifstream &inp)
{
    Polygon *p;
    try{
        int sides;
        inp >> sides;
        p = new Polygon(sides);
        for(int i=0; i < sides; ++i){
            int x, y;
            inp >> x >> y;
            p->_vertices[i].setPoint(x,y)
        }
    }catch(Polygon::Err e){
        if( e==Polygon::Err::FileError )
    }
    return p;
}
```

It should belong to class Polygon, but it cannot be called on a Polygon, as it should create the Polygon itself.

```
class Polygon {
public:
    enum Errors { ... };
    Polygon(int n);
    ...
    static Polygon* create(std::ifstream &inp);
private:
    vector<Point*> _vertices;
};
```

class-level method

calling a class-level method

```
t[i] = Polygon::create(inp);
```

Main program

Moving polygons in a sequence (c++ vector) along the same vector, then center calculation.

$A : t : \text{Polygon}^n, mp : \text{Point}, cout : \text{Point}^n$

$Pre : t = t_0 \wedge mp = mp_0$

notation for concatenation

$Post : mp = mp_0 \wedge t = \bigoplus_{i=1}^n < t_0[i].move(mp) >$

$\wedge cout = \bigoplus_{i=1}^n < t[i].center() >$

$cout := <>$

$i = 1 .. n$

$t[i].move(mp)$

$cout := cout \oplus t[i].center()$

Two summations (copy):

$i \in [m..n] \sim i \in [1 .. n]$

$s \sim t$

$f(i) \sim < t_0[i].move(mp) > < t[i].center() >$

$H, +, 0 \sim \text{Polygon}^n, \oplus, <> \text{Point}^n, \oplus, <>$

```
for( Polygon *p : t ){
    p->move( Point(20,20) );
    p->write();
    Point cp = p->center();
    cout << "(" << cp._x << "," << cp._y << ") \n";
}
```

Type-oriented solution

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <vector>
#include "polygon.h"
#include "point.h"
using namespace std;

int main()
{
    cout << "file name: "; string fn; cin>> fn;
    ifstream inp(fn.c_str());
    if(inp.fail()) { cout << "Wrong file name!\n"; exit(1); }
    int n; inp >> n;
    vector<Polygon*> t(n);
    for (unsigned int i=0; i<n; ++i) t[i] = Polygon::create(inp);
    for ( Polygon* p : t ){
        p->move(Point(20,20)); p->write();
        Point cp = p->center();
        cout << "(" << cp._x << ", " << cp._y << ") \n";
    }
    for ( Polygon* p : t ) delete p;
    return 0;
}
```

The code illustrates a type-oriented solution for processing polygons. It includes headers for iostream, fstream, cstdlib, vector, polygon.h, and point.h. It uses the std namespace. The main function reads a file name from standard input, creates a vector of size n containing pointers to Polygon objects, and then iterates over each polygon to move it to a center point (20, 20), write its state, and then calculate its center point again. Finally, it deletes each polygon object. Three annotations are shown: 'population' points to the line where the vector t is created; 'calculus' points to the loop where each polygon's move and write methods are called; and 'destruction' points to the loop where each polygon object is deleted.

Object-oriented solution

```
int main() {
    Application a;
    a.run();
    return 0;
}
```

```
class Application{
public:
    Application();
    void run();
    ~Application();
private:
    std::vector<Polygon*> t;
};
```

```
graph TD
    Application --> population[population]
    Application --> calculus[calculus]
    Application --> destruction[destruction]
```

```
Application::Application() {
    cout << "file name: " string fn; cin>> fn;
    ifstream inp(fn.c_str());
    if(inp.fail()) {
        cout << "Wrong file name!\n"; exit(1);
    }
    unsigned int n; inp >> n;
    t.resize(n);
    for(unsigned int i=0; i<n; ++i)
        t[i] = Polygon::create(inp);
}
```

```
void Application::run() {
    for ( Polygon* p : t ) {
        p->move(Point(20,20)); p->write();
        Point cp = p->center();
        cout << "(" << cp._x << ", "
              << cp._y << ") \n";
    }
}
```

```
Application::~Application() {
    for ( Polygon* p : t ) delete p;
}
```

Menu-controlled object-oriented solution

```
int main()
{
    Menu a;
    a.run();
    return 0;
}
```

```
class Menu{
public:
    Menu() { s = nullptr; }
    void run();
    ~Menu() { if(s!=nullptr) delete s; }
private:
    Polygon* s;

    void menuWrite();
    void case1();
    void case2();
    void case3();
    void case4();
};
```

```
void Menu::run()
{
    int v = 0;
    do{
        menuWrite();
        cin >> a; // validation!
        switch(a) {
            case 1: case1(); break;
            case 2: case2(); break;
            case 3: case3(); break;
            case 4: case4(); break;
        }
    }while(a != 0);
}
```

```
void Menu::menuWrite() {
    cout << "0 - exit\n";
    cout << "1 - create\n";
    cout << "2 - write\n";
    cout << "3 - move\n";
    cout << "4 - center\n";
}
```

methods would be: create, write, move, center

Menu items

input1.txt

4 1 1 -1 1 -1 -1 -1 1

input2.txt

3 0 0 -1 0 0 -1

```
void Menu::case1() {
    if(s!=nullptr) delete s;
    cout << "file name: "; string fn; cin>> fn;
    ifstream inp(fn.c_str());
    if(inp.fail()) { cout << "Wrong file name!\n"; return; }
    s = Polygon::create(inp);
}

void Menu::case2() {
    if(s==nullptr){ cout << "There is no polygon!\n"; return; }
    s->write();
}

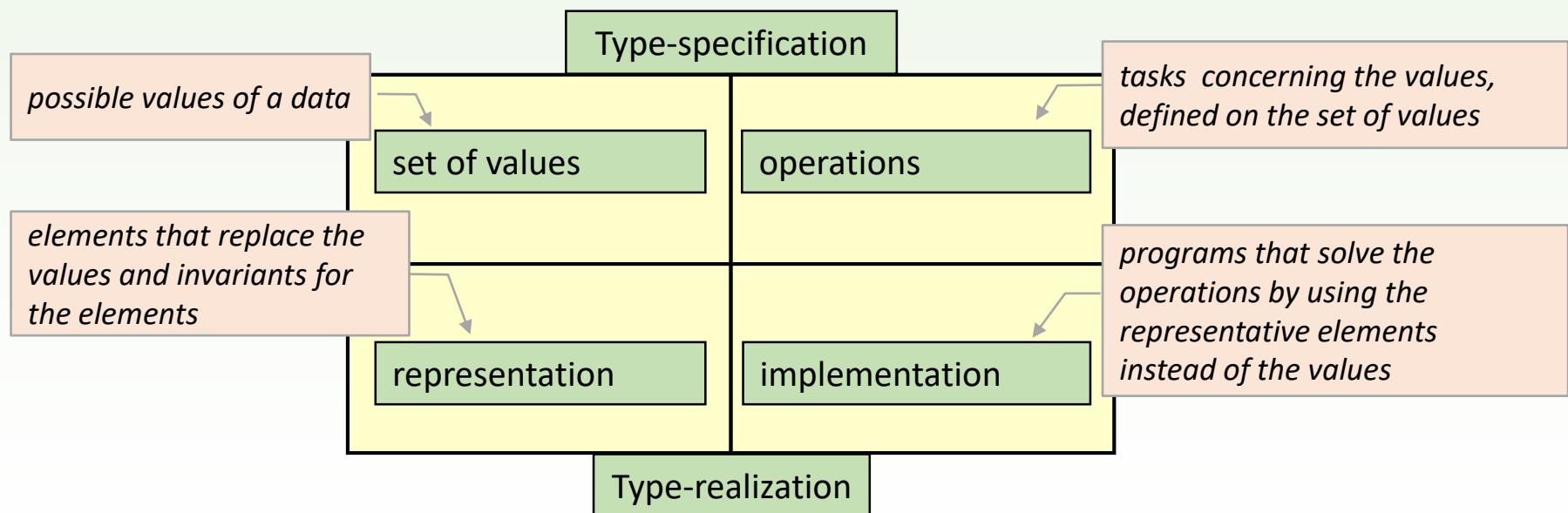
void Menu::case3() {
    if(s==nullptr){ cout << "There is no polygon!\n"; return; }
    s->move(Point(20,20));
}

void Menu::case4() {
    if(s==nullptr){ cout << "There is no polygon!\n"; return; }
    Point sp = s->center();
    cout << "(" << cp._x << "," << cp._y << ") \n";
}
```

Collections, enumerators,
basic algorithms
(algorithmic patterns)

Datatype

- ❑ Type of a data (specifically an object) is defined by the **set of its values** and its **operations**. It is called **specification**.
- ❑ Type realization shows how the values could be **represented** and how programs solve or **implement** the operations.



Type structure

basic type

complex type

record type

alternative type

iterative type

a value is represented by a group of values of other types

$T = \text{rec}(s_1:T_1, \dots, s_n:T_n)$
 i^{th} component of $t:T$ is $t.s_i$

a value is represented by a finite collection of values of another type, the elements of the collection are of the same type

$T = \text{it}(E)$

relation of elements that represent the values

a value is represented by one of the values of other types

$T = \text{alt}(s_1:T_1, \dots, s_n:T_n)$
if type of $t:T$ is T_i , then $t.s_i$ is true

Well-known iterative types

$h := h \cup \{e\}$ $h := h - \{e\}$ $l := e \in h$
 $h := \emptyset$ $l := h = \emptyset$
two kinds of element selection:
 $e := \text{mem}(h)$, $e \in h$
 $(h: \text{set}(E), e: E)$

iterative types

set(E)

set

sequence

array

tree

graph

universal

stack

queue

sequential input file

sequential output file

st, e, f : read

```

sf, df, f := read(f)
if f=> then sf := abnorm
else sf:=norm : df := f1 : f := <f2, ..., f|f|>

```

f : sequential input file (infile(E))

e : the read element (E)

st: indicates the success of reading (Status)

Status = {norm, abnorm}

infile(E)

```

f := <>
f := write(f, s)
f := f ⊕ s

```

f : sequential output file (outfile(E))

outfile(E)

Processing a collection

- ❑ **Collection** (container, collection, iteration) is an object, capable of storing elements. It provides operations for archiving and searching elements.
 - Like complex types, especially the **iterations**: set, sequence (stack, queue, file), array, tree, graph.
 - There are so-called **virtual collections**, too, the elements of which do not have to be stored: e.g. items of an integer-type interval or prime divisors of a natural number.
- ❑ **Processing a collection** means processing its elements.
 - Find the biggest item of a set.
 - How many negatives are in a sequence of numbers?
 - Traverse backwards every second item of an $[m .. n]$ interval.
 - Sum the prime divisors of a n natural number.

Enumeration

- ❑ Enumeration of the E -type elements of a collection can be considered as a sequence in set E^* . The **operations** of the traversal are the following:
 - *first()* : selects the first item of the enumeration, it actually starts the enumeration
 - *next()* : selects the next item of the enumeration
 - $I := end() \ (I:\mathbb{L})$: shows if the enumeration has ended
 - $e := current() \ (e:E)$: gets the current item of the enumeration

States of the enumeration

- ❑ An enumeration has different **states** (*pre-start*, *in process*, *finished*): the operations are only reasonable in certain states (otherwise, their effect is not defined).
- ❑ The **processing algorithm** guarantees that the operations are executed only (in that state) when they are reasonable.

`t : enor(E)`

`t.first()`

`¬t.end()`

`Process(t.current())`

`t.next()`

`e ∈ t`

`Process(e)`

```
for(t.first(); !t.end(); t.next())
{
    process(t.current());
}
```

foreach (forall) loop

```
for( auto e : t )
{
    process(e);
}
```

Enumeration with object

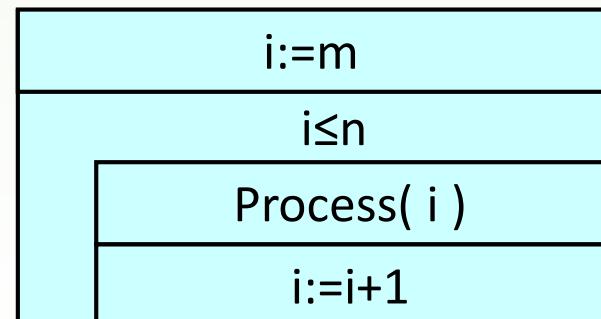
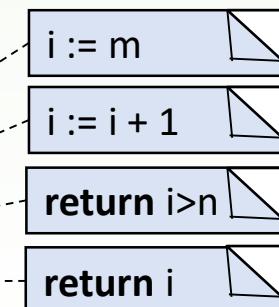
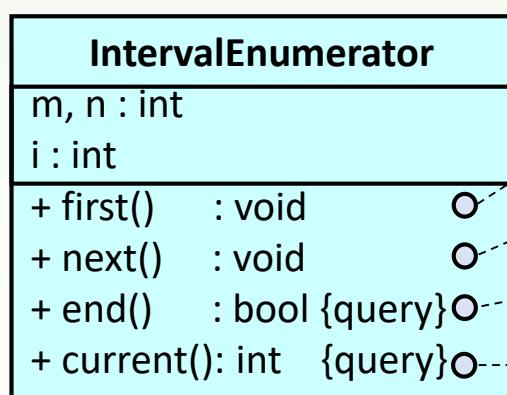
- ❑ Enumeration is done by a **distinct object** separated from the collection.
There can be more enumerator objects for one collection.
- ❑ **Type** of the enumerator object is denoted by *enor(E)*.
- ❑ **Realization** of an enumerator object depends on the type of the collection.
 - As the enumerator object has to know the traversed collection, its representation contains a **reference of the collection**.
 - Implementations of the operations usually need auxiliary data.
- ❑ It is worthy to **create the enumerator by a method of the collection** so that the collection is aware of being traversed.

Classic enumerator of an interval

Enumeration of integers in an interval **in ascending order**.

enor(\mathbb{Z})				
\mathbb{Z}^*	first()	next()	$i := \text{end}()$	$e := \text{current}()$
$m, n : \mathbb{Z}$ $i : \mathbb{Z}$	$i := m$	$i := i + 1$	$i := i > n$ $i : \mathbb{L}$	$e := i$ $e : \mathbb{Z}$

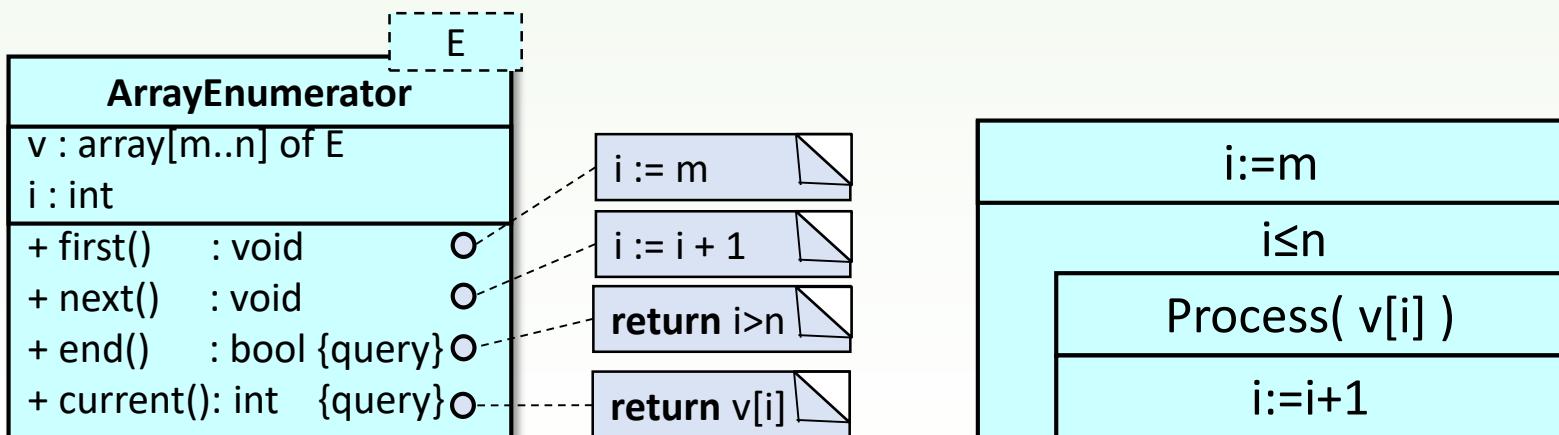
Usually, this class is not instantiated, the enumerator is variable i itself.



Classic enumerator of a vector

Enumeration of the items of a vector containing values from E,
from the beginning to the end

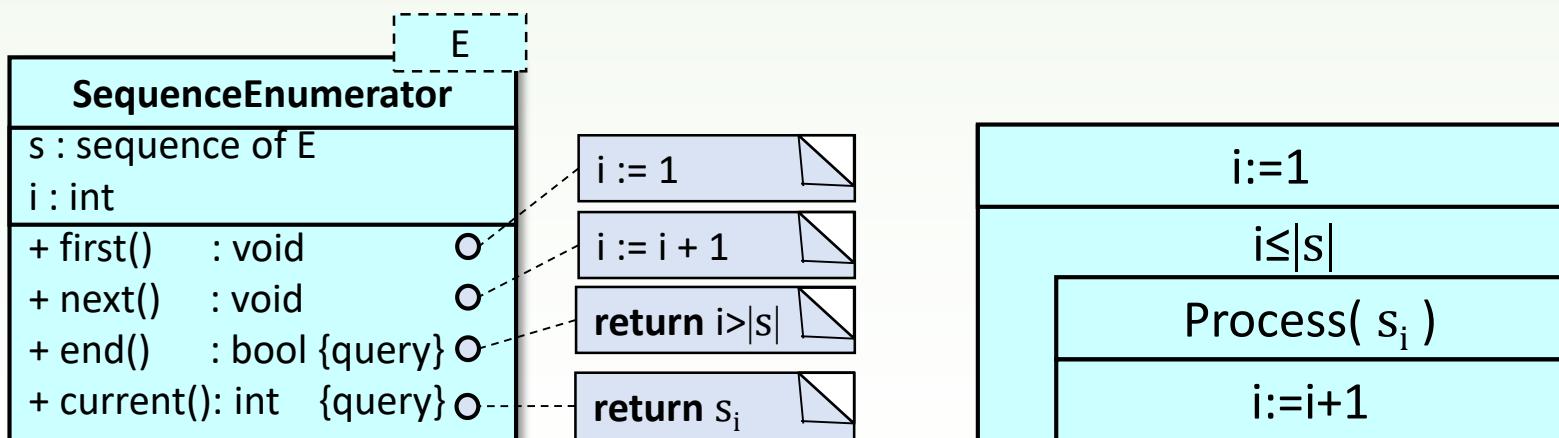
enor(E)				
E^*	first()	next()	$l := \text{end}()$	$e := \text{current}()$
$v : E^{m..n}$ $i : \mathbb{Z}$	$i := m$	$i := i + 1$	$l := i > n$	$e := v[i]$ $e : E$



Classic enumerator of a sequence

Enumeration of a finite sequence of values from E, from the beginning to the end

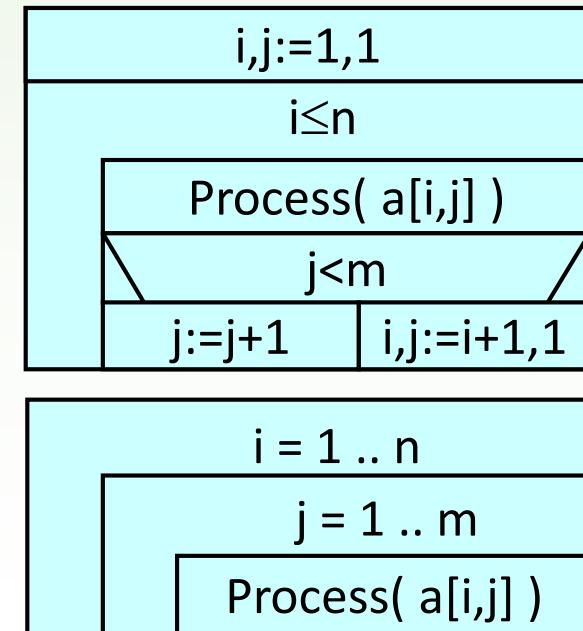
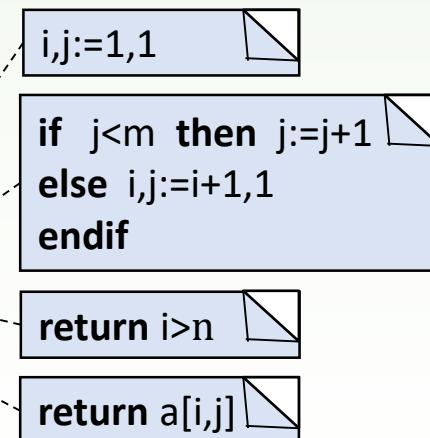
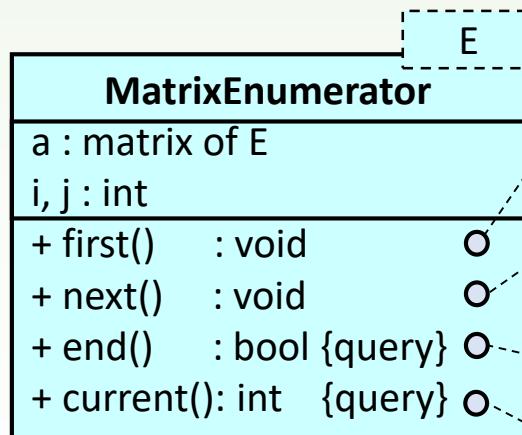
enor(E)				
E^*	first()	next()	$l := \text{end}()$	$e := \text{current}()$
$s : E^*$ $i : \mathbb{Z}$	$i := 1$	$i := i + 1$	$ l := s $ $ l : \mathbb{L}$	$e := s_i$ $e : E$



Row major enumerator of a matrix

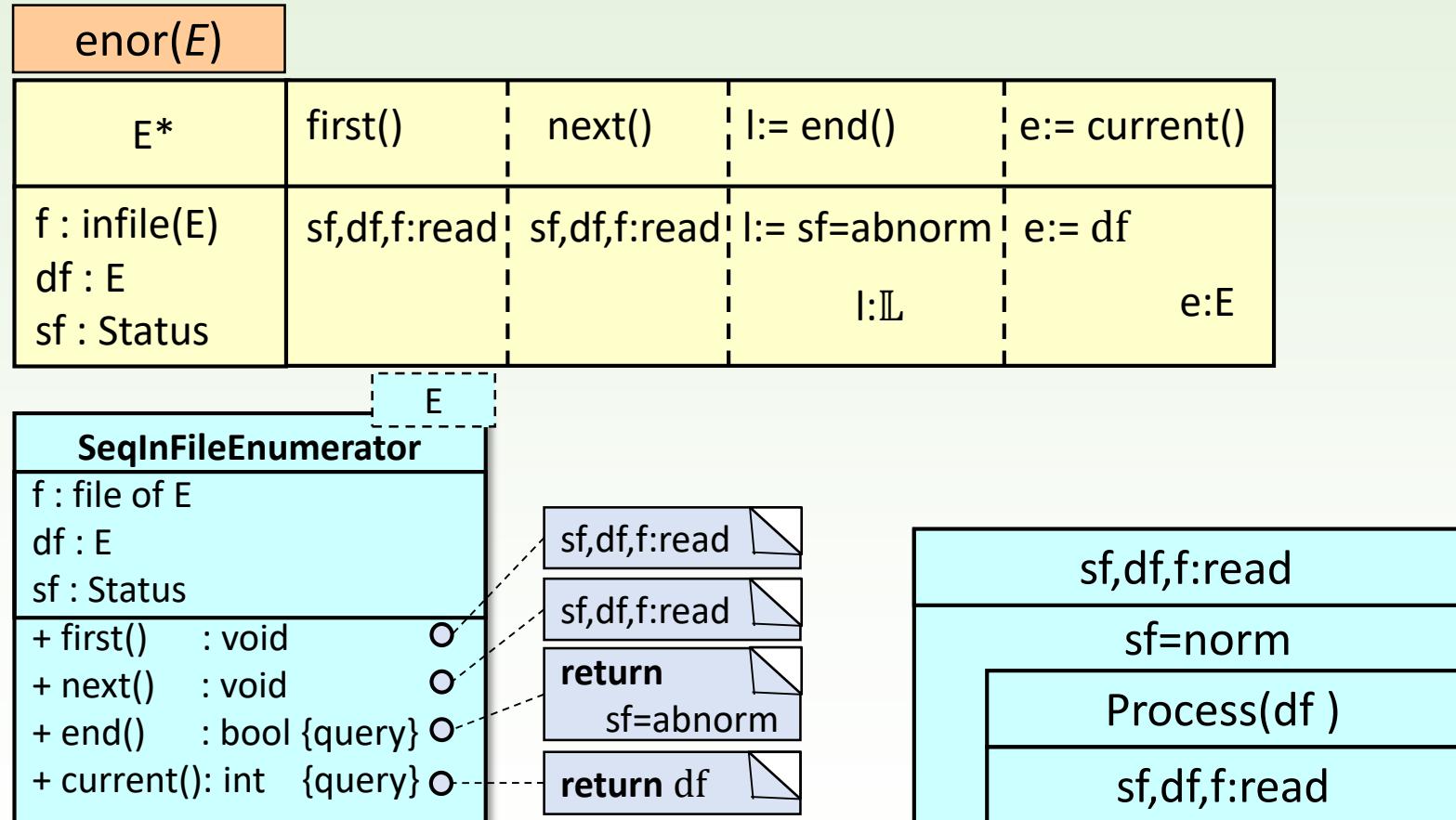
Enumeration of the items of a matrix with values from E in row major order.

enor(E)				
E^*	first()	next()	$i := \text{end}()$	$e := \text{current}()$
$a : E^{n \times m}$ $i, j : \mathbb{Z}$	$i, j := 1, 1$	if $j < m$ then $j := j + 1$ else $i, j := i + 1, 1$	$i := i > n$ $i : \mathbb{L}$	$e := a[i, j]$ $e : E$



Enumerator of a sequential input file

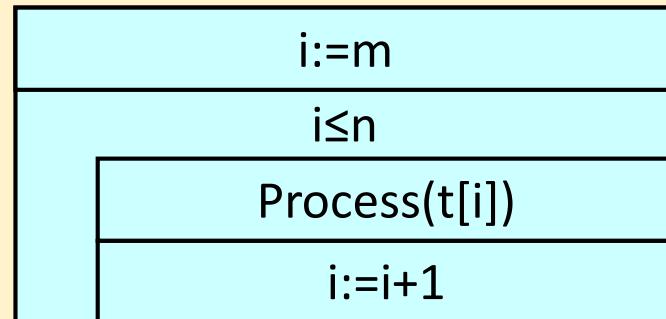
Enumeration of the items of a sequential input file with values from E.



Generalization of the algorithmic patterns

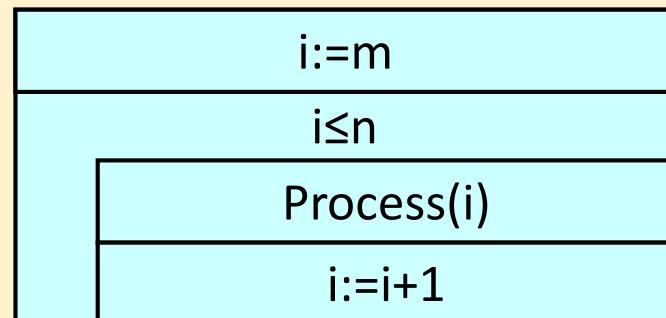
❑ Algorithmic patterns for arrays:

- $t : E^{m..n}$ ($E^{1..n} = E^n$)
- $f : E \rightarrow H$, cond: $E \rightarrow \mathbb{L}$



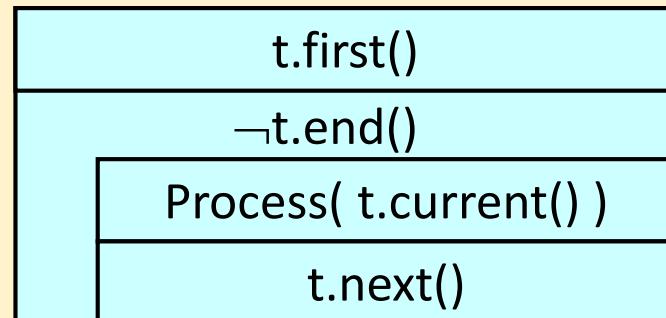
❑ Algorithmic patterns for functions defined on intervals:

- $[m .. n]$
- $f:[m .. n] \rightarrow H$, cond: $[m .. n] \rightarrow \mathbb{L}$



❑ Algorithmic patterns for enumerators:

- $t : \text{enor}(E)$
- $f : E \rightarrow H$, cond: $E \rightarrow \mathbb{L}$



Summation

Sum the values assigned to the elements of an enumeration.

A : $t:\text{enor}(E), s:H$

Pre : $t = t'$

Post: $s = \sum_{e \in t'} f(e)$

$f:E \rightarrow H$

$+:H \times H \rightarrow H$

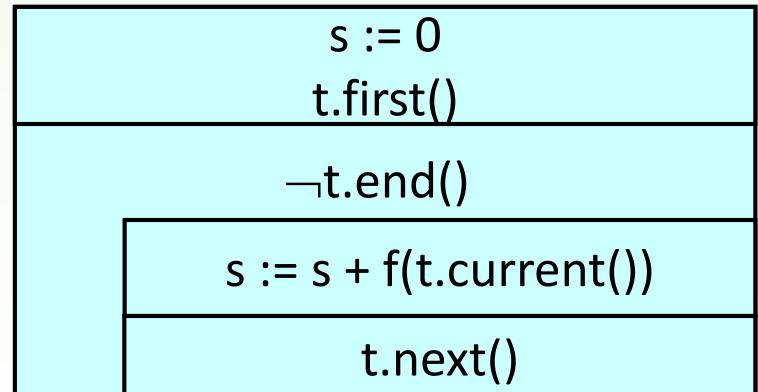
$0 \in H$

with left neutral element

$\sum_{e \in t'} f(e) = (\dots(f(e_1) + f(e_2)) + \dots) + f(e_n),$
where e_1, \dots, e_n are the elements of
enumeration t'

Special case: conditional summation

$\sum_{\substack{e \in t' \\ \text{cond}(e)}} g(e), \text{ so } f(e) = \begin{cases} g(e), & \text{if cond}(e) \\ 0, & \text{otherwise} \end{cases}$



Counting

Count the items with certain condition in an enumeration.

A : $t:\text{enor}(E), c:\mathbb{N}$

Pre : $t = t'$

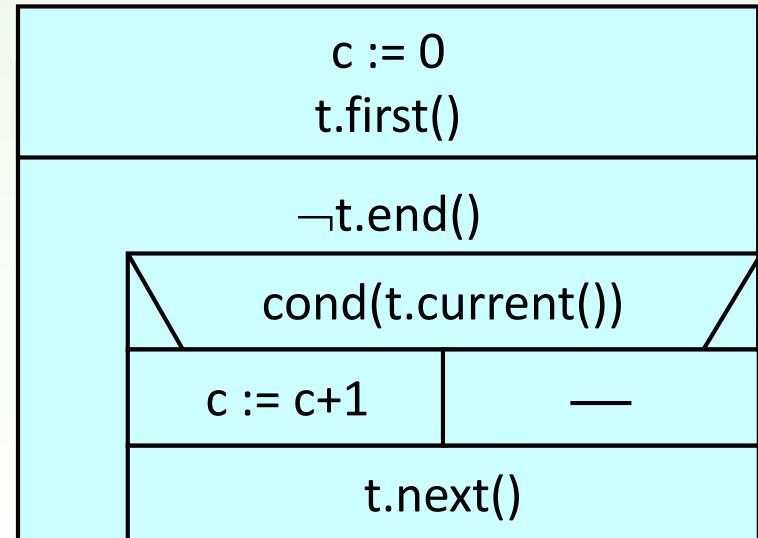
Post: $c = \sum_{e \in t'} \begin{cases} 1 & \text{cond}(e) \\ 0 & \text{otherwise} \end{cases}$

cond: $E \rightarrow \mathbb{L}$

Sum defined on the set of natural numbers

Counting is a special summation

$$\sum_{e \in t'} f(e), \text{ thus } f(e) = \begin{cases} 1, & \text{if cond}(e) \\ 0, & \text{otherwise} \end{cases}$$



Maximum search

Give the item of the highest value of an enumeration according to a given point of view.

$A : t:\text{enor}(E), \text{elem}:E, \text{max}:H$

$Pre : t = t' \wedge |t| > 0$

$Post: (\text{max} , \text{elem}) = \mathbf{MAX}_{e \in t'} f(e)$

$f:E \rightarrow H$

items of set H can be sorted

$\max = f(\text{elem}) = \mathbf{MAX}_{e \in t'} f(e)$
 $\wedge \text{elem} \in t'$

- MIN instead of MAX
- elem *can be skipped*,
max *not*

$t.\text{first}()$

$\max, \text{elem} := f(t.\text{current}()), t.\text{current}()$

$t.\text{next}()$

$\neg t.\text{end}()$

$f(t.\text{current}()) > \max$

$\max, \text{elem} := f(t.\text{current}()), t.\text{current}()$

$t.\text{next}()$

Selection (success is sure)

Find the first item of a given condition in an enumeration if it is sure that the enumerator contains such element.

A: $t:\text{enor}(E)$, $\text{elem}:E$

$\text{cond}: E \rightarrow \mathbb{L}$

Pre: $t = t' \wedge \exists e \in t : \text{cond}(e)$

Post: $(\text{elem}, t) = \text{SELECT}_{e \in t'} \text{cond}(e)$

At the end of the selection, the state of the enumerator is still „*in process*”, as it has remaining items

Searches the first item (it will be the *elem*) of enumerator t' for which the condition is satisfied.

Formally:

$\text{cond}(e_i) \wedge \forall_{k=1..i-1} \neg \text{cond}(e_k) \wedge \text{elem}=e_i$,
where e_1, e_2, \dots are items of enumerator t'

`t.first()`

`\neg cond(t.current())`

`t.next()`

`elem := t.current()`

Linear search (success is unsure)

Find the first item of a given condition in an enumeration.

$A : t:\text{enor}(E), l:\mathbb{L}, \text{elem}:E$

$\text{Pre} : t = t'$

$\text{Post} : (l, \text{elem}, t) = \text{SEARCH}_{e \in t'} \text{ cond}(e)$

$\text{cond}:E \rightarrow \mathbb{L}$

Enumeration of t only finishes if the search is unsuccessful, otherwise it remains in state „*in process*”.

Searches the first item (it will be the *elem*) of enumerator t' for which the condition is satisfied.

If the search is successful, the value of l changes to true, otherwise it remains false.

Formally:

$$l = \exists_{e \in t'} \text{cond}(e) \wedge (l \rightarrow \text{cond}(e_i) \wedge \forall_{k=1..i-1} \neg \text{cond}(e_k) \wedge \text{elem}=e_i, \text{ where } e_1, \dots, e_n \text{ are items of } t')$$

It is used for **decision**, too:

$$l = \text{SEARCH}_{e \in t'} \text{cond}(e) \text{ or } l = \exists_{e \in t'} \text{cond}(e)$$

$$l := \text{false}; t.\text{first}()$$

$$\neg l \wedge \neg t.\text{end}()$$

$$\text{elem} := t.\text{current}()$$

$$l := \text{cond}(\text{elem})$$

$$t.\text{next}()$$

Optimistic linear search

Check if a given condition stands for every element of an enumeration.
If not, give the first item that violates it.

$A : t:\text{enor}(E), l:\mathbb{L}, \text{elem}:E$

$\text{cond}:E \rightarrow \mathbb{L}$

$Pre : t = t'$

Enumeration of t only finishes if the search is successful, otherwise it remains in state „in process”.

$Post : (l, \text{elem}, t) = \forall \text{SEARCH}_{e \in t'} \text{cond}(e)$

If the condition stands for every item of the enumerator, then l remains true. Otherwise it changes to false. In this case, elem contains the first item of the enumeration that violates the condition. Formally:

$$l = \forall_{e \in t'} \text{cond}(e) \wedge (\neg l \rightarrow \neg \text{cond}(e_i) \wedge \forall_{k=1..i-1} \text{cond}(e_k)) \wedge \text{elem} = e_i, \text{ where } e_1, \dots, e_n \text{ are elements of } t'$$

It is used for **decision**, too:

$$l = \forall \text{SEARCH}_{e \in t'} \text{cond}(e) \text{ or } l = \forall_{e \in t'} \text{cond}(e)$$

$l := \text{true}; t.\text{first}()$

$l \wedge \neg t.\text{end}()$

$\text{elem} := t.\text{current}()$

$l := \text{cond}(\text{elem})$

$t.\text{next}()$

Conditional maximum search

Give the item of, according to a given point of view, the highest value in those items of an enumeration that satisfy a condition.

$A : t:\text{enor}(E), l:\mathbb{L}, \text{elem}:E, \text{max}:H$

$Pre : t = t'$

$Post : (l, \text{max}, \text{elem}) = \underset{\text{cond}(e)}{\text{MAX}}_{e \in t'} f(e)$

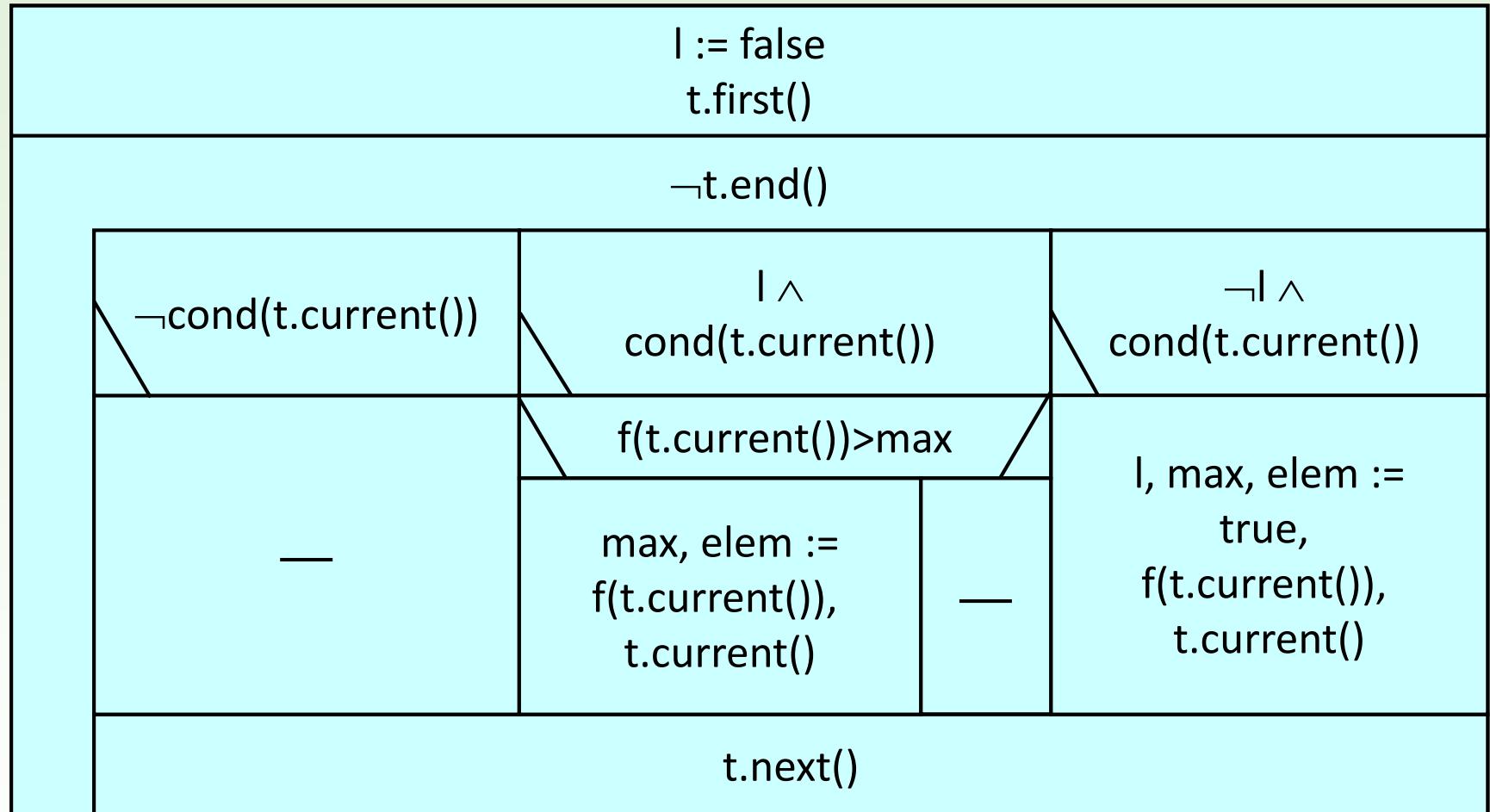
$f:E \rightarrow H$

$\text{cond}:E \rightarrow \mathbb{L}$

items of set H can be sorted

$$I = \exists_{e \in t'} \text{cond}(e) \wedge (I \rightarrow \text{max} = f(\text{elem}) = \underset{\text{cond}(e)}{\text{MAX}}_{e \in t'} f(e) \wedge \text{elem} \in t')$$

Conditional maximum search



- MIN instead of MAX
- elem can be skipped, max not

Steps of analogy

1. Forebode the algorithmic pattern that solves (part of) the task.
2. Specify the task by **executable postcondition** that predicts the solution.
3. Give the differences between the task and the algorithmic pattern:
 - type of the *enumerator* with the type of its *items*
 - concrete representatives of the *functions* ($f:[m..n] \rightarrow H$,
 $cond:[m..n] \rightarrow \mathbb{L}$)
 - *operation* of H , if needed
 - $(\mathbb{Z}, >)$ or $(\mathbb{Z}, <)$ instead of $(H, >)$
 - $(\mathbb{Z}, +, 0)$ or $(\mathbb{R}, *, 1)$ or $(\mathbb{L}, \wedge, \text{true})$ instead of $(H, +, 0)$
 - *renaming of the variables*
4. By applying the differences in the general algorithm of the algorithmic pattern, the solution of the task is given.

Aspects in testing

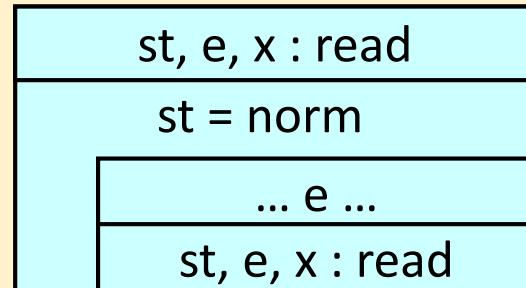
- Enumerator-based (for all of the patterns)
 - *length*: 0, 1, and more items
 - *first* and *last*: when the special element (to be summed or satisfying a condition or the maximal) is at the beginning or at the end of the enumerator.
- Role-based
 - *search*: exists or not exists an item satisfying *cond*
 - *max. search*: one maximum or more maxima
 - *summation*: loading
- Particularities of the operations that calculate functions *cond(i)* and *f(i)*.

Enumeration of a sequential input file

Enumeration of a sequential input file

- ❑ Items of a sequential input file `x:infile(E)` (can be considered as a sequence) can be enumerated via operation `st,e,x:read` (`e:E`, `st:Status={abnorm, norm}`).
- ❑ Operations of the enumeration:

- `first()` ~ `st, e, x : read`
- `next()` ~ `st, e, x : read`
- `current()` ~ `e`
- `end()` ~ `st=abnorm`



- ❑ Enumeration is based on `pre-reading strategy`: first reading, then examining if the reading was successful and if it was, then processing the item.
- ❑ In the specification, the enumeration might be denoted by `e∈x`.

Processing files

- ❑ In practice, there are a lot of problems where **sequences** have to be generated (**from sequences**). If sequences are e.g. in files (or are reachable in console), then it is worthy to handle them as **sequential input and output files**.
- ❑ Most common tasks:
 - **copy** and **elementwise process** (e.g. creating a report)
 - **multiple item selection**
 - **partitioning**
 - **union of sorted sequences**
- ❑ The common is that all of them are based on **summation**, and except the union (which needs a custom enumerator), all of them use a **sequential input file enumerator**.

Summation in file processing

General summation

$A : t:\text{enor}(E), s:H$

$Pre : t = t_0$

$Post: s = \sum_{e \in t_0} f(e)$

$$\begin{aligned} f &: E \rightarrow H \\ + &: H \times H \rightarrow H \\ 0 &\in H \end{aligned}$$

Special summation for files

$A : x:\text{infile}(E), y:\text{outfile}(F)$

$Pre : x = x_0$

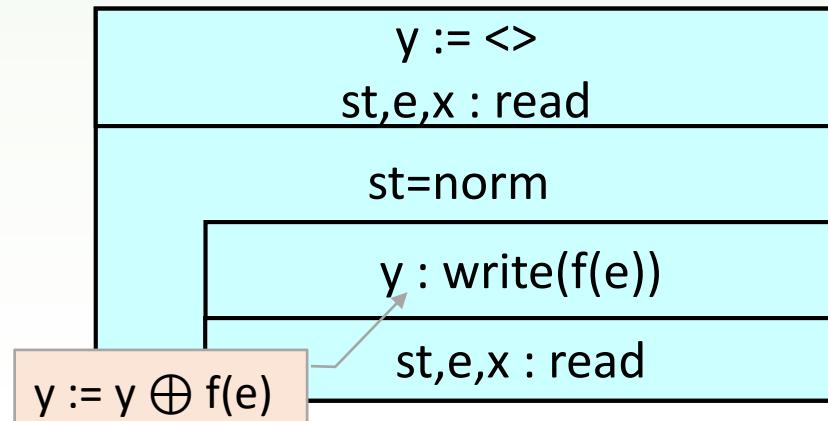
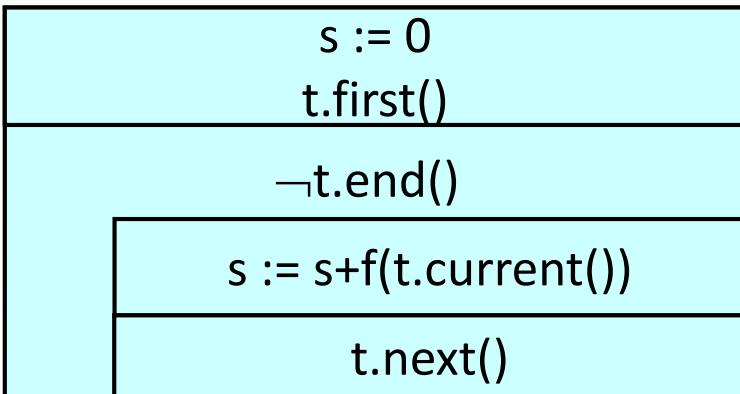
$Post: y = \bigoplus_{e \in x_0} f(e)$

$$\begin{aligned} f &: E \rightarrow F^* \\ \bigoplus &: F^* \times F^* \rightarrow F^* \\ <> &\in F^* \end{aligned}$$

Summation:

$t:\text{enor}(E) \sim x:\text{infile}(E)$
 $st,e,x : \text{read}$

$H,+,0 \sim F^*, \bigoplus, <>$



1st task

Transform a text: change every accented letter to unaccented in a sequential input file!

A : $x:\text{infile}(\text{Char})$, $y:\text{outfile}(\text{Char})$

Pre : $x = x_0$

Post: $y = \bigoplus_{ch \in x_0} \langle \text{transform}(ch) \rangle$

where $\text{transform} : \text{Char} \rightarrow \text{Char}$ and $\text{transform}(ch) = \dots$

Summation:

$t:\text{enor}(E) \sim x:\text{infile}(\text{Char})$
 $\quad\quad\quad st, ch, x : \text{read}$
 $e \sim ch$
 $f(e) \sim \langle \text{transform}(ch) \rangle$
 $H, +, 0 \sim \text{Char}^*, \oplus, \langle \rangle$

$y := \langle \rangle$

$st, ch, x : \text{read}$

$st = \text{norm}$

$y : \text{write}(\text{transform}(ch))$

$st, ch, x : \text{read}$

Grey box texting

- ❑ In case of summation we have to check
 - the enumerator (like in other patterns)
 - length: 0, 1, 2, and more items in the enumerator
 - “sides” of the enumerator: if there are at least 2 different items in the enumerator, then it is checkable
 - the loading, but the size of the output file equals to the size of the input file. It is not necessary.
 - ❑ The conversion has to be verified, too.

length-based: [input of 0, 1, 2, and more characters \(copy\)](#)

conversion-based: $x = <\text{\'a\'e\'i\'o\'\'o\'u\'u\'\u00f3}>$ \rightarrow $y = <\text{aeiouuu}>$

$$x = \langle aeioouuu \rangle \quad \rightarrow \quad y = \langle aeioouuu \rangle$$

$x = \langle bsmnz \rangle$ \rightarrow $y = \langle bsmnz \rangle$

x = <Ferenc Puskás ...>

C++

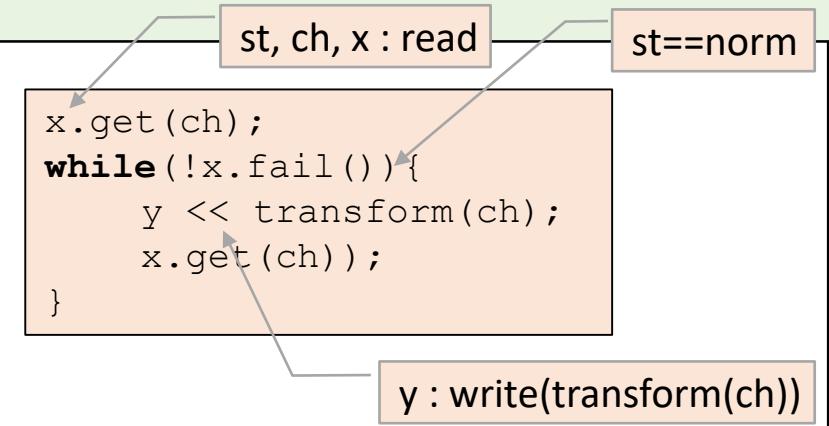
- ❑ Language C++ uses pre-reading strategy for processing a file.
- ❑ Implementations of reading a character (`st, ch, x : read`):
 - `x >> ch`
 - Does not read white spaces except if this automatism is switched off (`x.unsetf(ios::skipws)`).
 - `x.get(ch)`
 - Reads every character, even white spaces, too.
- ❑ In C++, operation `st==norm` is implemented as `!x.eof()`. Many times, using `!x.fail()` is more secure, because it indicates not just the end of file, but every type of unsuccessful reading, like the file is not correctly filled up.

C++ program

```
int main()
{
    ifstream x( "input.txt" );
    if ( x.fail() ) { ... }
    ofstream y( "output.txt" );
    if ( y.fail() ) { ... }

    char ch;
    while (x.get(ch)) {
        y << transform(ch);
    }

    return 0;
}
```



C++ program

```
char transform(char ch)
{
    char new_ch;
    switch (ch) {
        case 'á' : new_ch = 'a'; break;
        case 'é' : new_ch = 'e'; break;
        case 'í' : new_ch = 'i'; break;
        case 'ó' : case 'ö' : case 'ő' : new_ch = 'o'; break;
        case 'ú' : case 'ü' : case 'ű' : new_ch = 'u'; break;
        case 'Á' : new_ch = 'A'; break;
        case 'É' : new_ch = 'E'; break;
        case 'Í' : new_ch = 'I'; break;
        case 'Ó' : case 'Ö' : case 'Ő' : new_ch = 'O'; break;
        case 'Ú' : case 'Ü' : case 'Ű' : new_ch = 'U'; break;
        default   : new_ch = ch;
    }
    return new_ch;
}
```

2nd task

Assort the even numbers from a sequential input file containing integers.

$A : x:\text{infile}(\mathbb{Z}), \text{cout}:\text{outfile}(\mathbb{Z})$

$Pre : x = x_0$

$Post: \text{cout} = \bigoplus_{e \in x_0 \setminus \{x\}} \langle e \rangle$
 $2 \mid e$

Conditional summation:

$t:\text{enor}(E) \sim x:\text{infile}(\mathbb{Z})$
 $st, e, x : \text{read}$

$f(e) \sim \langle e \rangle$

$\text{cond}(e) \sim 2 \mid e$

$y \sim \text{cout}$

$H, +, 0 \sim \mathbb{Z}^*, \oplus, \langle \rangle$

$\text{cout} := \langle \rangle$

$st, e, x : \text{read}$

$st = \text{norm}$

$2 \mid e$

$\text{cout} : \text{write}(\langle e \rangle)$

$-$

$st, e, x : \text{read}$

Grey box testing

□ We have to check

- the enumerator
 - length: 0, 1, 2, and more items
 - “sides” of the enumerator: at least 2 different elements
- loading is not necessary
- condition of the assortment

length-based: input of 0, 1, 2, and more even numbers (copy)

condition-based: $x = \langle -100, -55, -2, -1, 0, 1, 2, 55, 100 \rangle$

→ $\text{cout} = \langle -100, -2, 0, 2, 100 \rangle$

C++ program

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream x;
    bool error = true;
    do{
        string fname;
        cout << "file name: ";
        cin >> fname;
        x.open(fname.c_str());
        if( (error=x.fail()) ) {
            cout << "Wrong file name!\n";
            x.clear();
        }
    }while(error);

    cout << "Selected even numbers: ";
    int e;
    while(x >> e) {
        if(0==e%2) cout << e << " ";
    }
    return 0;
}
```

After skipping the white spaces,
it reads the data of type of *e*.

st, e, x : read st==norm

x >> e;
while(!x.fail()) {
 if(0==e%2) cout << e;
 x >> e;
}

cout : write(<e>)

3rd Task

From the registry of a library, assort books of count 0 and those that were published before 2000.

A : $x:infile(Book), y:outfile(Book2), z:outfile(Book2)$

Book = rec(ID : \mathbb{N} , author: String, title : String, publisher : String,
year : String, count : \mathbb{N} , isbn : String)

Book2 = rec(ID : \mathbb{N} , author : String, title : String)

Pre : $x = x_0$

Post : $y = \bigoplus_{\substack{dx \in x_0 \\ dx.count=0}} \langle (dx.ID, dx.author, dx.title) \rangle \wedge$
 $z = \bigoplus_{\substack{dx \in x_0 \\ dx.year < "2000"} } \langle (dx.ID, dx.author, dx.title) \rangle$

Algorithm

Conditional summation:

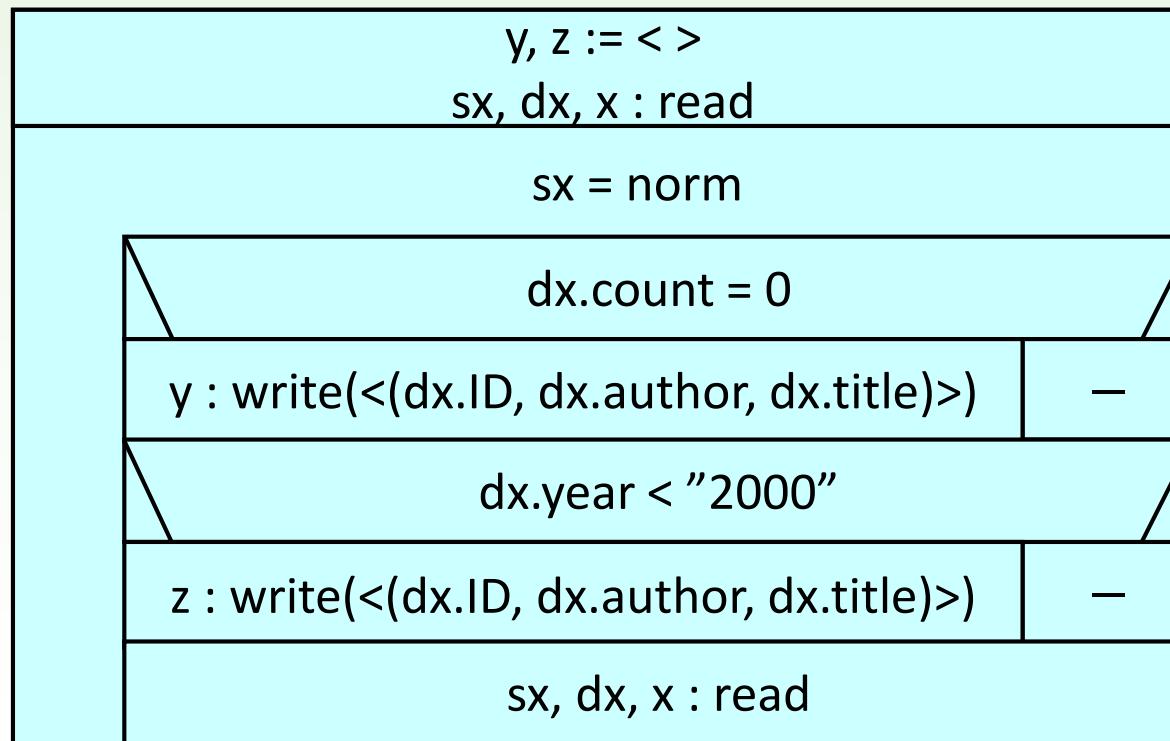
$t:enor(E) \sim x:infile(Book), sx,dx,x : read$

$e \sim dx$

$f_1(e) \sim \langle dx.ID, dx.author, dx.title \rangle, cond_1(i) \sim dx.count = 0$

$f_2(e) \sim \langle dx.ID, dx.author, dx.title \rangle, cond_2(i) \sim dx.year < "2000"$

$H,+,\oplus, \langle \rangle$



Grey box testing

❑ We have to check

- the enumerator
 - length: 0, 1, 2, and more items
 - “sides” of the enumerator: at least 2 different elements
- loading is not necessary
- conditions of the assortment

length-based: 0, 1, 2, and more books that satisfy the condition (copy)

condition-based: books of count zero and non zero and

books published before 2000 and after 1999

Implementation with operations read and write

```
bool read(Status &sx, Book &dx, ifstream &x);
void write(ofstream &x, const Book &dx);

int main()
{
    ifstream x("inp.txt");
    if (x.fail() ) { ... }
    ofstream y("out1.txt");
    if (y.fail() ) { ... }
    ofstream z("out2.txt");
    if (z.fail() ) { ... }

    Book dx;
    Status sx;
    while(read(sx,dx,x)) {
        if (0==dx.count)      write(y,dx);
        if (dx.year<"2000") write(z,dx);
    }
    return 0;
}
```

```
struct Book{
    int id;
    string author;
    string title;
    string publisher;
    string year;
    int count;
    string isbn;
};

enum Status{abnorm, norm};
```

```
read(sx,dx,x);
while(norm==sx) {
    if (0==dx.count)      write(y,dx);
    if (dx.year<"2000") write(z,dx);
    read(sx,dx,x);
}
```

Operations *read* and *write*

made up of lines,
strictly positioned input file

12 J. K. Rowling	Harry Potter II.	Animus	2000	0	963	8386	94	O
15 A. A. Milne	Winnie the Pooh	Móra	1936	10	963	11	1547	X

```
bool read(Status &sx, Book &dx, ifstream &x) {
    string line;
    getline(x, line);
    if (!x.fail()) {
        sx = norm;
        dx.id           = atoi(line.substr( 0, 4).c_str());
        dx.author       = line.substr( 5,14);
        dx.title        = line.substr(21,19);
        dx.publisher    = line.substr(42,14);
        dx.year         = line.substr(58, 4);
        dx.count        = atoi(line.substr(63, 3).c_str());
        dx.isbn         = line.substr(67,14);
    }
    else sx=abnorm;
    return norm==sx;
}
```

reads a line

transforms a character chain to integer

substring

creates a C-style string

returns a logical value

```
void write(ofstream &x, const Book &dx) {
    x << setw(4) << dx.id << ' '
    << setw(14) << dx.author << ' '
    << setw(19) << dx.title << endl;
}
```

positioned writing
#include <iomanip>

Implementation with classes

```
int main()
{
    try{
        Stock x("input.txt");
        Result y("output1.txt");
        Result z("output2.txt");

        Book dx;
        Status sx;
        while(x.read(dx, sx)) {
            if (0==dx.count)      y.write(dx);
            if (dx.year<"2000")  z.write(dx);
        }
    }catch(Stock::Errors e){
        if(Stock::FILE_ERROR==e) cout << ...
    }catch(Result::Errors e){
        if(Result::FILE_ERROR==e) cout << ...
    }
    return 0;
}
f.open(fname.c_str());
if(f.fail()) throw FILE_ERROR;
```

```
struct Book{
    int id;
    std::string author;
    std::string title;
    std::string publisher;
    std::string year;
    int count;
    std::string isbn;
};
```

```
enum Status{abnorm, norm};
class Stock{
public:
    enum Errors{FILE_ERROR};
    Stock(std::string fname);
    bool read(Book &dx, Status &sx);
private:
    std::ifstream x;
};

its body is unchanged
```

```
class Result{
public:
    enum Errors{FILE_ERROR};
    Result(std::string fname);
    void write(const Book &dx);
private:
    std::ofstream x;
};

its body is unchanged
```

4th Task

In a textfile, results of tests of students are stored. Results of one student are in one line, divided by whitespaces or tabs, data is given in the following order:

- neptun-code (6 characters),
- sequence of characters “+” and “-” without white space (non empty string)
- Results of one assignment and 4 tests (all of them between 0 and 5)

Give the final mark of those students who do not fail the course!

AA11XX	++++-+++++	5	5	5	5	5
CC33ZZ	++++---++-	2	1	0	5	5
BB22YY	--+----++-	2	2	3	3	5

Plan of the solution

A : $x : \text{infile(Student)}, y : \text{outfile(Evaluation)}$

Student = rec(neptun : String, pm : String, marks : {0..5}⁵)

Evaluation = rec(neptun : String, mark : {2..5})

Pre : $x = x_0$

Post : $y = \bigoplus_{\substack{dx \in x_0 \\ \text{cond}(dx)}} <dx.\text{neptun}, \text{avg}(dx)>$

$$\text{cond}(dx) = \forall \text{SEARCH } (dx.\text{marks}[i] > 1) \wedge \left(\sum_{i=1}^5 1 \leq \sum_{i=1}^{|dx.\text{pm}|} 1 \right)$$

$dx.\text{pm}[i] = '-' \quad dx.\text{pm}[i] = '+'$

$$\text{avg}(dx) = \left(\sum_{i=1}^5 dx.\text{marks}[i] \right) / 5$$

Conditional summation:

$t:\text{enor}(E) \sim x:\text{infile(Student)}$

$sx, dx, x : \text{read}$

$e \sim dx$

$f(e) \sim <(dx.\text{neptun}, \text{avg}(dx))>$

$H, +, 0 \sim \text{Evaluation}^*, \bigoplus, <>$

$y := < >$

$sx, dx, x : \text{read}$

$st = \text{norm}$

$\boxed{\text{cond}(dx)}$

$y : \text{write}(<(dx.\text{neptun}, \boxed{\text{avg}(dx)})>) \quad -$

$sx, dx, x : \text{read}$

Subprograms

Opt. linear search:

$t:\text{enor}(E) \sim i = 1 .. 5$
 $e \sim i$
 $\text{cond}(e) \sim \text{dx.marks}[i] > 1$

$$l := (\underset{i=1}{\overset{5}{\forall \text{SEARCH}}} \text{dx.marks}[i] > 1)$$

Two countings:

$t:\text{enor}(E) \sim i = 1 .. |\text{dx.pm}|$
 $e \sim i$
 $\text{cond1}(e) \sim \text{dx.pm}[i] = '+'$
 $\text{cond2}(e) \sim \text{dx.pm}[i] = '-'$

$$p, m := \sum_{i=1}^{|\text{dx.pm}|} 1, \sum_{i=1}^{|\text{dx.pm}|} 1$$

$\text{dx.pm}[i] = '+' \quad \text{dx.pm}[i] = '-'$

Summation:

$t:\text{enor}(E) \sim i = 1 .. 5$
 $e \sim i$
 $f(e) \sim \text{dx.marks}[i]$
 $H, +, 0 \sim \mathbb{N}, +, 0$

$$s := (\underset{i=1}{\overset{5}{\Sigma}} \text{dx.marks}[i]) / 5$$

$l := \text{cond}(dx)$

$l, i := \text{true}, 1$

$l \wedge i \leq 5$

$l := \text{dx.marks}[i] > 1$

$i := i + 1$

$p, m := 0, 0$

$i = 1 .. |\text{dx.pm}|$

$\text{dx.pm}[i] = '+'$

$p := p + 1$

$\text{dx.pm}[i] = '-'$

$m := m + 1$

$l := l \wedge p \geq m$

$a := \text{avg}(\text{dx.marks})$

$s := 0$

$i = 1 .. 5$

$s := s + \text{dx.marks}[i]$

$a := s / 5$

Grey box testing

Outer conditional summation:

<u>length-based</u> :	0, 1, 2, and more students who pass the course
“sides” of the <u>enumerator</u> :	done by the above
<u>loading</u> :	not needed
<u>cond()</u> and <u>f()</u> :	see below

Counting pluses and minuses:

<u>length-based</u> :	0, 1, 2, and more, only ‘+’
“sides” of the <u>enumerator</u> :	enumerations of 2 items, with ‘+’ and ‘-’ (4 cases)
<u>result-based</u> :	0, 1, and more ‘-’ with some ‘+’es

There is no failed test (optimistic linsearch) :

<u>length-based</u> :	not needed (length is 5)
“sides” of the <u>enumerator</u> :	only the first test is failed, only the last one is failed
<u>result-based</u> :	only 1s, there is 1, all of them at least 2

Sum of the marks:

<u>length-based</u> :	not needed (length is 5)
“sides” of the <u>enumerator</u> :	different marks at the beginning and at the end
<u>loading</u> :	not needed

C++ program

```
bool cond(const vector<int> &marks, const string &pm );
double avg(const vector<int> &marks);

int main() {
    try{
        InpFile x("input.txt");
        OutFile y("output.txt");
        Student dx;
        Status sx;
        while(x.read(dx,sx)) {
            if (cond(dx.marks, dx.pm)) {
                Evaluation dy(dx.neptun, avg(dx.marks));
                y.write(dy);
            }
        }
    }catch( InpFile::Errors er ) {
        if( er==InpFile::FILE_ERROR ) cout << ... ;
    }catch( OutFile::Errors er ) {
        if( er==OutFile::FILE_ERROR ) cout << ... ;
    }
    return 0;
}
```

C++ functions

```
bool cond(const vector<int> &marks, const string &pm) {
    bool l = true;
    for(unsigned int i=0; l && i<marks.size(); ++i) {
        l=marks[i]>1;
    }
    int p, m; p = m = 0;
    for(unsigned int i = 0; i<pm.size(); ++i) {
        if(pm[i]=='+') ++p;
        if(pm[i]=='-') ++m;
    }
    return l && m<=p;
}
```

```
double avg(const vector<int> &marks) {
    double s = 0.0;
    for(unsigned int i = 0; i< marks.size(); ++i) {
        s += marks[i];
    }
    return (0 == marks.size() ? 0 : s / marks.size());
}
```

Sequential input file

```
struct Student {  
    std::string neptun;  
    std::string pm;  
    std::vector<int> marks;  
};  
enum Status {abnorm, norm};  
  
class InpFile{  
public:  
    enum Errors{FILE_ERROR};  
    InpFile(std::string fname){  
        x.open(fname.c_str());  
        if(x.fail()) throw FILE_ERROR;  
    }  
    bool read( Student &dx, Status &sx);  
private:  
    std::ifstream x;  
};
```

```
bool InpFile::read(Student &dx, Status &sx)  
{  
    string line;  
    getline(x, line);  
    if (!x.fail() && line!="") {  
        sx=norm;  
        istringstream in(line);  
        in >> dx.neptun;  
        in >> dx.pm;  
        dx.marks.clear();  
        int mark;  
        while( in >> mark )  
            dx.marks.push_back(mark);  
    } else sx=abnorm;  
    return norm==sx;  
}
```

Sequential output file

```
struct Evaluation {
    std::string neptun;
    double mark;
    Evaluation(std::string str, double j) : neptun(str), mark(j) {}
};

class OutFile{
public:
    enum Errors{FILE_ERROR};
    OutFile(std::string fname) {
        x.open(fname.c_str());
        if(x.fail()) throw FILE_ERROR;
    }
    void write(const Evaluation &dx) {
        x.setf(std::ios::fixed);
        x.precision(2);
        x << dx.neptun << std::setw(7) << dx.mark << std::endl;
    }
private:
    std::ofstream x;
};
```

#include <iomanip>

Task and program modification

In the textfile, lines begin with the name of the students which consists of optional number of (but at least one) parts (separators in between).

Muhammad Ali	AA11XX	++++++	5	5	5	5	5
Cher	CC33ZZ	++++-+++	2	1	0	5	1
Cristiano Ronaldo dos Santos Aveiro	BB22YY	----+---	2	4	4	0	0

```
int main() {
    try{
        InpFile x("input.txt");
        OutFile y("output.txt");
        Student dx;
        Status sx;
        while(x.read(dx,sx)) {
            if (dx.has) {
                Evaluation dy(dx.neptun, dx.result);
                y.write(dy);
            }
        }
    ...
}
```

```
struct Student {
    std::string name;
    std::string neptun;
    bool has;
    double result;
};
```

Reading varying number of data

```
bool InpFile::read(Student &dx, Status &sx)
{
    string line, str;
    getline(f, line);
    if (!f.fail() && line!="") {
        sx=norm;
        istringstream in(line);
        in >> dx.name;
        in >> dx.neptun;
        in >> str;
        while( !( '+'== str[0] || '-'== str[0]) ) {
            dx.name += " " + dx.neptun;
            dx.neptun = str;
            in >> str;
        }
        vector<int> marks;
        int mark;
        while( in >> mark ) marks.push_back(mark);
        dx.has = cond(marks, str);
        dx.result = avg(marks);
    } else sx=abnorm;
    return norm==sx;
}
```

filling dx based on variable *line*

for reading data from the line (#include <iostream>)

If str does not start with + or -, then it is part of the name, or it is the neptun code

we thought it was a neptun code, but it is part of the name

str is considered to be a neptun code

private methods of class InpFile

Reading from input files

x : infile(E)	st, data, x : read	st = abnorm
E ≡ char // characters without separation	x.get(data); x >> data; //x.unsetf(ios::skipws)	x.eof() x.fail()
E ≡ <basic type> // basic values divided by separators	x >> data;	x.fail()
E ≡ struct(s1 : <basic type>, s2 : <basic type>, sn : <basic type> ⁿ , ...) // record of fixed number of basic types // divided by separators	x >> data.s1 >> data.s2; for(int i=0; i<n; ++i) { x >> data.sn[i]; }	x.fail()
E ≡ line // line-buffered data // number of data varies	string data; getline(x, data); istringstream is(data); is >> ...	x.fail()

Custom enumerators

Types of enumerators

- ❑ Until now, only **standard enumerations** of well-known collections were shown, where creating an enumerator class is not necessary.
- ❑ **Custom enumerator** is that
 - does not traverse the items of the collection in a usual way, or
 - uses more items of the collection in one step, or
 - merges more collections
- ❑ An enumerator may be considered as custom, if a standard enumerator
 - does not start with operation `first()` because it is already in state “in operation”, or
 - ends before reaching the last item of the collection, or
 - there is no real collection behind the enumerator

1st task

Given two function $f : \mathbb{N} \rightarrow \mathbb{R}$ and $g : \mathbb{N} \rightarrow \mathbb{R}$, and a number $e \in \mathbb{R}$. There exist two arguments, i and j , for which $f(i) + g(j) = e$ holds. Give such i and j !

$A : e : \mathbb{R}, i : \mathbb{N}, j : \mathbb{N}$

$Pre : e = e_0 \wedge \exists i, j \in \mathbb{N} : f(i) + g(j) = e$

$Post : Pre \wedge f(i) + g(j) = e$

Unfortunately, there is no clue of the algorithmic pattern in this specification.

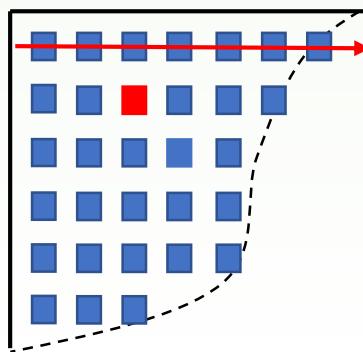
Idea

Arrange the values of $f(i)+g(j)$ in an infinite **matrix**, where indexes start from 0 and value $f(i)+g(j)$ is located in the i^{th} row and j^{th} column in the matrix.

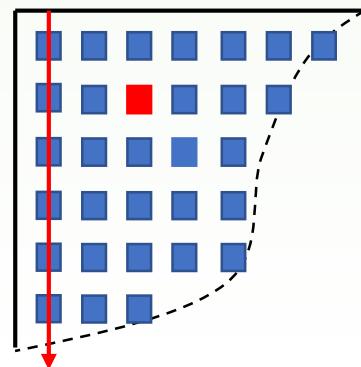
Define an enumerator that traverses the indexes of this matrix.

Standard enumeration (row- and column major order) is not good, since the rows and columns are infinite long. It would not traverse the second and higher rows/columns.

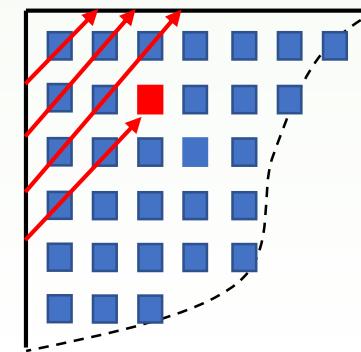
row-major order



column-major order



antidiagonal order



Plan

There is no collection behind this enumerator,
just a theoretical matrix traversed in an unusual way.

$A : t:\text{enor}(\mathbb{N} \times \mathbb{N}), e:\mathbb{R}, i:\mathbb{N}, j:\mathbb{N}$

$Pre : t = t_0 \wedge e = e_0 \wedge \exists i,j \in \mathbb{N} : f(i) + g(j) = e$

$Post : e = e_0 \wedge (i,j) = \text{SELECT}_{(i,j) \in t_0} (f(i) + g(j) = e)$

Selection:

$t:\text{enor}(E) \sim t:\text{enor}(\mathbb{N} \times \mathbb{N})$
 $e \sim (i,j)$
 $\text{cond}(e) \sim f(i) + g(j) = e$

$t.\text{first}()$

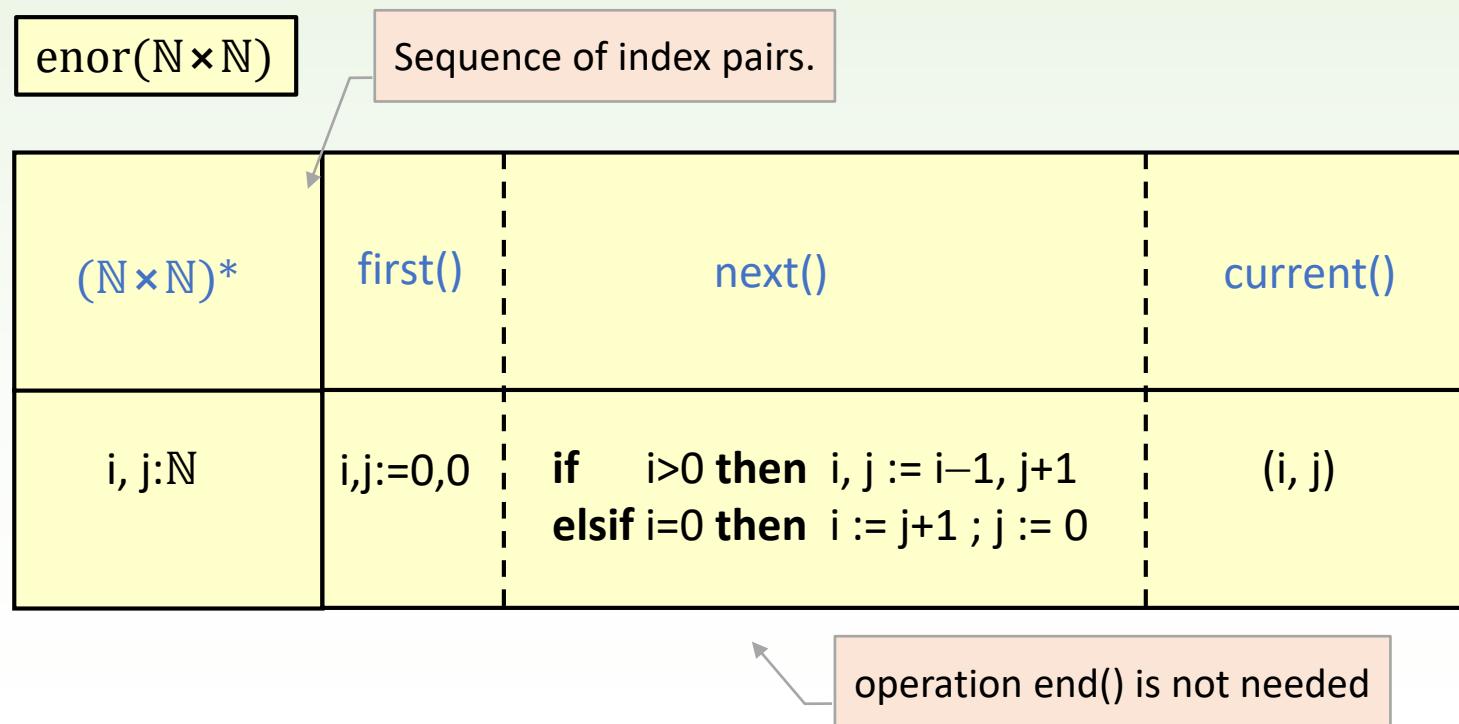
$\neg \text{cond}(t.\text{current}())$

$t.\text{next}()$

$i, j := t.\text{current}()$

Enumerator of the pair of indexes

The enumerator pretends to give the next row- and column indexes of an existing matrix.



Plan

There is no collection behind this enumerator,
just a theoretical matrix traversed in an unusual way.

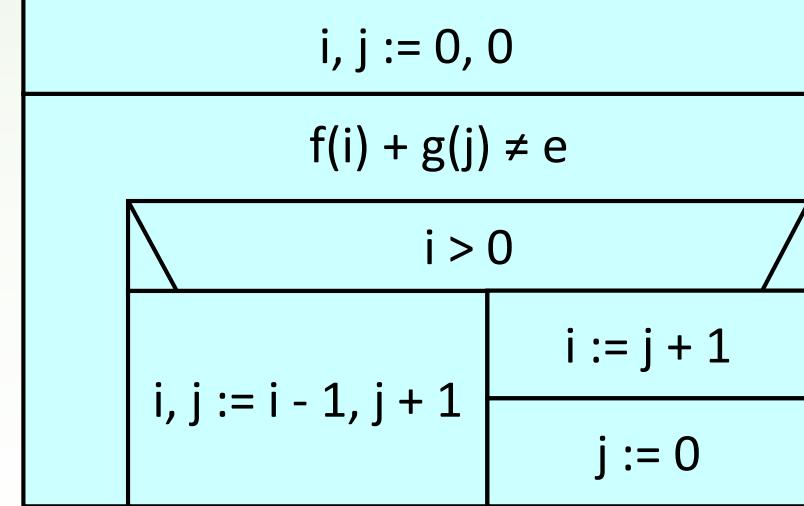
$A : t:\text{enor}(\mathbb{N} \times \mathbb{N}), e:\mathbb{R}, i:\mathbb{N}, j:\mathbb{N}$

$Pre : t = t_0 \wedge e = e_0 \wedge \exists i,j \in \mathbb{N} : f(i) + g(j) = e$

$Post : e = e_0 \wedge (i,j) = \text{SELECT}_{(i,j) \in t_0} (f(i) + g(j) = e)$

Selection:

$t:\text{enor}(E) \sim t:\text{enor}(\mathbb{N} \times \mathbb{N})$
 $e \sim (i,j)$
 $\text{cond}(e) \sim f(i) + g(j) = e$



Testing the selection

□ Testing of selection means testing the enumerator.

- Length-based: the enumerator is infinite, but it can be tested that the searched item is the first, second, or umpteenth item. Or the searched item is in the first, second, or umpteenth diagonal.
- It can be considered that the searched item is at the beginning, in the middle, or at the end of a diagonal.
- Beginning of the enumeration: $f(0)+g(0)=e$

Program

```
int main()
{
    double e = read<double>("Give a real number: ",
                             "This is not a real number!", all);

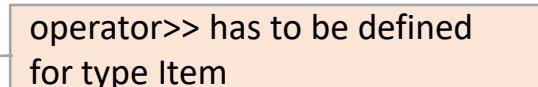
    int i, j;
    i = j = 0;
    while( f(i)+g(j) !=e ) {
        if(i>0) { --i; ++j; }
        else      { i = j+1; j = 0; }
    }

    cout << "The given number is equal to the sum f("
        << i << ") + g(" << j << ")\n";

    return 0;
}
```

The code illustrates the use of function templates and exception handling. It defines a function `all` that returns `true`. The `main` function reads a real number from the user. It then uses a `while` loop to find two integers `i` and `j` such that `f(i) + g(j)` equals the input number. If it finds such a pair, it prints the result. If no such pair exists, it prints an error message. The `cout` statement uses the `<<` operator to print the values of `i` and `j`.

Template for reading

```
template <typename Item>
Item read( const std::string &msg, const std::string &err, bool valid(Item))
{
    Item n;
    bool wrong;
    do{
        std::cout << msg;
        std::cin >> n; 
        if((wrong = std::cin.fail())) std::cin.clear();
        std::string tmp = "";
        getline(std::cin, tmp);
        wrong = wrong || tmp.size()!=0 || !valid(n);
        if(wrong) std::cout << err << std::endl;
    }while(wrong);
    return n;
}
```

2nd task

On a trip, somebody's smartwatch measured the altitude in every minute, and stored it in a sequential input file. What percent of the trip went uphill?

$A : f:\text{infile}(\mathbb{R}), v:\mathbb{R}$

$\text{Pre} : f = f_0 \wedge |f_0| \geq 2$

$\text{Post} : v = (\sum_{i=2 \dots |f_0|} 1) / (\sum_{i=2 \dots |f_0|} 1)$
 $f_0[i] > f_0[i-1]$

Two successive items are referred, but reading of a sequential input file does not support it.

$A : t:\text{enor}(\mathbb{R} \times \mathbb{R}), v:\mathbb{R}$

$\text{Pre} : t = t_0 \wedge |t_0| > 0$

$\text{Post} : v = (\sum_{(\text{first}, \text{second}) \in t_0} 1) / (\sum_{(\text{first}, \text{second}) \in t_0} 1)$
first < second

Instead of a sequential input file, a custom enumerator would be useful that enumerates the successive pairs of the file.

Buffered enumerator

The enumerator pretends to read the next two successive items of the file.

$\text{enor}(\mathbb{R} \times \mathbb{R})$	A sequence of successive pairs from the original file.			
$(\mathbb{R} \times \mathbb{R})^*$	<code>first()</code>	<code>next()</code>	<code>current()</code>	<code>end()</code>
<code>f: infile(\mathbb{R})</code> <code>first, second : \mathbb{R}</code> <code>st : Status</code>	<code>st, first, f : read</code> <code>st, second, f: read</code>	<code>first := second</code> <code>st, second, f : read</code>	<code>(first, second)</code>	<code>ist = abnorm</code>

Plan

$A : t: \text{enor}(\mathbb{R} \times \mathbb{R}), v:\mathbb{R}$

$Pre : t = t_0 \wedge |t_0| > 0$

$Post : v = (100 \cdot \sum_{\substack{\text{first}, \text{second} \\ \text{first} < \text{second}}} 1) / (\sum_{\text{first}, \text{second} \in t_0} 1)$

Counting and summation
are in the same loop.

Counting:

$t: \text{enor}(E) \sim$	$t: \text{enor}(\mathbb{R} \times \mathbb{R})$
$e \sim$	$(\text{first}, \text{second})$
$\text{cond}(e) \sim$	$\text{second} > \text{first}$

Summation:

$t: \text{enor}(E) \sim$	$t: \text{enor}(\mathbb{R} \times \mathbb{R})$
$e \sim$	$(\text{first}, \text{second})$
$f(e) \sim$	1
$H, +, 0 \sim$	$\mathbb{N}, +, 0$

$st, \text{first}, f : \text{read}$
$st, \text{second}, f : \text{read}$
$c, d := 0, 0$

$st = \text{norm}$

$\text{first} < \text{second}$	
$c := c + 1$	$-$

$d := d + 1$

$\text{first} := \text{second}$
$st, \text{second}, f : \text{read}$

$v := 100 \cdot c / d$

Testing counting and summation

- ❑ Counting and summation use the same enumerator. These cases do not have to be tested twice:
 - length-based: one, two, or more (always uphill)
 - beginning of the enumeration: uphill just at the beginning
 - end of the enumeration : uphill just at the end
- ❑ Result-based (counting):
 - there is no uphill
 - there is only one minute of uphill
 - there are more uphills
- ❑ Testing the memory loading at the summation is not necessary.

Program

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

    int first, second;
    int c = 0; int d = 0;
    for( f >> first >> second; !f.fail(); first = second, f >> second) {
        if( first < second ) ++c;
        ++d;
    }

    cout.setf(ios::fixed);
    cout.precision(2);
    cout << "Rate of the uphill part of the trip: "
        << (100.0*c)/d << "%" << endl;

    return 0;
}
```

```
f >> first >> second;
while( !f.fail() )
    if( first < second ) ++c;
    ++d;
    first = second;
    f >> second;
```

real division, since the dividend is double

3rd task

On a trip, somebody's smartwatch measured the altitude in every minute, and stored it in a sequential input file. How long was the longest consecutive uphill?

A : $f:\text{infile}(\mathbb{R})$, $\text{max}:\mathbb{N}$

Pre : $f = f_0 \wedge$

$\exists i \in [1 .. |f|] : f[i] > f[i-1]$

Post : ?

It should be a maximum search in the length of the consecutive uphills, but they cannot be read from the file. It is difficult to specify the task precisely.

Idea



- ❑ It would be nice to enumerate the length of the consecutive uphills. Then it would be easy to find the maximum.
- ❑ For that, it should be seen in which minute slopes the road upwards and downwards. Based on the original file, it is an easy task.

t:

2 1 3 2

b:

0 1 1 0 1 0 1 1 1 0 0 1 1

f:

450 432 444 448 437 445 439 448 456 463 458 450 457 464

Plan

Suppose to have an enumerator that enumerates the length of the consecutive uphills.

$A : t:\text{enor}(\mathbb{N})$, $\text{max}:\mathbb{N}$

$Pre : t = t_0 \wedge |t_0| > 0$

$Post : \text{max} = \mathbf{MAX}_{e \in t_0} e$

Maximum search:

$t:\text{enor}(E) \sim t:\text{enor}(\mathbb{N})$

$f(e) \sim e$

$H, > \sim \mathbb{N}, >$

$t.\text{first}()$

$\text{max} := t.\text{current}()$

$t.\text{next}()$

$\neg t.\text{end}()$

$t.\text{current}() > \text{max}$

$\text{max} := t.\text{current}()$

$t.\text{next}()$

Testing the maximum search

❑ Enumerator

- length-based: one, two, or more
- beginning of the enumerator: first is the maximum
- end of the enumerator: last is the maximum

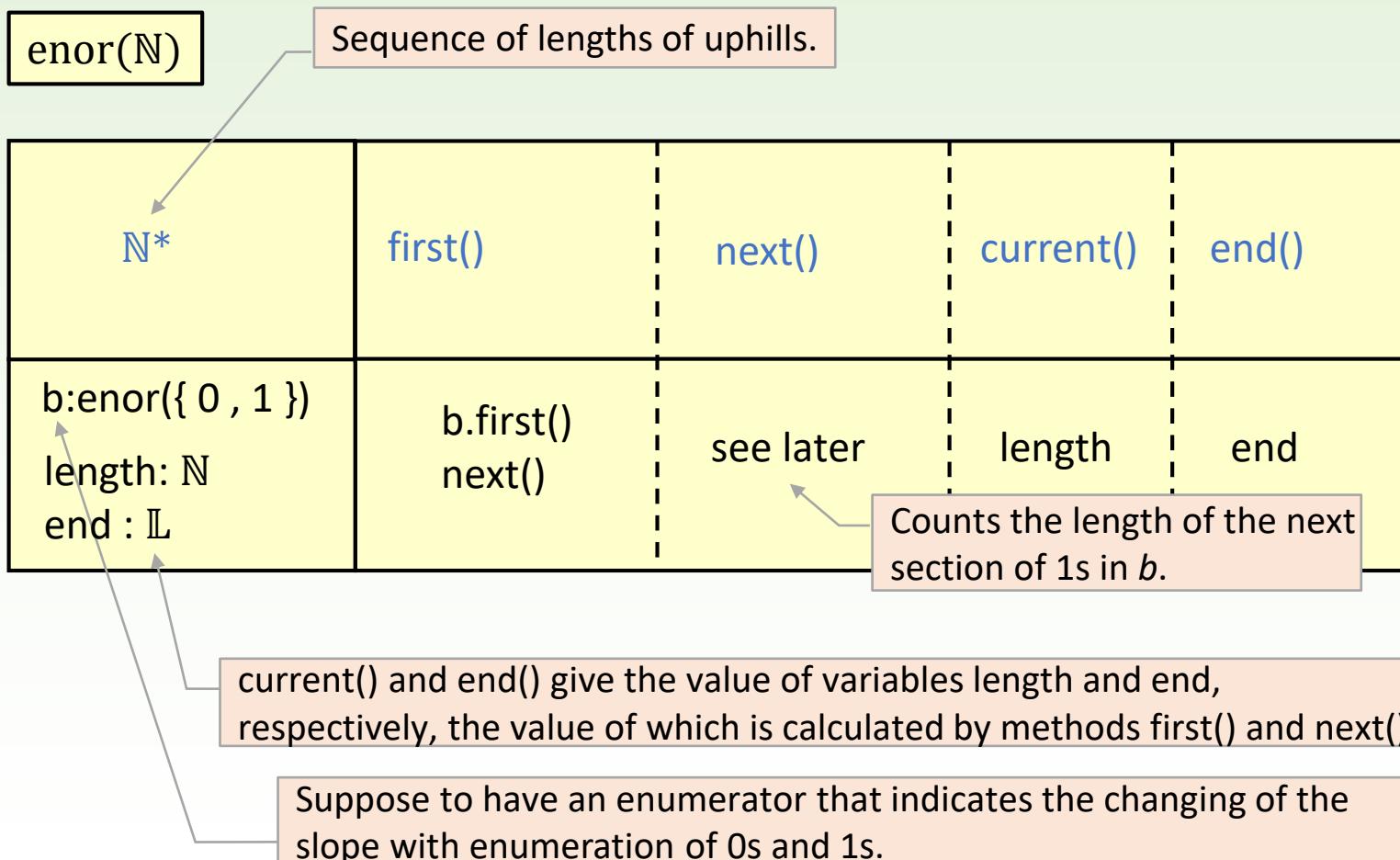
❑ Result-based:

- One maximum
- More maxima

Main program

```
int main()
{
    LengthEnumerator t("input.txt");
    t.first();
    int max = t.current();
    for( t.next(); !t.end(); t.next() ) {
        if( max < t.current() ) max = t.current();
    }
    cout << "The length of the longest uphill: " << max << endl;
    return 0;
}
```

Abstract enumerator of lengths



Operation next()

Counts the length of the next section of 1s, if there is any.

$A : b:\text{enor}(\{0, 1\}), \text{length}:\mathbb{N}, \text{end}:\mathbb{L}$

- b' – initial state of b
- b'' – state of b after jumping over the 0s
- b – final state of the enumerator

$\text{Pre} : b = b' \wedge b'$ is „in process” ($\neg b.\text{end}()$)

Looks for the beginning of the next section of 1s or the end of the enumerator in b' .

$\text{Post} : (e'', b'') = \text{SELECT}_{e \in (b'.\text{current}(), b')} (b'.\text{end}() \vee e = 1) \wedge$

This notation (instead of $e \in b'$) means that $b'.\text{current}()$ is accessible without $b'.\text{first}()$ since b' is already “in process”.

$\wedge \text{end} = b''.\text{end}()$

$\wedge (\neg \text{end} \rightarrow (\text{length}, b = \sum_{e \in (e'', b'')} 1))$

Length of the next section of 1s is given by a summation which holds as long as $e = 1$ (here $e = b'.\text{current}()$). It can stop before reaching the end of b . For that, the previously processed enumeration is used, where $e'' = b''.\text{current}()$.

If there is another section of 1s, then $e'' = b''.\text{current}() = 1$ and $\neg b''.\text{end}()$.

Notation for specification

Enumeration until the end:

t:enor(E)	Σ, MAX	h:set(E)	i:[m .. n]	x:infile(E)
$r = \boxtimes_{e \in t_0} f(e)$		$r = \boxtimes_{e \in h_0} f(e)$	$r = \boxtimes_{i=m..n} f(i)$	$r = \boxtimes_{dx \in x_0} f(dx)$
$r, t = \boxtimes_{e \in t_0} \text{cond}(e)$	$r, h = \boxtimes_{e \in h_0} \text{cond}(e)$	$r, i = \boxtimes_{i=m..n} \text{cond}(i)$	$r, (sx, dx, x) = \boxtimes_{dx \in x_0} \text{cond}(dx)$	

SEARCH, SELECT

sx, dx, x (like h or i) indicates the actual and the final state of the enumerator.

Enumeration as long as a condition holds:

t:enor(E)	h:set(E)	i:[m .. n]	x:infile(E)
$\dots = \boxtimes_{e \in t_0}^{\text{cond}(e)} f(e)$	$\dots = \boxtimes_{e \in h_0}^{\text{cond}(e)} f(e)$	$\dots = \boxtimes_{i=m..n}^{\text{cond}(i)} f(i)$	$\dots = \boxtimes_{dx \in x_0}^{\text{cond}(dx)} f(dx)$

$\neg t.\text{end}() \wedge \text{cond}(e)$

$h \neq \emptyset \wedge \text{cond}(e)$

$i \leq n \wedge \text{cond}(i)$

$sx = \text{norm} \wedge \text{cond}(e)$

Continue an enumerator previously stopped:

t:enor(E)	h:set(E)	i:[m .. n]	x:infile(E)
$\dots = \boxtimes_{e \in (t'.\text{current}(), t')} f(e)$	$\dots = \boxtimes_{e \in h'} f(e)$	$\dots = \boxtimes_{i=i'+1..n} f(i)$	$\dots = \boxtimes_{dx \in (dx', x')} f(dx)$

$sx, dx, x : \text{read}$
 enumerates (sx, dx) pairs, but instead of
 $(sx, dx) \in x_0$,
 $dx \in x_0$ is easier.

Operation next()

$$\begin{aligned}
 e'', b'' = & \text{SELECT}_{e \in (b'.\text{current}(), b')} (b.\text{end}() \vee e=1) \wedge \text{end} = b''.\text{end}() \\
 & e = 1 \\
 \wedge & (\neg \text{end} \rightarrow (\text{length}, b = \sum_{e \in (e'', b'')} 1))
 \end{aligned}$$

Selection with enumerator "in process"

$t:\text{enor}(E) \sim b:\text{enor}(\mathbb{L})$

without $\text{first}()$

$\text{cond}(e) \sim b.\text{end}() \vee b.\text{current}()=1$

Summation as long as a condition holds with enumerator "in process"

$t:\text{enor}(E) \sim b:\text{enor}(\mathbb{L})$

without $\text{first}()$

while $b.\text{current}()=1$

$s \sim \text{length}$

$f(e) \sim 1$

$H,+,\emptyset \sim \mathbb{N},+,0$

$\neg b.\text{end}() \wedge b.\text{current}() \neq 1$

$b.\text{next}()$

$\text{end} := b.\text{end}()$

$\neg \text{end}$

$\text{length} := 0$

$\neg b.\text{end}() \wedge b.\text{current}()=1$

$\text{length} := \text{length} + 1$

$b.\text{next}()$

Grey box testing of next()

❑ Selection with enumerator “in process”:

- length-based: zero, one, or more slope without uphill
- beginning of the enumerator: uphill right at the beginning
- end of the enumerator: uphill just at the end
- Condition of selection: there is or there is no uphill

❑ Summation as long as a condition holds with enumerator “in process”:

- length-based: zero, one, or longer uphill at the beginning
- beginning of the enumerator: uphill just at the beginning
- end of the enumerator: uphill just at the end
- Loading: not necessary

Enumerator class of lengths

```
class LengthEnumerator{
public:
    LengthEnumerator(const std::string &fname) : _b(fname) {}
    void first() { _b.first(); next(); }
    int current() const { return _length; }
    bool end() const { return _end; }
    void next();
private:
    BitEnumerator _b;
    int _length;
    bool _end;
};
```

```
void LengthEnumerator::next()
{
    for( ; !_b.end() && !_b.current(); _b.next() );
    if ( (_end = _b.end()) ) return;
    for( _length = 0 ; !_b.end() && _b.current(); _b.next() ) ++_length;
}
```

Abstract enumerator of slope changing

enor({ 0 , 1 })

Sequence of 0s and 1s the length of which is size of the original file – 1 (processes two successive elements in the file). If the slope goes upwards, the actual item of the enumerator is 1 (0 otherwise).

{ 0 , 1 }*	first()	next()	current()	end()
f: infile(\mathbb{R}) first, second: \mathbb{R} st: Status	st, first, f: read st, second, f: read	first:= second st, second, f : read	first < second 1 0	st = abnorm

Enumerator class of slope changing

```
class BitEnumerator{
public:
    enum Errors { FILEERROR };
    BitEnumerator(const std::string &fname) {
        _f.open(fname);
        if(!_f.fail()) throw FILEERROR;
    }
    void first() { _f >> _first >> _second; }
    void next() { _first = _second; _f >> _second; }
    int current() const { return (_first < _second ? 1 : 0); }
    bool end() const { return _f.fail(); }
private:
    std::ifstream _f;
    int _first, _second;
};
```

Relationship between objects

Types of relationships

- ❑ When objects **communicate with each other** (they call the methods of each other synchronously or asynchronously, they send signals to each other, or they operate with the attributes of the other), then they establish relationship between each other.
- ❑ There are five types of relationships between objects:
 - dependency
 - association
 - aggregation or shared aggregation
 - composition or composite aggregation
 - inheritance

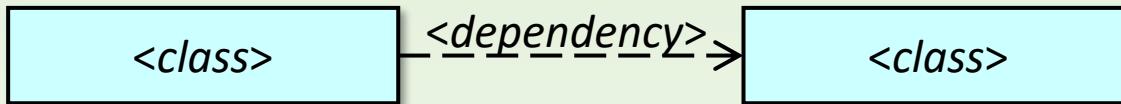
Abstraction

- ❑ *Objects* and their relationships (*link*) are represented by object diagrams, but objects and the properties of their relationship are shown in a class diagram, on a higher abstraction level.
- ❑ Class diagram is the abstraction of the object diagram.
- ❑ Unfortunately, in most cases, the programming languages do not provide tools to describe the relationships between objects (except the inheritance), they only know the abstraction of objects, which are the classes.

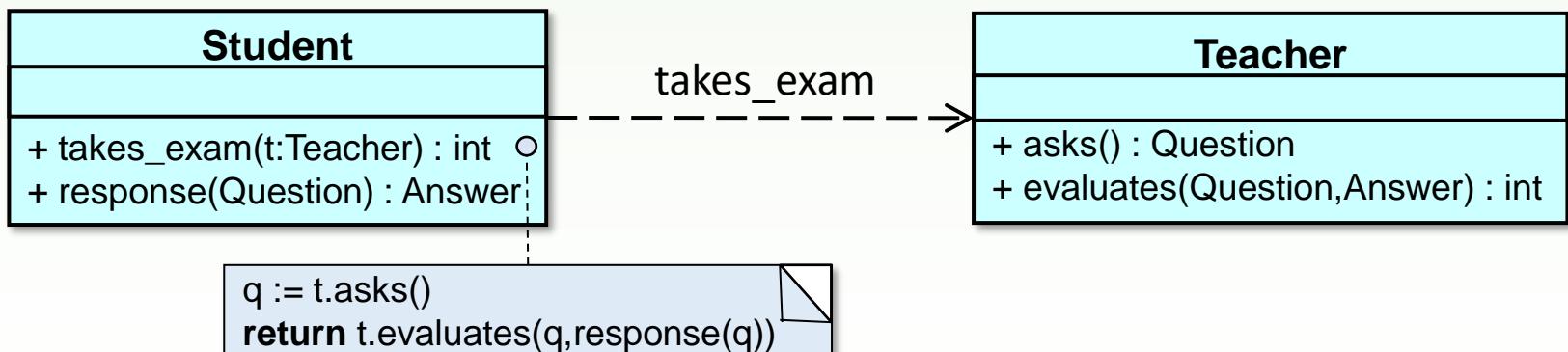
4

Important criterion of object orientation is **abstraction**.

Dependency



- ❑ When a method of a class gets in touch with an object of another class **temporary**, e.g. gets it **as a parameter** or locally **instantiates** it to **call its method**, to **send a signal** to it, or to **forward the reference** of the object (for example at exception throwing).
- ❑ When a method of a class calls a class-level method of an other class.

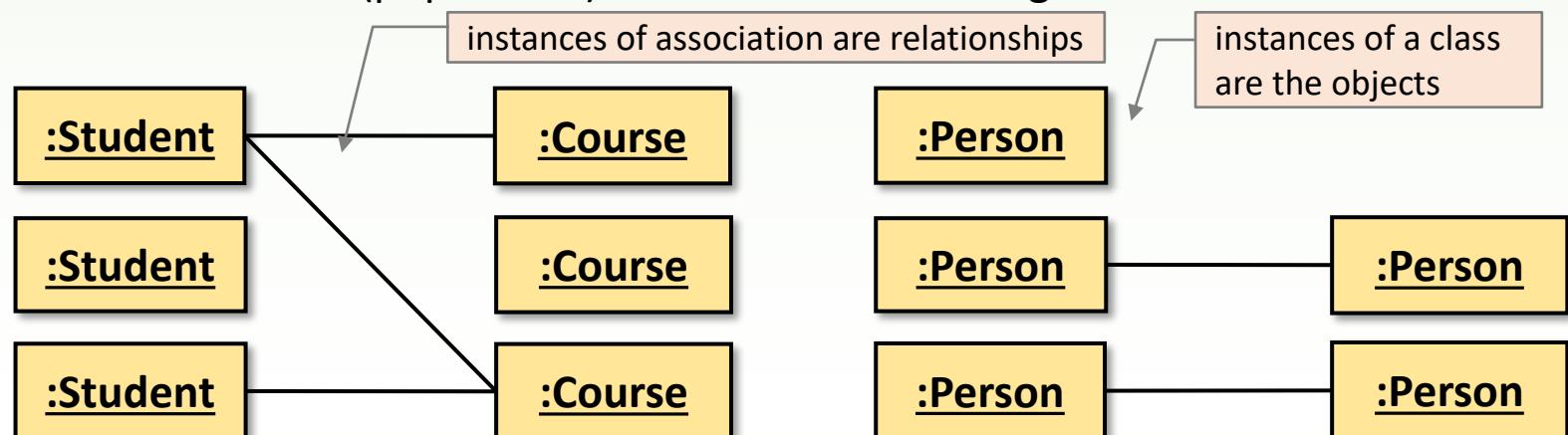


Association

- Expresses a **long-term** relationship between objects (permanent dependency between objects).
- One association might describe several relationships.



Possible instantiation (population) of the above class diagrams:

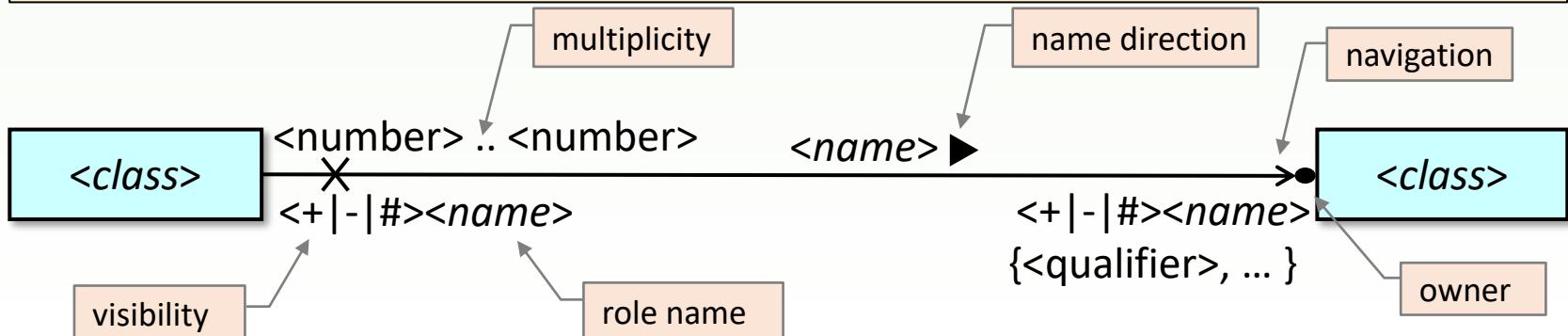


Properties of association

❑ An association might have several properties, like

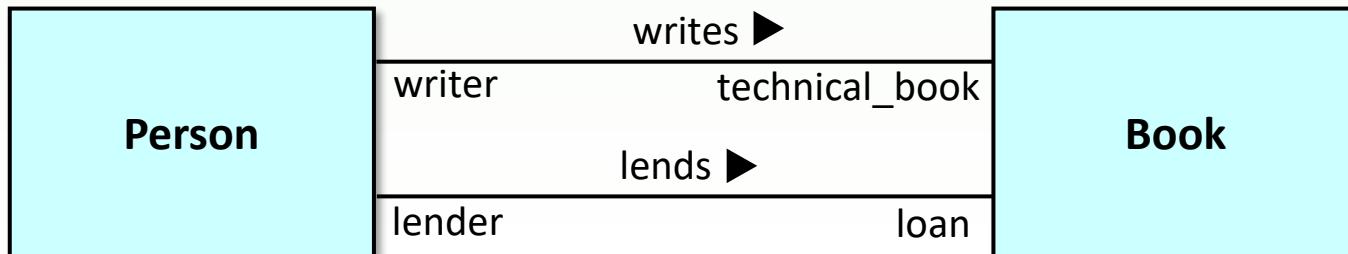
- name
- name direction
- multiplicity
- arity (binary or n-ary associations)
- navigation
- end names of the association (role names)
- visibility and owner of the end names of the association

❑ If a property is missing, it means that it is not clear yet.



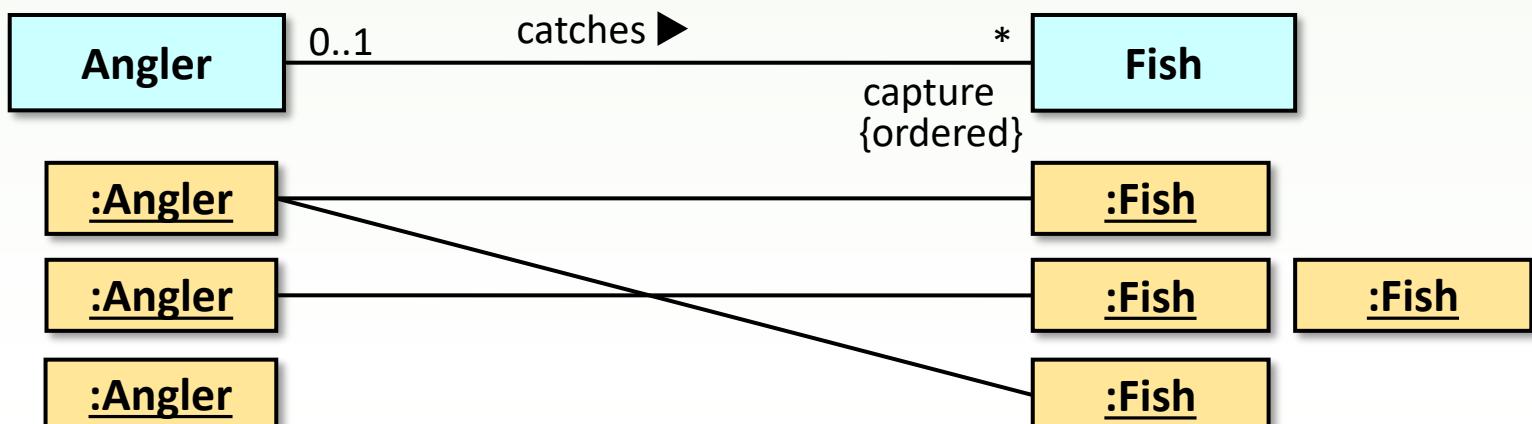
Name and name direction

- ❑ Association is usually described by a complex sentence, where the predicate is the **name of the association**, the rest stand for the **end names** (role names).
- ❑ **Binary** (between two objects) associations are usually described by
 - a sentence of **subjective, predicate, and object**, where the end names are the subjective and the object.
 - The black triangle (like an arrow) refers to the object of the sentence: it is the **name direction**.



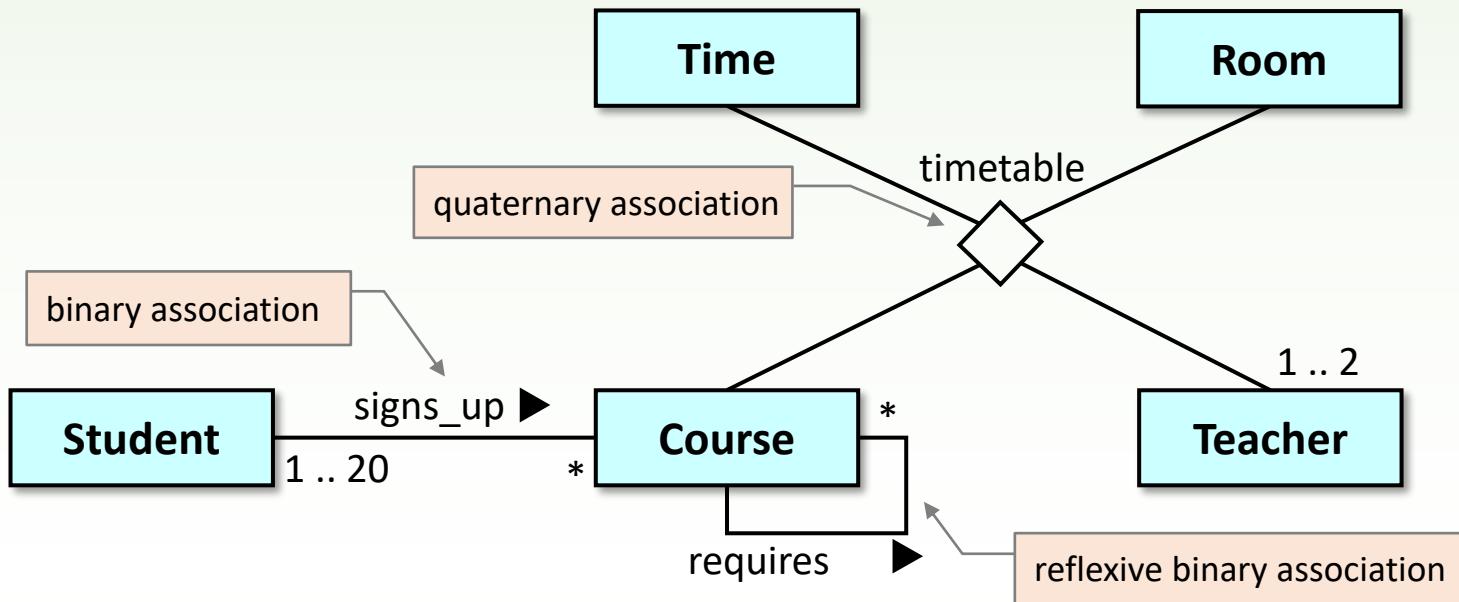
Multiplicity

- ❑ Multiplicity shows that how many (min .. max) objects of the class with the multiplicity can establish relationship simultaneously with an object of the other class in the association.
 - multiplicity 1 can be skipped
 - instead of 0 .. *, notation * is used, where * is an arbitrary natural number
- ❑ If the multiplicity is „many”, it might be prescribed that the objects at the many side are
 - all unique {unique},
 - in a given order {ordered}.



Arity

- ❑ Arity means that how many type of objects are connected.
- ❑ Until now, only binary associations were shown, where 2 type of objects were connected.
 - Same object may be present in more relationships.
 - Reflexive association connects two objects of the same class.



Navigation

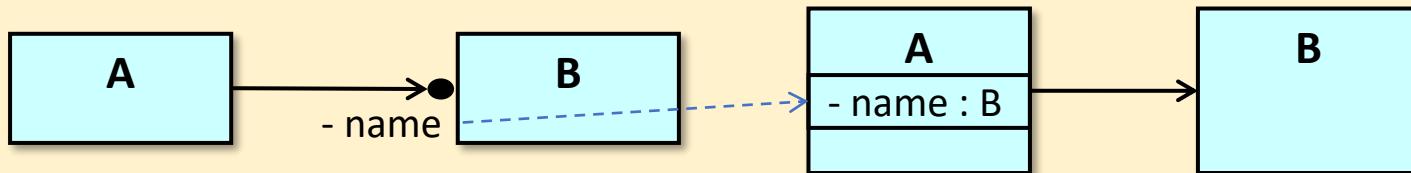
- ❑ Navigation shows which object should **reach** the other **effectively**.
 - Effective navigation in a given direction is denoted by an arrow at the given end of association.
 - x denotes the **non-supported direction of the navigation**.
 - Unsigned association refers to an **undefined** navigation.
- ❑ Navigation direction and name direction are different expressions, their directions might differ.



```
class Person {  
private:  
    IdentityCard *_ic;  
public:  
    enum Error {ESCAPE};  
    IdentityCard showIdentityCard() const {  
        if (_id != nullptr) return *_ic;  
        else throw ESCAPE;  
    }  
};
```

Owner of the end name

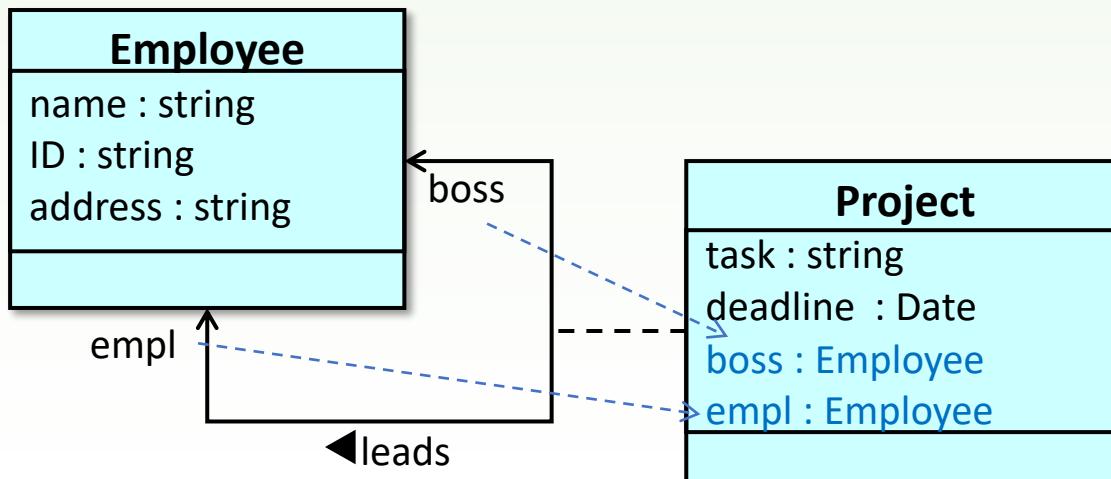
- ❑ Objects in a relationship might be referred by the end names ([role names](#)). Where are the references stored?
Who is the owner of the role name?
 - It might be the [association](#) itself: the relationship stores the objects and the storage can be accessed by all of the related objects.
 - It might be [the other class\(es\) on the other side](#): in this case, the attribute name in the other class is the role name. It is denoted by a *black fleck* on the side of the role name.



- ❑ [Visibility](#) of the role name (private, protected, public) shows if the name is public or, only can be seen by the owner.
- ❑ [Multiplicity](#) of a class shows if it is a collection or not.

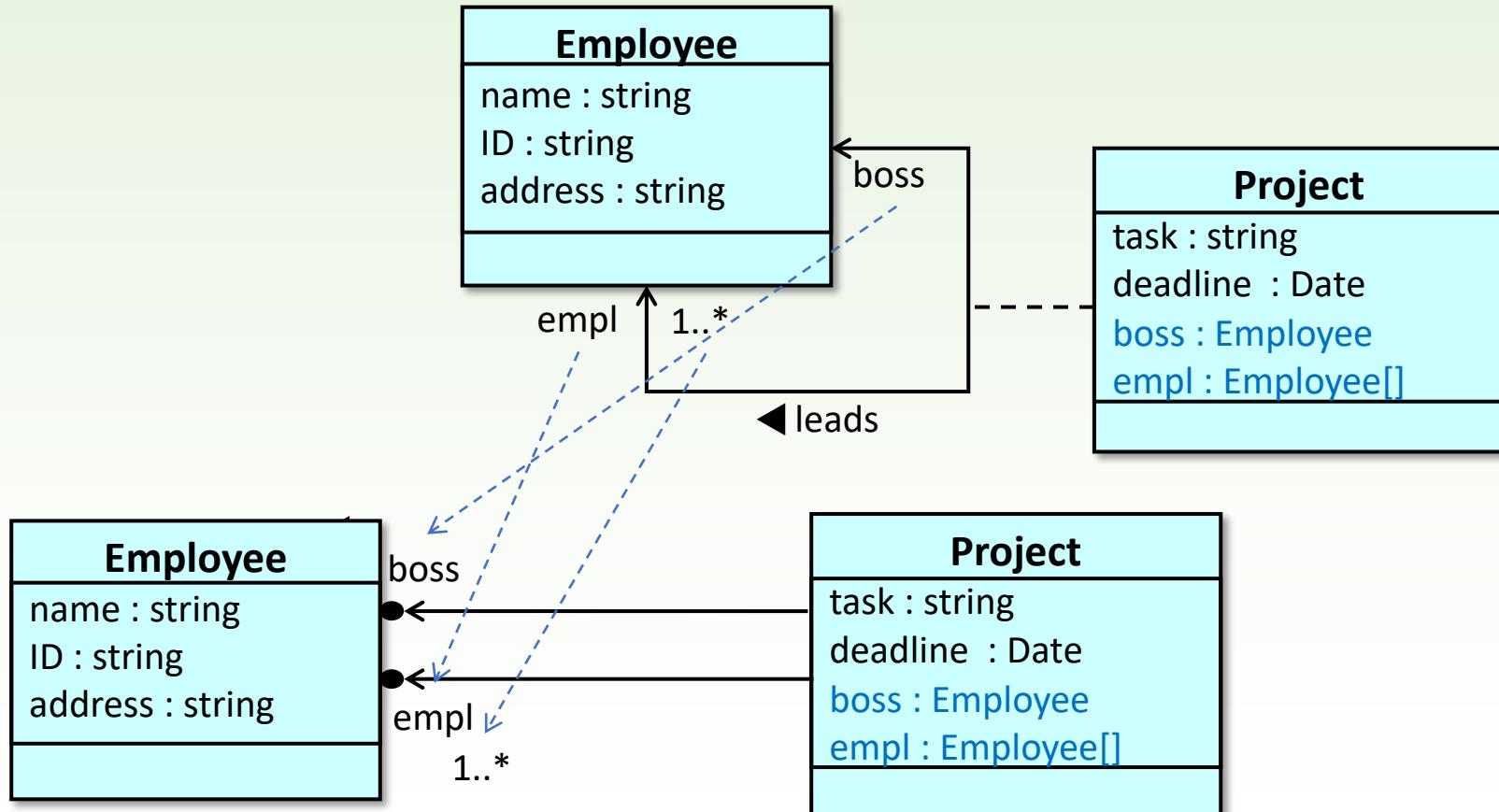
Association class

- ❑ In UML, it is possible to define a class to describe a relationship. Instances are relationships which are accessible by the connected objects, thus objects reach the information in it.
- ❑ When a role name is owned by the association, the role name is an attribute of the association class.
- ❑ Leading OO languages do not support it.



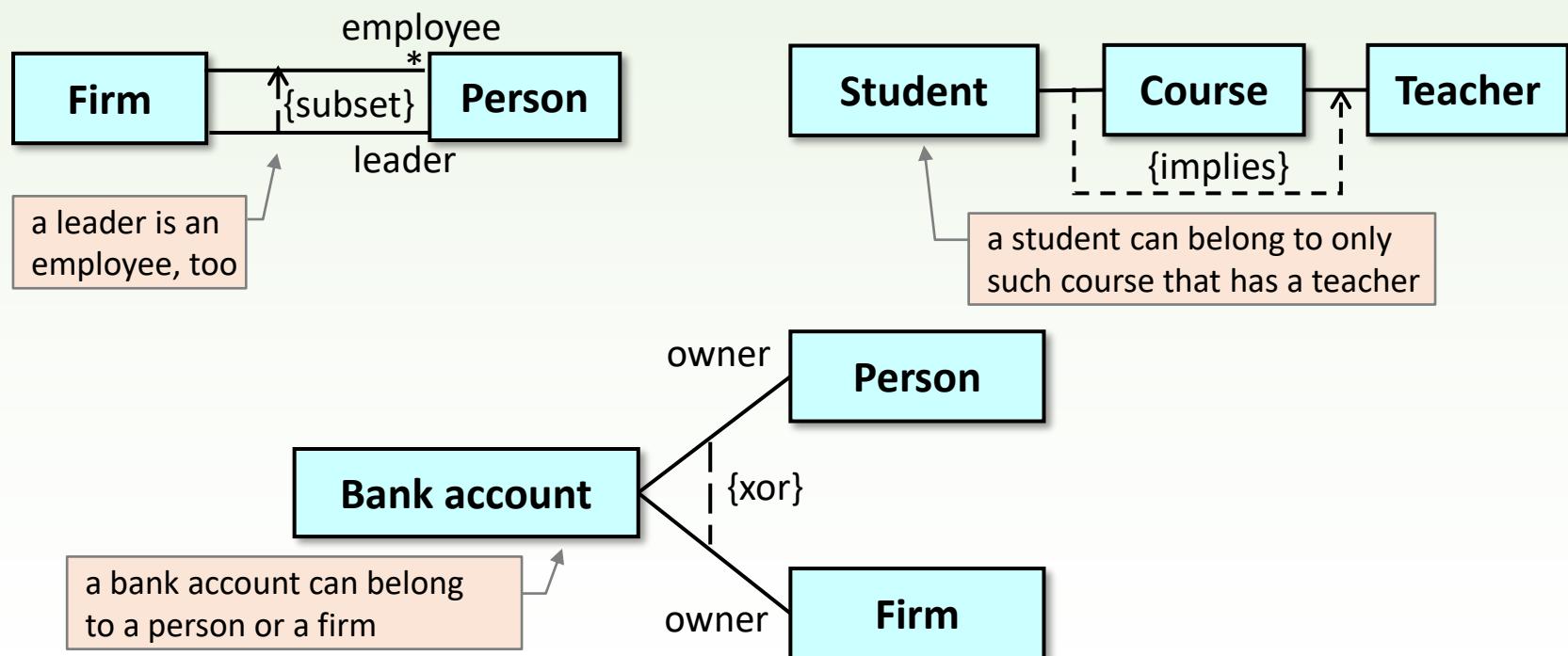
Elimination of an association class

- Replace the association class with a standard class.

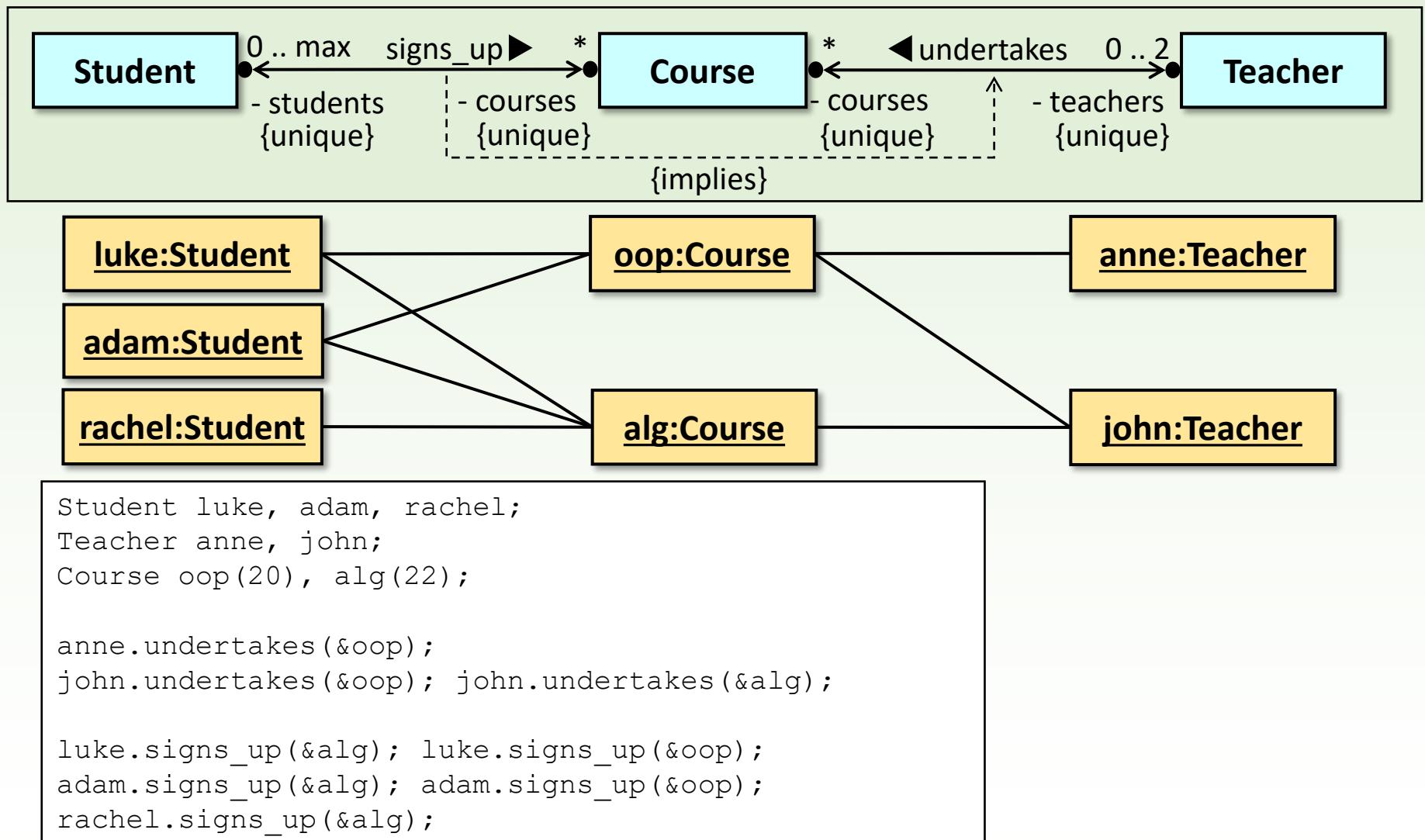


Relationship of associations

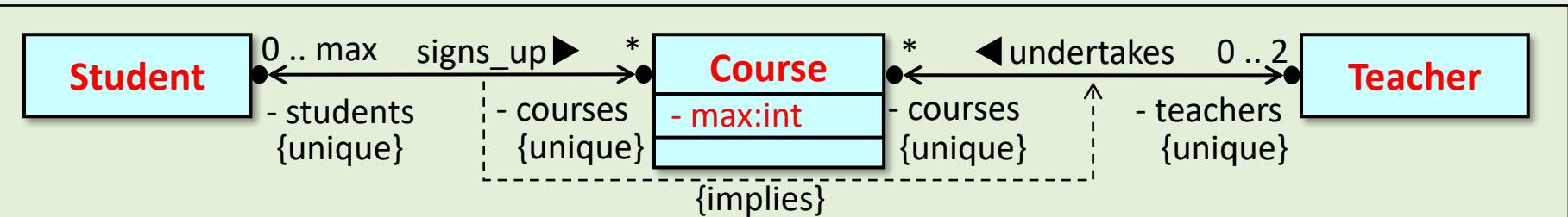
- Logical conditions (subset, and, or, xor, implies, ...) might be given, which restrict the associations of an object.



Example



Classes



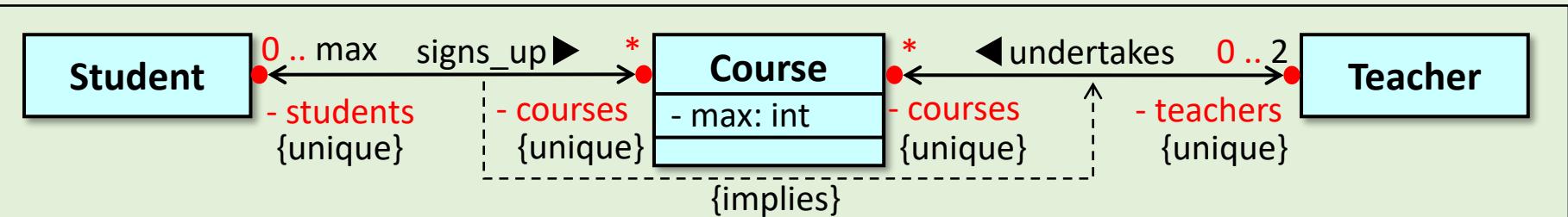
```
class Course {
private:
    unsigned int _max;
    ...

public:
    Course(unsigned int max) : _max(max) {}
};
```

```
class Student {
private:
    ...
public:
    ...
};
```

```
class Teacher {
private:
    ...
public:
    ...
};
```

Classes

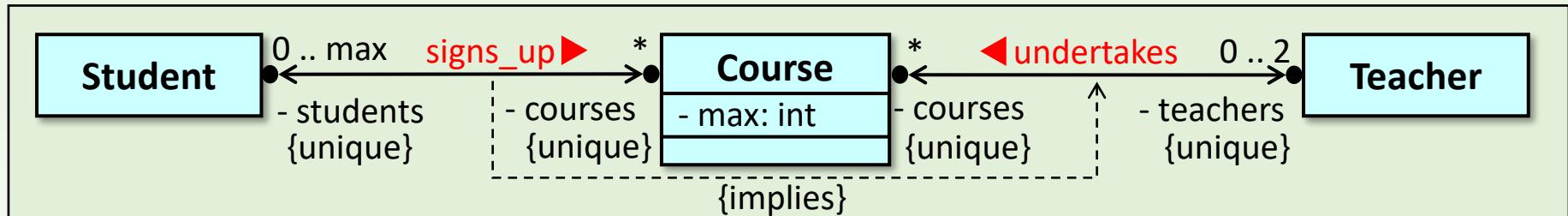


```
class Course {
private:
    unsigned int _max;
    std::vector<Teacher*> _teachers; // 0 .. 2
    std::vector<Student*> _students; // 0 .. max
public:
    Course(unsigned int max) : _max(max) {}
};
```

```
class Student {
private:
    std::vector<Course*> _courses;
public:
    ...
};
```

```
class Teacher {
private:
    std::vector<Course*> _courses;
public:
    ...
};
```

Classes



```
class Course {  
private:  
    unsigned int _max;  
    std::vector<Teacher*> _teachers; // 0 .. 2  
    std::vector<Student*> _students; // 0 .. max  
public:  
    Course(unsigned int max) : _max(max) {}  
};
```

```
class Student {  
private:  
    std::vector<Course*> _courses;  
public:  
    void signs_up(Course* pc);  
};
```

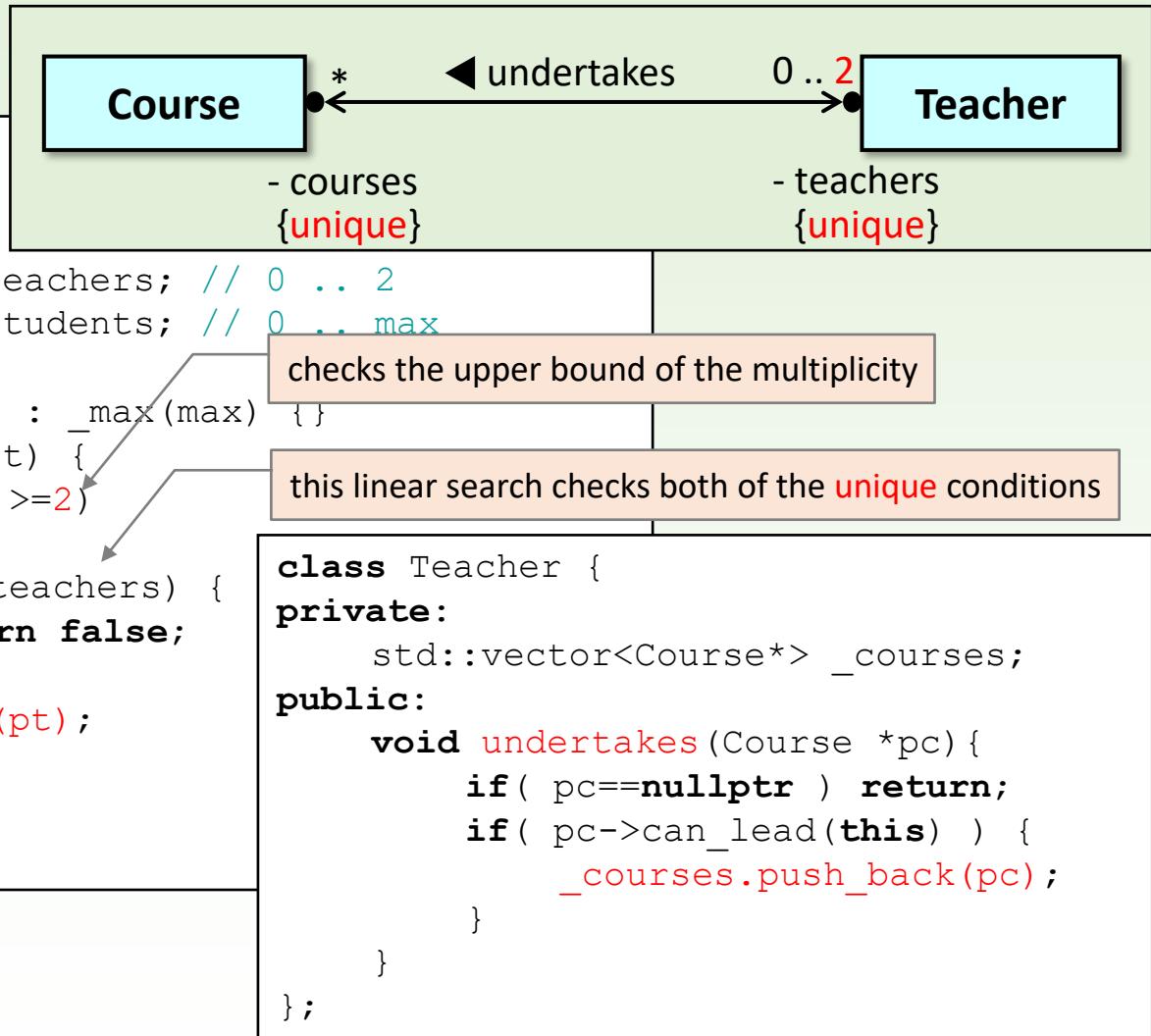
```
class Teacher {  
private:  
    std::vector<Course*> _courses;  
public:  
    void undertakes(Course* pc);  
};
```

Methods

```

class Course {
private:
    unsigned int _max;
    std::vector<Teacher*> _teachers; // 0 .. 2
    std::vector<Student*> _students; // 0 .. max
public:
    Course(unsigned int max) : _max(max) {}
    bool can_lead(Teacher *pt) {
        if (_teachers.size() >= 2)
            return false;
        for (Teacher *p : _teachers) {
            if (p==pt) return false;
        }
        _teachers.push_back(pt);
        return true;
    }
};

```

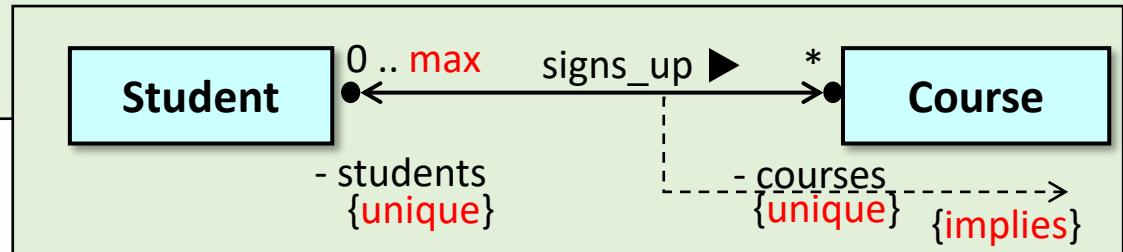


Methods

```

class Course {
private:
    unsigned int _max;
    std::vector<Teacher*> _teachers; // 0 .. 2
    std::vector<Student*> _students; // 0 .. max
public:
    Course(unsigned int max) : _max(max) {}
    bool can_lead(Teacher *pt) { ... }
    bool has_teacher() const { return _teachers.size() > 0; }
    bool can_sign_up(Student *ps) {
        if (_students.size() >= _max)
            return false;
        for (Student *p : _students)
            if (p == ps) return false;
        _students.push_back(ps);
        return true;
    }
};

```



checks the upper bound of the multiplicity

checks the implies condition

this linear search checks both of the unique conditions

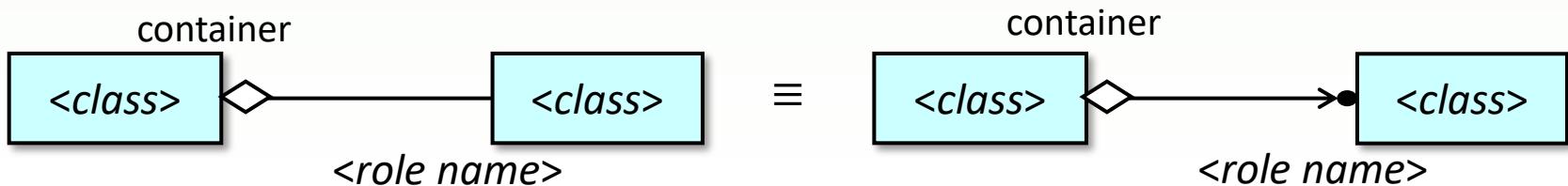
```

class Student {
private:
    std::vector<Course*> _courses;
public:
    void signs_up(Course *pc) {
        if (pc == nullptr
            || !pc->has_teacher())
            return;
        if (pc->can_sign_up(this))
            _courses.push_back(pc);
    }
};

```

Aggregation

- ❑ It is a binary association expressing that an object **contains** or **owns** another object:
 - It is **asymmetric**, **non-reflexive**, **transitive**, and cannot contain loop neither indirectly (an object cannot contain itself).
 - An object **can be part of many other objects** – even simultaneously, too –, and if the container is destroyed, the contained object can live on.
- ❑ We agree that for lack of notation,
 - the relationship is navigated in the direction of the contained class,
 - role name of the contained class belongs to the container.

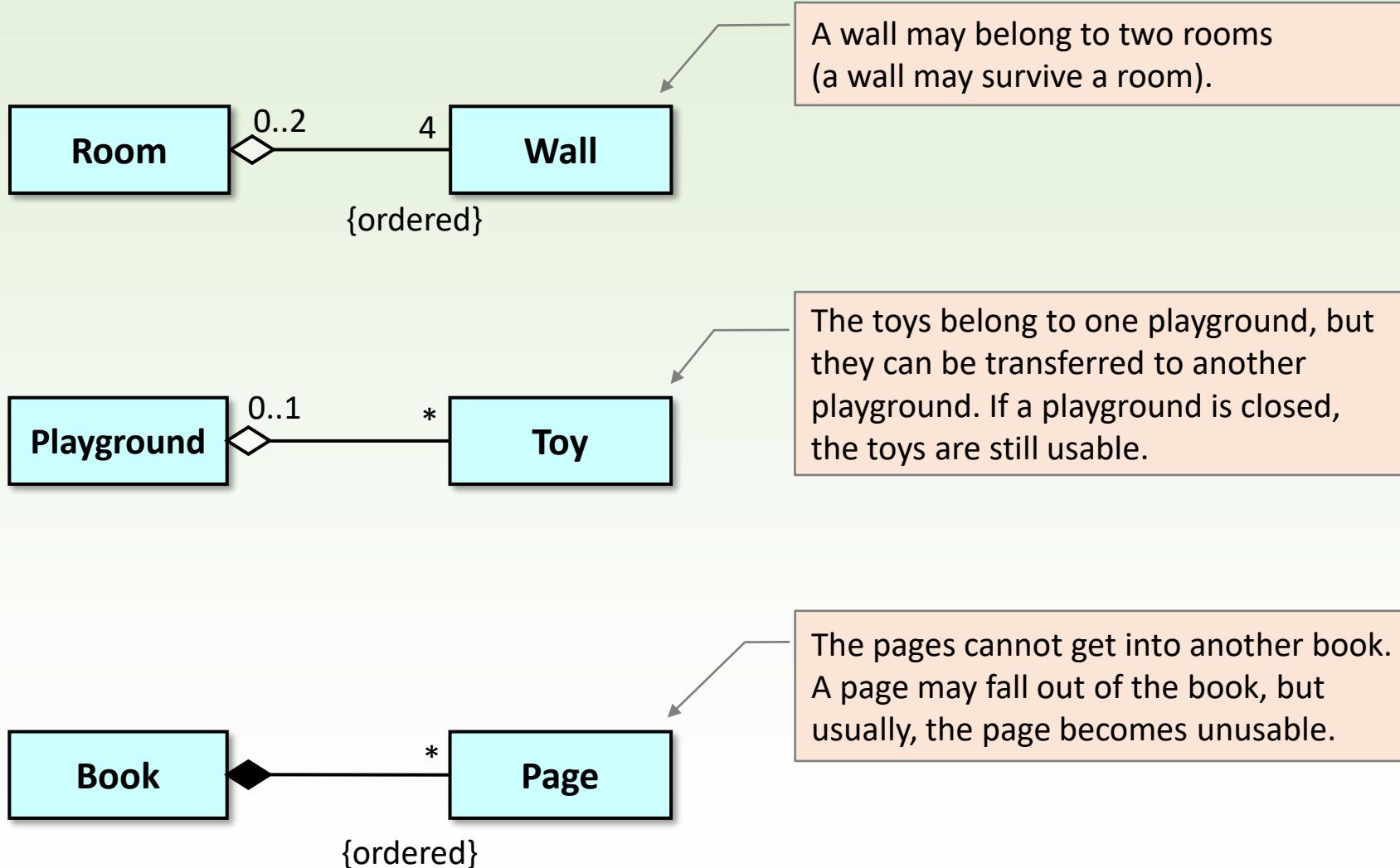


Composition

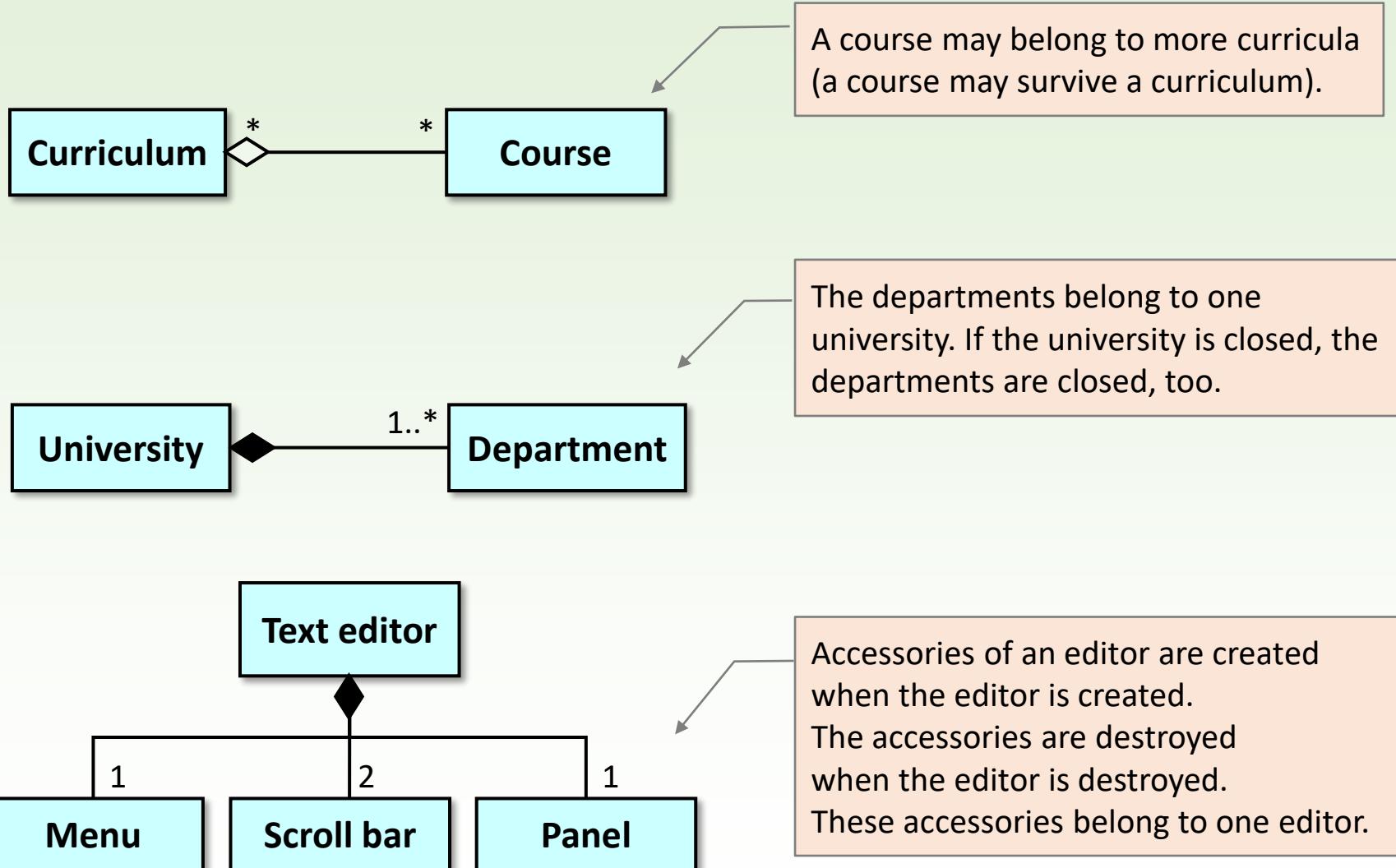
- ❑ Composition is a special aggregation where the contained objects **can belong to only one container**, and the contained objects cannot exist by themselves.
- ❑ Several explanations are known, e.g.: the contained object
 - **is always part of a container**,
 - **cannot change its owner**, and
 - **can only be created and destroyed by the container**: usually its constructor instantiates it and its destructor destroys it.



Examples

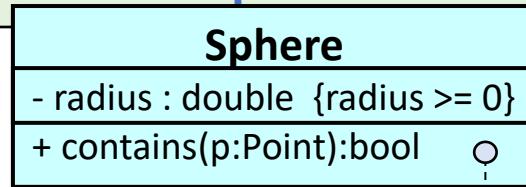


Examples

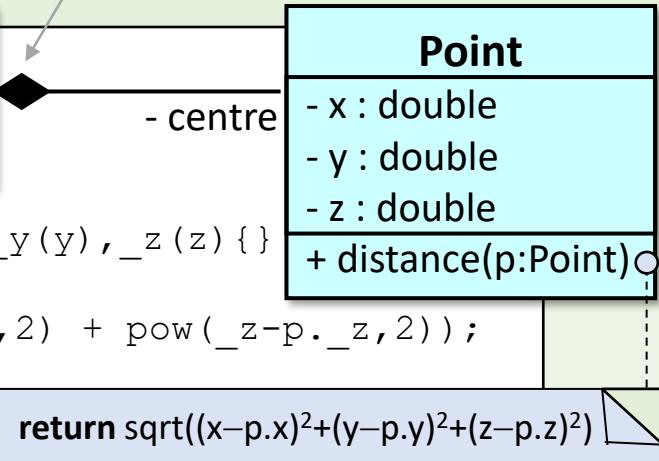


Ex.: point in a sphere

```
class Point {
private:
    double _x, _y, _z;
public:
    Point(double x, double y, double z) : _x(x), _y(y), _z(z) {}
    double distance(const Point &p) const {
        return sqrt(pow(_x-p._x,2) + pow(_y-p._y,2) + pow(_z-p._z,2));
    }
};
```



The centre belongs to the sphere, it is created and destroyed together with the sphere.



```
class Sphere{
private:
    Point _centre;
    double _radius;
public:
    enum Errors{ILLEGAL_RADIUS};
    Sphere(const Point &c, double r): _centre(c), _radius(r) {
        if (_radius<0.0) throw ILLEGAL_RADIUS;
    }
    double contains(const Point &p) const {
        return _centre.distance(p) <= _radius;
    }
};
```

composition

Copies Point c for the automatically created centre: in this case, the centre and the point in c are two different objects.

```

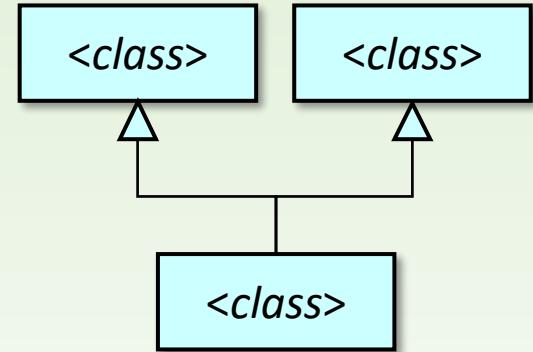
Point p(-12.0, 0.0, 23.0);
Point c(-12.3, 0.0, 23.4);
Sphere s1(c, 1.0);
cout << s1.contains(p) << endl;
cout << c.distance(p) << endl;
  
```

How to implement a relationship

- ❑ In case of **association**, one of the objects creates the relationship by one of its methods (named after the association's name or by a constructor). The method gets the other objects' reference as parameters or instantiates them.
- ❑ In case of **aggregation**, the container creates the relationship.
- ❑ In case of **composition**, the container's constructor creates the relationship.
 - If the contained element can be replaced, it is done by a method.
 - In stricter explanations, the constructor instantiates and the destructor destroys the contained element.

Inheritance

- If an object is similar to another, (it has the **same attributes and methods**), then its class may be inherited from the classes of the similar objects: inherits their properties, and it might modify and augment them.



- In modeling, there are two reasons for inheritance:
 - **Generalization**: to create a **base class** (superclass) to describe common properties of similar, already existing classes.
 - **Specialization**: to create a **subclass** by inheritance.

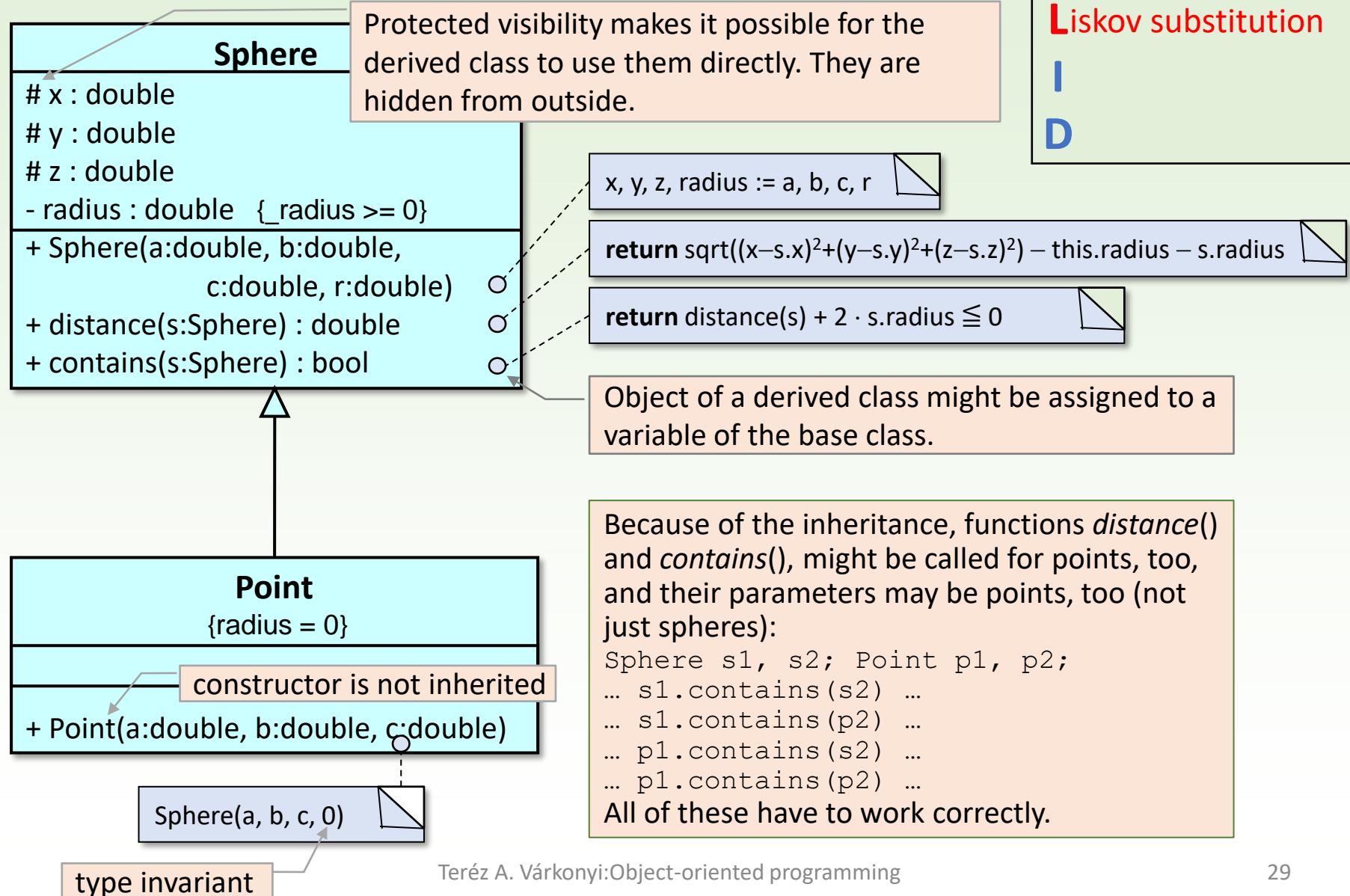
5

A well-known criterion of object-orientation is **inheritance**: classes may be derived from already existing classes.
Object of a subclass might be assigned to a variable of the base class.

Inheritance and visibility

- ❑ In the child class, public and protected attributes of the parent might be referred. Private attributes cannot be accessed, but indirectly, through inherited methods they might be modified.
- ❑ Inheritance may be public, protected, or private.
 - **Public**, if the visibilities of the protected and public attributes are inherited as they are defined in the base class. (In UML this is default, in language C++, not.)
 - **Protected**, if the public and protected members become protected in the child.
 - **Private**, if the public and protected members become private in the child.

Ex.: point from sphere



Single responsibility
O
Liskov substitution
I
D

C++ : point from sphere

```
class Sphere{
protected:
    double _x, _y, _z;
private:
    double _radius;
public:
    enum Errors{ILLEGAL_RADIUS};
    Sphere(double a, double b, double c, double r):
        _x(a),_y(b),_z(c),_radius(r) { if(_radius<0.0) throw ILLEGAL_RADIUS; }
    double distance(const Sphere& s) const
    { return sqrt(pow((_x-s._x),2) + pow((_y-s._y),2) + pow((_z-s._z),2))
        - _radius - s._radius; }
    bool contains(const Sphere& s) const
    { return distance(s) + 2 * s._radius <= 0; }
};
```

public inheritance

```
class Point : public Sphere {
public:
    Point(double a, double b, double c) : Sphere(a, b, c, 0.0) {}
```

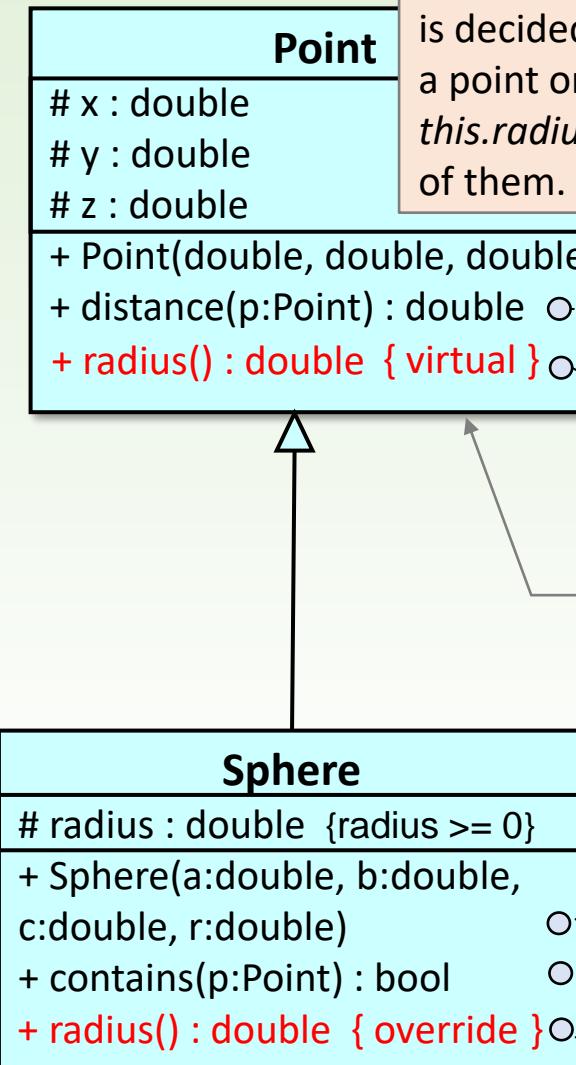
```
Point p(0,0,0);
Sphere s(1,1,1,1);

cout << s.contains(p) << endl;
cout << s.distance(p) << endl;
cout << s.contains(s) << endl;
cout << s.distance(s) << endl;
cout << p.contains(s) << endl;
cout << p.distance(s) << endl;
cout << p.contains(p) << endl;
cout << p.distance(p) << endl;
```

Sphere

Point
{radius = 0}

Example: sphere from point



These calls cannot be expounded obviously, as it is decided **runtime** if variables *p* and *this* refer to a point or a sphere.

this.radius() and *p.radius()* depend on the type of them.

Single responsibility

Liskov substitution

Interface segregation

Dependency inversion

return $\sqrt{(x-p.x)^2+(y-p.y)^2+(z-p.z)^2}$ – ~~radius()~~ – ~~p.radius()~~

return $\sqrt{(x-p.x)^2+(y-p.y)^2+(z-p.z)^2}$

return 0.0

If it is called on spheres, it does not give correct result, it violates the Liskov-principle.

In OO languages, if *radius()* is virtual, then when it is called on a pointer or a reference variable (e.g. variable *this*), that version is run where the pointer or reference points at.

Point(a,b,c); radius := r

return $distance(p) + 2 \cdot p.radius() \leq 0$

return $distance(p) \leq radius$

return *radius*

C++ : sphere from point

```

class Point {
protected:
    double _x, _y ,_z;
public:
    Point(double a, double b, double c) : _x(a), _y(b), _z(c) {}
    double distance(const Point &p) const
    { return sqrt(pow(( _x-p._x),2)+pow(( _y-p._y),2)+pow(( _z-p._z),2))
      - radius() - p.radius(); }
    virtual double radius() const { return 0.0; }
};

```

```

class Sphere : public Point {
private:
    double _radius;
public:
    enum Errors{ILLEGAL_RADIUS};
    Sphere(double a, double b, double c, double r) : Point(a,b,c), _radius(r)
    { if(_radius<0.0) throw ILLEGAL_RADIUS; }
    bool contains(const Point &p) const { return distance(p)+2*p.radius()<=0; }
    double radius() const override { return _radius; }
};

```

```

Point p(0,0,0);
Sphere g(1,1,1,1);

cout << p.distance(p) << endl;
cout << g.distance(p) << endl;
cout << p.distance(g) << endl;
cout << g.distance(g) << endl;

cout << g.contains(p) << endl;
cout << g.contains(g) << endl;

```

Polymorphism and dynamic binding

- ❑ If a method of the parent is overridden in a child (**override**), that method has several "shapes" (it is **polymorphic**).
- ❑ An instance of a child might be assigned to a variable of the parent, so it has to be **decided runtime** if the variable refers to an instance of the parent or the child (late or **dynamic binding**).
- ❑ If a polymorphic virtual method of the parent is called on a pointer or reference variable, then the type of the object (child or parent) determines which version of that method is run (**runtime polymorphism**).

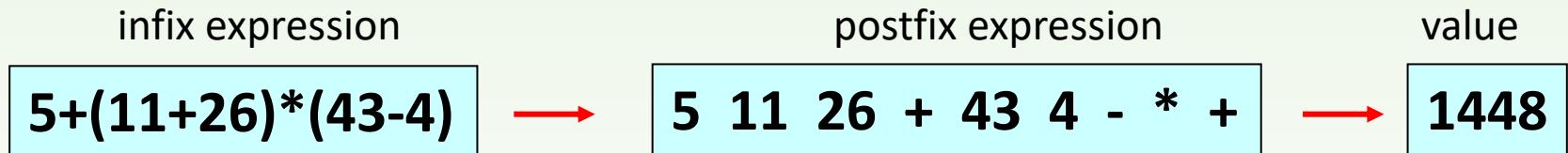
6

Typical criterion of object-orientation is **runtime (subtype) polymorphism** together with **dynamic binding**.

Inheritance and data structures

Task

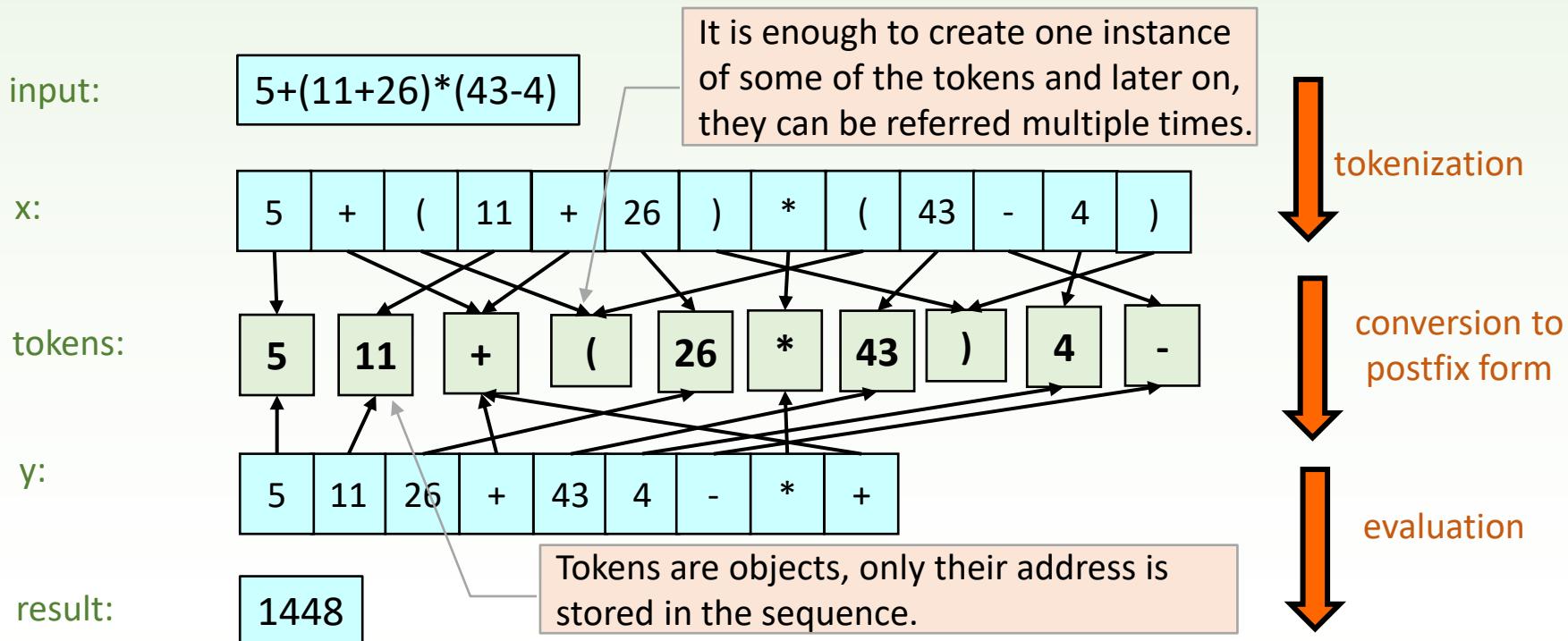
Transform an **infix** expression to a **postfix** expression (Reverse Polish Notation - RPN) and calculate its **value**.



To transform an infix expression and to evaluate it, usually two **stacks** are needed. In the first one, the operators and the open parentheses are stored. In the second one, the operands and the partial results are put.

Plan

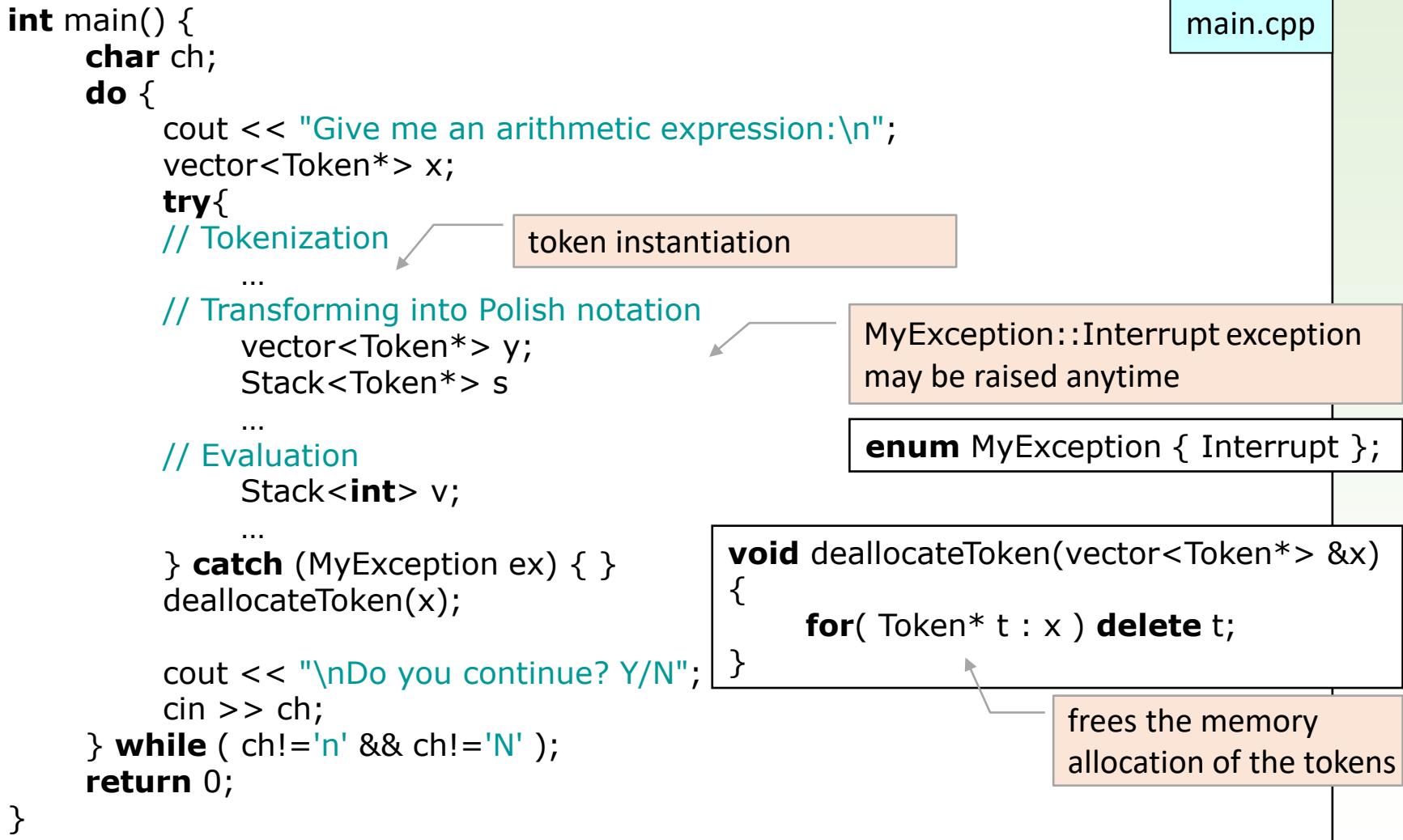
1. **Tokenize** the infix expression and put the tokens into an x sequence.
2. **Convert to Polish notation**: x is converted into a y sequence in which the tokens are in postfix form. For the conversion, a stack is used.
3. **Evaluate** the y sequence by using a second stack.



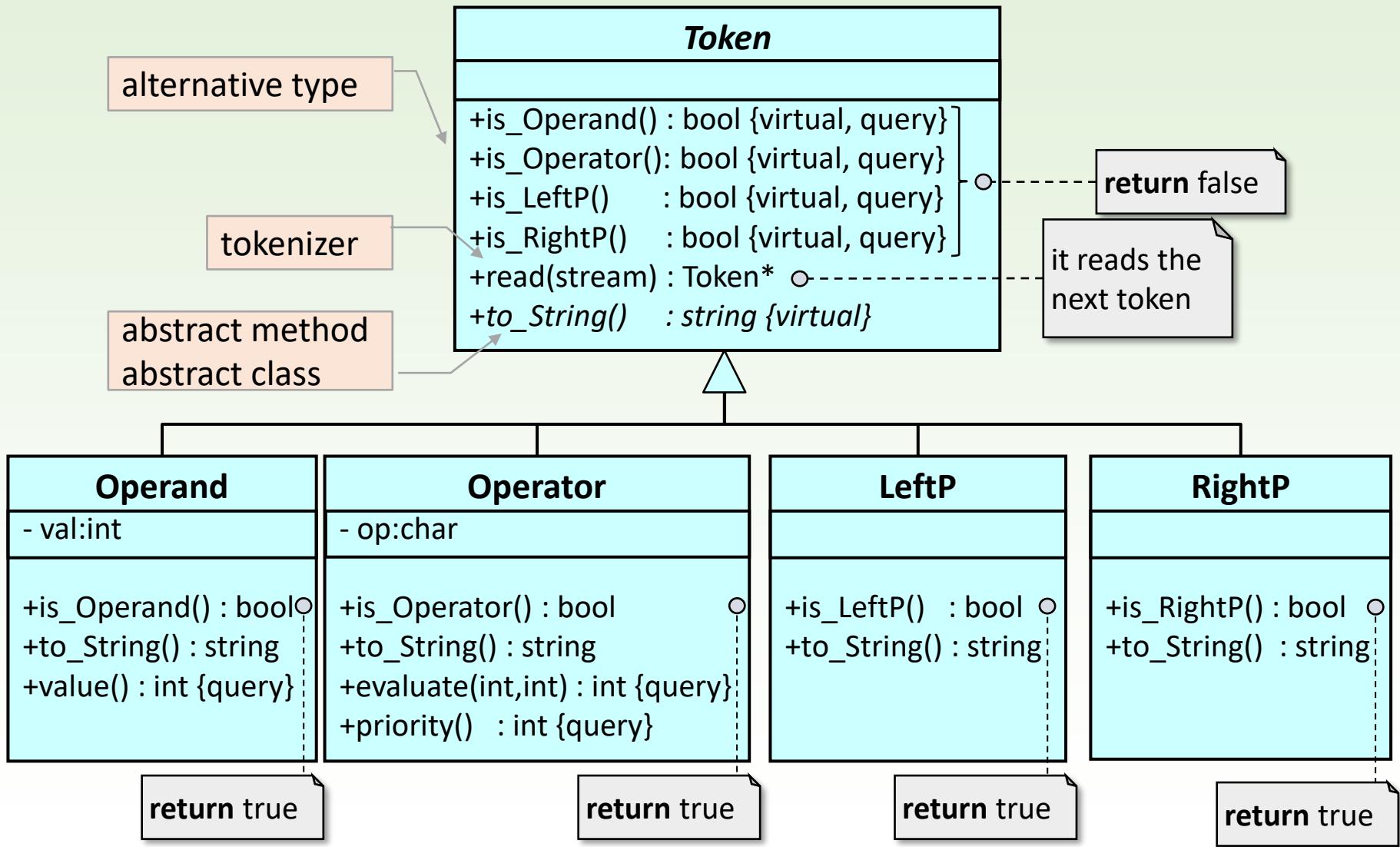
Objects to be used

- string:** an infix expression given in a standard input (`fstream`)
- tokens:** specific tokens (`Token`), as operands (`Operand`), operators (`Operator`), and parentheses (`LeftP`, `RightP`)
- sequences:** pointers pointing at the tokens (`vector<Token*>`):
 - for the tokenized infix expression (`x`),
 - for the tokenized postfix expression (`y`)
- stacks:** for storing the pointers of the tokens (`Stack<Token*>`),
for storing the numbers (`Stack<int>`)

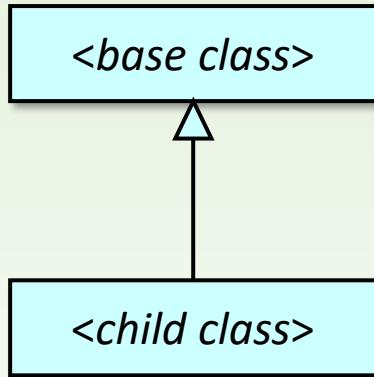
Main program



Token class and its children

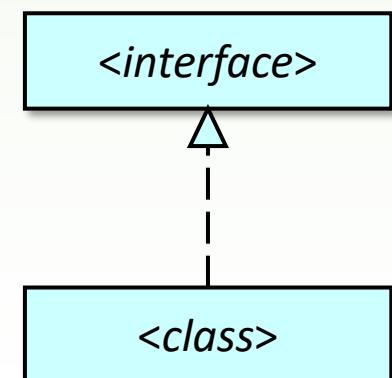


Abstract class, interface



- ❑ **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**.



Token class

```
class Token
{
public:
    class IllegalElementException{
private:
    char _ch;
public:
    IllegalElementException(char c) : _ch(c){}
    char message() const { return _ch; }
};

virtual ~Token();
virtual bool is_LeftP();
virtual bool is_RightP();
virtual bool is_Operand();
virtual bool is_Operator();
virtual bool is_End();

virtual std::string to_String() const = 0;

friend
std::istream& operator>>(std::istream&, Token*&);

};
```

for exception handling

Why is the destructor virtual?

not presented in the specification

token.h

Operand class

```
class Operand: public Token
{
private:
    int _val;
public:
    Operand(int v) : _val(v) {}

    bool is_Operand() const override { return true; }

    std::string to_String() const override {
        std::ostringstream ss;
        ss << _val;
        return ss.str();
    }

    int value() const { return _val; }
};
```

conversion

token.h

LeftP class (one instance is enough)

```
class LeftP : public Token
{
private:
    LeftP(){}
    static LeftP *_instance;
public:
    static LeftP *instance() {
        if (_instance == nullptr) _instance = new LeftP();
        return _instance;
    }
    bool is_LeftP()           const override { return true; }
    std::string to_String()   const override { return "("; }
};
```

private constructor
points at the only one instance
creator method
private constructor is called here

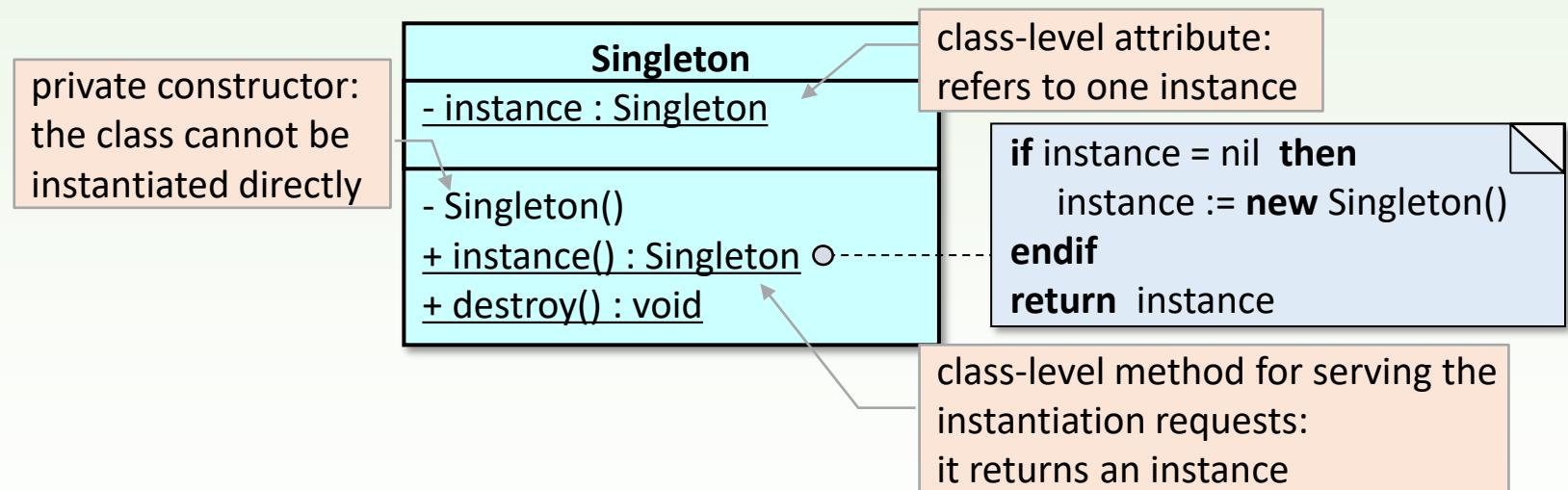
token.h

```
LeftP *_LeftP::_instance = nullptr;
```

token.cpp

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.

RightP, End singleton classes

```
class RightP : public Token {  
private:  
    RightP(){}  
    static RightP *_instance;  
public:  
    static RightP *instance() {  
        if (_instance == nullptr) _instance = new RightP();  
        return _instance;  
    }  
    bool is_ RightP()  
    std::string to_String()  
};
```

```
RightP *RightP::_instance = nullptr;  
End     *End::_instance = nullptr;
```

token.cpp

```
class End : public Token {  
private:  
    End(){}  
    static End *_instance;  
public:  
    static End *instance() {  
        if (_instance == nullptr) _instance = new End();  
        return _instance;  
    }  
    bool is_ End()  
    std::string to_String()  
};
```

```
const override { return true; }  
const override { return ")" };
```

token.h

Operator class

```
class Operator: public Token
{
private:
    char _op;
public:
    Operator(char o) : _op(o) {}

    bool is_Operator() const override { return true; }
    std::string to_String() const override {
        string ret;
        ret = _op;
        return ret;
    }
    virtual int evaluate(int leftValue, int rightValue) const;
    virtual int priority() const;
};
```

conversion

token.h

Methods of class Operator

```
int Operator::evaluate(int leftValue, int rightValue) const
{
    switch(_op){
        case '+': return leftValue+rightValue;
        case '-': return leftValue-rightValue;
        case '*': return leftValue*rightValue;
        case '/': return leftValue/rightValue;
        default: return 0;
    }
}

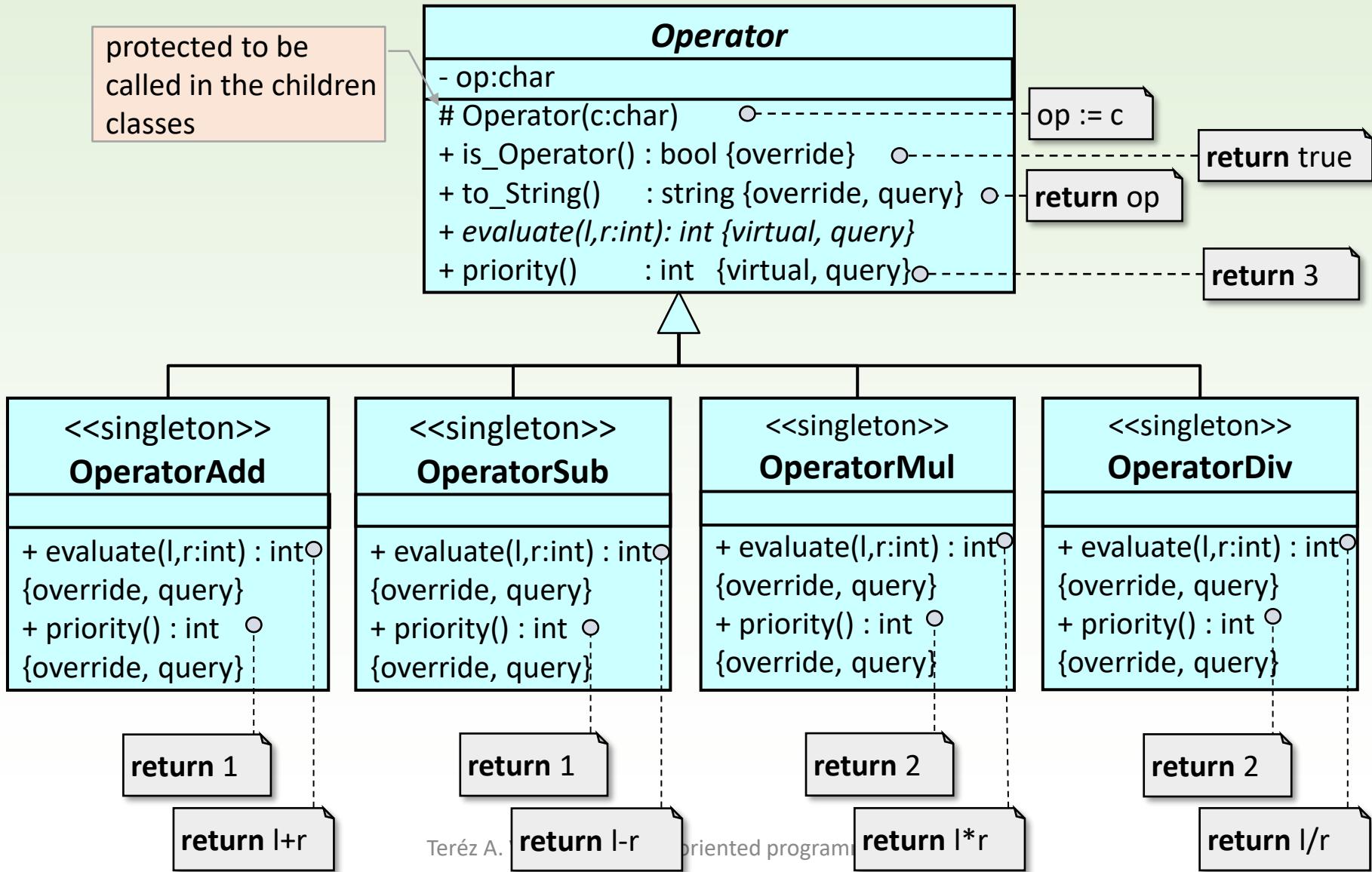
int Operator::priority() const
{
    switch(_op){
        case '+': case '-': return 1;
        case '*': case '/': return 2;
        default: return 3;
    }
}
```

token.cpp

Single responsibility
Open-Close
Liskov substitution
Interface segregation
Dependency inversion

this code does not
satisfy the open-close
principle

Operator classes



Abstract Operator class

```
class Operator: public Token
{
private:
    char _op;
protected:
    Operator(char o) : _op(o) {}
public:
    bool is_Operator()    const override { return true; }
    std::string to_String() const override {
        string ret;
        ret = _op;
        return ret;
    }
    virtual int evaluate(int leftValue, int rightValue) const = 0;
    virtual int priority() const { return 3; }
};
```

token.h

Singleton operator classes

```
class OperatorAdd: public Operator
{
private:
public:
public:
};

class OperatorSub: public Operator
{
private:
public:
public:
};

class OperatorMul: public Operator
{
private:
public:
public:
};

class OperatorDiv: public Operator
{
private:
public:
public:
};

int evaluate(int leftValue, int rightValue) const override {
    return leftValue / rightValue;
}

int priority() const override { return 2; }

OperatorAdd* OperatorAdd::_add = nullptr;
OperatorSub* OperatorAdd::_sub = nullptr;
OperatorMul* OperatorAdd::_mul = nullptr;
OperatorDiv* OperatorAdd::_div = nullptr;
```

token.h

No need for conditionals

token.cpp

Tokenizer operator

```
istream& operator>> (istream &s, Token* &t){  
    char ch;  
    s >> ch;  
    switch(ch){  
        case '0' : case '1' : case '2' : case '3' : case '4':  
        case '5' : case '6' : case '7' : case '8' : case '9':  
            s.putback(ch);  
            int intval;  
            s >> intval;    back to the a buffer  
            t = new Operand(intval); break;  
        case '+' : t = OperatorAdd::instance(); break;  
        case '-' : t = OperatorSub::instance(); break;  
        case '*' : t = OperatorMul::instance(); break;  
        case '/' : t = OperatorDiv::instance(); break;  
        case '(' : t = LeftP::instance(); break;  
        case ')' : t = RightP::instance(); break;  
        case ';' : t = End::instance(); break;  
        default: if(!s.fail()) throw new Token::IllegalElementException(ch);  
    }  
    return s;  
}
```

singletons

an expression is ended by a semicolon

token.cpp

Tokenization

```
// Tokenization

try{
    Token *t;
    cin >> t;
    while( !t->is_End() ){
        x.push_back(t);
        cin >> t;
    }
} catch(Token::IllegalElementException *ex){
    cout << "Illegal character: " << ex->message() << endl;
    delete ex;
    throw Interrupt;
}
```

token reading

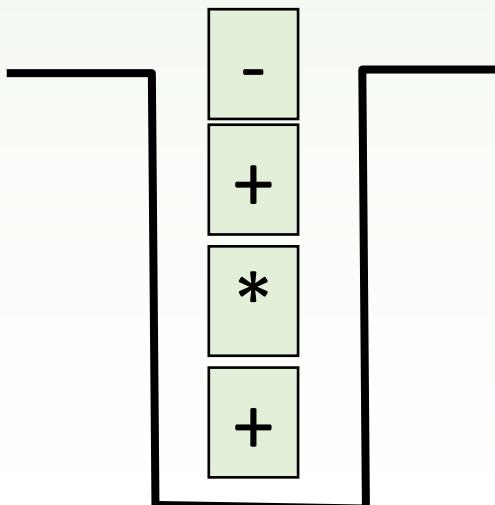
the reading throws it

main.cpp

Convert to Polish notation

Left parentheses and operators are put into a stack. The operator with lower priority has to swap with the higher priority operators in the stack. Every other token is copied into the output sequence. In case of a right parenthesis, the content of the stack until the first left parenthesis is put into the output sequence. When we reach the end of the input sequence, the content of the stack is put into the output sequence.

5 + (11 + 26) * (43 - 4)



$x.\text{first}()$; $y := <>$

$\neg x.\text{end}()$

$t := x.\text{current}()$

$t.\text{is_Operand}()$	$t.\text{is_LeftP}()$	$t.\text{is_RightP}()$	$t.\text{is_Operator}()$
$y.\text{push_back}(t)$	$s.\text{push}(t)$	$\neg s.\text{top}().\text{is_LeftP}()$ $y.\text{push_back}(s.\text{top}())$ $s.\text{pop}()$	$\neg s.\text{empty}() \wedge$ $\neg s.\text{top}().\text{is_LeftP}() \wedge$ $s.\text{top}().\text{priority}() \geq t.\text{priority}()$ $y.\text{push_back}(s.\text{top}())$ $s.\text{pop}()$
		$s.\text{pop}()$	$s.\text{push}(t)$
$x.\text{next}()$			

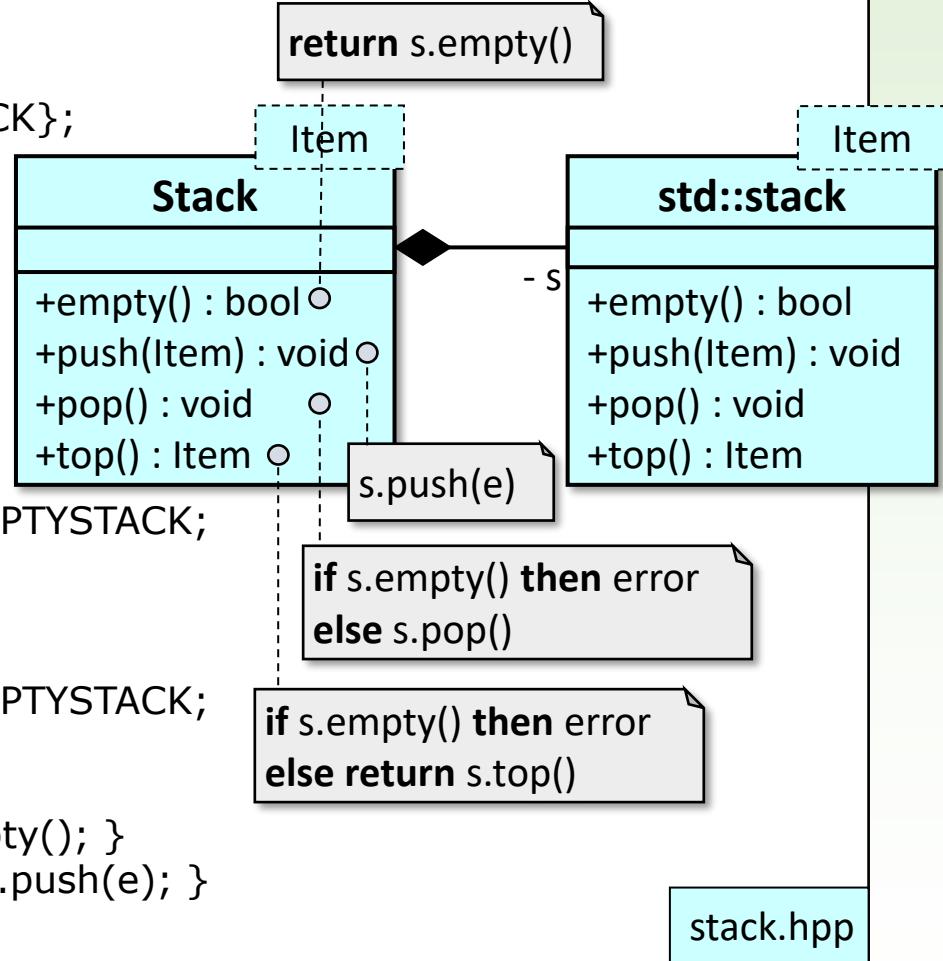
$\neg s.\text{empty}()$

$y.\text{push_back}(s.\text{top}()) ; s.\text{pop}()$

Template for the stack

```
#include <stack>
```

```
enum StackExceptions{EMPTYSTACK};  
  
template <typename Item>  
class Stack  
{  
private:  
    std::stack<Item> s;  
public:  
    void pop() {  
        if( s.empty() ) throw EMPTYSTACK;  
        s.pop();  
    }  
    Item top() const {  
        if( s.empty() ) throw EMPTYSTACK;  
        return s.top();  
    }  
    bool empty() { return s.empty(); }  
    void push(const Item& e) { s.push(e); }  
};
```



Creating the postfix expression

```
// Transforming into polish form

vector<Token*> y;
Stack<Token*> s;

for( Token *t : x ){
    if      ( ...
    else if ( ...
    else if ( ...
    else if ( ...
    else if ( ...
}
while( !s.empty() ){
    if( s.top()->is_LeftP() ){
        cout << "Syntax error!\n";
        throw Interrupt;
    else{
        y.push_back(s.top());
        s.pop();
    }
}

}
```

enumeration

see the next slide

error when we have more left parentheses than right

main.cpp

Creating the postfix expression

```
if      ( t->is_Operand() ) y.push_back(t);
else if ( t->is_LeftP() )    s.push(t);
else if ( t->is_RightP() ){
    try{
        while( !s.top()->is_LeftP() ) {
            y.push_back(s.top());
            s.pop();
        }
        s.pop();
    }catch(StackExceptions ex){
        if(ex==EMPTYSTACK){
            cout << "Syntax error!\n";
            throw Interrupt;
        }
    }
}else if ( t->is_Operator() ) {
    while( !s.empty() && s.top()->is_Operator() &&
           ((Operator*)s.top())->priority()>=((Operator*)t)->priority() ) {
        y.push_back(s.top());
        s.pop();
    }
    s.push(t);
}
```

error when we have more right parentheses than left

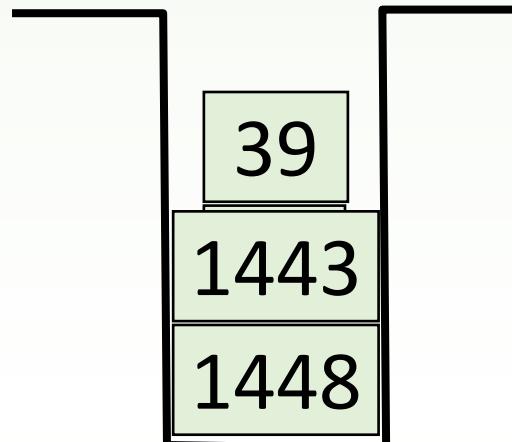
"static casting": s.top()->priority() is not good, as in Token there is no priority().

main.cpp

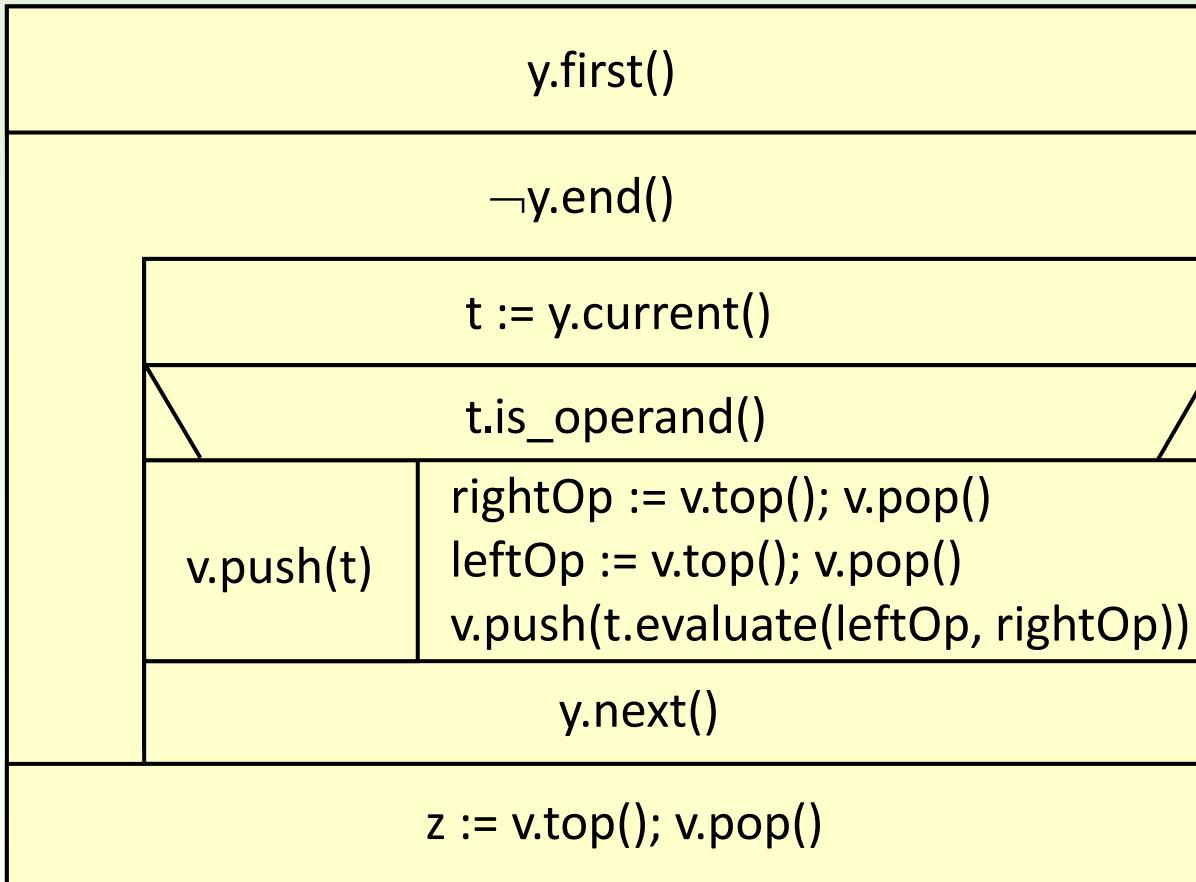
Evaluation of the postfix expression

Operands of the postfix expression (in order of reading) are put into a stack. In case of operator, the top two numbers are taken and processed according to the type of the operator. The result is put back into the stack. At the end of the process, the result can be found in the stack.

5	11	26	+	43	4	-	*	+
---	----	----	---	----	---	---	---	---



Evaluation



Evaluation

```
// Evaluation
try{
    Stack<int> v;
    for( Token *t : y ){
        if ( t->is_Operand() ) {
            v.push( ((Operand*)t)->value() );
        } else{
            int rightOp = v.top(); v.pop();
            int leftOp  = v.top(); v.pop();
            v.push(((Operator*)t)->evaluate(leftOp, rightOp));
        }
    }
    int result = v.top(); v.pop();
    if( !v.empty() ){
        cout << "Syntax error!";
        throw Interrupt;
    }
    cout << "The value of the expression: " << result << endl;
} catch( StackExceptions ex ){
    if( ex==EMPTYSTACK ){
        cout << "Syntax error! ";
        throw Interrupt;
    }
}
```

enumeration

static casting

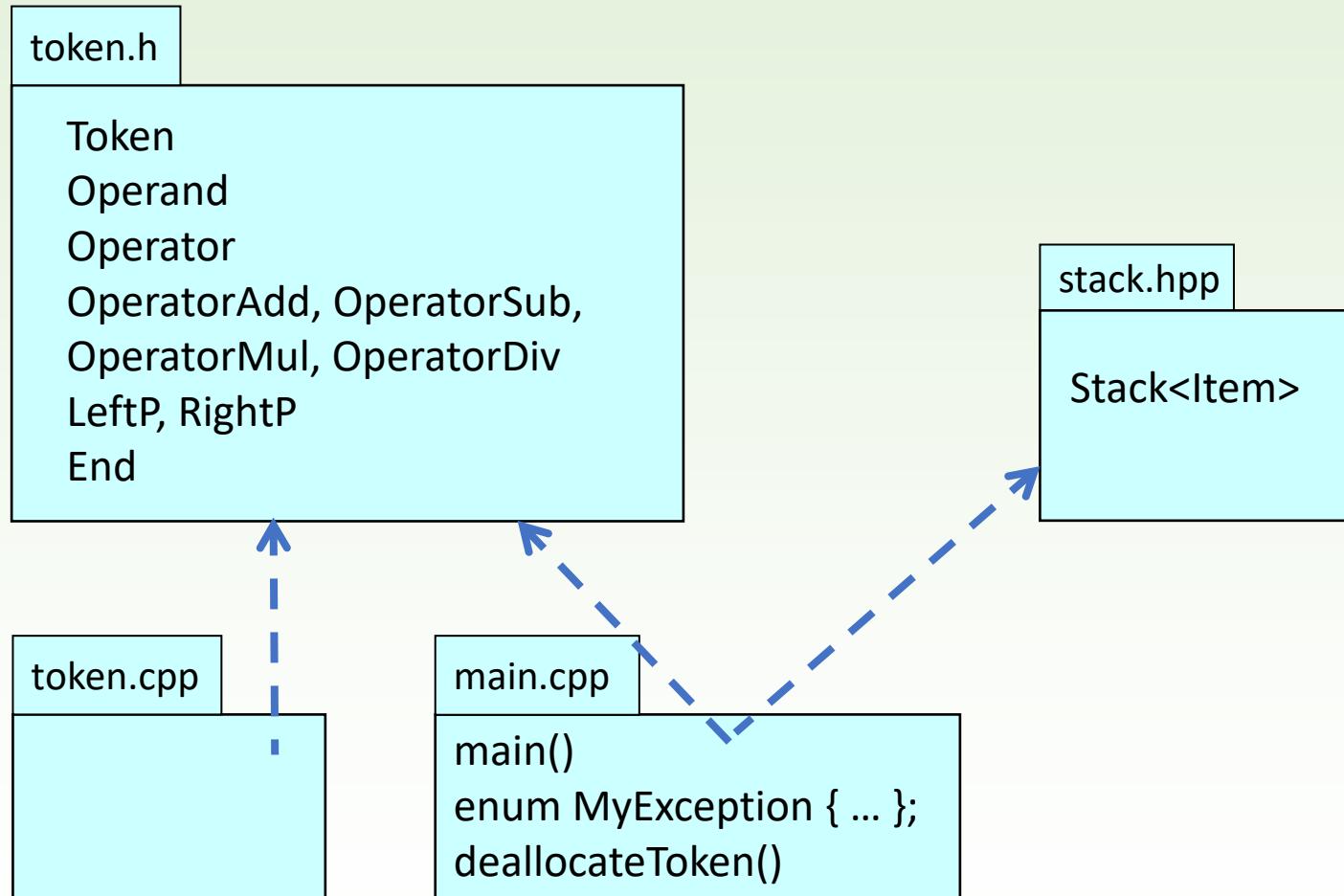
static casting

error when we have too many operands

error when we do not have enough operands

main.cpp

Package diagram

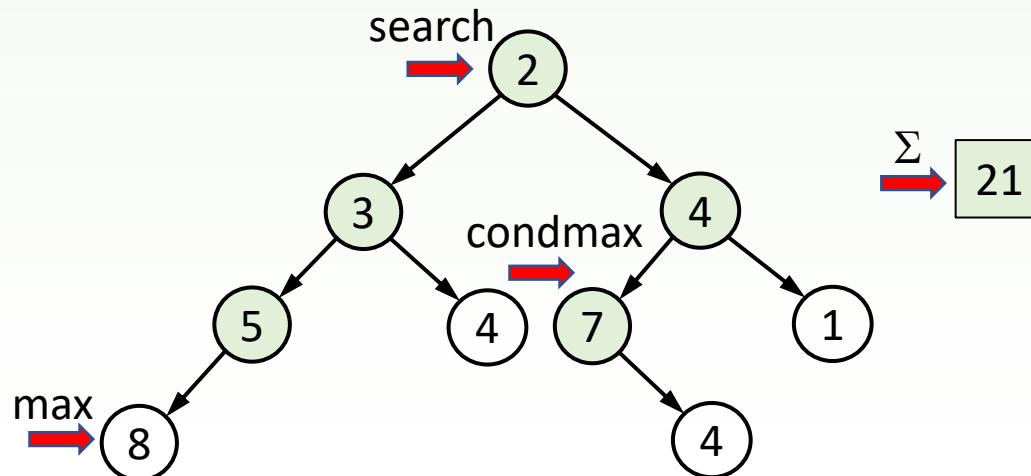


Task: traversal of a binary tree

Read some numbers from a standard input and build randomly a **binary tree** from them. Then print the tree's elements to a standard output based on different **traversal strategies**. Finally,

- sum up the internal nodes,
- find the **maximums** of the internal nodes and all of the nodes,
- find the "first" **even element**!

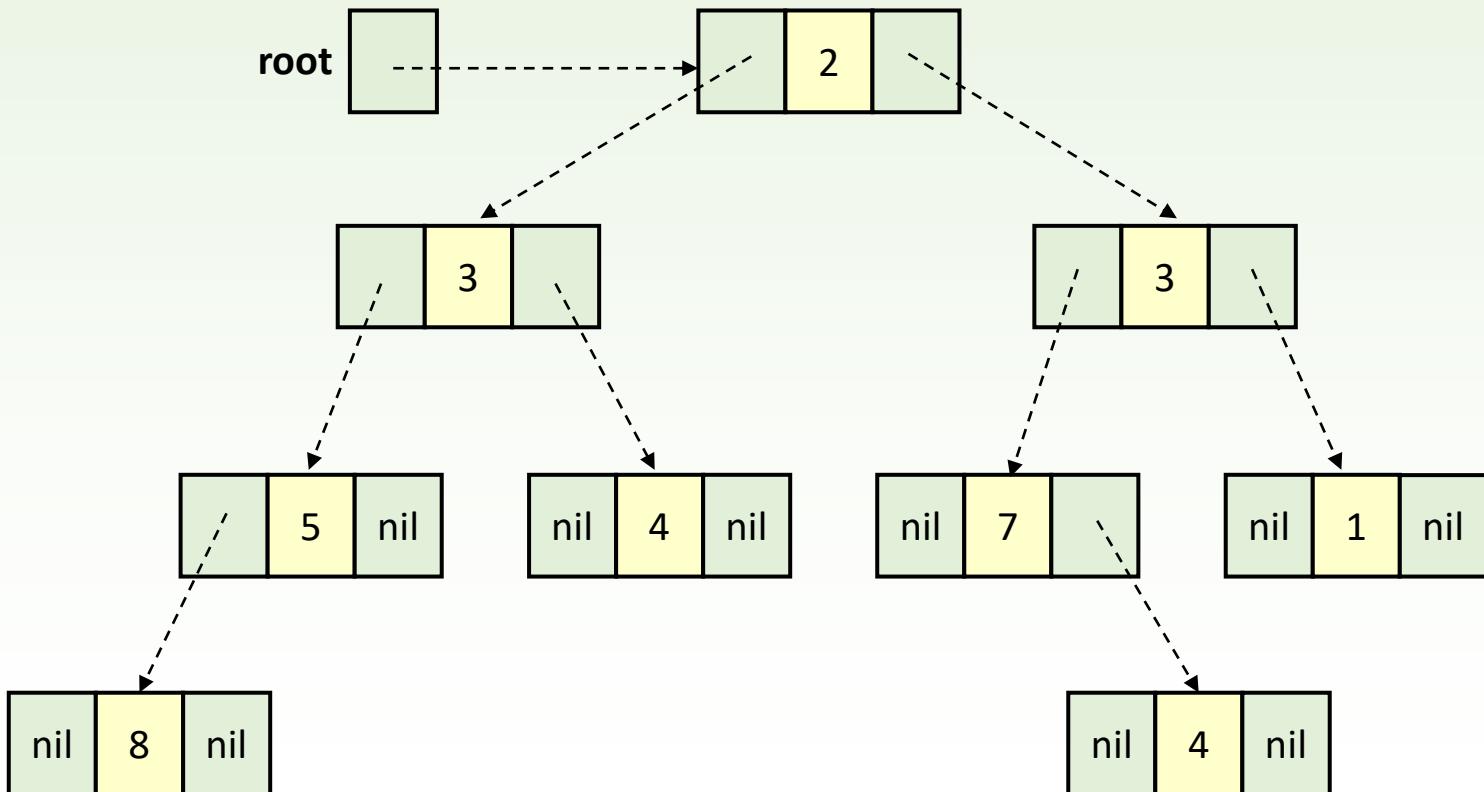
The tree is planned so that the summation, maximum search, and linear search are implemented easily. To be able to change the type of the nodes, the tree becomes a template.



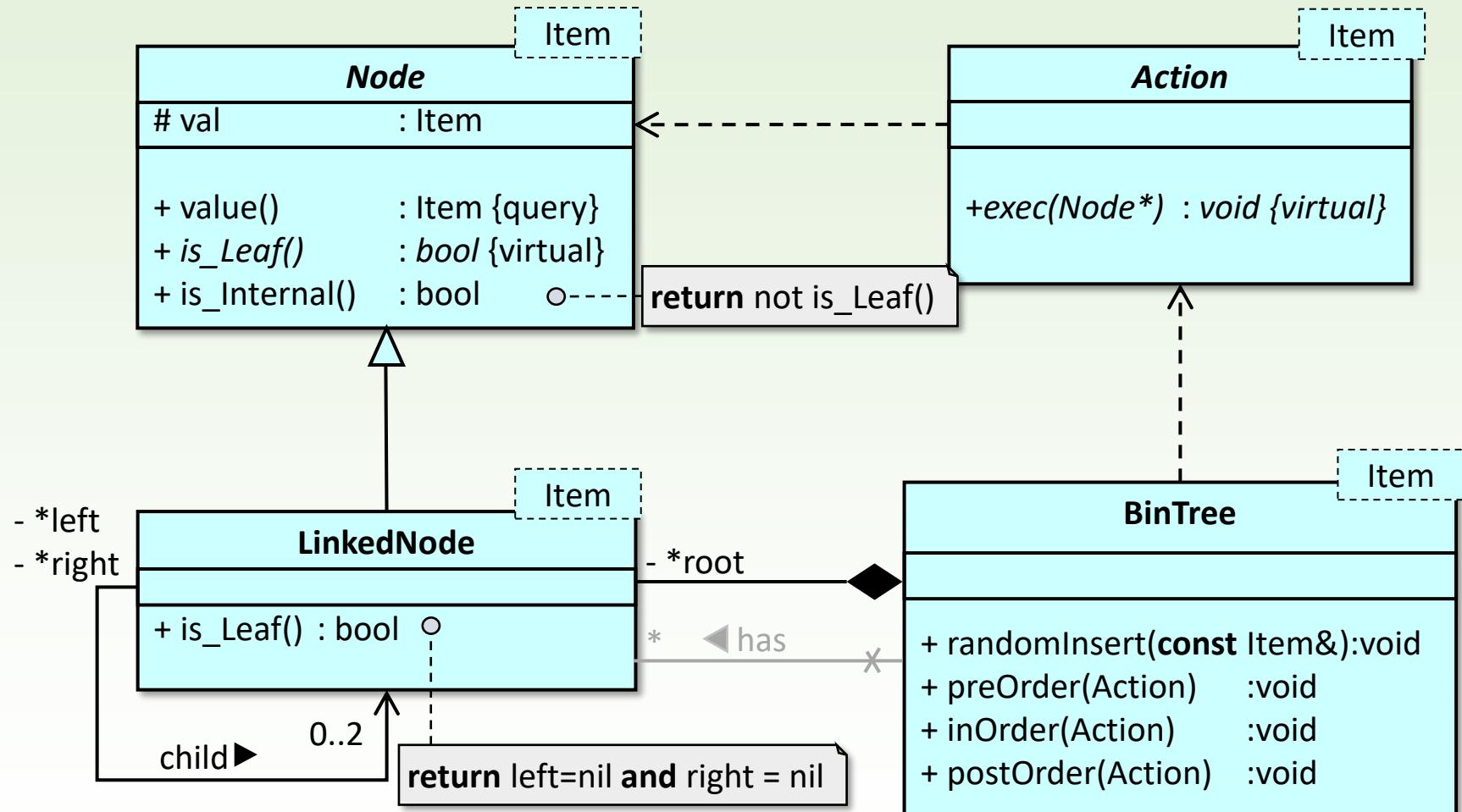
Chained binary tree

```
class BinTree  
protected:  
    LinkedNode *_root;  
    ...  
};
```

```
template <typename Item>  
struct LinkedNode {  
    LinkedNode *_left;  
    Item _val;  
    LinkedNode *_right;  
};
```



Class diagram



Template class of a node

```
template <typename Item>
class Node {
public:
    Item value() const { return _val; }
    virtual bool is_Leaf() const = 0;
    bool is_Internal() const { return !is_Leaf(); }
    virtual ~Node(){}
protected:
    Node(const Item& v): _val(v){}
    Item _val;
};

template <typename Item> class BinTree;
template <typename Item>
class LinkedNode: public Node<Item>{
public:
    friend class BinTree;
    LinkedNode(const Item& v, LinkedNode *l, LinkedNode *r):
        Node<Item>(v), _left(l), _right(r){}
    bool is_Leaf() const override
    { return _left==nullptr && _right==nullptr; }
private:
    LinkedNode *_left;
    LinkedNode *_right;
};

BinTree class is defined after
LinkedNode. To avoid circular
reference, BinTree is just
indicated here.

LinkedNode allows BinTree to
see its private attributes and
methods.

bintree.hpp
```

Template class of the binary tree

```
template <typename Item>
class BinTree{
public:
```

```
    BinTree():_root(nullptr) {srand(time(nullptr));}
    ~BinTree();
```

```
    void randomInsert(const Item& e);
```

```
    void preOrder (Action<Item>*todo){ pre (_root, todo); }
```

```
    void inOrder (Action<Item>*todo){ in (_root, todo); }
```

```
    void postOrder(Action<Item>*todo){ post(_root, todo); }
```

```
protected:
```

```
    LinkedNode<Item>* _root;
```

```
    void pre (LinkedNode<Item>*r, Action<Item> *todo);
```

```
    void in (LinkedNode<Item>*r, Action<Item> *todo);
```

```
    void post (LinkedNode<Item>*r, Action<Item> *todo);
```

```
};
```

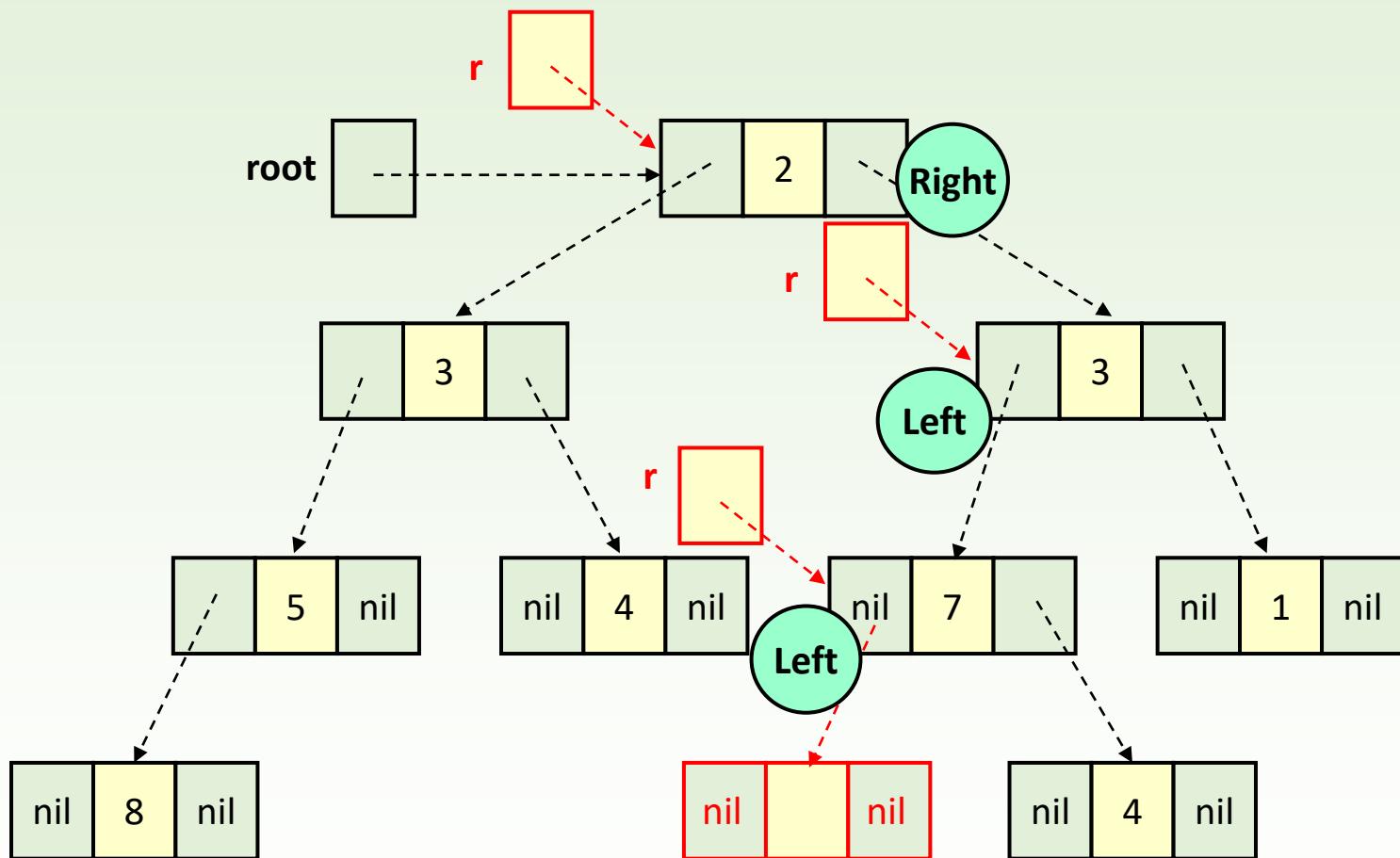
```
template <typename Item>
class Action{
public:
    virtual void exec(Node<Item> *node) = 0;
    virtual ~Action(){}
};
```

initialization of the
random generator :
`#include <time.h>`
`#include <cstdlib>`

with a given action, starting from the root,
they traverse the nodes

bintree.hpp

Inserting a new node into the tree



Inserting a new node into the tree

```
void BinTree<Item>::randomInsert(const Item& e)
{
    if(_root==nullptr) _root = new LinkedNode<Item>(e, nullptr, nullptr);
    else {
        LinkedNode<Item> *r = _root;
        int d = rand();           ← random generator
        while(d&1 ? r->_left!=nullptr : r->_right!=nullptr){
            if(d&1) r = r->_left;
            else   r = r->_right;
            d = rand();           ← Is the last bit of the random number 1?
        }
        if(d&1) r->_left = new LinkedNode<Item>(e, nullptr, nullptr);
        else   r->_right= new LinkedNode<Item>(e, nullptr, nullptr);
    }
}
```

bintree.hpp

Building the tree

```
#include <iostream>
#include "bintree.hpp"

using namespace std;

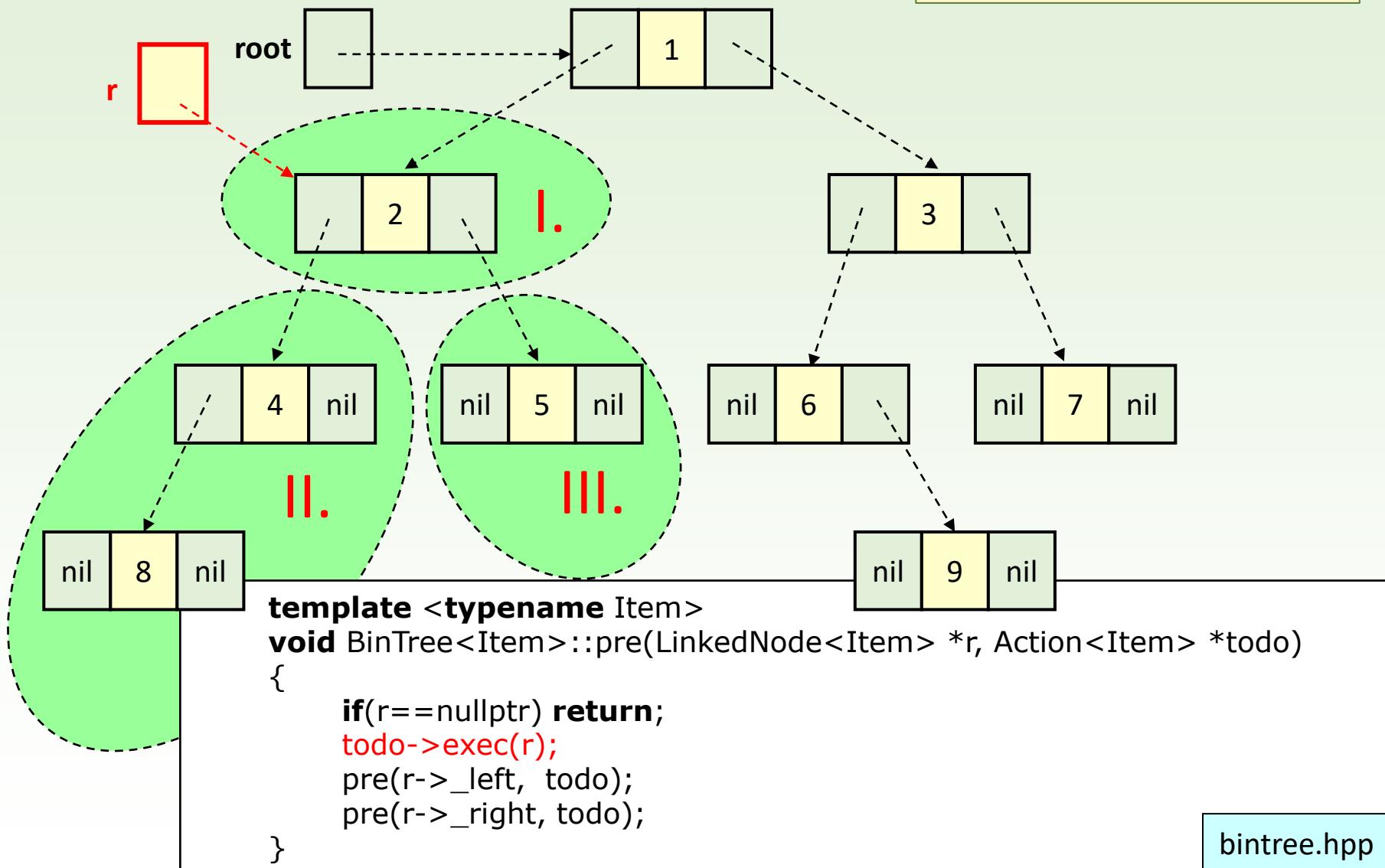
int main()
{
    BinTree<int> t;
    int i;
    while(cin >> i){
        t.randomInsert(i);
    }

    return 0;
}
```

bintree.hpp

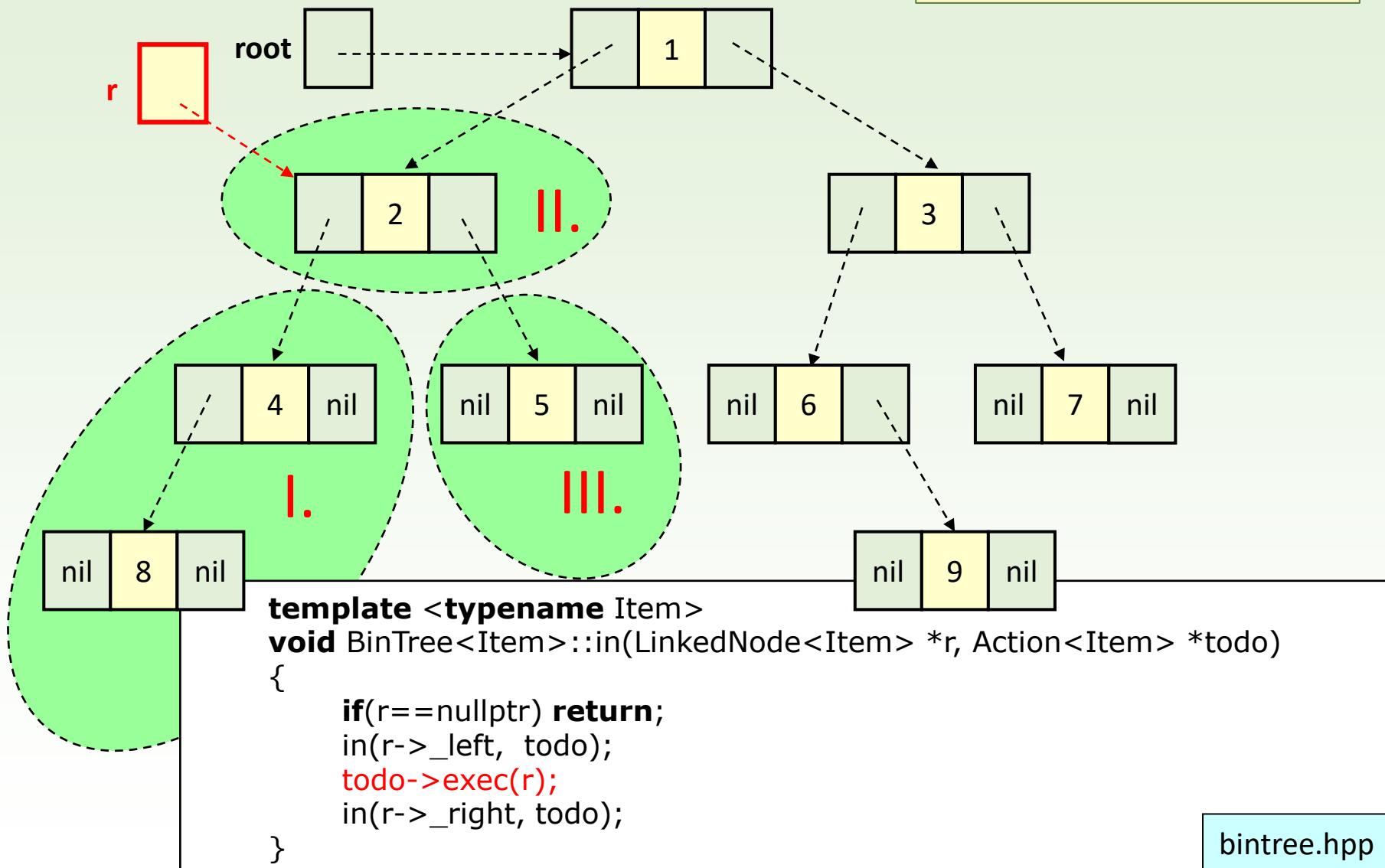
Preorder traversal

1 2 4 8 5 3 6 9 7



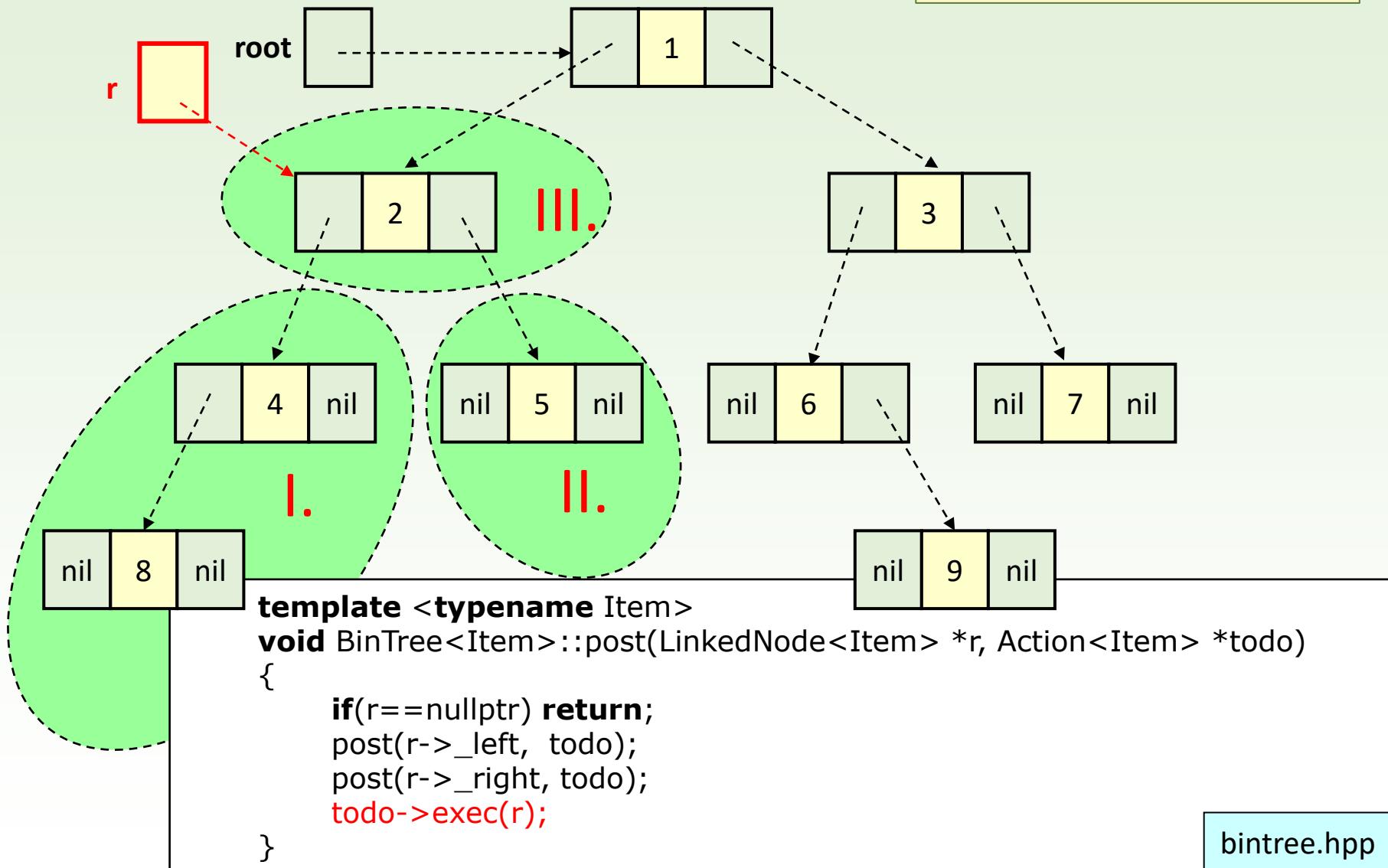
Inorder traversal

8 4 2 5 1 6 9 3 7



Postorder traversal

8 4 5 2 9 6 7 3 1



Template class of printing

```
template <typename Item>
class Printer: public Action<Item>{
public:
    Printer(ostream &o): _os(o){}
    void exec(Node<Item> *node) override {
        _os << '[' << node->value() << ']';
    }
private:
    ostream& _os;
};
```

Output operator has to be defined
for the datatype replacing Item.

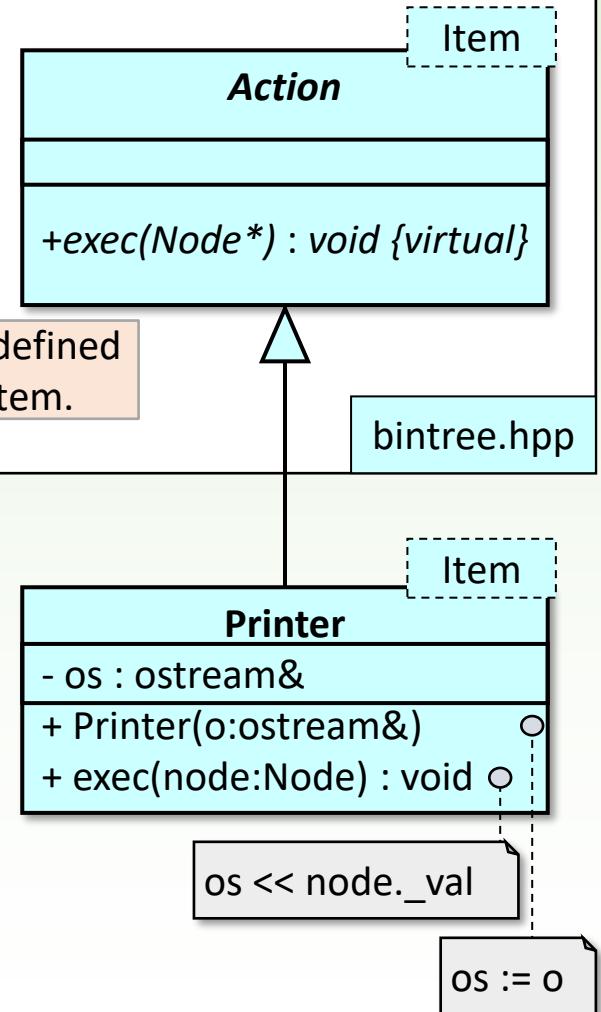
```
BinTree<int> t = build();

Printer<int> print(cout);

cout << "Preorder traversal :";
t.preOrder (&print); cout << endl;

cout << "Inorder traversal :";
t.inOrder (&print); cout << endl;

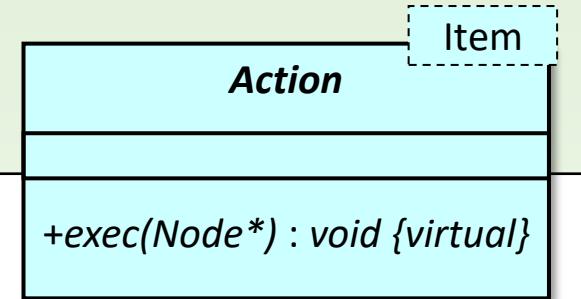
cout << "Postorder traversal :";
t.postOrder (&print); cout << endl;
```



Destructor: traversal with delete

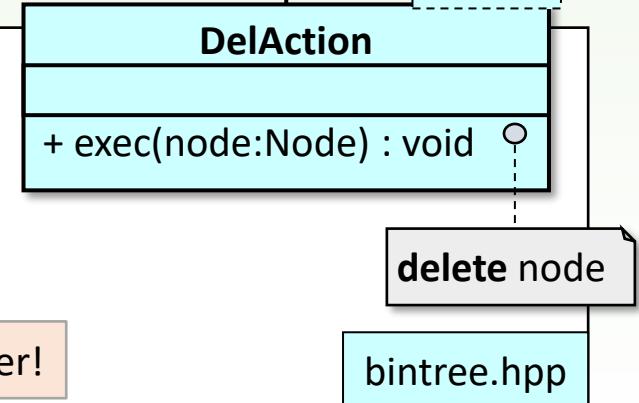
```
protected:  
class DelAction: public Action<Item> {  
public:  
    void exec(Node<Item> *node) override {delete node;}  
};
```

in `BinTree<Item>`'s protected part.



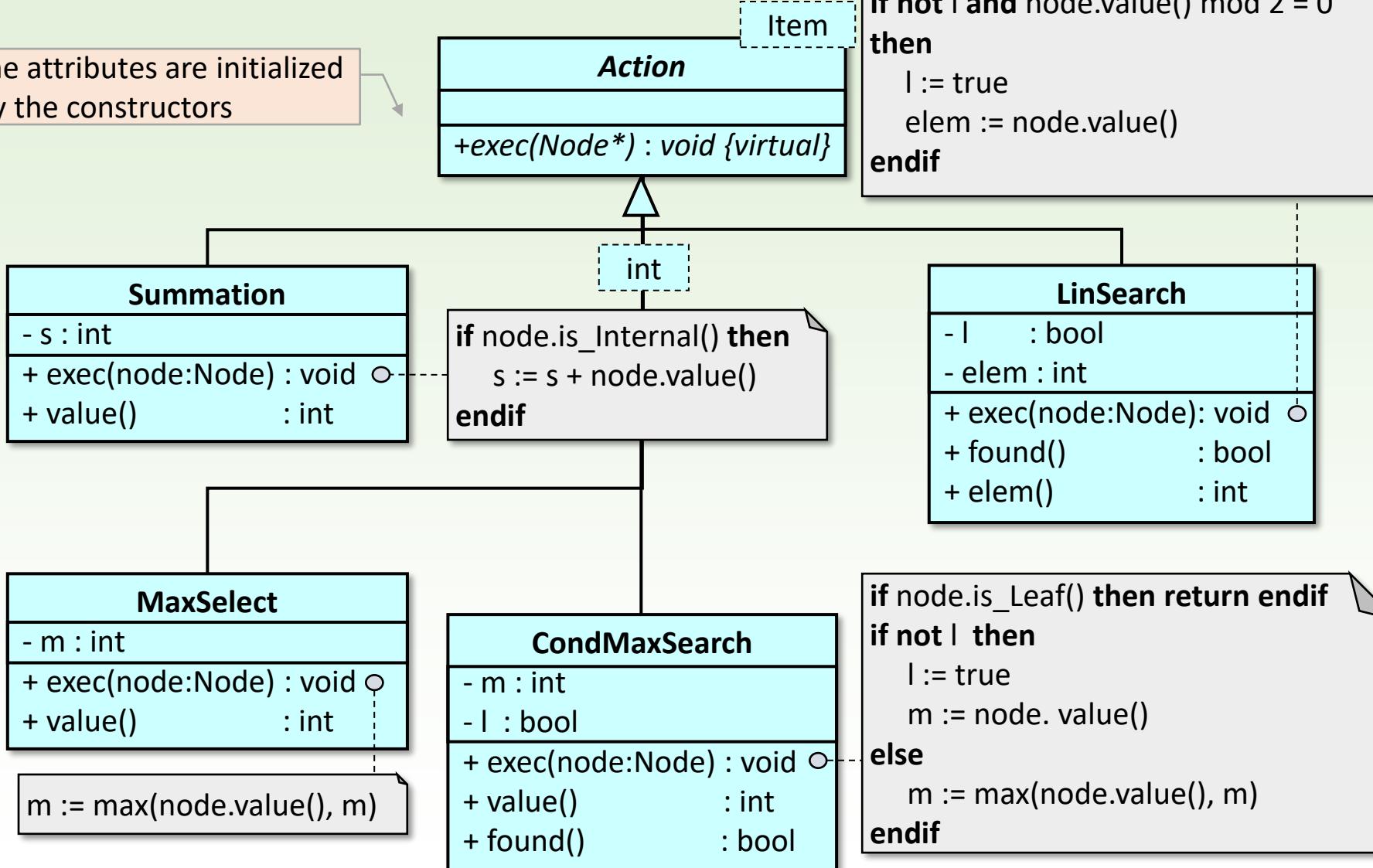
```
template <typename Item>  
BinTree<Item>::~BinTree()  
{  
    DelAction del;  
    post(_root, &del);  
}
```

It works only with postorder!



Other actions' classes

the attributes are initialized by the constructors



If not l and node.value() mod 2 = 0
then

$$l := \text{true}$$

$$\text{elem} := \text{node.value()}$$

endif

LinSearch

- l : bool
- elem : int
- +exec(node:Node) : void
- +found() : bool
- +elem() : int

CondMaxSearch

if node.is_Leaf() then return endif

if not l then

$$l := \text{true}$$

$$m := \text{node.value()}$$

else

$$m := \max(\text{node.value}(), m)$$

endif

Summation

```
class Summation: public Action<int>{
public:
    void Summation():_s(0){}
    void exec(Node<int> *node) override {
        if(node->Is_Internal()) _s += node->value(); }
    int value() const {return _s;}
private:
    int _s;
};
```

```
BinTree<int> t = build();

Summation sum;
t.preOrder(&sum);
cout << "Sum of the elements of the tree: "
     << sum.value() << endl;
```

Linear search

```
class LinSearch: public Action<int>{
public:
    void LinSearch():_l(false){}
    void exec(Node<int> *node) override {
        if (!l && node->value()%2==0){
            l = true;
            _elem = node->value();
        }
    }
    bool found() const {return _l;}
    int elem() const {return _elem;}
private:
    bool _l;
    int _elem;
};
```

```
BinTree<int> t = build();

LinSearch search;
t.preOrder(&search);

if (search.found()) cout << search.elem() << " is an";
else cout << "There is no";
cout << " even element in the tree.";
```

Maximum search

```
class MaxSelect: public Action<int>{
public:
    MaxSelect(int &i) : _m(i){}
    void exec(Node<int> *node) override
    { _m = max( _m, node->value() ); }
    int value() const {return _m;}
private:
    int _m;
};
```

```
BinTree<int> t = build();
MaxSelect max(t.rootValue());
t.preOrder(&max);

cout << "Maxima of the elements of the tree: " << max.value();
```

It shoud raise an exception if
the tree is empty (without root)

```
template <class Item> class BinTree {
...
public:
    enum Exceptions{NOROOT};
    Item rootValue() const {
        if( _root==nullptr ) throw NOROOT;
        return _root->value();
    }
}
```

bintree.hpp

Conditional maximum search

```
BinTree<int> t = build();
CondMaxSearch max;
t.preOrder(&max);
```

```
class CondMaxSearch: public Action<int>{
public:
    struct Result {
        int m;
        bool l;
    };
    CondMaxSearch(){_r.l = false;}
    virtual void exec(Node<int> *node) {
        if(node->is_Leaf()) return;
        if(!_r.l){
            _r.l = true;
            _r.m = node->value();
        }else{
            _r.m = max( _r.m, node->value() );
        }
    }
    Result value(){return _r;}
private:
    Result _r;
};
```

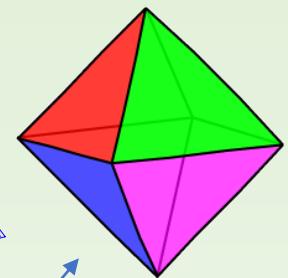
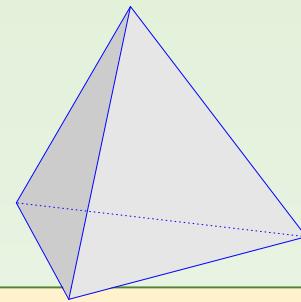
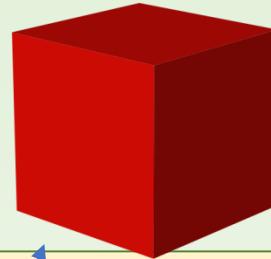
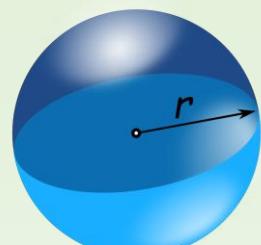
```
cout << "Maxima of the elements of the tree: " << endl;
if(max.value().l) cout << max.value().m << endl;
else             cout << "none" << endl;
```

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

Volume of geometrical shapes

Competition of creatures

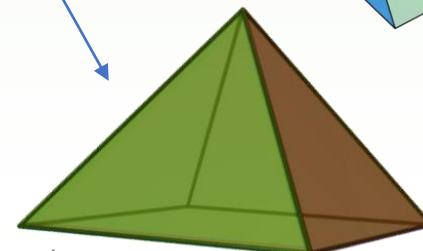
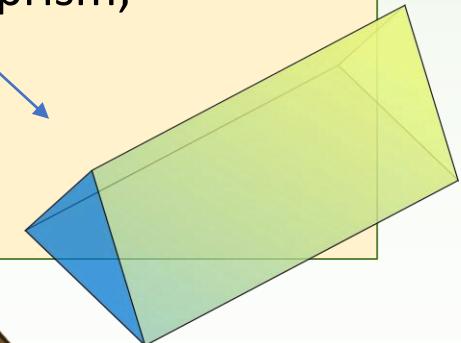
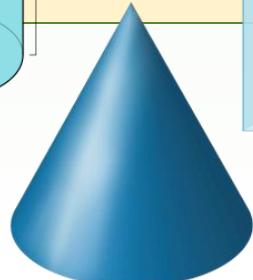
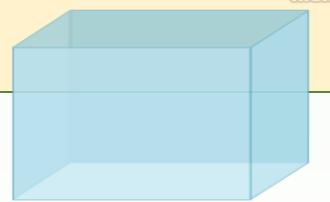
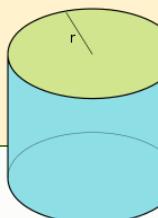
1st task



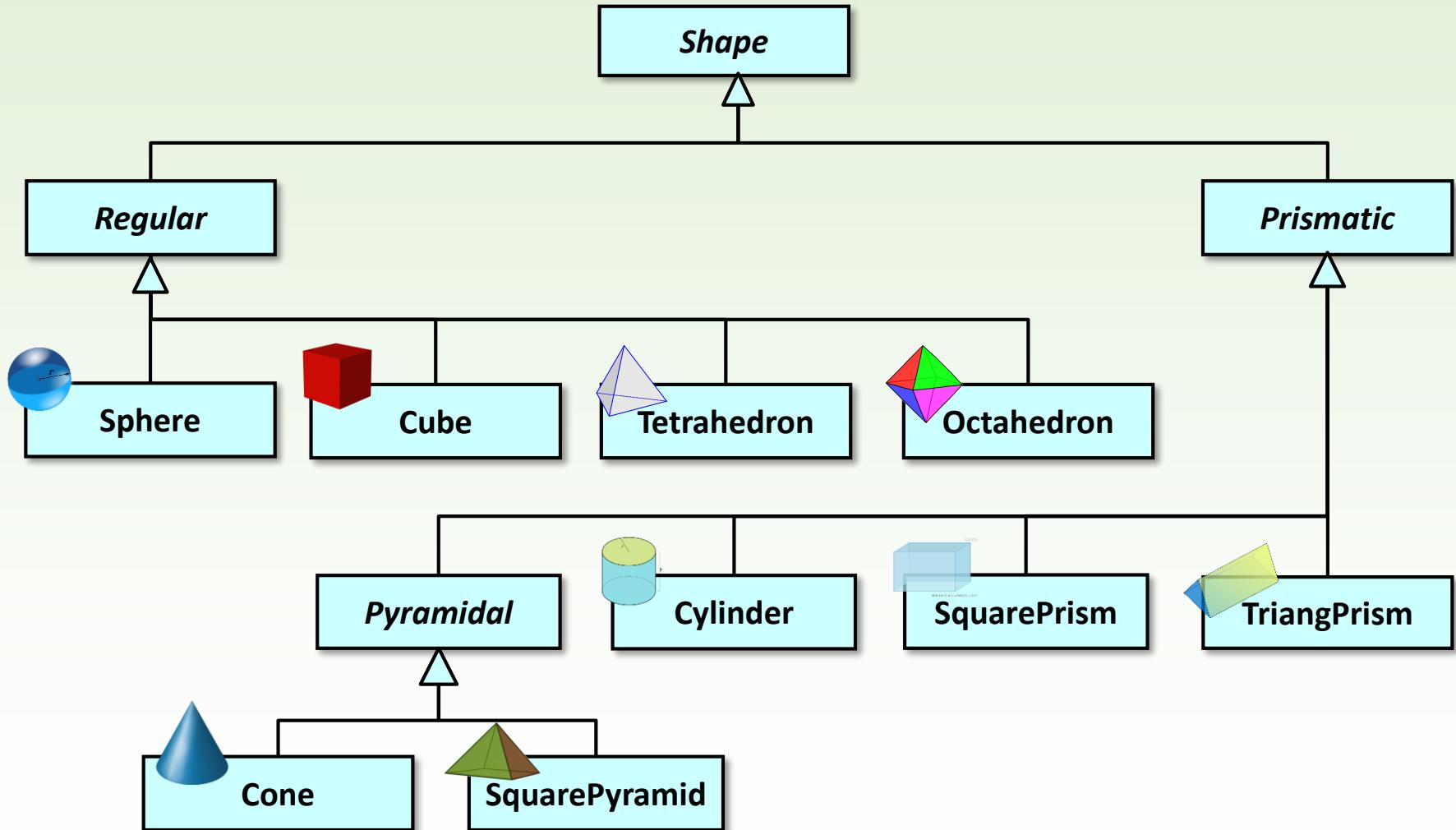
Create a program to calculate the volume of different geometrical shapes and to calculate how many objects of the different types were created.

Possible types:

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



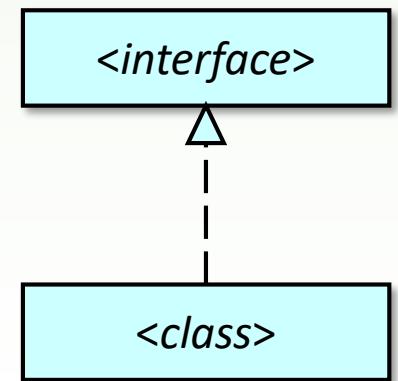
Class diagram



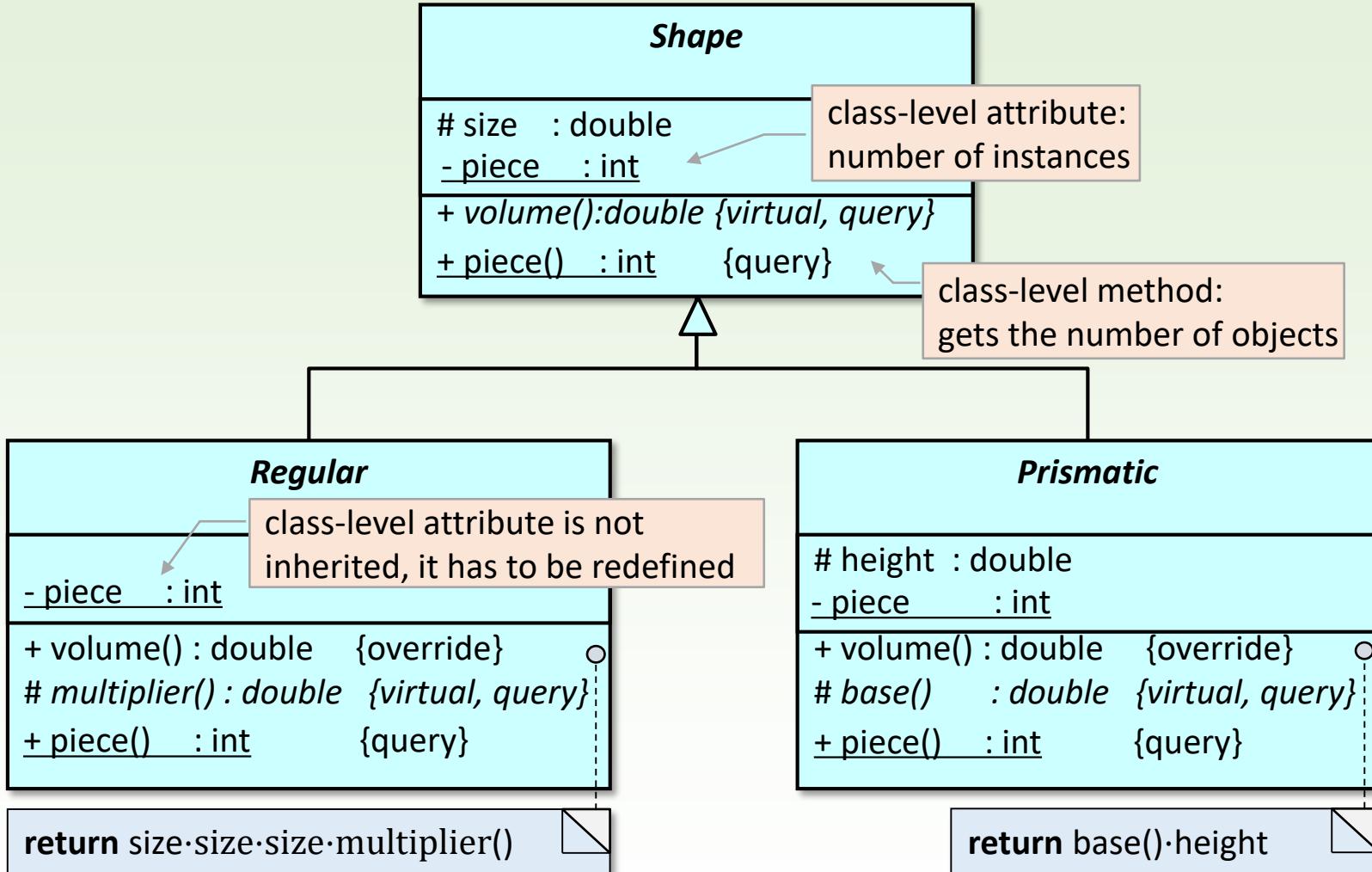
Abstract class, interface

- ❑ **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;
};
```

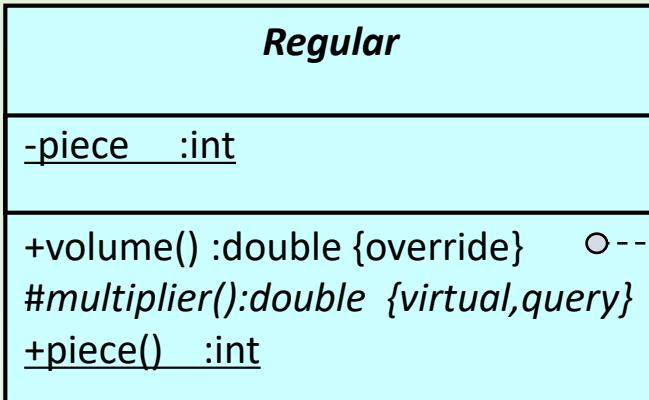
Without this, the constructor would automatically call the empty constructor of the base class which does not exist.

```
int Regular::_piece = 0;

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

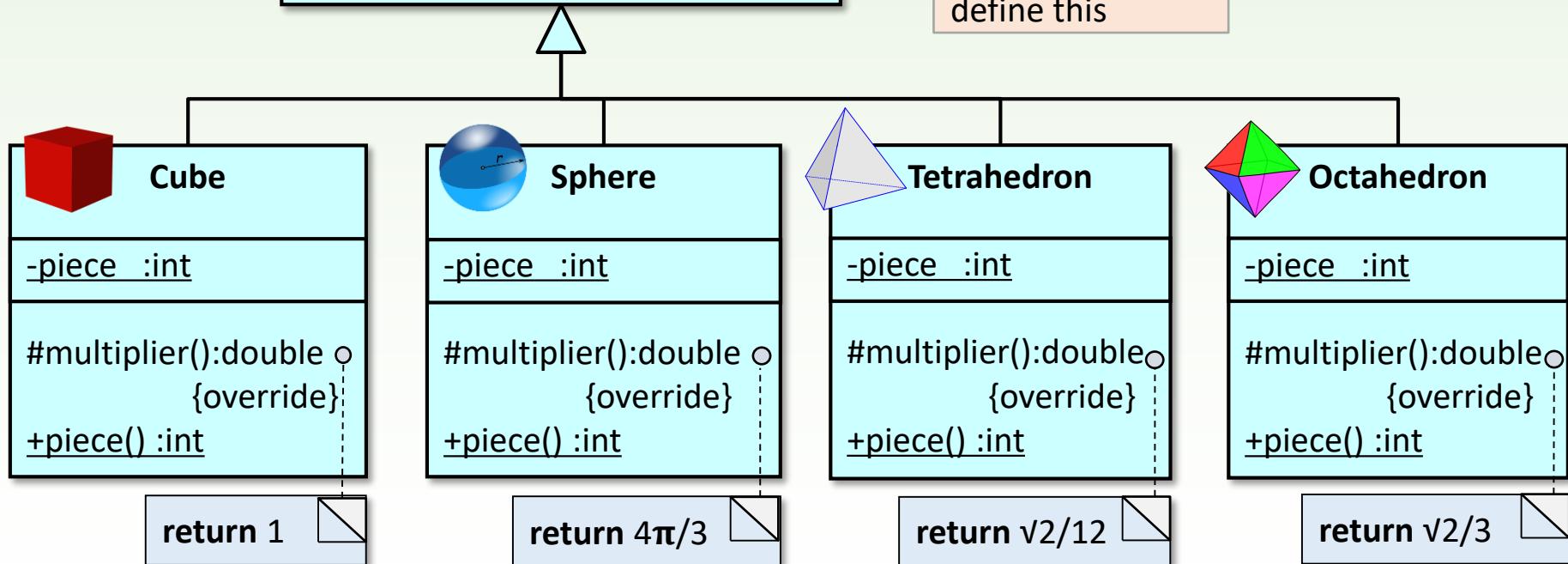
the destructor automatically calls the destructor of the base class

Regular shapes



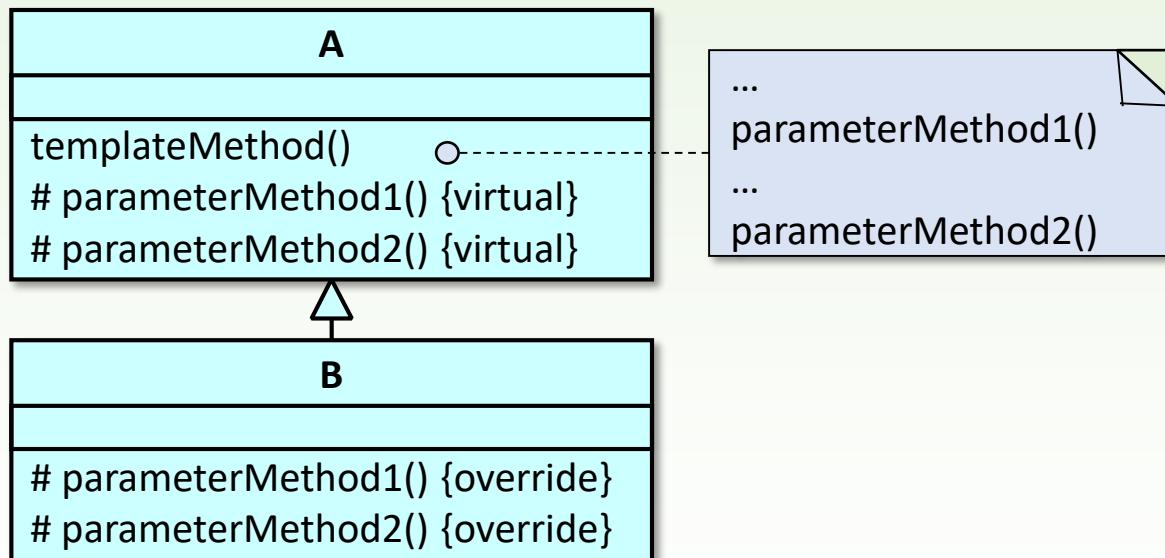
return size·size·size·multiplier()

subclasses have to
define this



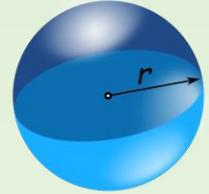
Template method design pattern

- 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



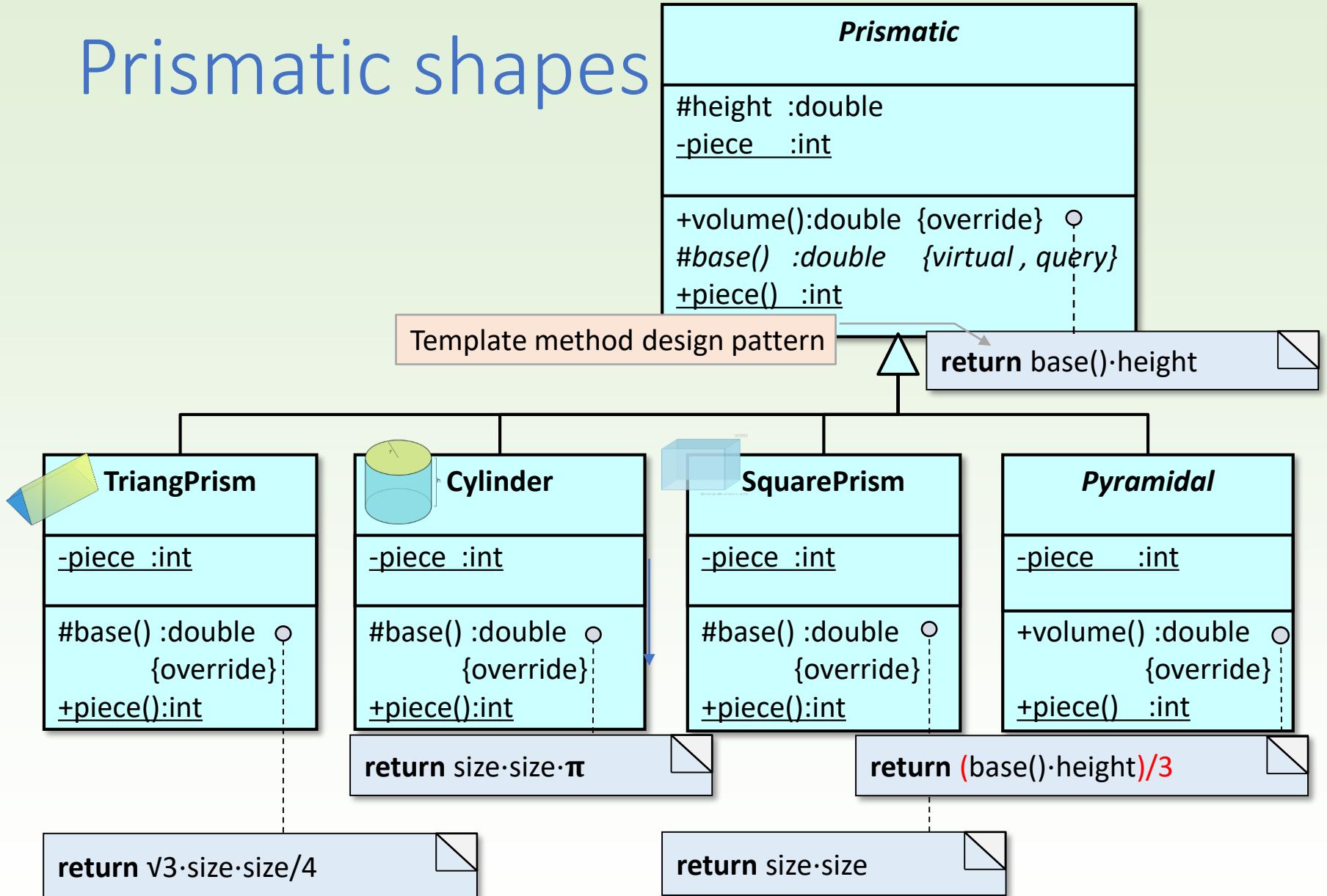
```
class Sphere : public Regular
{
public:
    Sphere(double size);
    ~Sphere();
    static int piece() { return _piece; }
protected:
    double multiplier() const override { return _multiplier; }
private:
    constexpr static double _multiplier = (4.0 * 3.14159) / 3.0;
    static int _piece;
};
```

constant class-level expression

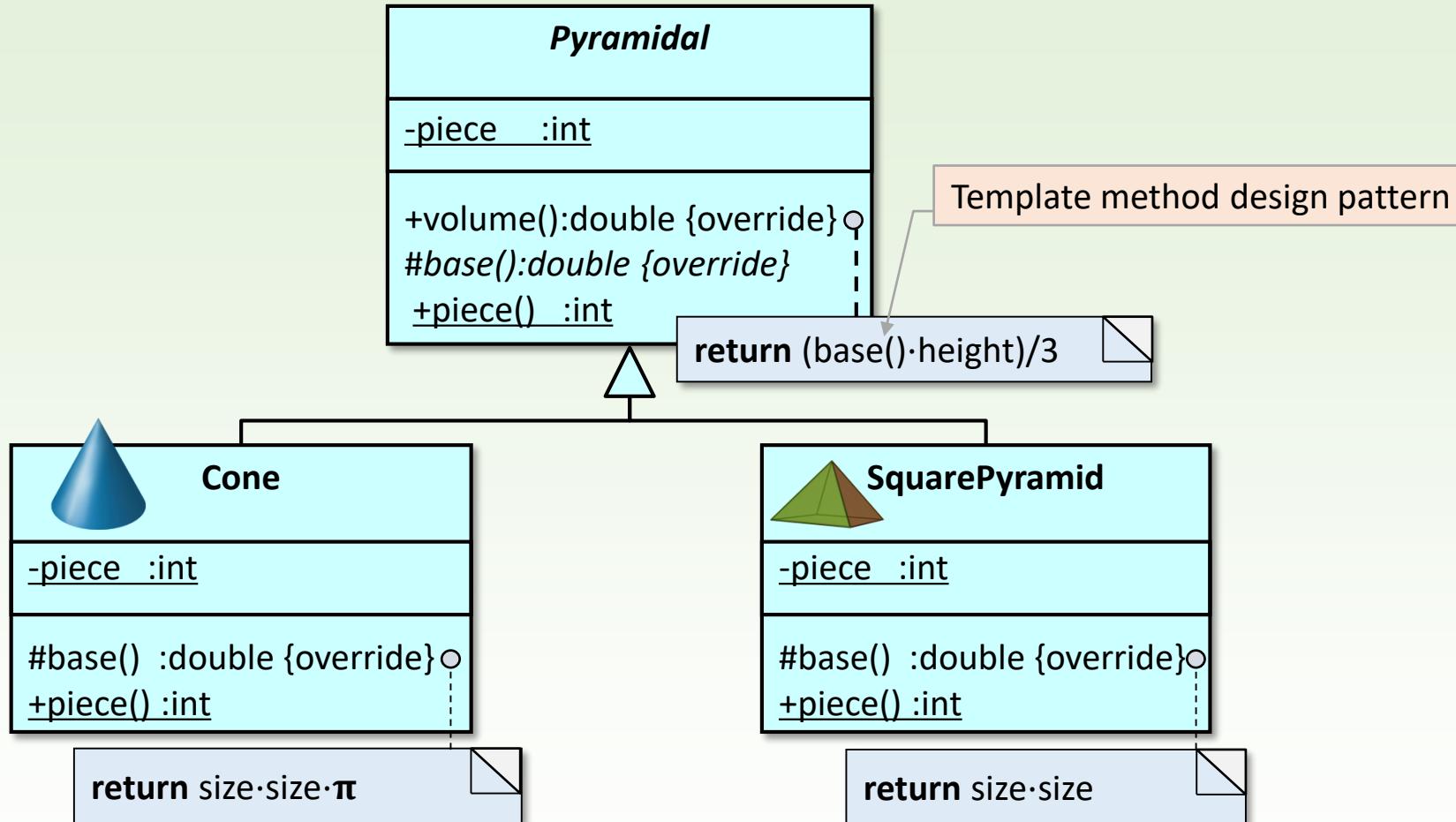
```
int Sphere::_piece = 0;

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

```
#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();
}
```

8	shapes.txt
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

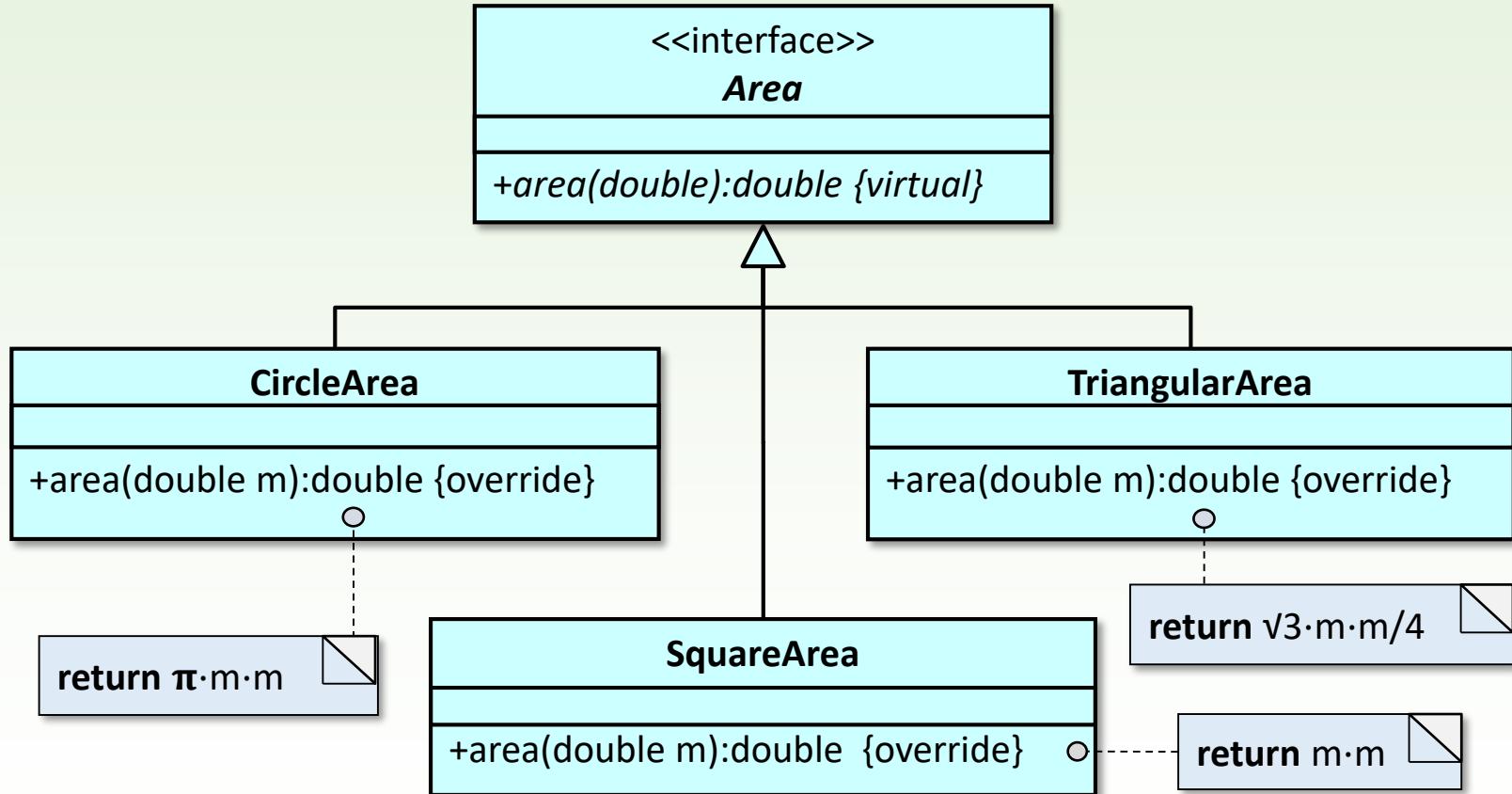
```
...
for ( Shape *p : shapes ) {
    cout << p->volume() << endl;
}
print_statistics();
for ( Shape *p : shapes ) delete p;
print_statistics();
}

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.

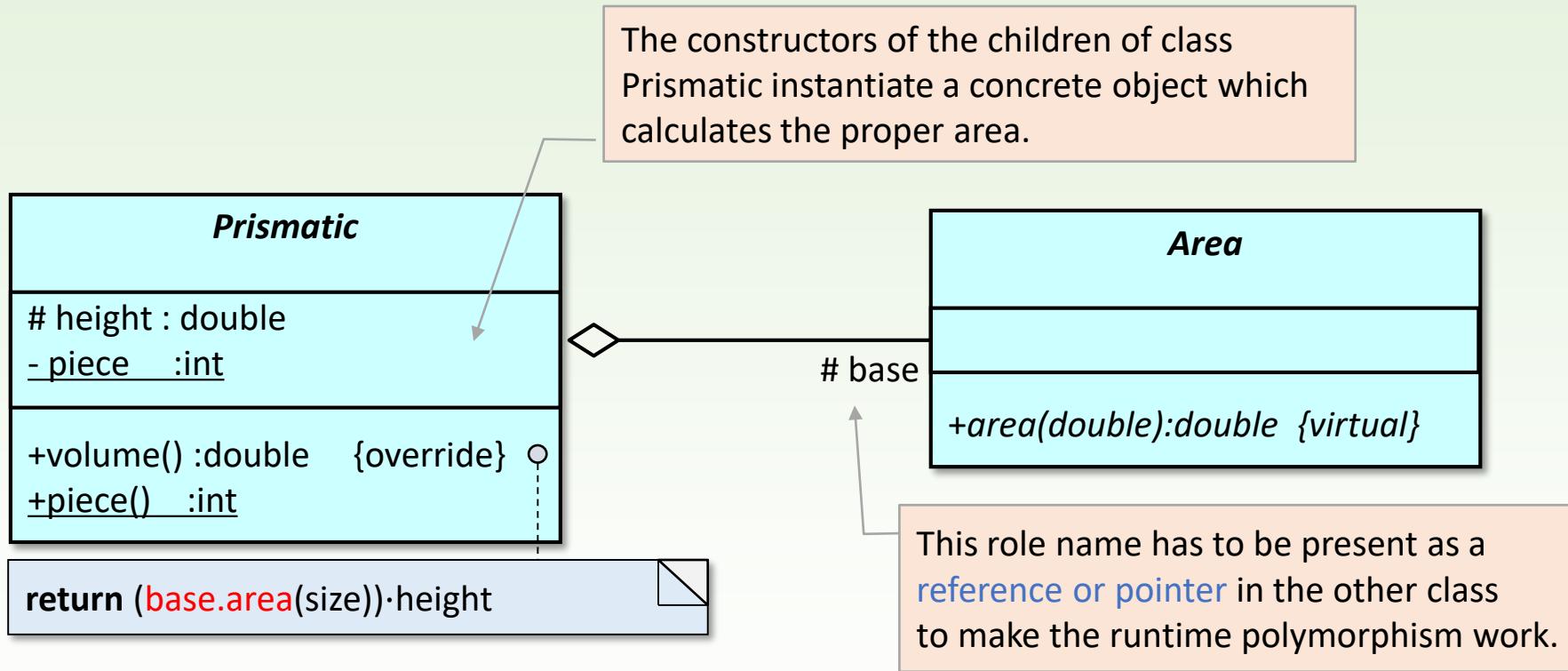
The destructor of the proper shape runs instead of the virtual destructor of base class Shape, because of runtime polymorphism.

Redundancy elimination: objects instead of methods for area calculus



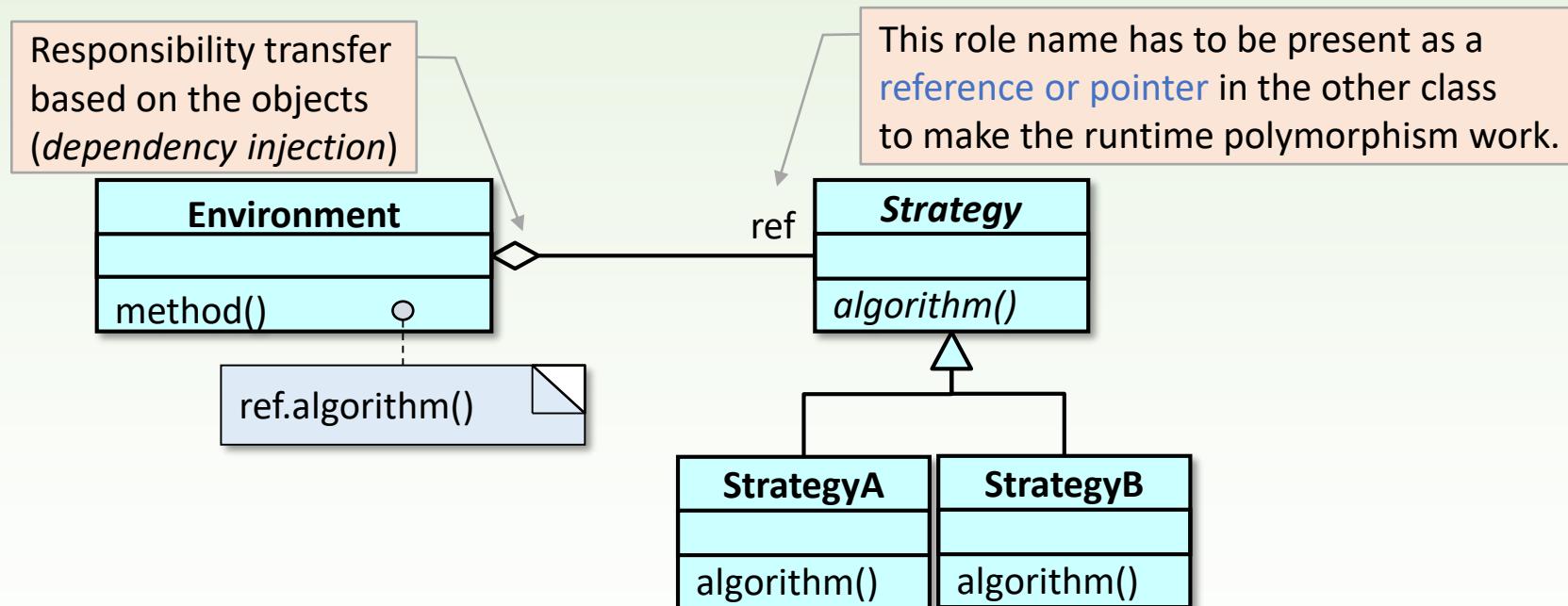
Dependency injection

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

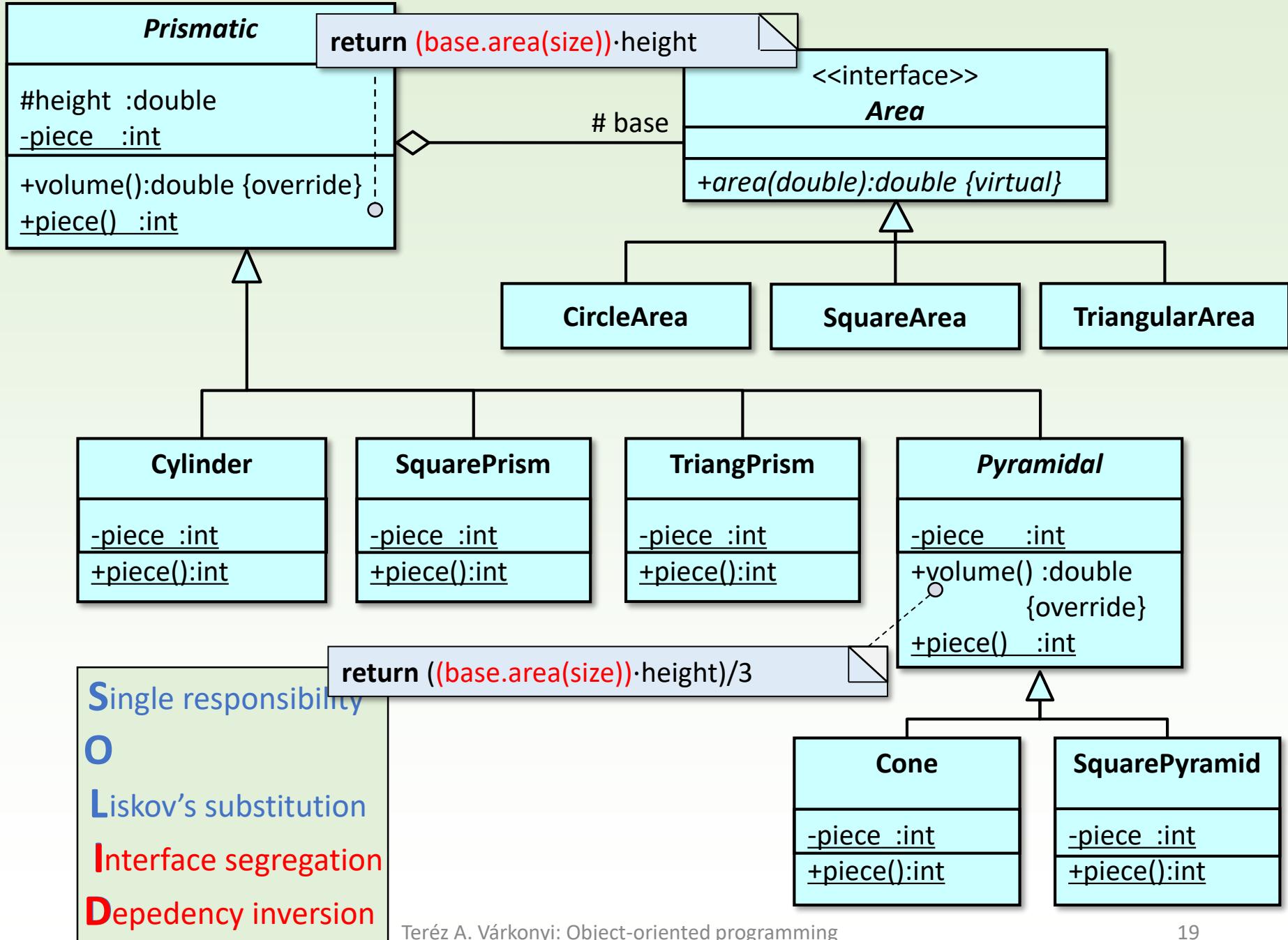


Strategy design pattern

- ❑ 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.



```
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;
}
```

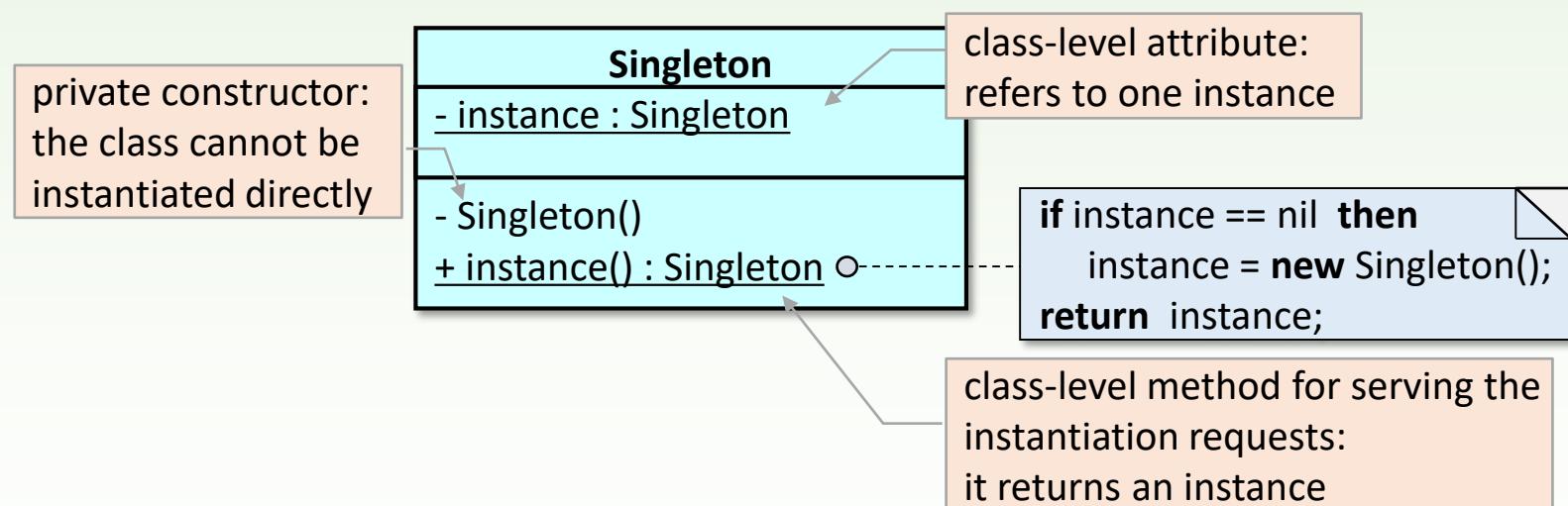
```
TriangularPrism::TriangularPrism(...) : Prismatic(...) {
    ++_piece; _base = new TriangularArea();
}
TriangularPrism::~TriangularPrism() {
    --_piece; delete _base;
}
```

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.

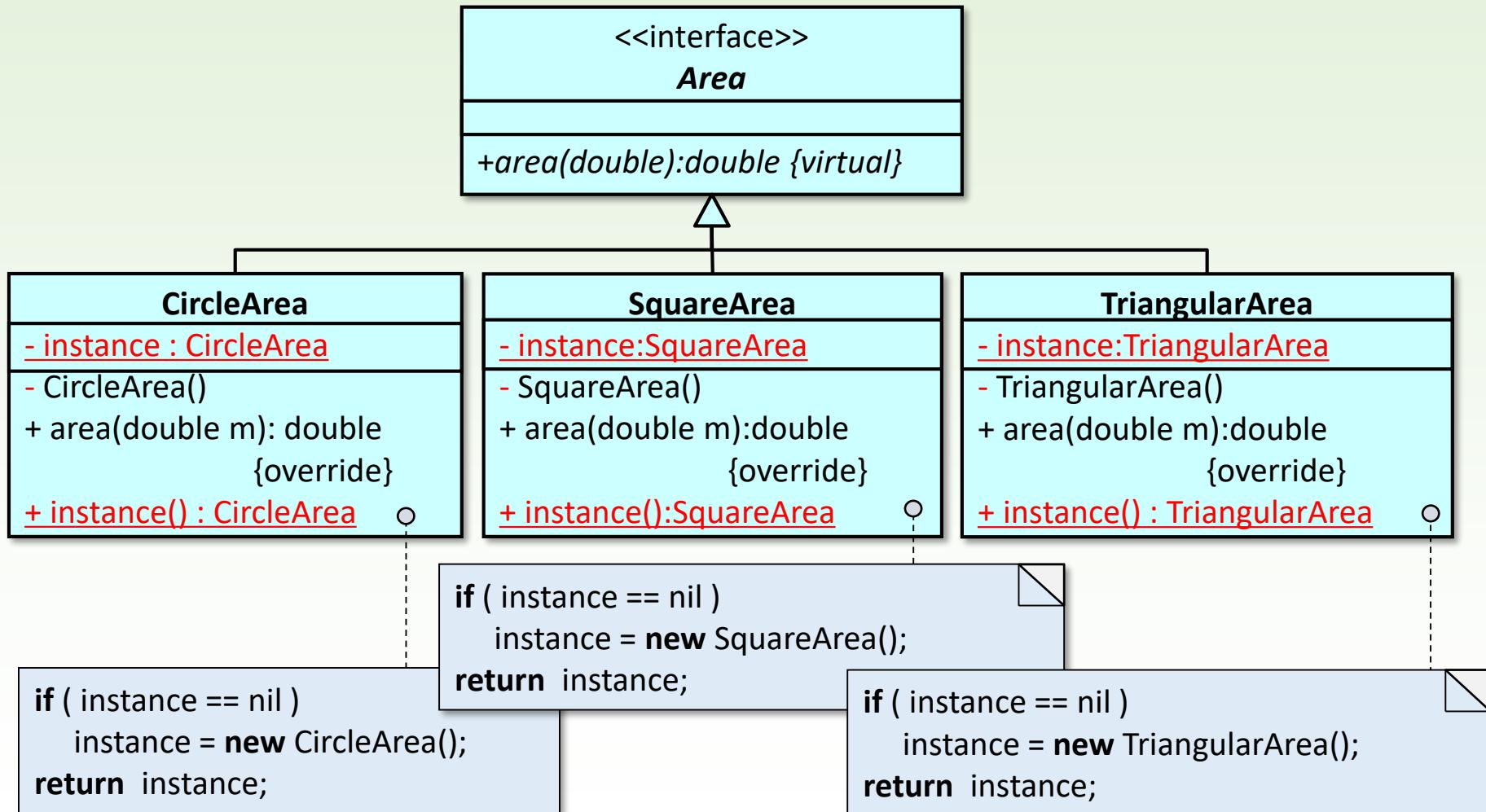
Singleton design pattern

- ❑ 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;
}

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;
}

```

instead of `_base = new CircleArea()`

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

$A : \text{track: Ground}^m, \text{creatures: Creature}^n, \text{alive: String}^*$

$Pre : \text{creatures} = \text{creatures}_0 \wedge \text{track} = \text{track}_0$

$Post : \forall i \in [1..n] : (\text{creatures}[i], \text{track}_i) = \text{transmute}(\text{creatures}_0[i], \text{track}_{i-1})$
 $\wedge \text{track} = \text{track}_n$

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

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

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

alive := <>

$i = 1 .. n$

$\text{creatures}[i], \text{track} := \text{transmute}(\text{creatures}[i], \text{track})$

$\text{creatures}[i].alive()$

alive : write ($\text{creatures}[i].name()$)

-

old \ominus new ::= new

Double summation (one conditional):

$t:\text{enor}(E) \sim i = 1 .. n$

$f(e)_1 \sim \text{transmute}(\text{creatures}[i], \text{track})$

$f(e)_2 \sim <\text{creatures}[i].name()>$

$\text{cond}(e)_2 \sim \text{creatures}[i].alive()$

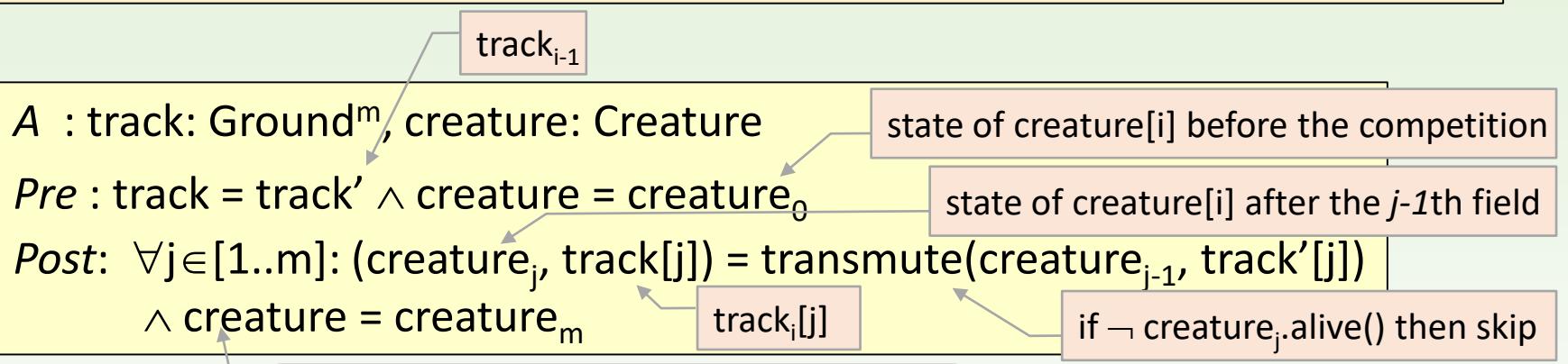
$H,+_0 \sim \text{Creature}^n \times \text{Ground}^m, (\oplus, \ominus),$

$(\langle \rangle, \text{track}_0)$

$H,+_0 \sim \text{String}^*, \oplus, \langle \rangle$

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.



creature, track := transmute(creature, track)

$j := 1$

creature.alive() \wedge $j \leq m$

creature.transmute(track[j])

$j := j+1$

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

Double summation as long as a condition:

$t:enor(E) \sim j = 1 .. m$

as long as creature.alive()

$f(e) \sim \text{transmute}(\text{creature}, \text{track}[j])$

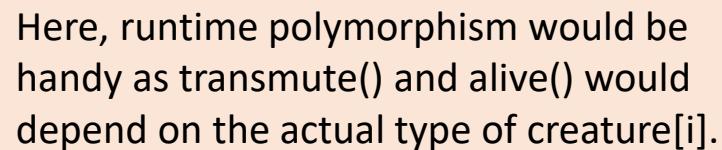
$H,+,<0> \sim \text{Creature} \times \text{Ground}^m, (\ominus, \oplus),$
 $(\text{creature}_0, \langle \rangle)$

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;
}

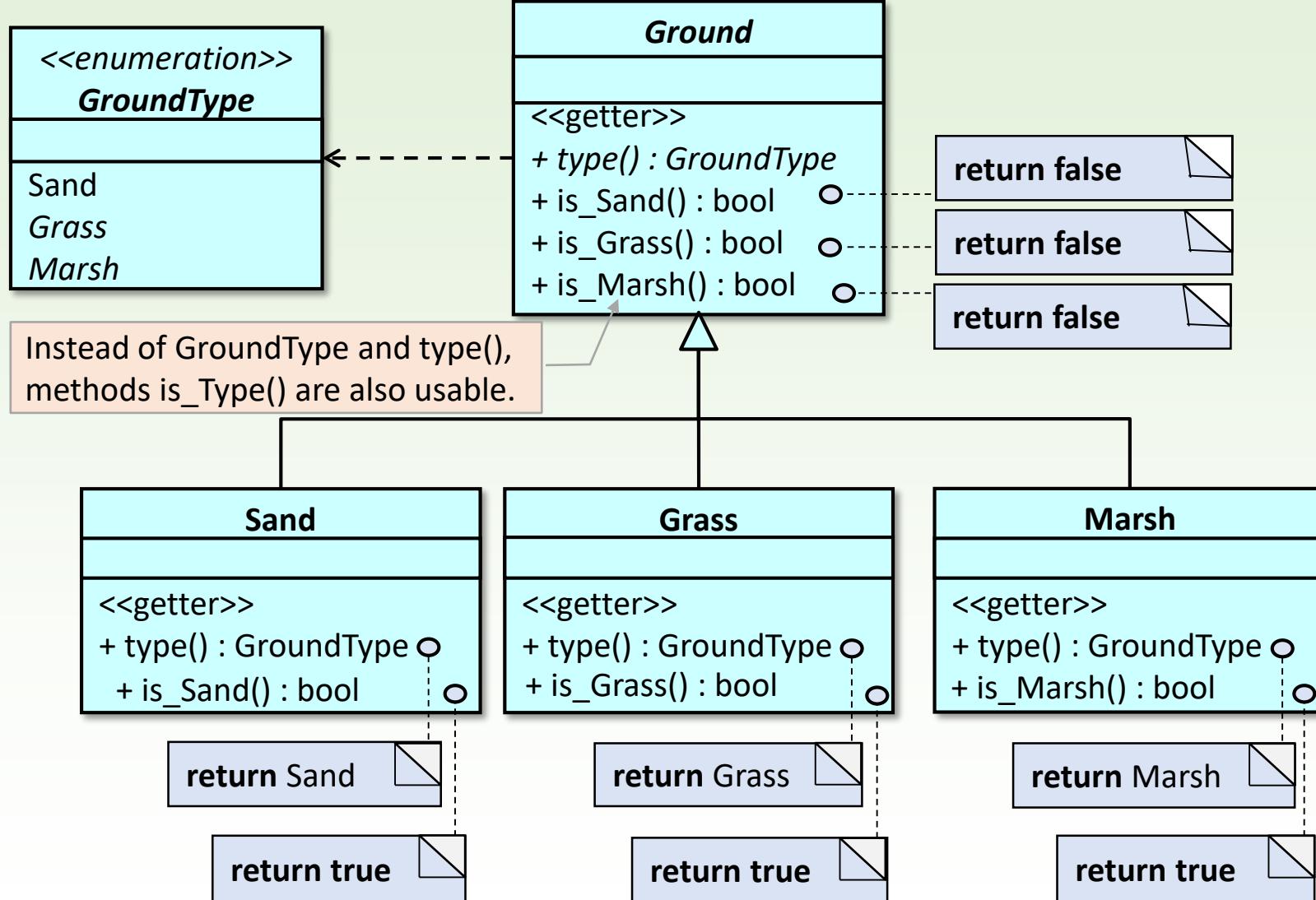
// Destruction
...
```

main.cpp

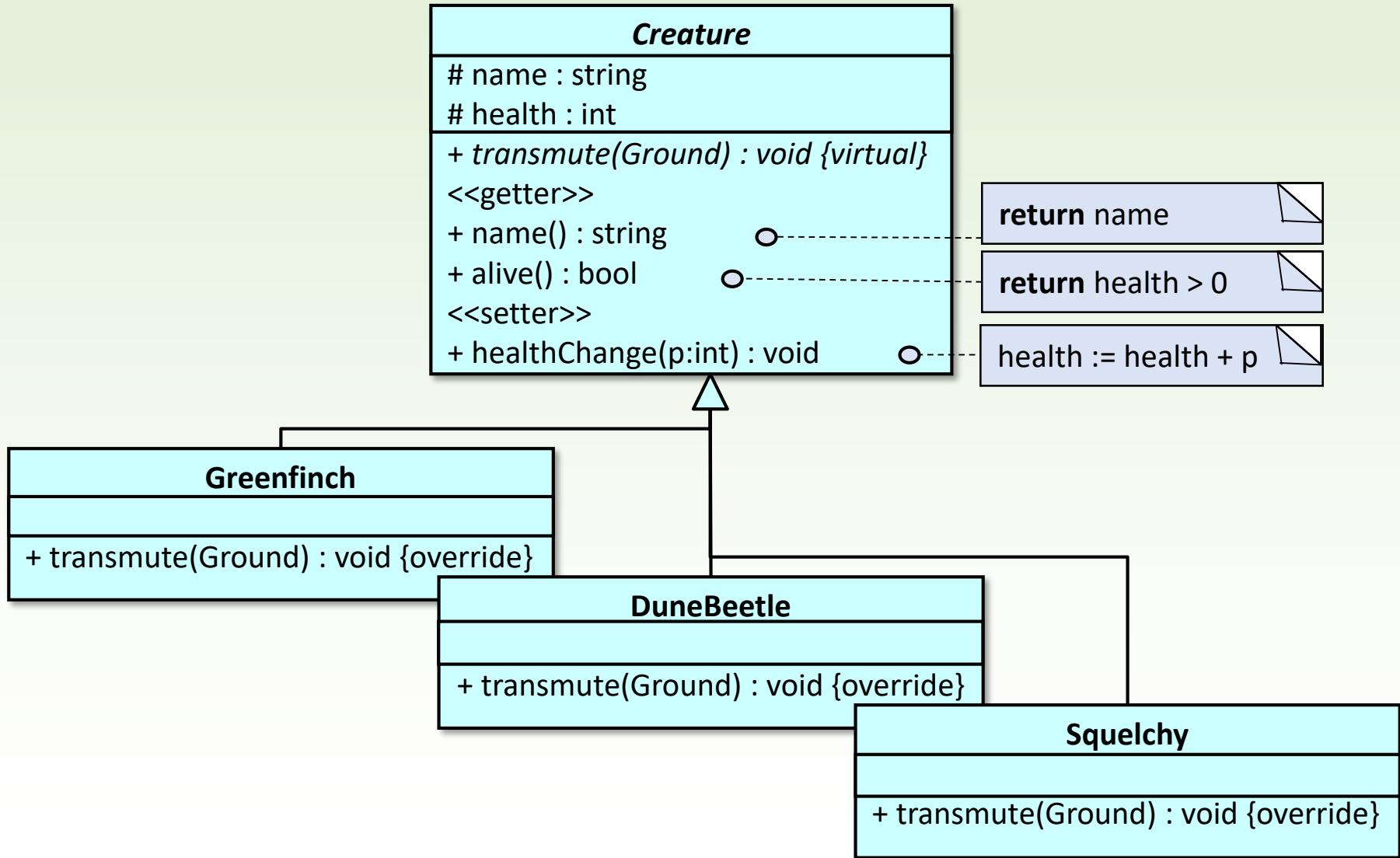


Here, runtime polymorphism would be handy as `transmute()` and `alive()` would depend on the actual type of `creature[i]`.

Recognizing the subtypes



Inheritance of creatures



Creatures

```
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 () {}
};
```

type of the ground is integer

creature.h

```
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;
        }
    }
}
```

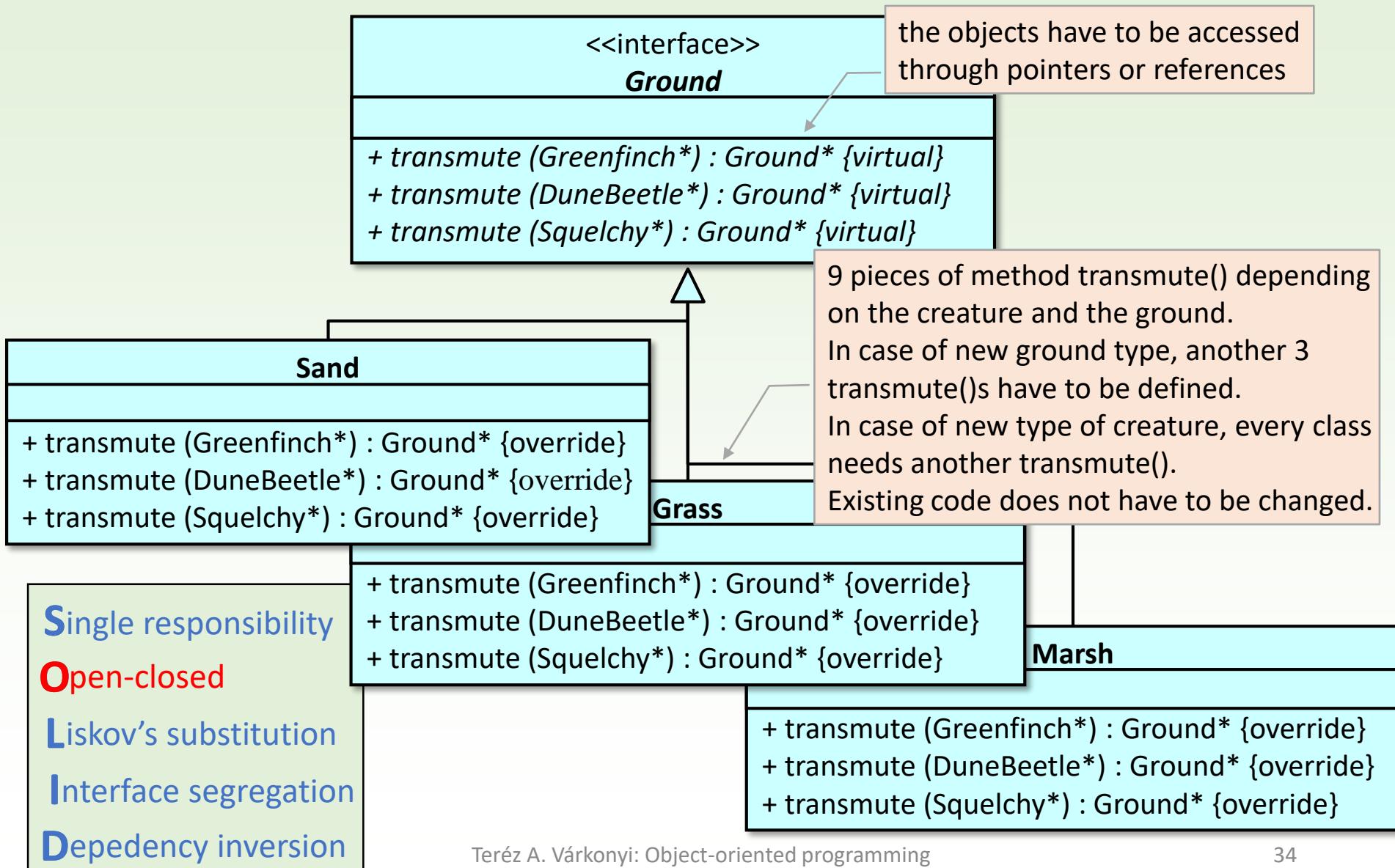
Critiques:

- 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

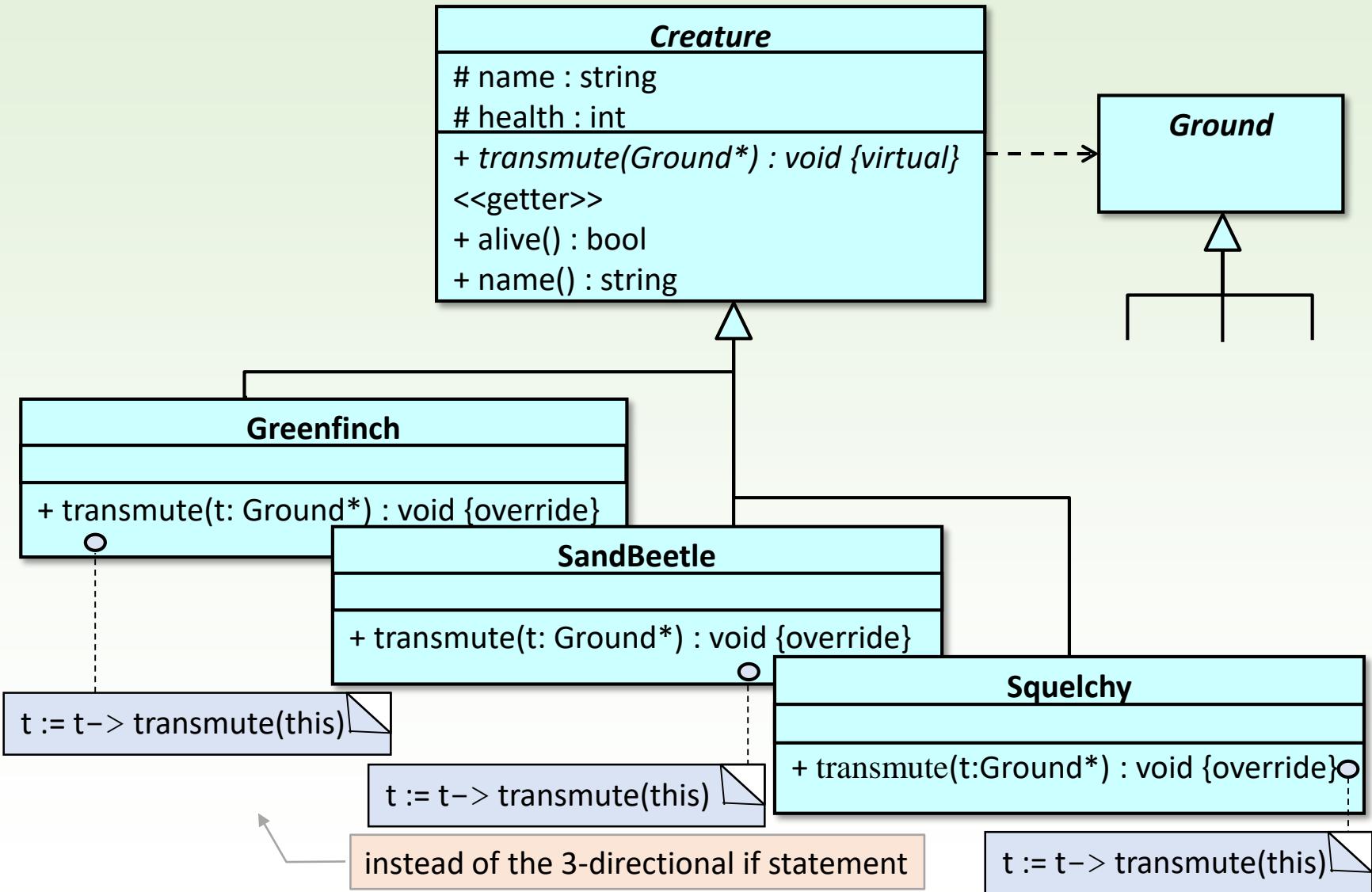
```
void Squelchy::transmute(int &ground) {
    if (alive()) {
        switch(ground) {
            case 0: _health -=5; break;
            case 1: _health -=2; ground = 2; break;
            case 2: _health +=6; break;
        }
    }
}
```

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

Inheritance of grounds

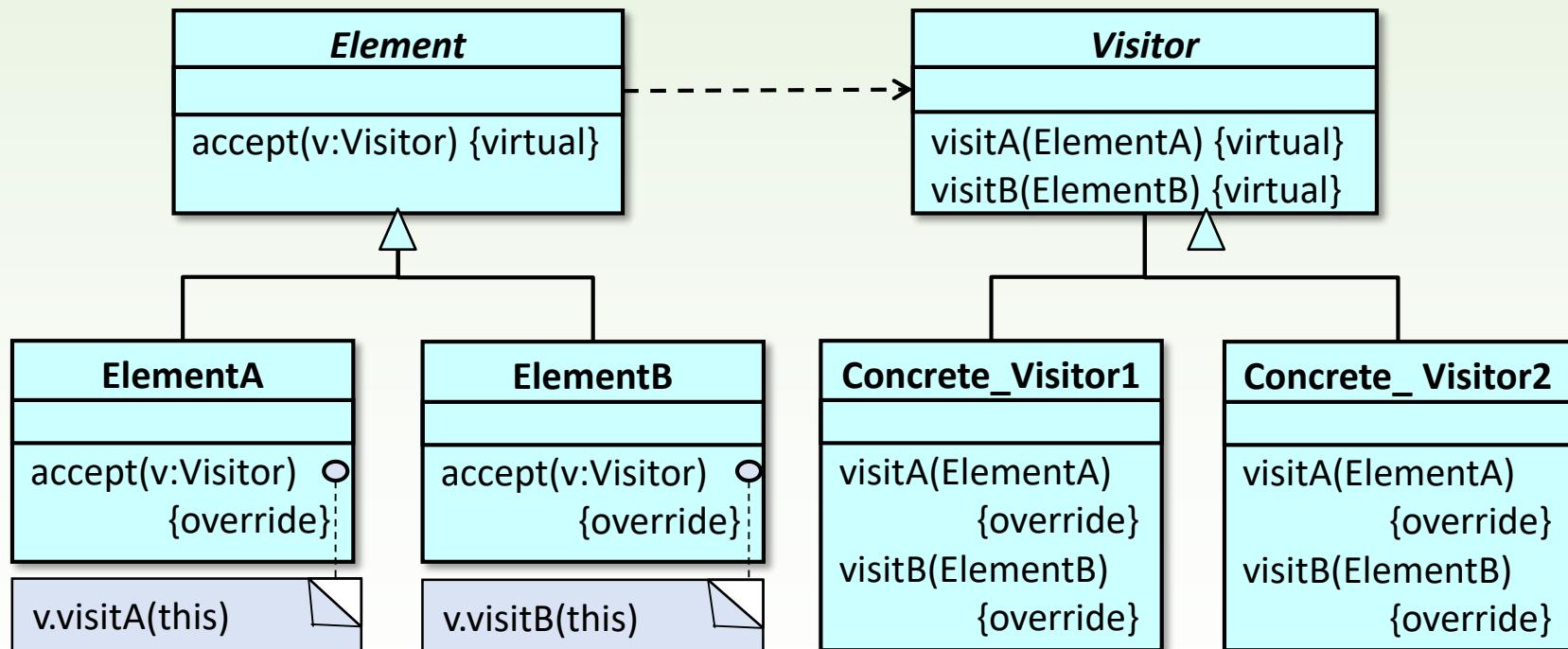


Class diagram of creatures (again)



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 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); }  
};
```

```
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

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();
}
```

```
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;
}
```

instead of
new Sand
new Grass
new Marsh

ground.cpp

Population

```
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

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

main.cpp

Packages

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

```
#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 { ... }
```

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

```
class Creature { ... };
```

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

creature.h

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

...
```

ground.h

ground.cpp

Design patterns II. (Bridge, Iterator, Factory)

Set and its iteration

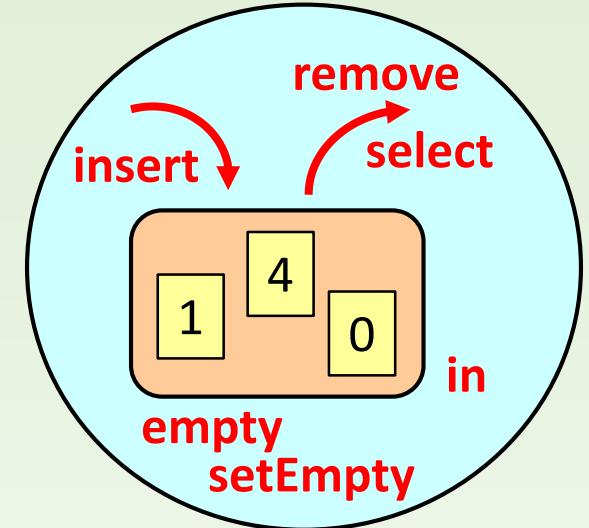
Class-templates

1st task

Write a program to create **sets of integers**.

- Representation of a set depends on if there is an upper bound (**max**) for its items or not.
 - Usually, items are stored in a **sequence**.
 - Specifically, a the set os represented by a **boolean array**, where a number is in the set if the numberth item of the set is *true*.
- It is worthy to hide the representation from the user: when a set is created, the user decides if there is an upper bound for the items, or not, but the program does not say anything about the different representations (the user does not need to know them).

Two representations



Array

	0	1	...	4	max
vect	true	true	false	false	true
size	3				

Sequence

	1	...	size
seq	1	4	0

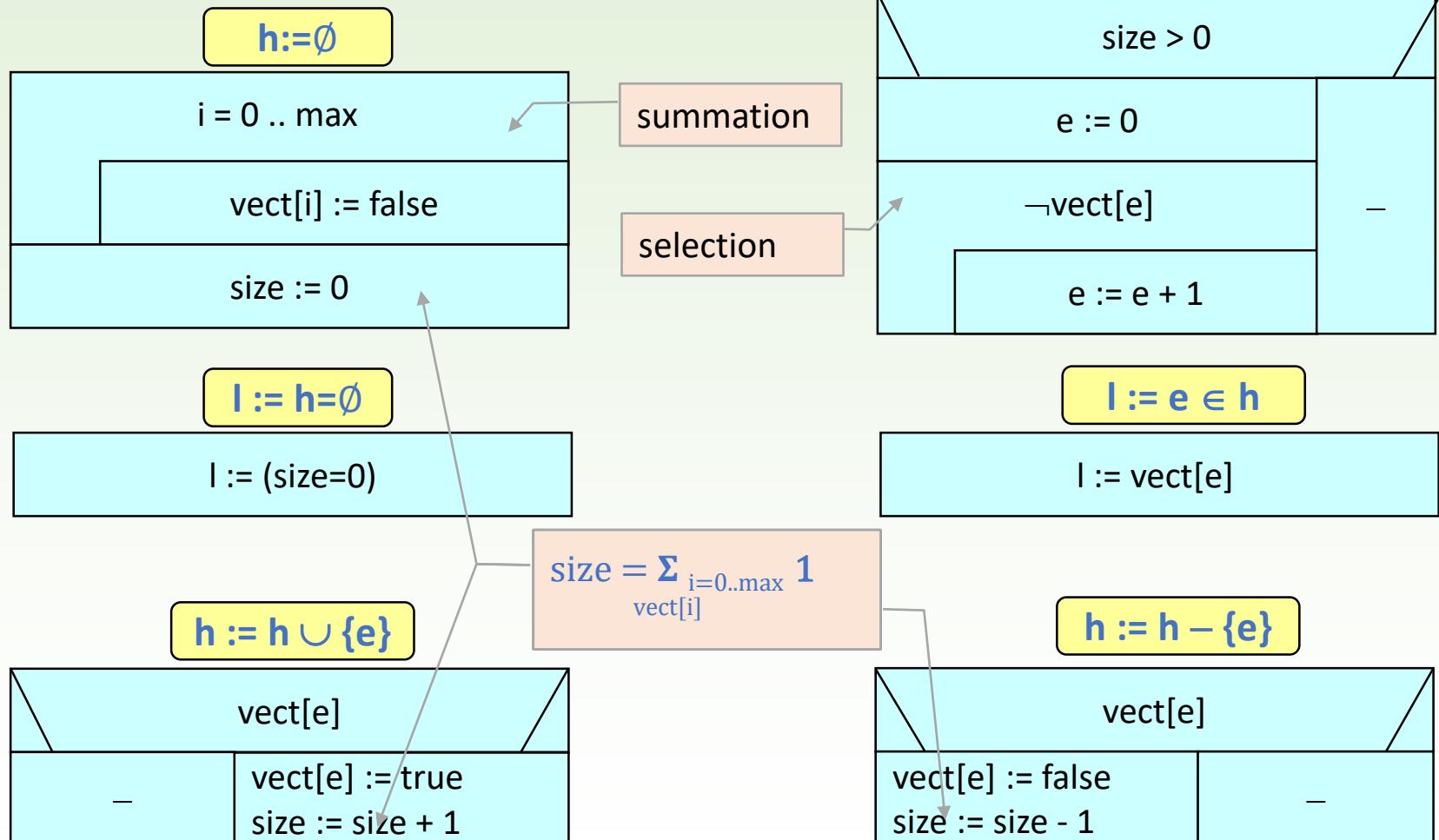
1. Fixed-size array and the number of the stored items.
2. The computational time of the operations is mainly constant, except select and setEmpty (they are linear).

1. Dynamic sequence.
2. The computational time of the operations is mainly linear, except select and setEmpty (they are constant).

Array representation

Type-specification	
<p>set([0..max])</p> <p>Sets containing natural numbers between 0 and max.</p> <p>(Formally: power set $\mathcal{P}\{0..max\}$)</p>	<p>setEmpty $h := \emptyset$ $h: \text{set}([0..max])$</p> <p>insert $h := h \cup \{e\}$ $h: \text{set}([0..max]), e: \mathbb{N}$</p> <p>remove $h := h - \{e\}$ $h: \text{set}([0..max]), e: \mathbb{N}$</p> <p>select $e := \text{mem}(h)$ $h: \text{set}([0..max]), e: \mathbb{N}$</p> <p>empty $l := h = \emptyset$ $h: \text{set}([0..max]), l: \mathbb{L}$</p> <p>in $l := e \in h$ $h: \text{set}([0..max]), e: \mathbb{N}, l: \mathbb{L}$</p>
<p>vect : $\mathbb{L}^{0..max}$</p> <p>size : \mathbb{N}</p> <p>invariant: $\text{size} = \sum_{i=0..max} 1$</p>	programs of the operations
type-realization	

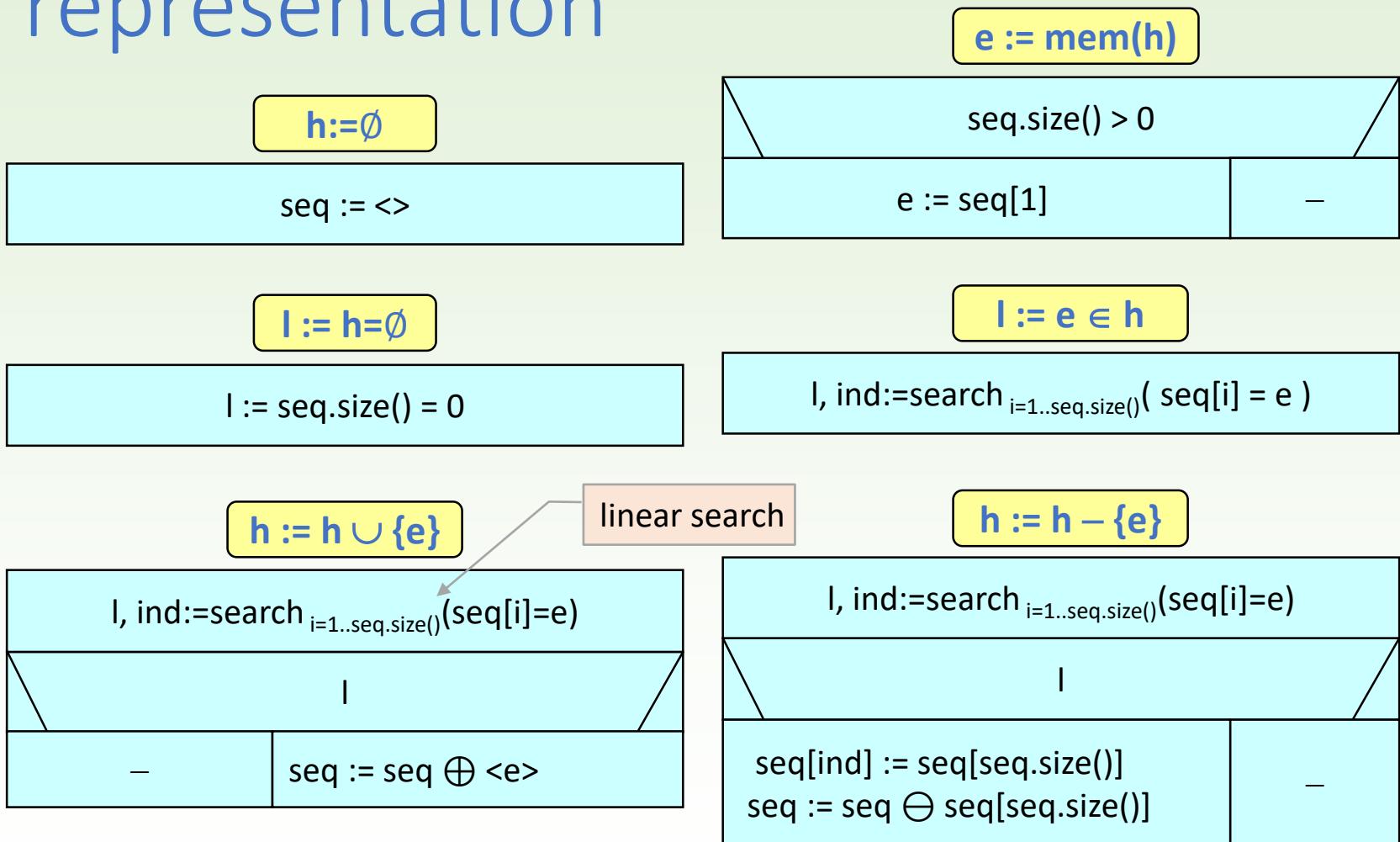
Operations in array representation



Sequence representation

type-specification		operations	
type values	Sets containing natural numbers. (Formally: $\{ h \in \mathcal{P}(\mathbb{N}) \mid h < \infty \}$, finite items of $\mathcal{P}(\mathbb{N})$ power set.)	setEmpty insert remove select empty in	
representation	seq : \mathbb{N}^*	vector<int> in C++	implementation
programs of the operations			type-realization

Operations in sequence representation

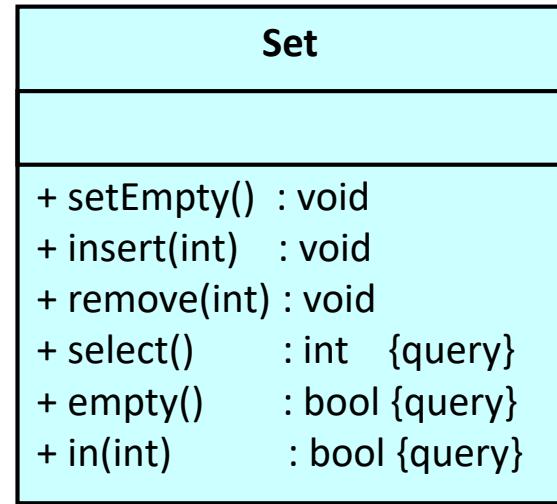


Public part of class Set

```
class Set
{
public:

    void setEmpty();
    void insert(const int &e);
    void remove(const int &e);
    int select() const;
    bool empty() const;
    bool in(int e) const;

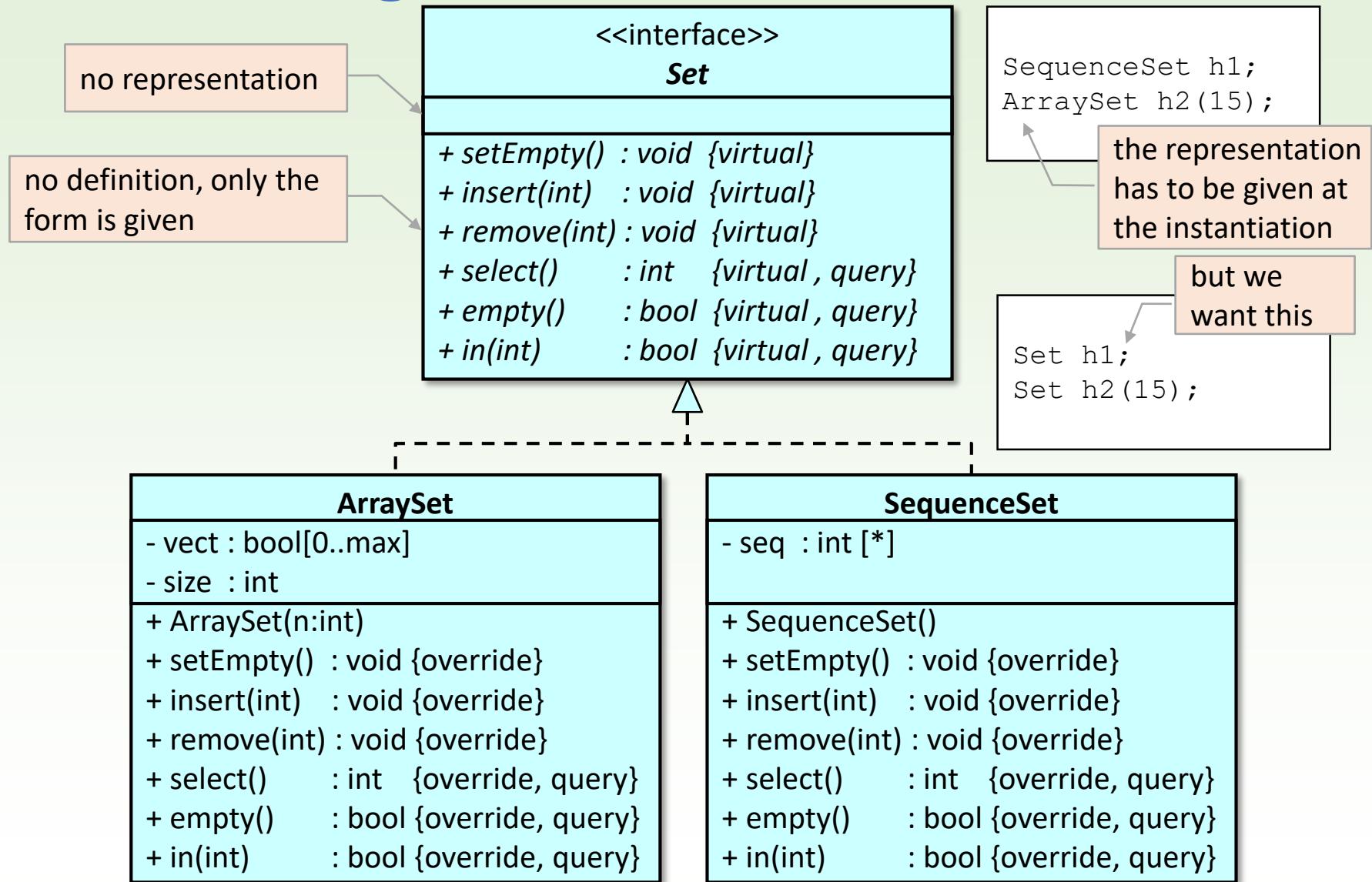
private:
    ...
};
```



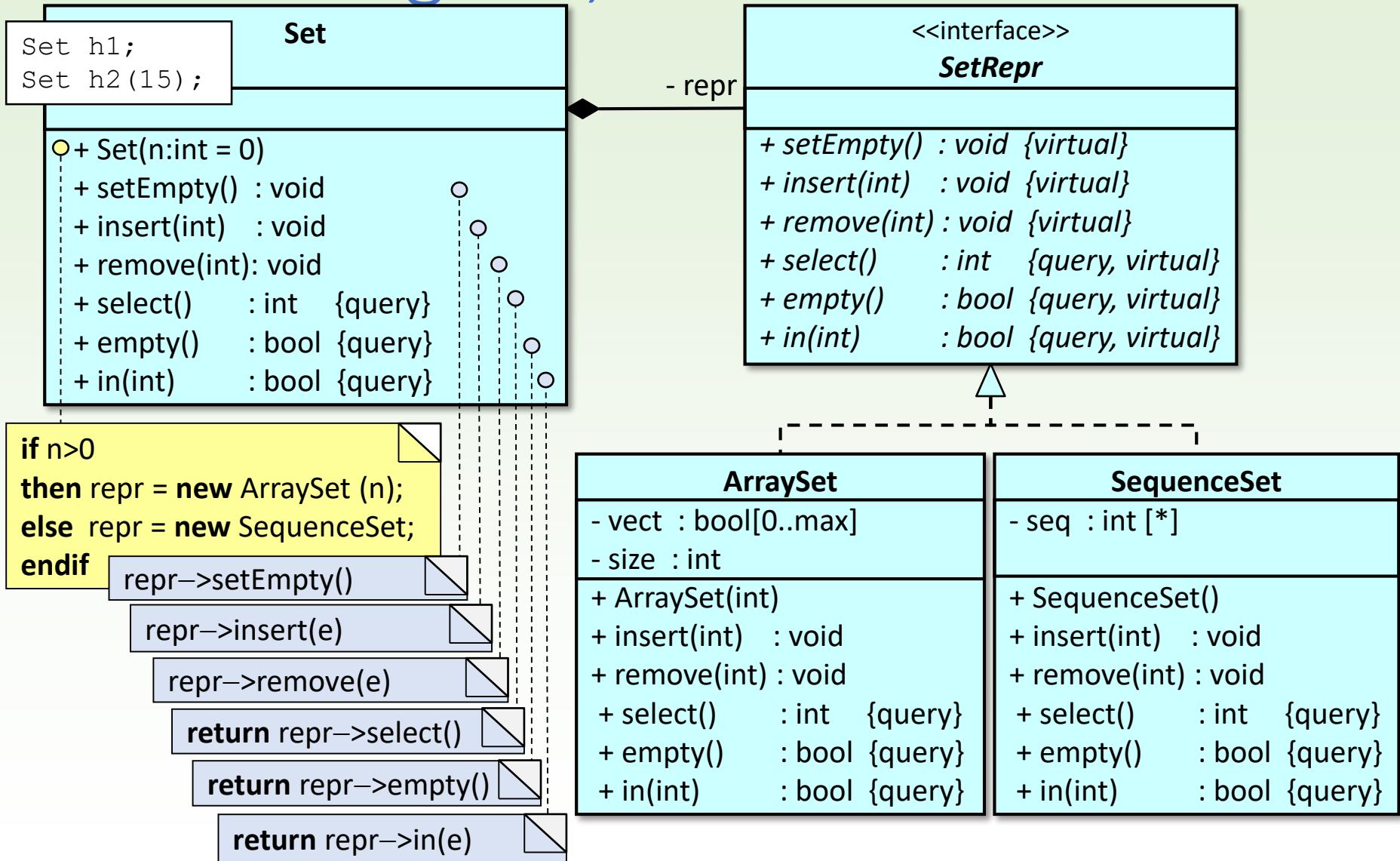
How to implement both representations ??

set.h

Class diagram, 1st version

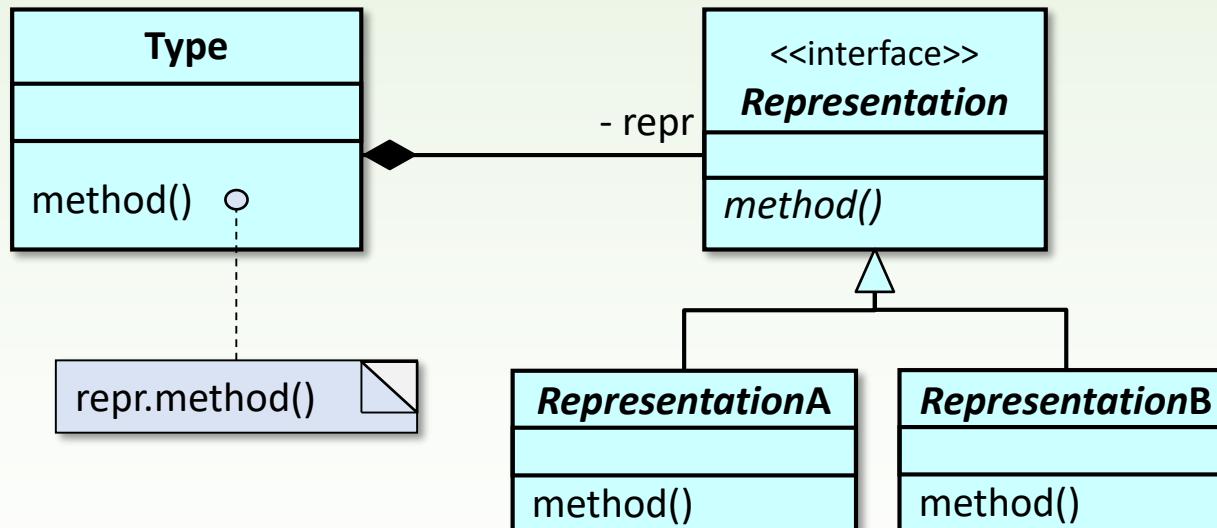


Class diagram, 2nd version



Bridge design pattern

- The representation of a class is separated from the class itself, so that it may be changed flexibly.



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

Class Set, inline way

```
#include "setrepr.h"
#include "array_set.h"
#include "sequence_set.h"

class Set {
public:
    Set(int n = 0) {
        if (n>0) _repr = new ArraySet(n);
        else      _repr = new SequenceSet;
    }
    ~Set() { delete _repr; }
    void setEmpty()          { _repr->setEmpty(); }
    void insert(int e)       { _repr->insert(e); }
    void remove(int e)       { _repr->remove(e); }
    int select() const { return _repr->select(); }
    bool empty() const { return _repr->empty(); }
    bool in(int e) const { return _repr->in(e); }
private:
    SetRepr *_repr;
    Set(const Set& h);
    Set& operator=(const Set& h);
};
```

Set
+ Set(n:int = 0)
+ setEmpty() : void
+ insert(int) : void
+ remove(int) : void
+ select() : int {query}
+ empty() : bool {query}
+ in(int) : bool {query}

The default copy constructor and assignment operator would not work well.
If this is private, they cannot be used.
Later on, they can be defined and made public.

set.h

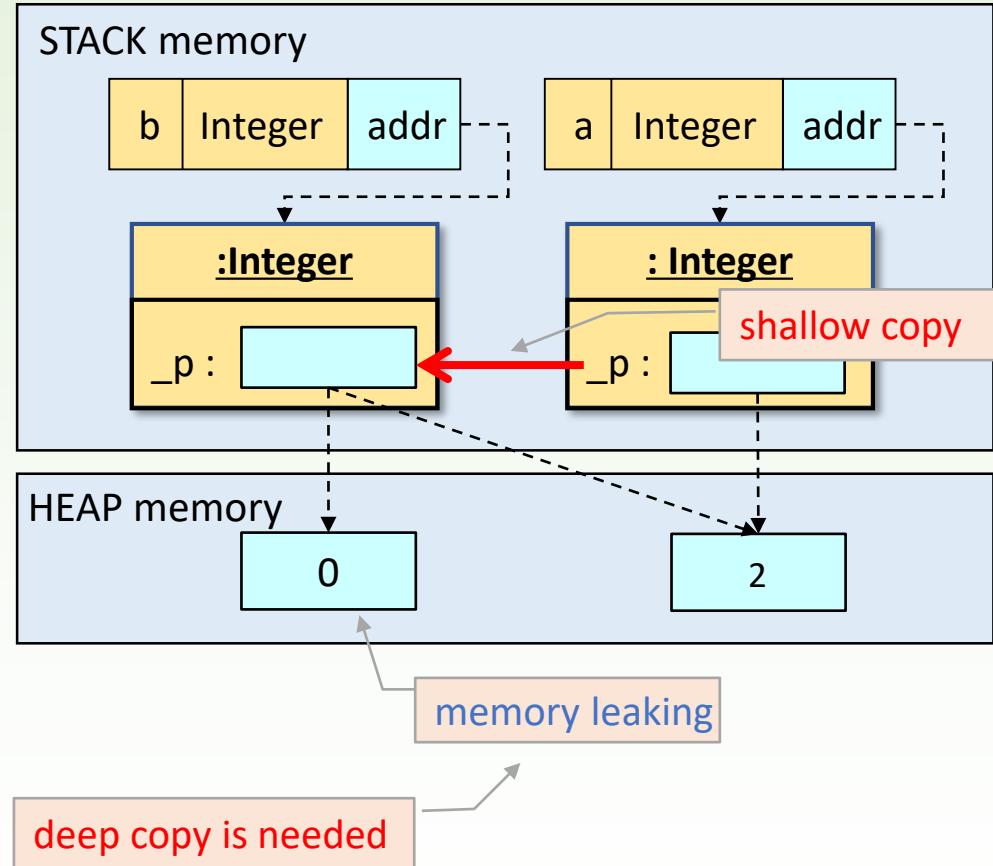
Side note: default copy and assignment

```
class Integer {  
public:  
    Integer() { _p = new int; *_p = 0; }  
    ~Integer() { delete _p; }  
    void addl(int i) { *_p += i; }  
    int get() const { return *_p; }  
private:  
    int *_p;  
};
```

```
Integer a, b;  
b = a; assignment operator
```

```
b.add(2);  
cout << a.get();
```

```
copy constructor  
Integer b = a;
```



Interface of the representation

```
<<interface>>
```

```
SetRepr
```

```
+ setEmpty() : void {virtual}  
+ insert(int) : void {virtual}  
+ remove(int) : void {virtual}  
+ select()    : int   {virtual, query}  
+ empty()     : bool  {virtual, query}  
+ in(int)      : bool  {virtual, query}
```

```
class SetRepr  
{  
public:  
    virtual void setEmpty() = 0;  
    virtual void insert(int e) = 0;  
    virtual void remove(int e) = 0;  
    virtual int select() const = 0;  
    virtual bool empty() const = 0;  
    virtual bool in(int e) const = 0;  
    virtual ~SetRepr() {}  
};
```

```
setrepr.h
```

Array representation

```
#include "setrepr.h"
#include <vector>

class ArraySet : public SetRepr {
public:
    ArraySet (int n) : _vect(n+1), _size(0) {
        setEmpty();
    }
    void setEmpty()      override;
    void insert(int e)   override;
    void remove(int e)   override;
    int select() const override;
    bool empty() const override;
    bool in(int e) const override;
private:
    std::vector<bool> _vect;
    int _size;
};
```

array_set.h

<<interface>>

SetRepr

```
+ setEmpty() : void {virtual}
+ insert(int) : void {virtual}
+ remove(int) : void {virtual}
+ select()    : int {virtual, query}
+ empty()     : bool {virtual, query}
+ in(int)     : bool {virtual, query}
```



ArraySet

```
- vect : bool[0..max]
- size : int
```

```
+ ArraySet(int)
+ setEmpty() : void
+ insert(int) : void
+ remove(int) : void
+ select()    : int {query}
+ empty()     : bool {query}
+ in(int)     : bool {query}
```

Exceptions

```
#include <exception>
#include <sstream>

class EmptySetException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Empty set";
    }
};

class IllegalElementException : public std::exception {
private:
    int _e;
public:
    IllegalElementException(int e) : _e(e) {}
    const char* what() const noexcept override {
        std::ostringstream os;
        os << "Illegal element: " << _e;
        std::string str = os.str();
        char * msg = new char[str.size() + 1];
        std::copy(str.begin(), str.end(), msg);
        msg[str.size()] = '\0';
        return msg;
    }
};
```

setrepr.h

Methods of ArraySet

```
int ArraySet::select() const
{
    if(_size==0) throw EmptySetException();
    int e;
    for(e=0; !_vect[e]; ++e);
    return e;
}
```

```
bool ArraySet::empty() const
{
    return _size==0;
}
```

```
bool ArraySet::in(int e) const
{
    if(e<0 || e>int(vect.size())-1) throw IllegalElementException(e);
    return _vect[e];
}
```

```
Set h(100);
try {
    h.insert(101);
} catch(exception &ex) {
    cout << ex.what() << endl;
}
```

exception instantiation
and throw

exception instantiation
and throw

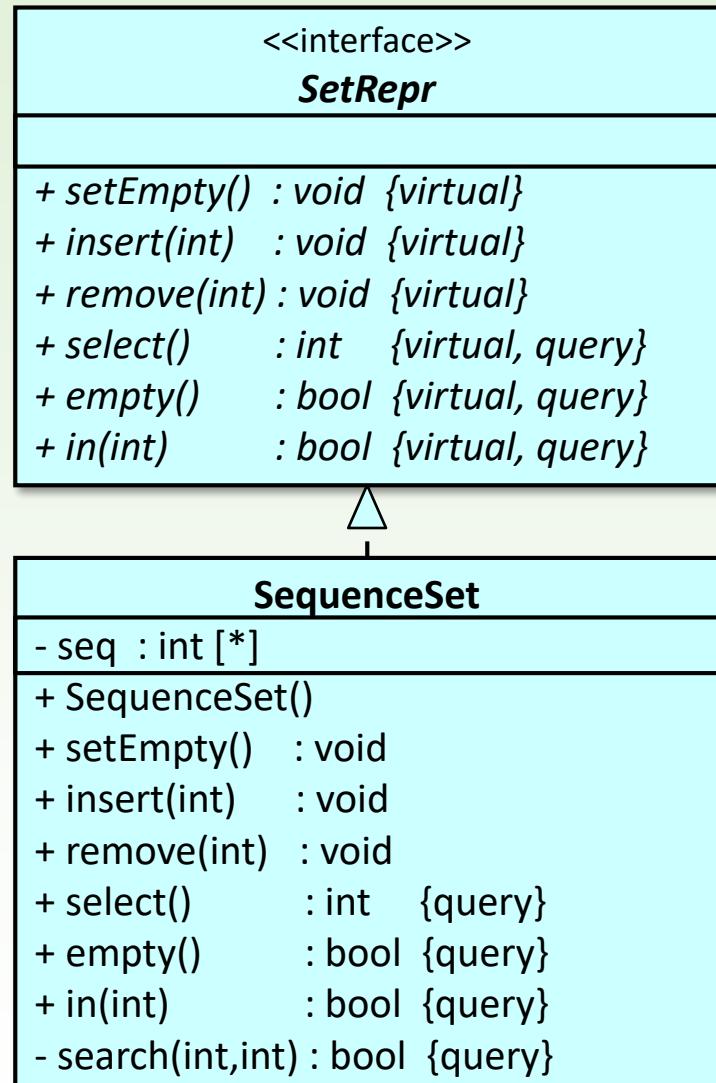
array_set.cpp

Sequence representation

```
#include "setrepr.h"
#include <vector>

class SequenceSet : public SetRepr{
public:
    SequenceSet () { _seq.clear(); }
    void setEmpty() override;
    void insert(int e) override;
    void remove(int e) override;
    int select() const override;
    bool empty() const override;
    bool in(int e) const override;
private:
    std::vector<int> _seq;
    bool search(int e,
                unsigned int &ind) const;
};
```

sequence_set.h



2nd task

Search for an item in a set of natural numbers which is greater than at least three other items in the set. (The search is obviously unsuccessful if there are at last 3 items in the set.)

- A possible solution would be a [linear search](#) for an item, where a [counting](#) determines the number smaller items in the set.
- The [enumeration](#) of the items is used for both of the algorithmic patterns.

Specification

A : h:set(\mathbb{N}), l: \mathbb{L} , n: \mathbb{N}

Pre: $h = h_0$

Post: $l, n = \text{SEARCH}_{e \in h_0} (\text{NoLess}(h_0, e) \geq 3)$

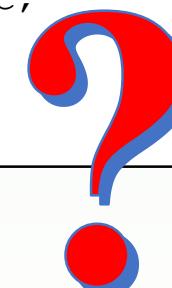
$$\text{NoLess}(h_0, e) = \sum_{\substack{u \in h_0 \\ e > u}} 1$$

```
bool l = false;
int n;
for( ; !l && !h.empty(); h.remove(n)) {
    n = h.select();
    int c = 0;
    for( ; !h.empty(); h.remove(h.select())) {
        if(n > h.select()) ++c;
    }
    l = c >= 3;
}
```

standard enumeration of a set:
first() ~ -
next() ~ remove(current())
end() ~ empty()
current() ~ select()

Not a good solution:

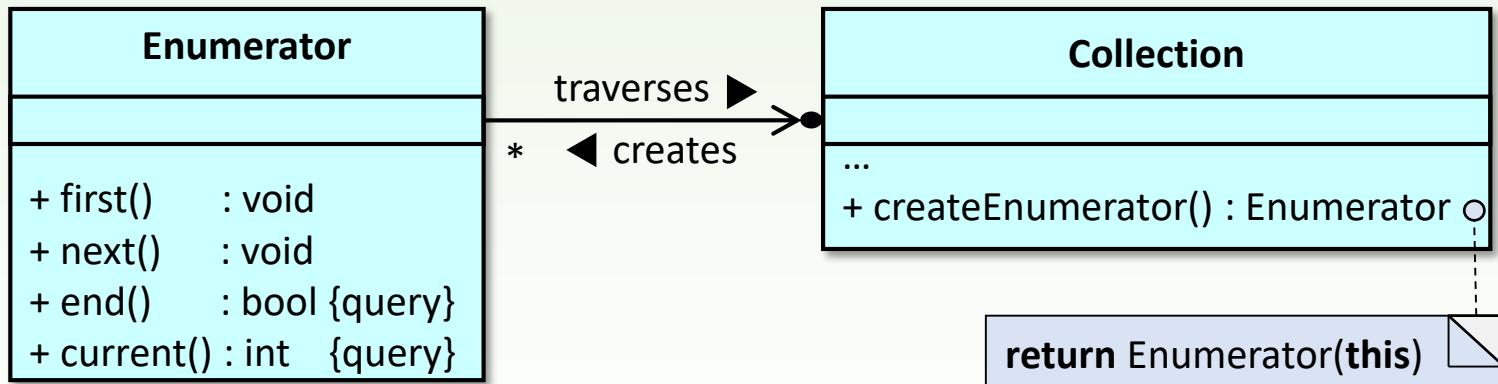
- the standard enumeration modifies the set, the inner loop removes all of the items
- the two enumerations are not independent, the two enumerations are interlocked



main.cpp

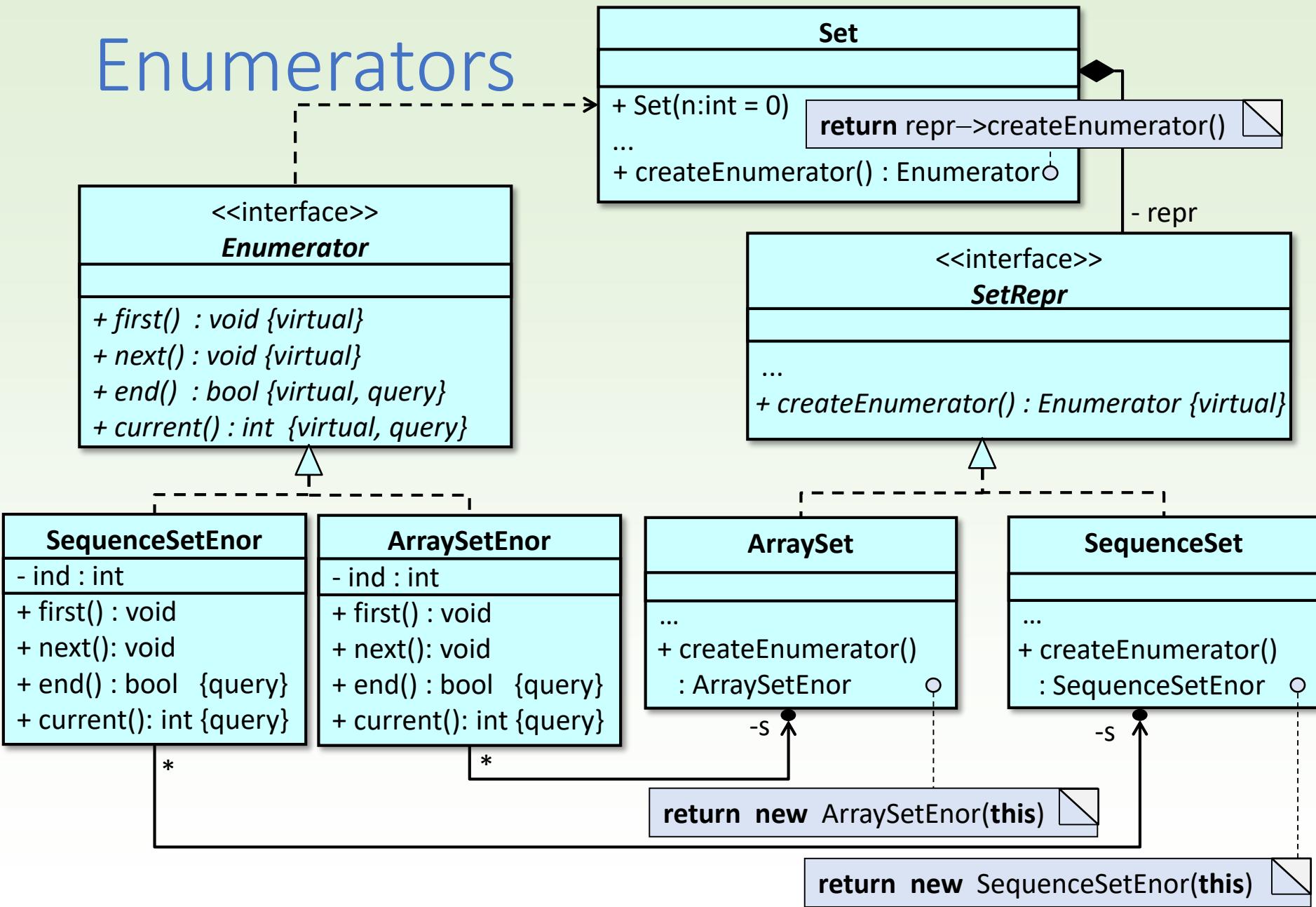
Iterator design pattern

- ❑ The enumeration (traversal) of a collection is done by an independent object (enumerator) which can access the collection (refers to it or to a constant copy). The enumerator object is created by the collection.



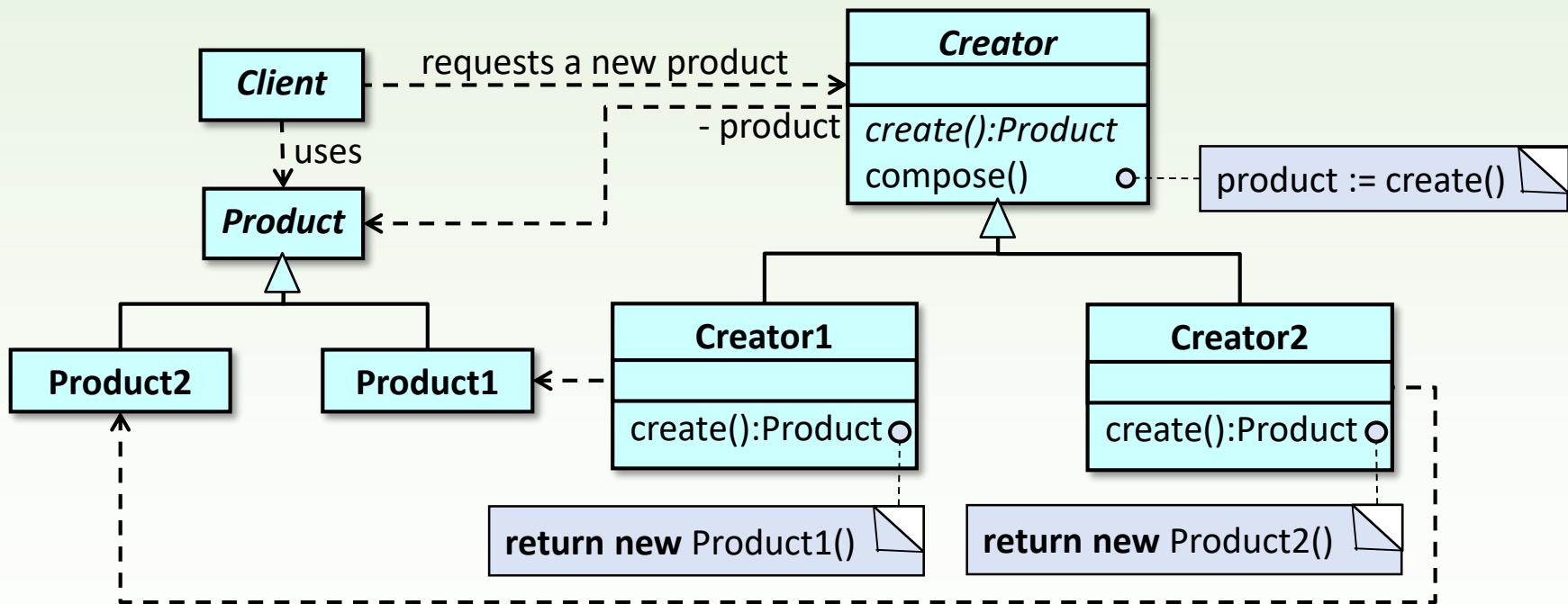
Design patterns are class diagram patterns that help object-oriented modeling. They play a significant role in reusability, modifiability, and ensuring efficiency.

Enumerators



Factory method design pattern

- The client does not know what kind of product-object it needs. It transfers the responsibility to one of the supporting subclasses.



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

Main program

```
Set h;  
// Reading data  
...  
bool l = false;  
int n;  
Enumerator* enor1 = h.createEnumerator();  
for(enor1->first(); !l && !enor1->end(); enor1->next()) {  
    n = enor1->current();  
    int c = 0;  
    Enumerator* enor2 = h.createEnumerator();  
    for(enor2->first(); !enor2->end(); enor2->next()) {  
        if(n > enor2->current()) ++c;  
    }  
    l = (c >= 3);  
}  
  
if (l) cout << "The number you are looking for: " << n << endl;  
else cout << "There is no such number.\n";
```

in the background:
repr->createEnumerator()
return new SequenceSetEnor(this)

main.cpp

Enumerator of SequenceSet

```
class SequenceSet{
public:
    ...
class SequenceSetEnor : public Enumerator{
public:
    SequenceSetEnor(SequenceSet *h) : _s(h) { }
    void first()           override { _ind = 0; }
    void next()            override { ++_ind; }
    bool end()             const override { return _ind == _s->_seq.size(); }
    int current() const override { return _s->_seq[_ind]; }
private:
    SequenceSet *_s;
    unsigned int _ind;
};

Enumerator* createEnumerator() override{
    return new SequenceSetEnor(this);
}
};
```

Enumeration of the items is realized by enumeration of the sequence which represents the set.

<<interface>>

Enumerator

```
+ first()   : void {virtual}
+ next()    : void {virtual}
+ end()     : bool {virtual,query}
+ current() : int {virtual,query}
```

SequenceSetEnor

- s : SequenceSet

- ind : int

```
+ first() : void
+ next(): void
+ end() : bool {query}
+ current(): int {query}
```

sequence_set.h

Enumerator of ArraySet

```
class ArraySet{  
public:
```

Enumeration of the items is realized by enumeration of the indexes of the array where the corresponding value is true.

```
...  
  
class ArraySetEnor : public Enumerator{  
public:  
    ArraySetEnor(ArraySet *h) : _s(h) {}  
    void first() override { _ind = -1; next(); }  
    void next() override {  
        for(++_ind; _ind<_s->vect.size() && !_s->vect[_ind]; ++_ind);  
    }  
    bool end() const override { return _ind == _s->vect.size(); }  
    int current() const override { return _ind; }  
private:  
    ArraySet *_s;  
    unsigned int _ind;  
};
```

```
Enumerator* createEnumerator() override{  
    return new ArraySetEnor(this);  
}  
};
```

<<interface>>

Enumerator

```
+ first() : void {virtual}  
+ next() : void {virtual}  
+ end() : bool {virtual,query}  
+ current() : int {virtual,query}
```

ArraySetEnor

```
- s : ArraySet  
- ind : int  
+ first() : void  
+ next(): void  
+ end() : bool {query}  
+ current(): int {query}
```

array_set.h

3rd task

Make the enumeration secure

- Problem: the enumeration might be wrong if the set is changed during the enumeration.
- Critical operations: `setEmpty()`, `insert()`, `remove()`, assignment operator, destructor.
- Solution: Do not let the critical operations to be run.

```
Set h;  
...  
Enumerator * enor = h.createEnumerator();  
for(enor->first(); !enor->end(); enor->next()) {  
    int e = enor->current();  
    h.remove(e);  
}
```

If sequence representation is used, deleting the current item causes an error.



main.cpp

Locking the operations

```
class Set {  
public:
```

```
...  
void Set::remove(int e)  
{  
    if(_repr->getEnumCount() != 0) throw UnderTraversalException();  
    _ref->remove(e);  
}  
~Set() {
```

```
    if(_repr->getEnumCount() != 0) throw UnderTraversalException();  
    delete repr;  
}
```

```
};
```

```
class UnderTraversalException : public std::exception {  
public:  
    const char* what() const noexcept override {  
        return "Under traversal";  
    }  
};
```

setrepr.h

The critical operation throws an exception if it is under traversal.

```
class SetRepr {  
public:
```

```
    SetRepr(): _enumeratorCount(0) {}
```

```
...
```

```
    int getEnumCount() const { return _enumeratorCount; }
```

```
protected:
```

```
    int _enumeratorCount;
```

```
};
```

set.h

this is not an interface any more, but an abstract class

number of the active enumerators

setrepr.h

ArraySet

```
class ArraySet : public SetRepr {
public:
    ArraySet(int n): SetRepr(), _vect(n+1), _size(0) {
        setEmpty();
    }
    ...
}

class ArraySetEnor : public Enumerator{
public:
    ArraySetEnor(ArraySet *h): _s(h)
    { ++(_s->enumeratorCount); }
    ~ArraySetEnor() { --(_s->enumeratorCount); }
    ...
};

Enumerator * createEnumerator() override {
    return new ArraySetEnor(this);
}
};
```

sets the counter of the enumerators to zero

When a new enumerator is instantiated for the ArraySet object, its enumerator-counter is increased.

When the enumerator is destroyed, the enumerator-counter is decreased.

array_set.h

SequenceSet

```
class SequenceSet : public SetRepr{  
public:  
    SequenceSet(): SetRepr() {}  
    ...  
    class SequenceSetEnor : public Enumerator{  
        public:  
            SequenceSetEnor(SequenceSet *h): _s(h)  
            { ++(_s->_enumeratorCount); }  
            ~ SequenceSetEnor() { --(_s->_enumeratorCount); }  
            ...  
    };  
    Enumerator* createEnumerator() override {  
        return new SequenceSetEnor(this);  
    }  
};
```

sets the counter of the enumerators to zero

When a new enumerator is instantiated for the SequenceSet object, its enumerator-counter is increased.

When the enumerator is destroyed, the enumerator-counter is decreased.

sequence_set.h

4th task

- ❑ Create **class templates**, where the type of the items (in the set) can be given as an input parameter.
- ❑ Remark: in case of array representation, only natural numbers with upper bound can be stored, the template parameter has to be **int**.

```
Set<int>    h1(100);  
Set<int>    h2;  
Set<string> h3;
```

in compilation time, the class template is instantiated as a class,
in runtime, the class is instantiated as an object

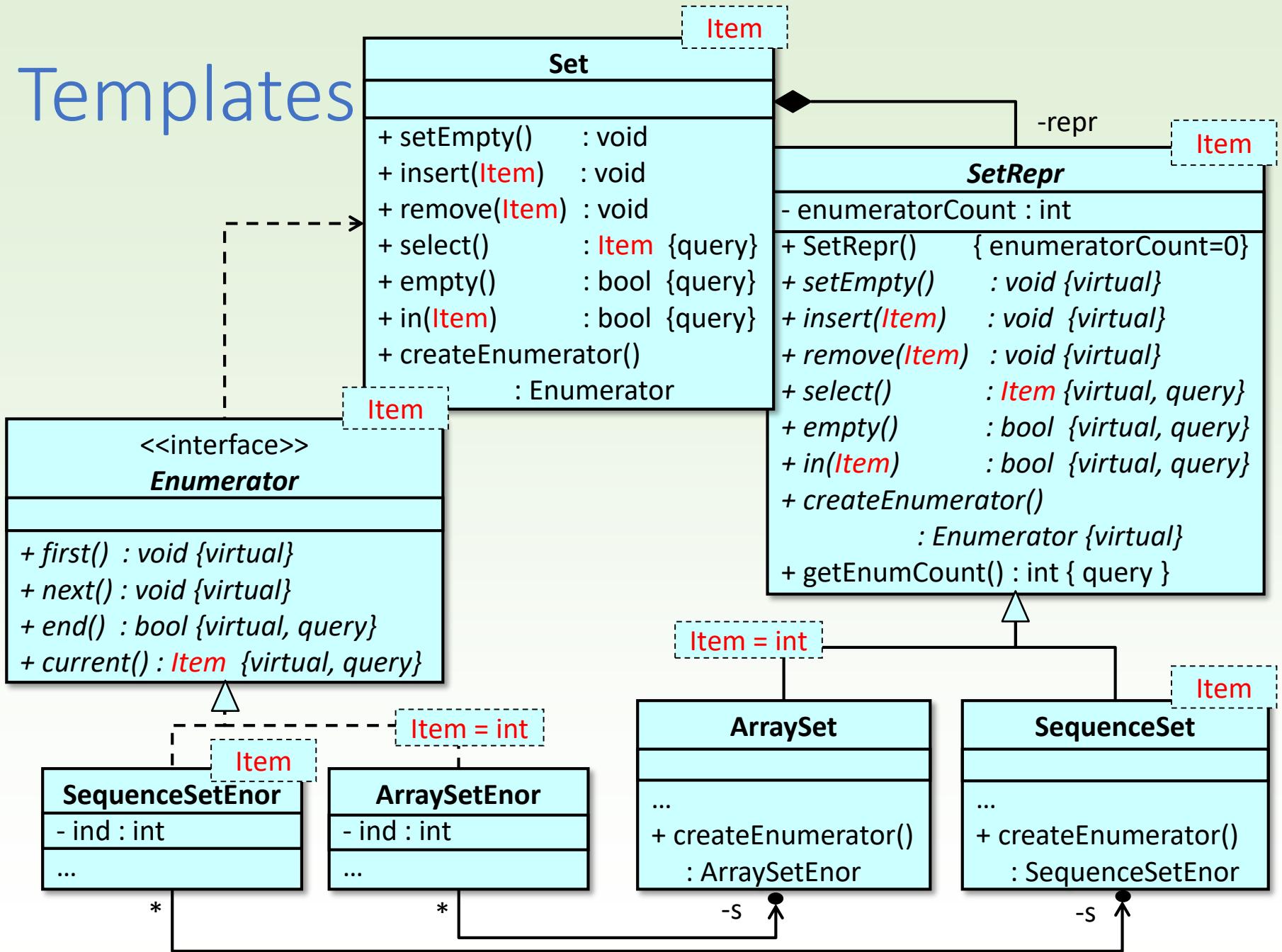
```
h1.insert(12);  
h2.insert(123);  
h3.insert("apple");
```

The enumerator classes are instances of class templates, too,
the type parameter of which has to match the type of the set.

```
Enumerator<int> *enor1 = h1.createEnumerator();  
Enumerator<string> *enor2 = h3.createEnumerator();
```

main.cpp

Templates



Interface-template of the representation

indicates that this is a template and gives the parameters of the template

```
template <typename Item>
class SetRepr {
public:
    SetRepr(): _enumeratorCount(0) {}
    virtual ~SetRepr() {};
    virtual void setEmpty() = 0;
    virtual void insert(Item e) = 0;
    virtual void remove(Item e) = 0;
    virtual Item select() const = 0;
    virtual bool empty() const = 0;
    virtual bool in(Item e) const = 0;
    virtual Enumerator<Item>* createEnumerator() = 0;
    int getEnumCount() const { return _enumeratorCount; }
protected:
    int _enumeratorCount;
};
```

Item	<i>SetRepr</i>
	# enumeratorCount : int
+ SetRepr()	{ enumeratorCount=0}
+ <i>setEmpty()</i>	: void {virtual}
+ <i>insert(Item)</i>	: void {virtual}
+ <i>remove(Item)</i>	: void {virtual}
+ <i>select()</i>	: <i>Item</i> {virtual, query}
+ <i>empty()</i>	: bool {virtual, query}
+ <i>in(Item)</i>	: bool {virtual, query}
+ <i>createEnumerator()</i>	: Enumerator {virtual}
+ <i>getEnumCount()</i>	: int {query }

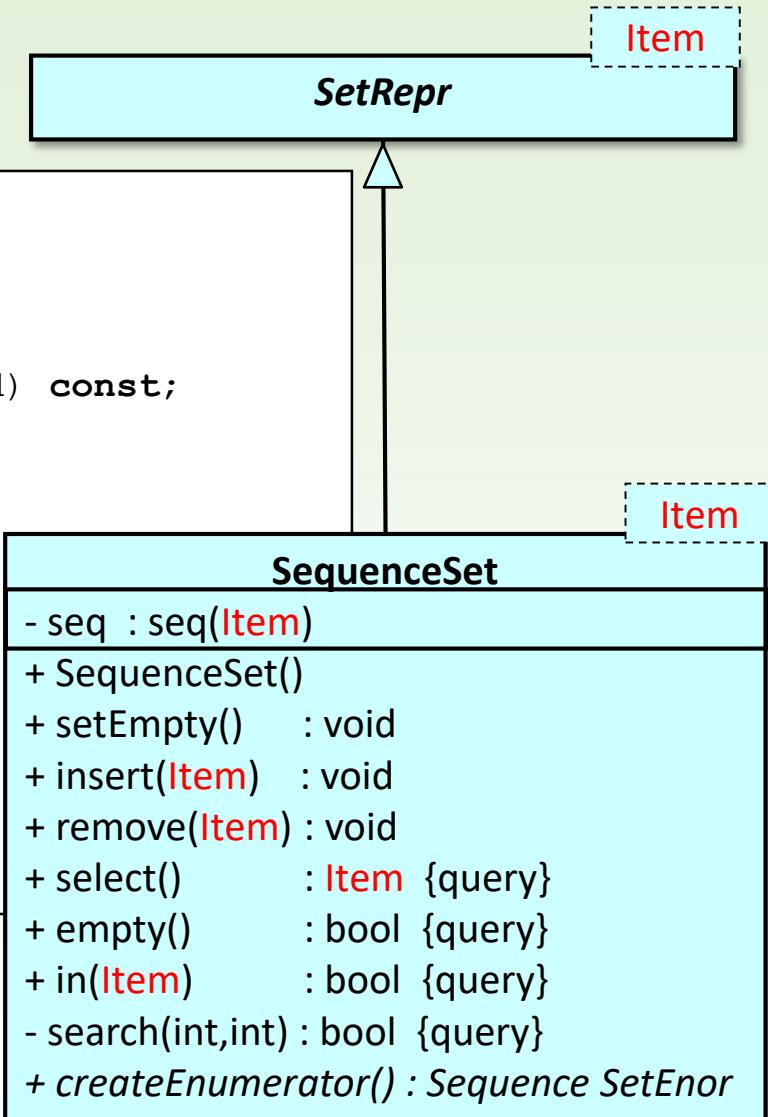
setrepr.h

Template of SequenceSet

```
template <typename Item>
class SequenceSet : public SetRepr<Item>{
private:
    std::vector<Item> _seq;
    bool search(Item e, unsigned int &ind) const;
public:
    SequenceSet () { _seq.clear(); }
    void setEmpty() override;
    void insert(Item e) override;
    void remove(Item e) override;
    Item select() const override;
    bool empty() const override;
    bool in(Item e) const override;
    ...
};
```

sequence_set.hpp

Class template definition (.h) and
template-method definitions (.cpp)
go to the same file.



Methods of template SequenceSet

```
...
template <typename Item>
void SequenceSet<Item>::setEmpty() { _seq.clear(); }

template <typename Item>
void SequenceSet<Item>::insert(int e)
{
    unsigned int ind;
    if(!search(e,ind)) _seq.push_back(e);
}

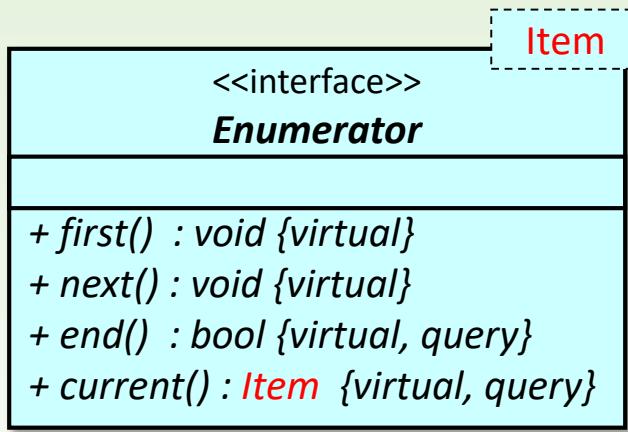
template <typename Item>
void SequenceSet<Item>::remove(int e)
{
    unsigned int ind;
    if(search(e,ind)) {
        _seq[ind] = _seq[_seq.size()-1];
        _seq.pop_back();
    }
}

template <typename Item>
int SequenceSet<Item>::select() const
{
    if(_seq.size()==0) throw EmptySetException();
    return _seq[0];
}
```

Not just a class, but a function may be template, too.
Specifically, methods of a class template are templates, too.

sequence_set.hpp

Interface-template of the enumerator



```
template <typename Item>
class Enumerator {
public:
    virtual void first() = 0;
    virtual void next() = 0;
    virtual bool end() const = 0;
    virtual Item current() const = 0;
    virtual ~Enumerator() {};
```

enumerator.h

Template of SequenceSetEnor

```
template <typename Item>
class SequenceSet : public SetRepr<Item>{
public:
    ...
    because of the embedding, this is a
    template with parameter Item: indicating
    that this is a template is not necessary
    class SequenceSetEnor : public Enumerator<Item>{
public:
    SequenceSetEnor(SequenceSet<Item> *h) : _s(h) {}
    void first()           override { _ind = 0; }
    void next()            override { ++_ind; }
    bool end()             const override { return _ind == _s->_seq.size(); }
    Item current()         const override { return _s->_seq[_ind]; }
private:
    SequenceSet<Item> *_s;
    unsigned int _ind;
};

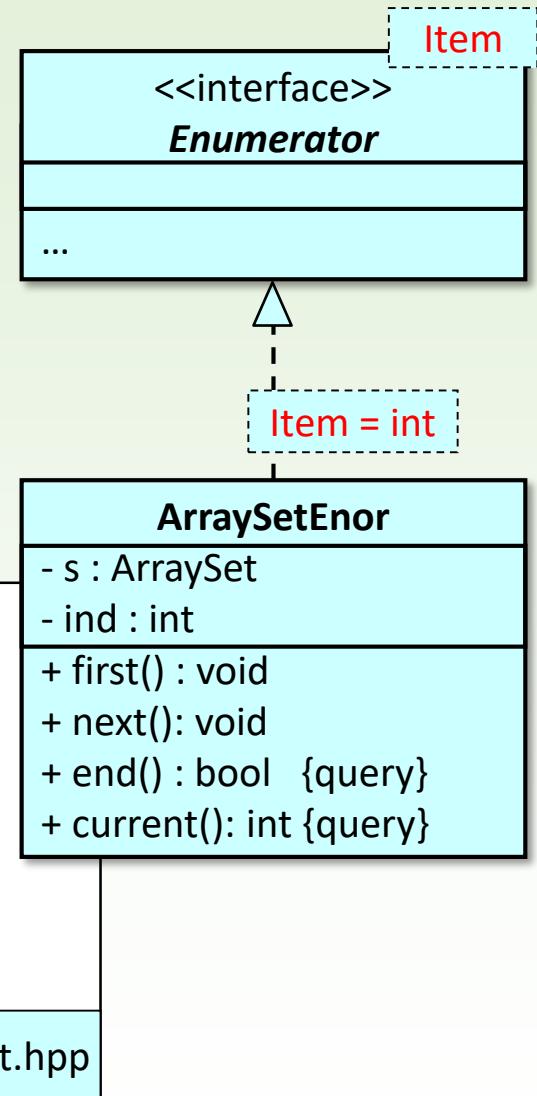
Enumerator<Item>* createEnumerator() override
    return new SequenceSetEnor<Item> (this);
};

Item
```

Diagram illustrating the class hierarchy and interface:

- Item** (dashed border): A placeholder type.
- <<interface>> Enumerator** (solid border): An interface with methods `first()`, `next()`, `end()`, and `current()`.
- SequenceSetEnor** (solid border): A concrete class that implements the `Enumerator` interface. It has a private member `_s` of type `SequenceSet<Item>` and a private member `_ind` of type `unsigned int`. It overrides the `first()`, `next()`, `end()`, and `current()` methods.
- sequence_set.hpp** (bottom right): The header file containing the code.

Enumerator of ArraySet



```
class ArraySet : public SetRepr<int>{
public:
    ...
    class ArraySetEnor : public Enumerator<int>{
        ...
    };
};

Enumerator<int>* createEnumerator() override{
    return new ArraySetEnor(this);
}
```

Class template Set

```
template <typename Item>
class Set {
public:
    Set(int n = 0) {
        if (0 == n) _repr = new SequenceSet<Item>;
        else _repr = new ArraySet(n);
    }
    ~Set() { delete _repr; }
    void setEmpty() { _repr->setEmpty(); }
    void insert(Item e) { _repr->insert(e); }
    void remove(Item e) { _repr->remove(e); }
    Item select() const { return _repr->select(); }
    bool empty() const { return _repr->empty(); }
    bool in(Item e) const { return _repr->in(e); }

    Enumerator<Item>* createEnumerator() { _repr->createEnumerator(); }

private:
    SetRepr<Item> *_repr;

    Set(const Set& h) ;
    Set& operator=(const Set& h);
};
```

Compilation error:
This assignment may cause error if *Item* is not int.
Though, this assignment is needed only if *Item*=int.

Item

Set
<code>+ Set(n:int = 0)</code>
<code>+ setEmpty() : void</code>
<code>+ insert(Item) : void</code>
<code>+ remove(Item) : void</code>
<code>+ select() : Item {query}</code>
<code>+ empty() : bool {query}</code>
<code>+ in(Item) : bool {query}</code>

set.h

Instantiation depending on a parameter instead of a conditional

```
template <typename Item>
class Set {
public:
    Set(int n = 0) { _repr = createSetRepr<Item>(n); }
    ...
private:
    SetRepr<Item>* _repr;

    static SetRepr<Item>* createSetRepr(int n) {
        return new SequenceSet<Item>;
    }
};
```

a factory design pattern template instantiates the representation

General creator template for class Set<Item>

set.h

```
template<>
inline SetRepr<int>* Set<int>::createSetRep(int n) {
    if (n > 0) return new ArraySet(n);
    else         return new SequenceSet<int>;
}
```

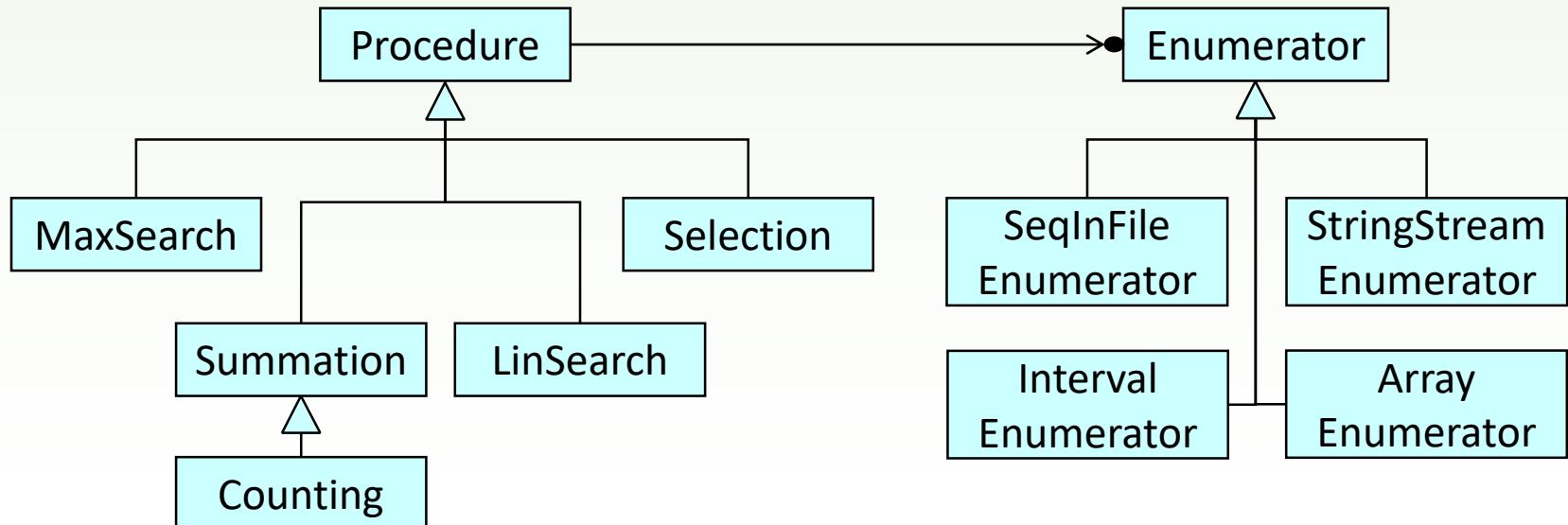
Special creator template for class Set<int>

set.h

Library of class templates
of the algorithmic patterns

Goal

- ❑ Create a **library of codes to describe the algorithmic patterns in general** (Class Template Library). By using it, the programs may be implemented by putting in a minimal effort (without loops).
- ❑ The solution is done by a so-called **activity object**,
 1. which is **inherited** from a class of the Library,
 2. in which the **template parameters** of the parent are given and the **methods** are overridden,
 3. to which an **enumerator object** is connected during runtime.



Prime loop of the alg. patterns

```
template <typename Item>
void run()
{
    if(_enor==nullptr) throw MISSING_ENUMERATOR;

    init();
    for( first(); loopCond(); _enor->next())
    {
        body(_enor->current());
    }
}
```

type of the enumerated items

pointer of the enumerator object

default: _enor->first()

default: !_enor->end() && whileCond(_enor->current())

initialization

t.first()

-t.end()

body(t.current())

t.next()

default: true

This method is feasible to implement any algorithmic pattern if the called methods (`init()`, `body()`, sometimes `loopCond()`, or `whileCond`) are overridden in a proper way, and if attribute `_enor` points at a usable enumerator object.
For that, `run()` has to be a template method, with parameter methods mentioned above. The logic is based on the Template method design pattern.

Prime class of the algorithmic patterns

```
template <typename Item>
class Procedure {
protected:
    Enumerator<Item> * _enor;

    Procedure():_enor(nullptr) {}
    virtual void init() = 0;
    virtual void body(const Item& current) = 0;
    virtual void first() { _enor->first(); }
    virtual bool whileCond(const Item& current) const { return true; }
    virtual bool loopCond() const
    { return !_enor->end() && whileCond(_enor->current()); }

public:
    enum Exceptions { MISSING_ENUMERATOR };
    virtual void run() final;
    virtual void addEnumerator(Enumerator<Item>* en) final { _enor = en; }
    virtual ~Procedure() {}

};
```

type of the enumerated items

pointer of the enumerator object

parameters methods (of template method run()) to be overridden

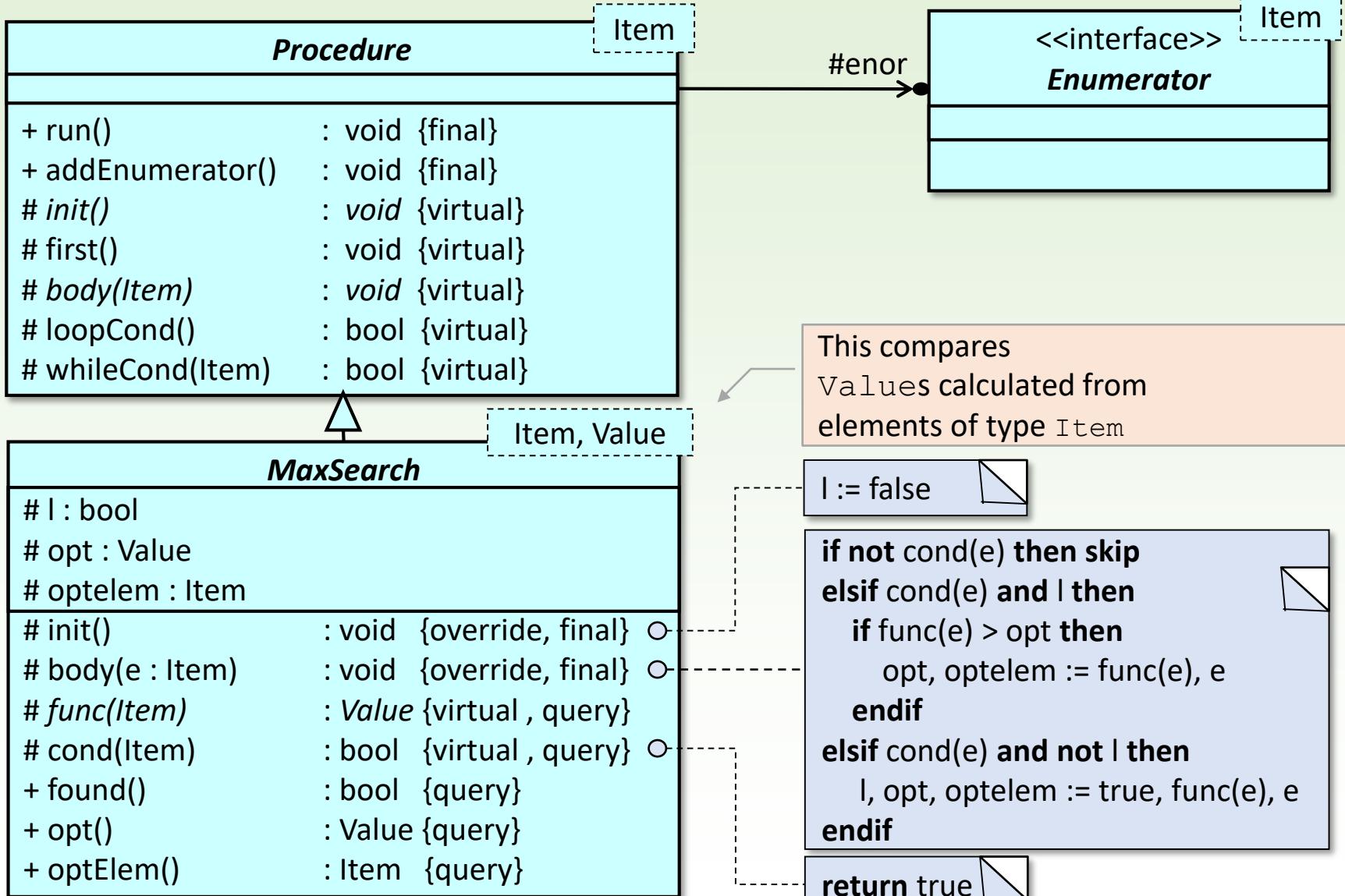
prime loop

cannot be overridden in the children

adding a concrete enumerator object

procedure.hpp

General Maximum search



Class of (Conditional) Max. search

```
template <typename Item, typename Value = Item,
          typename Compare = Greater<Value> >
class MaxSearch : public Procedure<Item>
{
protected:
    bool _l;
    Item _optelem;
    Value _opt;
    Compare _better;

    void init() override final{ _l = false; }
    void body(const Item& e) override final;

    virtual Value func(const Item& e) const = 0;
    virtual bool cond(const Item& e) const { return true; }

public:
    bool found() const { return _l; }
    Value opt() const { return _opt; }
    Item optElem() const { return _optelem; }
};
```

parameter of comparison

attribute which is responsible for the comparison

final override of the methods
in the prime loop

new methods to be overridden

getters of the result

maxsearch.hpp

Loop body of (Cond.) Max. search

```
template <typename Item, typename Value, typename Compare>
void MaxSearch<Item, Value, Compare>::body(const Item& e)
```

```
{  
    if ( !cond(e) ) return;  
    Value val = func(e);  
    if (_l){  
        if (_better(val,_opt)){  
            _opt = val;  
            _optelem = e;  
        }  
    }  
    else {  
        _l = true;  
        _opt = val;  
        _optelem = e;  
    }  
}
```

condition of the Maximum search

creates a comparable value from the enumerated item

Object `_better` of type `Compare` shows if `val` is better than `_opt`. For that, the type which substitutes the template parameter `Compare`, has to implement `operator()`.

maxsearch.hpp

Classes for comparison

```
template <typename Value>
class Greater{
public:
    bool operator() (const Value& l, const Value& r)
    {
        return l > r;
    }
};

template <typename Value>
class Less{
public:
    bool operator() (const Value& l, const Value& r)
    {
        return l < r;
    }
};
```

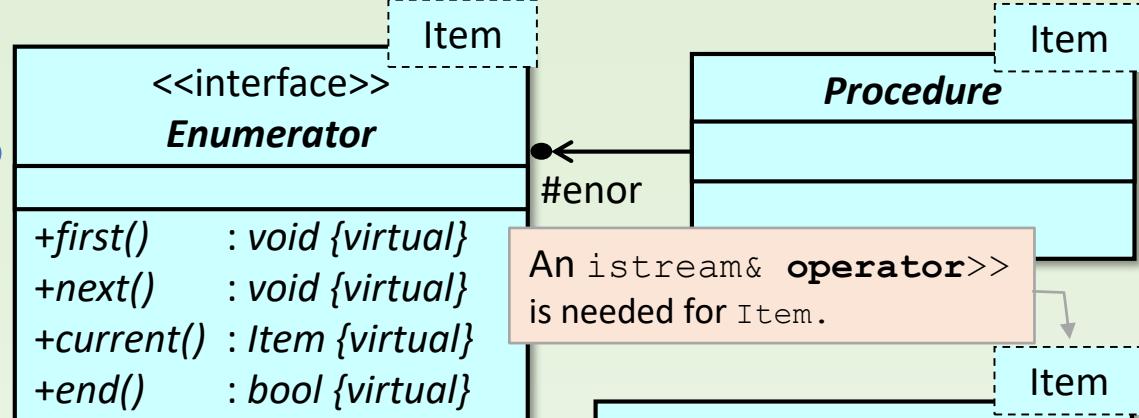
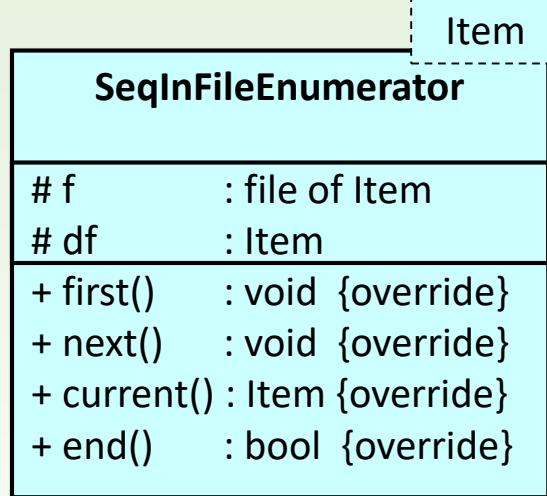
If _better if of type Greater<int>, then _better(2,5) means $2 > 5$.

If _better if of type Less<int>, then _better(2,5) means $2 < 5$.

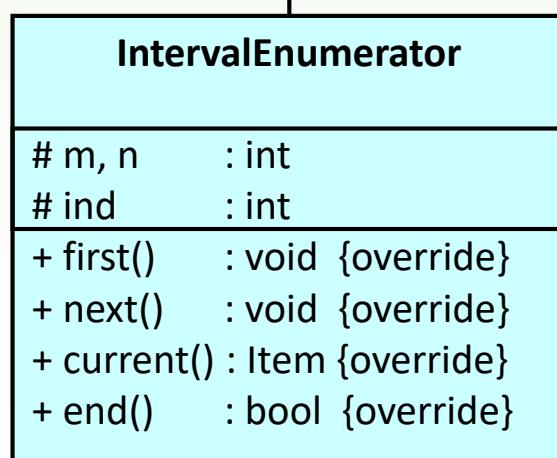
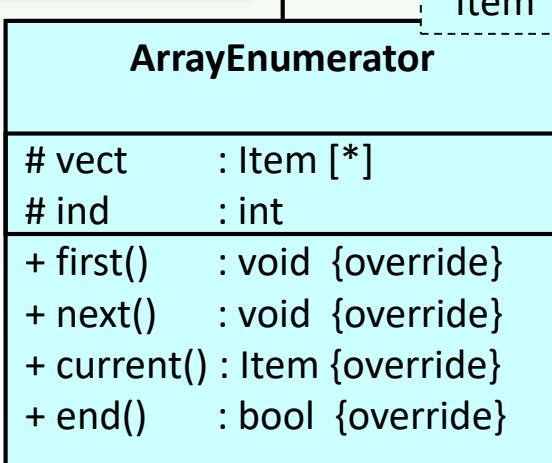
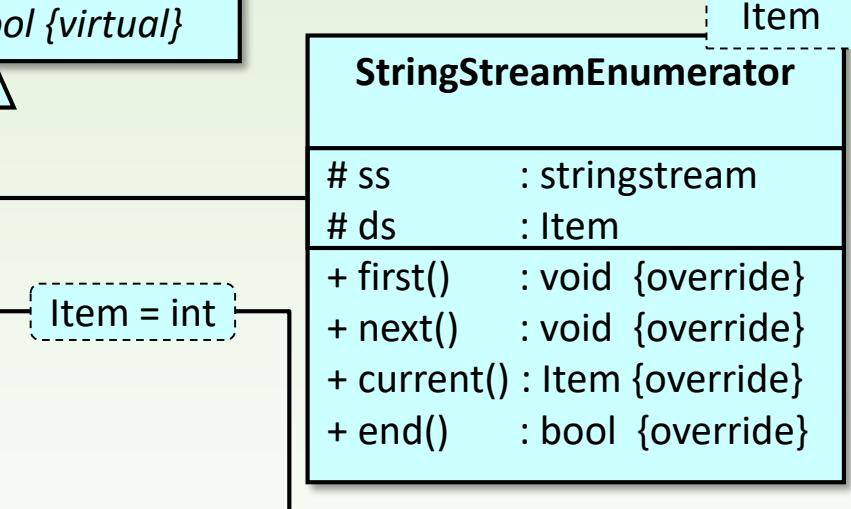
maxsearch.hpp

Enumerators

An `istream& operator>>`
is needed for `Item`.



An `istream& operator>>`
is needed for `Item`.



Interval- and array enumerator

```
class IntervalEnumerator : public Enumerator<int> {
protected:
    int _m, _n;
    int _ind;
public:
    IntervalEnumerator(int m, int n) : _m(m), _n(n) {}
    void first()           override { _ind = m; }
    void next()            override { ++_ind; }
    bool end()             const override { return _ind > _n; }
    Item current()         const override { return _ind; }
};
```

intervalenumerator.hpp

```
template <typename Item>
class ArrayEnumerator : public Enumerator<Item> {
protected:
    const std::vector<Item> *_vect;
    unsigned int _ind;
public:
    ArrayEnumerator(const std::vector<Item> *v) : _vect(v) {}
    void first()           override { _ind = 0; }
    void next()            override { ++_ind; }
    bool end()             const override { return _ind >= _vect->size(); }
    Item current()         const override { return (*_vect)[_ind]; }
};
```

arrayenumerator.hpp

Sequential input file enumerator

```
template <typename Item>
class SeqInFileEnumerator : public Enumerator<Item>{
protected:
    std::ifstream _f;
    Item _df;
public:
    enum Exceptions { OPEN_ERROR };

    SeqInFileEnumerator(const std::string& str) {
        _f.open(str);
        if(_f.fail()) throw OPEN_ERROR;
    }
    void first() override { next(); }
    void next() override { _f >> _df; }
    bool end() const override { return _f.fail(); }
    Item current() const override { return _df; }
};
```

An `istream& operator>>` is needed for `Item`.

seqinfilenumerator.hpp

In the Library, this enumerator is a bit more complex: it has a specialization which in case of enumerating characters (`Item = char`), switches off the automatism that skips the whitespaces. On the other hand, in case of reading line-by-line, it skips the empty lines.

StringStreamEnumerator

```
template <typename Item>
class StringStreamEnumerator : public Enumerator<Item> {
protected:
    std::stringstream _ss;
    Item             _df;
public:
    StringStreamEnumerator(std::stringstream& ss) { _ss << ss.rdbuf(); }

    void first()           final override { next(); }
    void next()            final override { _ss >> _df; }
    bool end()             const final override { return !_ss; }
    Item current()         const final override { return _df; }
};
```

An `istream& operator>>` is needed for `Item`.

stringstreamenumerator.hpp

1st task

Given a text file containing integers. Find the biggest odd number in the file.

A : $f:\text{infile}(\mathbb{N})$, $l:\mathbb{L}$, $\text{max}:\mathbb{N}$

Pre : $f = f_0$

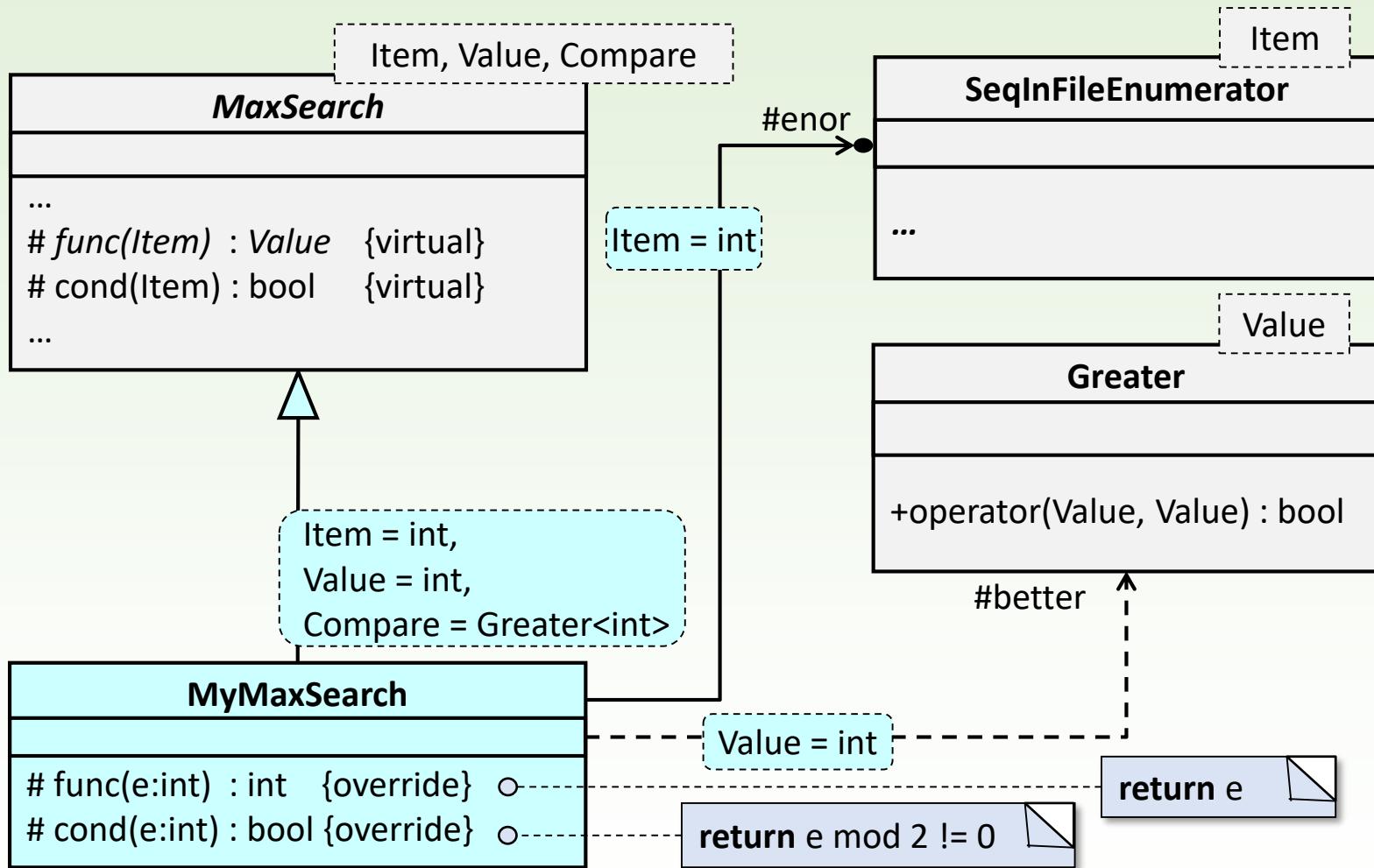
Post : l , $\text{max} = \underset{\begin{array}{c} e \in f_0 \\ 2 \nmid e \end{array}}{\text{MAX}} e$

Conditional maximum search

$t:\text{enor}(E)$	\sim	$f:\text{infile}(\mathbb{N})$
$f(e)$	\sim	e
$H, >$	\sim	$\mathbb{N}, >$
$\text{cond}(e)$	\sim	$2 \nmid e$

E / Item	$\sim \mathbb{N} / \text{int}$
H / Value	$\sim \mathbb{N} / \text{int}$
$f(e) / \text{func}(e)$	$\sim e$
$\text{cond}(e) / \text{cond}(e)$	$\sim 2 \nmid e / e \% 2 != 0$

Class diagram of the solution



Implementation

```
class MyMaxSearch : public MaxSearch<int>{
protected:
    int func(const int& e) const override { return e; }
    bool cond(const int& e) const override { return e % 2 != 0; }
};
```

```
int main() {
    try {
        MyMaxSearch pr;
        SeqInFileEnumerator<int> enor("input.txt");
        pr.addEnumerator(&enor);
        pr.run();

        if (pr.found())
            cout << "The biggest odd number:" << pr.optElem();
        else
            cout << "There is no odd number!";
    } catch(SeqInFileEnumerator<int>::Exceptions ex) {
        if (SeqInFileEnumerator<int>::OPEN_ERROR == ex )
            cout << "Wrong file name!";
    }
    return 0;
}
```

activity object

enumerator object

12 -5 23
44 130 56 3 input.txt
-120

2nd task

In a traffic count, it was measured every hour how many passengers entered a metro station. (The measurement started on a Monday, and was not recorded all the time.) When it was recorded, time was coded by a number of four digits: the first two digits indicate the number of days passed since the measurement started, the second two digits indicate the hour. The measurements are stored in a sequential input file as time code-count pairs.

On the weekends, when (which hour of which day) was the least busy hour?

0108 23	input.txt
0112 44	
0116 130	
0207 120	
...	

Specification

$A : f:\text{infile(Pair)}, l:\mathbb{L}, \text{min}:\mathbb{N}, \text{elem:Pair}$
 $\text{Pair} = \text{rec(timestamp : } \mathbb{N}, \text{number} : \mathbb{N})$

$\text{Pre} : f = f_0$

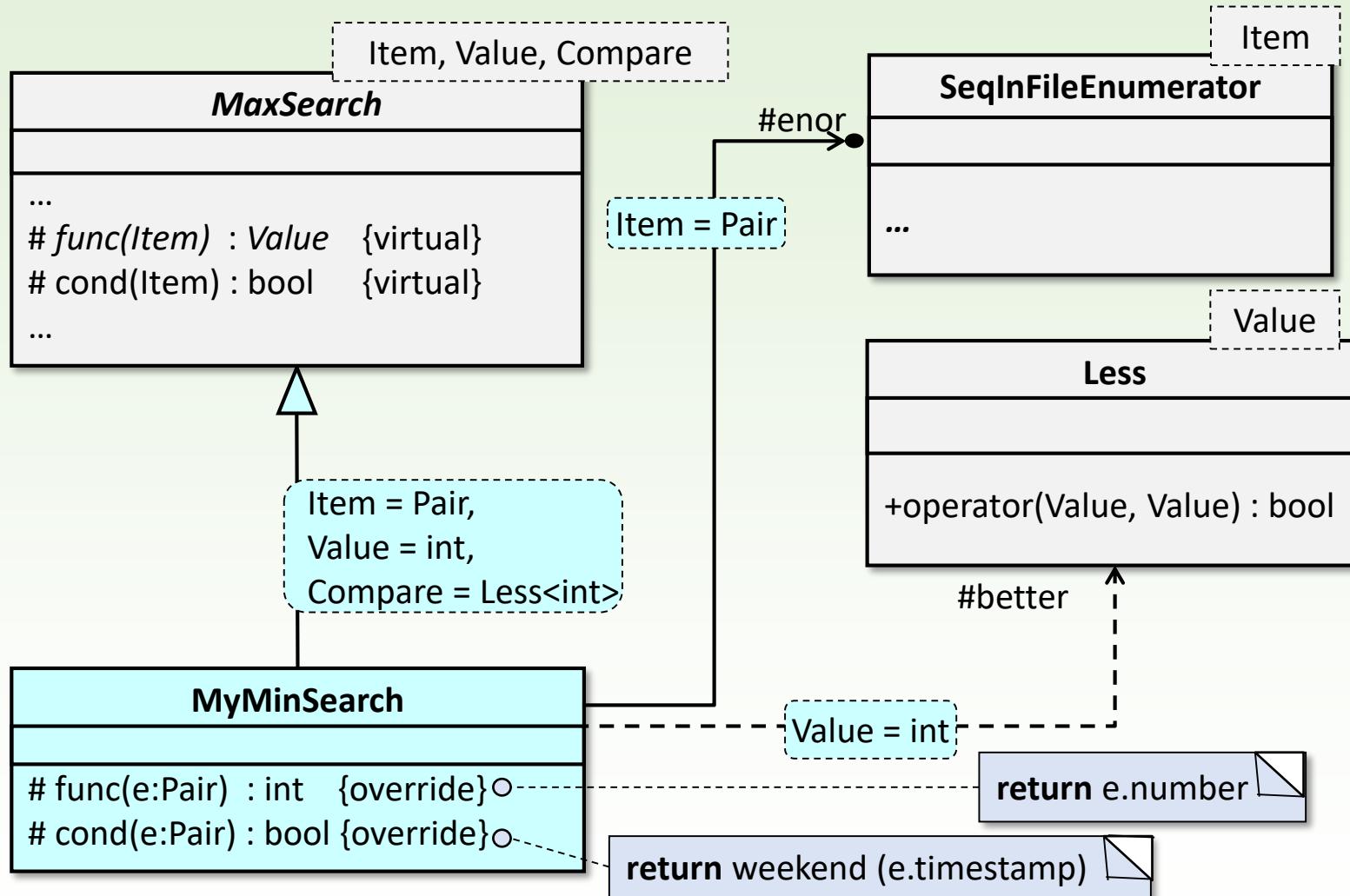
$\text{Post} : l, \text{min}, \text{elem} = \mathbf{MIN}_{e \in f_0} e.\text{number}$
weekend ($e.\text{timestamp}$)

Conditional maximum search

$t:\text{enor(Item)}$	\sim	$f:\text{infile(Pair)}$
$\text{func}(e)$	\sim	$e.\text{number}$
$H, >$	\sim	$\mathbb{N}, <$
$\text{cond}(e)$	\sim	weekend ($e.\text{timestamp}$)

 $e.\text{time} / 100 \% 7 == 6 \mid\mid e.\text{time} / 100 \% 7 == 0$

Class diagram of the solution



Type Pair

```
struct Pair{
    int timestamp;
    int number;

    int day() const { return timestamp / 100; }
    int hour() const { return timestamp % 100; }
};

istream& operator>>(istream& f, Pair& df)
{
    f >> df.timestamp >> df.number;
    return f;
}
```

0108 23
0112 44
0116 130
0207 120
...

input.txt

It is needed for the enumeration of
elements of type Pair

Main program

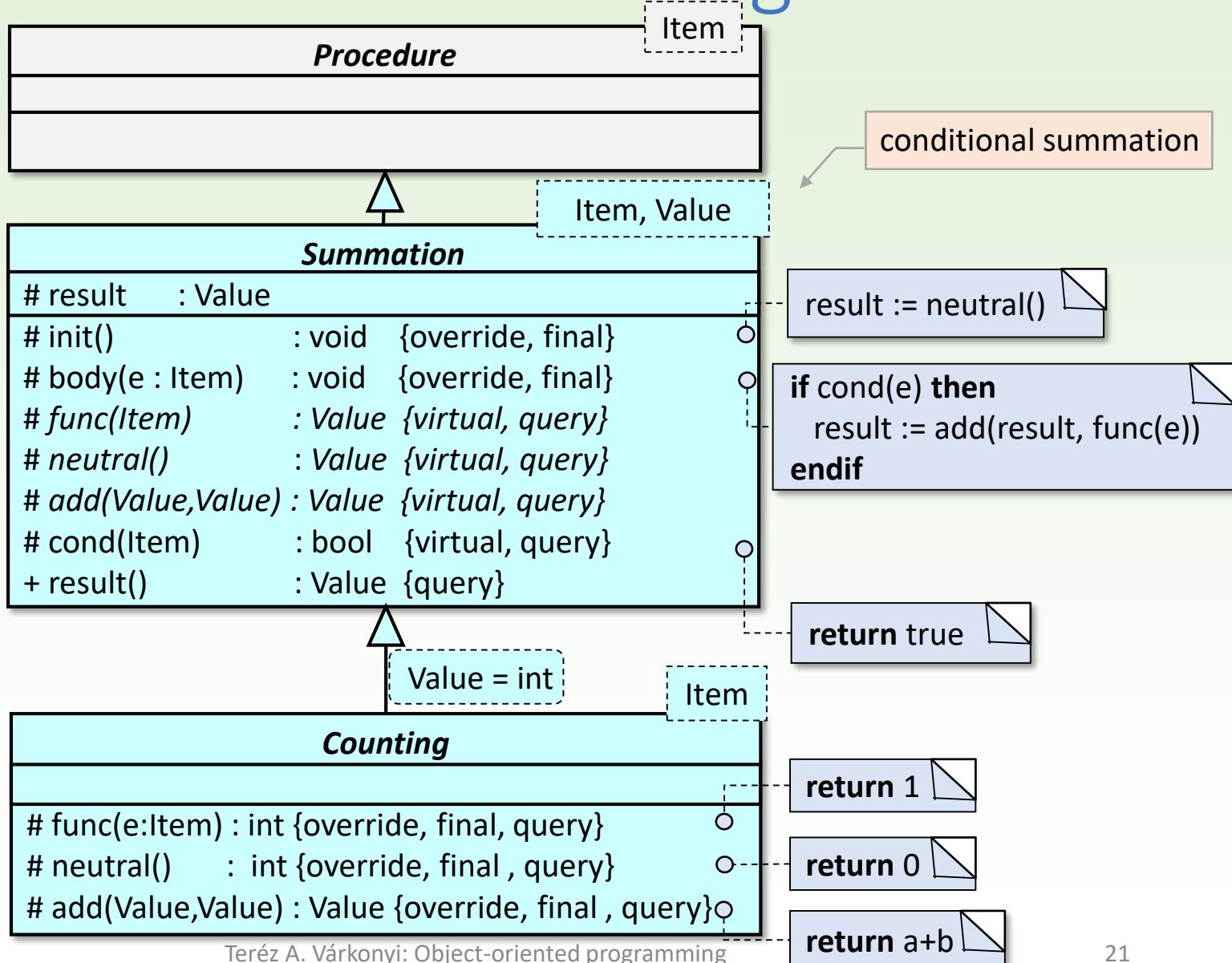
```
class MyMinSearch: public MaxSearch<Pair, int, Less<int> > {
protected:
    int func(const Pair &e) const override { return e.number; }
    bool cond(const Pair &e) const override
    { return e.day() % 7 == 6 || e.day() % 7 == 0; }
};

int main()
{
    try {
        SeqInFileEnumerator<Pair> enor("input.txt");
        MyMinSearch pr;
        pr.addEnumerator(&enor);
        pr.run();

        if (pr.found()) {
            Pair p = pr.optElem();
            cout << "The least busy hour was the " << p.hour()
                << ". hour of the " << p.day() << ". day when "
                << pr.opt() << " people entered the metro station.\n";
        } else cout << "There is no weekend data.\n";
    } catch(SeqInFileEnumerator<int>::Exceptions ex) {
        if (SeqInFileEnumerator<int>::OPEN_ERROR == ex )
            cout << "File open error!";
    }
    return 0;
}
```

minimum search
according to numbers
with this condition

Summation and Counting



Class of Summation and Counting

```
template < typename Item, typename Value = Item >
class Summation : public Procedure<Item>{
private:
    Value _result;
protected:
    void init() override final { _result = neutral(); }
    void body(const Item& e) override final {
        if(cond(e)) _result = add(_result, func(e));
    }
    virtual Value func(const Item& e) const = 0;
    virtual Value neutral() const = 0;
    virtual Value add( const Value& a, const Value& b) const = 0;
    virtual bool cond(const Item& e) const { return true; }
public:
    Value result() const { return _result; }
};
```

summation.hpp

```
template < typename Item >
class Counting : public Summation<Item, int> {
protected:
    int func(const Item& e) const final override { return 1; }
    int neutral() const final override { return 0; }
    int add (const int& a, const int& b) const final override {
        return a + b;
    }
};
```

counting.hpp

Creating sequences (ostream, vector) with Summation

Summation is overdefined for special cases of Value. Neutral() and add() are already defined. It is used for copying, listing, or assorting, when the output is a sequence (ostream or vector).

```
class Summation<Item, ostream> : public Procedure<Item, ostream> {
private:
    std::ostream *_result;
public:
    Summation(std::ostream *o) : _result(o) {}
protected:
    ...
    virtual string func(const Item& e) const = 0;
    virtual bool cond(const Item& e) const { return true; }
};
```

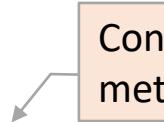
only func() and cond() may be overridden

concatenating strings created by method func() to a given ostream

Creating sequences (ostream, vector) with Summation

Summation is overdefined for special cases of Value. Neutral() and add() are already defined. It is used for copying, listing, or assorting, when the output is a sequence (ostream or vector).

```
class Summation<Item, vector<Value>> :public Procedure<Item, vector<Value>>
{
private:
    std::vector<Value> _result;
public:
    Summation() {}
    Summation(const std::vector<Value> &v) : _result(v) {}
protected:
    ...
    virtual Value func(const Item& e) const = 0;
    virtual bool cond(const Item& e) const { return true; }
};
```



Concatenating elements created by
method func() to a given vector.

3rd task – only C++

Concatenate the content of a text file containing integers to an existing vector, then write the vector to the console.

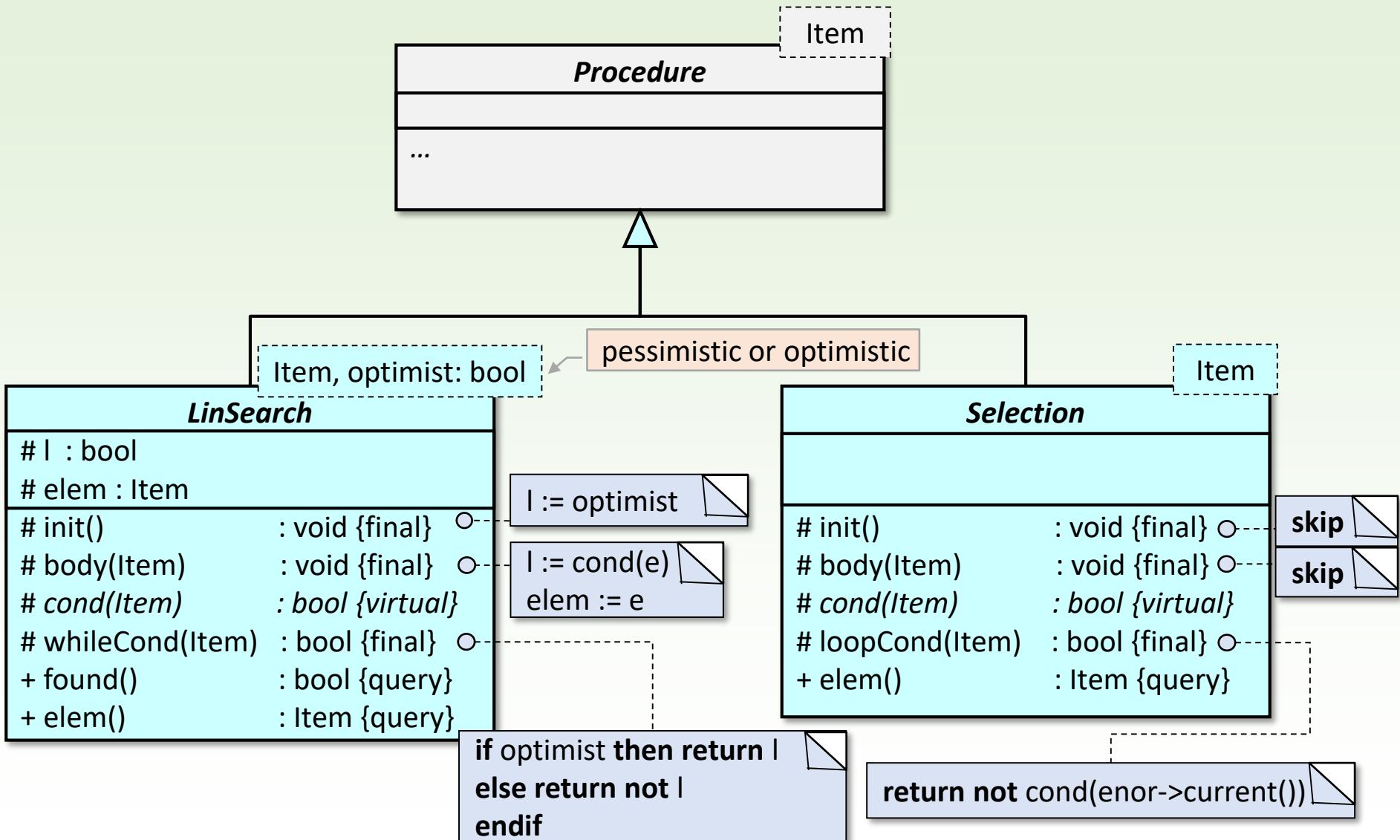
```
class Concat : public Summation<int, vector<int> > {
public:
    Concat(const vector<int> &v) : Summation<int, vector<int> >(v) {}
    int func(const int &e) const override { return e; }
};

class Write : public Summation<int, ostream > {
public:
    Write(ostream* o) : Summation<int, ostream>(o) {}
    string func(const int &e) const override {
        ostringstream os;
        os << e << " ";
        return os.str();
    }
}

vector<int> v = { -17, 42 };
Concat pr1(v);
SeqInFileEnumerator<int> enor1("input.txt");
pr1.addEnumerator(&enor1);
pr1.run();

Write pr2(&cout);
ArrayEnumerator<int> enor2(&pr1.result());
pr2.addEnumerator(&enor2);
pr2.run();
```

Linear search and Selection



Class of Linear search

```
template < typename Item, bool optimist>
class LinSearch : public Procedure<Item> {
protected:
    bool _l;
    Item _elem;

    void init()           override final { _l = optimist; }
    void body(const Item& e) override final { _l = cond(_elem = e); }
    bool whileCond(const Item& e) const override final {
        return optimist ? _l : !_l;
    }
    virtual bool cond(const Item& e) const = 0;

public:
    bool found() const { return _l; }
    Item elem() const { return _elem; }
};
```

linsearch.hpp

Class of Selection

```
template < typename Item >
class Selection : public Procedure<Item> {
protected:
    void init()           override final { }
    void body(const Item& e) override final { }
    bool loopCond()       const override final {
        return !cond(Procedure<Item>::_enor->current());
    }
    virtual bool cond(const Item& e) const = 0;
public:
    Item result() const { return Procedure<Item>::_enor->current(); }
};
```

selection.hpp

4th task

Lines of a text file contain recipes where a recipe consists of the name of the food (string) and the ingredients. An ingredient is given by its name (string), quantity (number), and unit (string). Example:

```
semolina_pudding milk 1 liter semolina 13 spoon  
                          butter 60 gram sugar 5 spoon
```

How many recipes need sugar?



one line of the file

A : $f:\text{infile}(\text{Recipe}) , c:\mathbb{N}$

 Recipe = rec(name : String, ingredients : Ingredient*)

 Ingredient = rec(substance : String, quantity: \mathbb{R} , unit : String)

Pre : $f = f_0$

Post : $c = \sum_{\substack{e \in f_0 \\ \text{has_sugar}(e)}} 1$, where

subtask: is there sugar in the ingredients?

ie.ingredients
 $\text{has_sugar}(e) = \text{SEARCH}_{i=1}^{\text{e.ingredients}} \text{e.ingredients}[i].\text{substance} = \text{"sugar"}$

Counting

$t:\text{enor}(\text{Item}) \sim f:\text{infile}(\text{Recipe})$
 $\text{cond}(e) \sim \text{has_sugar}(e)$

Linear search

$t:\text{enor}(\text{Item}) \sim \text{Ingredient}^* (i=1..*)$
 $\text{cond}(e) \sim$
 $\text{e.ingredients}[i].\text{substance} = \text{"sugar"}$

First plan of the solution

main

```
pr : MyCounting
enor : SeqInFileEnumerator<Recipe>("input.txt")
pr.addEnumerator(&enor)
pr.run()
return pr.result()
```

Item = Recipe

MyCounting : Counting

```
# cond(e:Recipe) : bool {override}
```

```
pr : MyLinSearch
enor: ArrayEnumerator<Ingredient>(e.vect)
pr.addEnumerator(&enor)
pr.run()
return pr.found()
```

Item = Ingredient

MyLinSearch : LinSearch

```
# cond(e: Ingredient):bool {override}
```

operator>>(is:istream, e:Recipe)

```
getline(is, line)
ss : stringstream(line)
ss >> e.name
pr : Copy
enor: StringStreamEnumerator<Ingredient>(ss)
pr.addEnumerator(&enor)
pr.run()
e.vect := pr.result()
```

Ingredient

```
substance : string
quantity : int
unit : string
```

operator>>(is:istream, e:Ingredient)

```
is >> e.substance
>> e.quantity
>> e.unit
```

Item = Ingredient

Value = Ingredient [*]

Copy : Summation

```
# func(e:Ingredient) : Ingredient {override}
```

```
return e.substance="sugar"
```

```
return e
```

Second plan of the solution

main

```

pr : MyCounting
enor : SeqInFileEnumerator<Recipe>("input.txt")
pr.addEnumerator(&enor)
pr.run()
return pr.result()

```

Item = Recipe

MyCounting : Counting

```
# cond(e:Recipe) : bool {override}
```

return e.has_sugar

Recipe

```
name : string
has_sugar : bool
```

Ingredient

```
substance : string
quantity : int
unit : string
```

operator>>(is:istream, e:Recipe)

```

getline(is, line)
ss : stringstream(line)
ss >> e.name
pr : MyLinSearch ○
enor: stringstream enumerator<Ingredient>(ss)
pr.addEnumerator(&enor)
pr.run()
e.has_sugar := pr.found()

```

operator>>(is:istream, e:Ingredient)

```

is >> e.substance
>> e.quantity
>> e.unit

```

Item = Ingredient

MyLinSearch : LinSearch

```
# cond(e: Ingredient):bool {override}○
```

return e.substance="sugar"

5th task

Observations of asteroids are stored in a text file.

Every line contains one observation: ID of the asteroid (string), date (string), mass of the asteroid (thousand tons), distance between the asteroid and Earth (100.000 kms). 1 asteroid may have more observations.

AXS0076 2015.06.13. 2000 5230

The file is ordered by ID.

Give those asteroids with their greatest observed mass that were closer to Earth than 1 billion kms at every observation.

A : $f:\text{inFile(Observation)}, \text{cout:outfile(String} \times \mathbb{N})$

Observation = rec(id:String, date:String, mass: \mathbb{N} , distance: \mathbb{N})

Pre : $f = f' \wedge f \nearrow_{\text{id}}$

A' : $t:\text{enor(Asteroid)}, \text{cout:outfile(String} \times \mathbb{N})$

Asteroid = rec(id:String, mass: \mathbb{N} , near: \mathbb{L})

Pre : $t = t_0$

Post : $\text{cout} = \bigoplus_{\substack{e \in t_0 \\ e.\text{near}}} \langle (e.\text{id}, e.\text{mass}) \rangle$

Summation (assortment)

$t:\text{enor(Item)} \sim t:\text{enor(Asteroid)}$

$\text{func}(e) \sim (e.\text{id}, e.\text{mass})$

$H, +, 0 \sim (\text{String} \times \mathbb{N})^*, \bigoplus, \langle \rangle$

$\text{cond}(e) \sim e.\text{near}$

Plan of the solution

main

```
pr : Assortment(&cout)
enor : AsteroidEnumerator("input.txt")
pr.addEnumerator(&enor)
pr.run()
```

Item = Asteroid
Value = ostream

Assortment : Summation

```
# cond(e:Asteroid) : bool {override}
# func(e:Asteroid) : string {override}
```

return string(e)

return e.near

Asteroid

```
id : string
mass : int
near : bool
```

f := new SeqInFileEnumerator<Observation>(str)

Item = Asteroid

AsteroidEnumerator : Enumerator

```
- f : SeqInFileEnumerator<Observation>
- current : Asteroid
- end : bool
```

+ AsteroidEnumerator(str:string)

+~AsteroidEnumerator()

first():void {override}

next():void {override}

current() : Asteroid {override}

end():bool {override}

delete f

f.first()
next()

return current

return end

**operator>>(is:istream,
e:Observation)**

```
is >> e.id
>> e.date
>> e.mass
>> e.distance
```

Reading the data of an asteroid

Method next() calculates the greatest mass of the asteroid and that if it was closer to Earth than 1 billion kms at every observation or not.

A : f:inFile(Observation), e:Observation, st:Status, current:Asteroid, end: \perp

Observation = rec(id:String, date:String, mass:N, distance:N)

Asteroid = rec(id:String, mass:N, near:L)

Pre : $f = f' \wedge e = e' \wedge st = st' \wedge f \nearrow_{id}$

Post : end = (st'=abnorm) \wedge (\neg end \rightarrow current.id = e'.id \wedge

$e.id = current.id$
 $(current.mass, st, e, f) = \text{MAX}_{e \in (e', f')} e.mass \wedge$

the two enumerations cannot
be done in sequence, they
have to be merged into one loop

$e.id = current.id$
 $(current.near, st'', e'', f'') = \text{SEARCH}_{e \in (e', f')} e.distance < 10000))$

these two processes may
stop in different states

Merging two algorithmic patterns

The Maximum search which calculates the greatest mass of the asteroid and the Linear search which decides if the asteroid was near all the time have to be put into the same loop.

cannot be merged

Maximum search

$t:\text{enor}(\text{Item}) \sim f:\text{infile}(\text{Observation})$
without first()
as long as the same id
 $\text{func}(e) \sim e.\text{mass}$
 $H, > \sim \mathbb{N}, >$

Optimistic linear search

$t:\text{enor}(\text{Item}) \sim f:\text{infile}(\text{Observation})$
without first()
as long as the same id
 $e.\text{distance} < 10000$



cond(e)

Summation

$t:\text{enor}(\text{Item}) \sim f:\text{infile}(\text{Observation})$
without first()
as long as the same id
 $\text{func}(e) \sim e.\text{mass}$
 $H, +, 0 \sim \mathbb{N}, \text{max}, 0$

can be merged

Summation

$t:\text{enor}(\text{Item}) \sim f:\text{infile}(\text{Observation})$
without first()
as long as the same id
 $\text{func}(e) \sim e.\text{distance} < 10000$
 $H, +, 0 \sim \mathbb{L}, \wedge, \text{true}$



Plan of method next()

next

```

end := f.end()
if ( end ) then return
endif
current.id := f.current().id
pr : DoubleSummation (current.id)
pr.addEnumerator(f)
pr.run()
current.max := pr.result().mass
current.near := pr.result().near

```

Asteroid

id : string
 mass : int
 near : bool

Observation

id : string
 date : string
 mass : int
 distance : int

Result

mass : int
 near : bool

Result(int,bool)

DoubleSummation : Summation

- id : string
- + DoubleSummation(str : string) ○
- # func(Observation e) : Result {override} ○
- # neutral() : Result {override} ○
- # add(Result a, Result b) : Result {override} ○
- # whileCond(Observation e) : bool {override} ○
- # first() : void {override} ○

skip

return e.id = id

id := str

return Result(
e.mass,
e.distance < 10000)

return Result(0, true)

return Result(
max(a.mass, b.mass),
a.near **and** b.near)

Behavior of objects

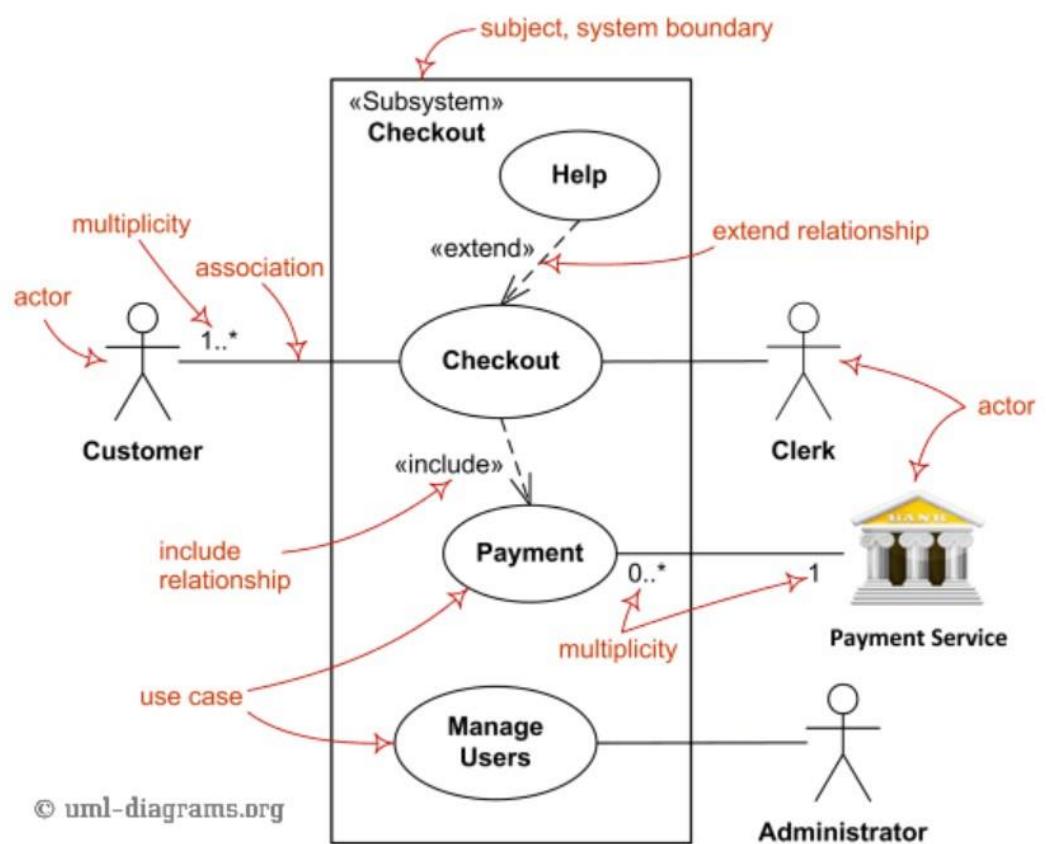
Behavioral views of UML

Behavioral views

- ❑ To describe the dynamical behavior of objects, UML has introduced several views. In this lecture, the followings are presented:
 - Use case diagram
 - Communication diagram
 - Sequence diagram
 - State machine diagram

Use case diagram

- For a planned system, it shows
 - its goal,
 - its functionality (what it is capable of),
 - who it serves (actors)
 - what requirements it has for the environment



Specifiers of the relationships of the use cases

❑ Precedence of the use cases

- **precede**: the order of the activities a user might trigger
- **invoke**: an activity which follows a user activity, but cannot be triggered directly

❑ Extensions of a use case

- **include**: independently triggerable, divided part of a user activity without which the container is incomplete (abstract).
- **extend**: a complete activity which might extend optionally a user activity and which is never abstract

❑ Inheritance between activities or actors

❑ Multiplicity might be denoted

User story

- ❑ A use case diagram does not provide an acceptable image of the system to be implemented.
- ❑ In a tabular description (called user story) which goes *by user groups* ("AS a ..."), every user activity has to be explained in detail:
 - *name* of the activity,
 - what *prerequisites* it assumes (*GIVEN*)
 - what *event* triggers it (*WHEN*)
 - its *effects* and result (*THEN*).

AS a ...		
case		description
activity	GIVEN	assumed precondition
	WHEN	triggering event
	THEN	effects

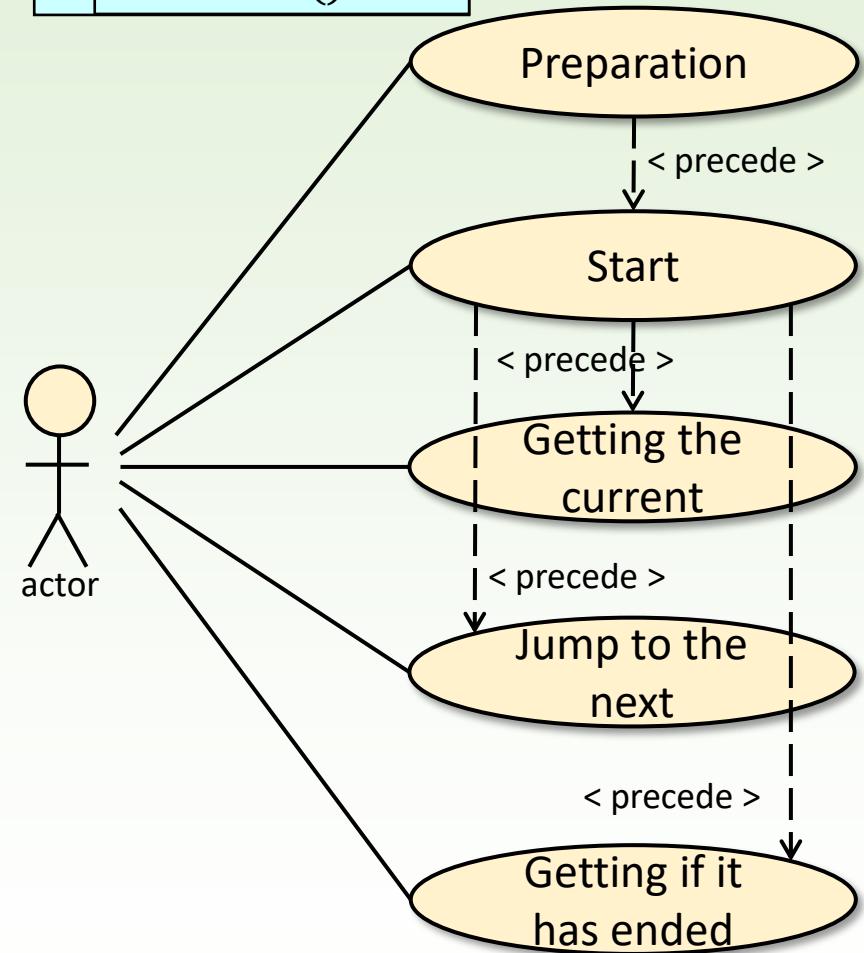
first()

→end()

... current() ...

next()

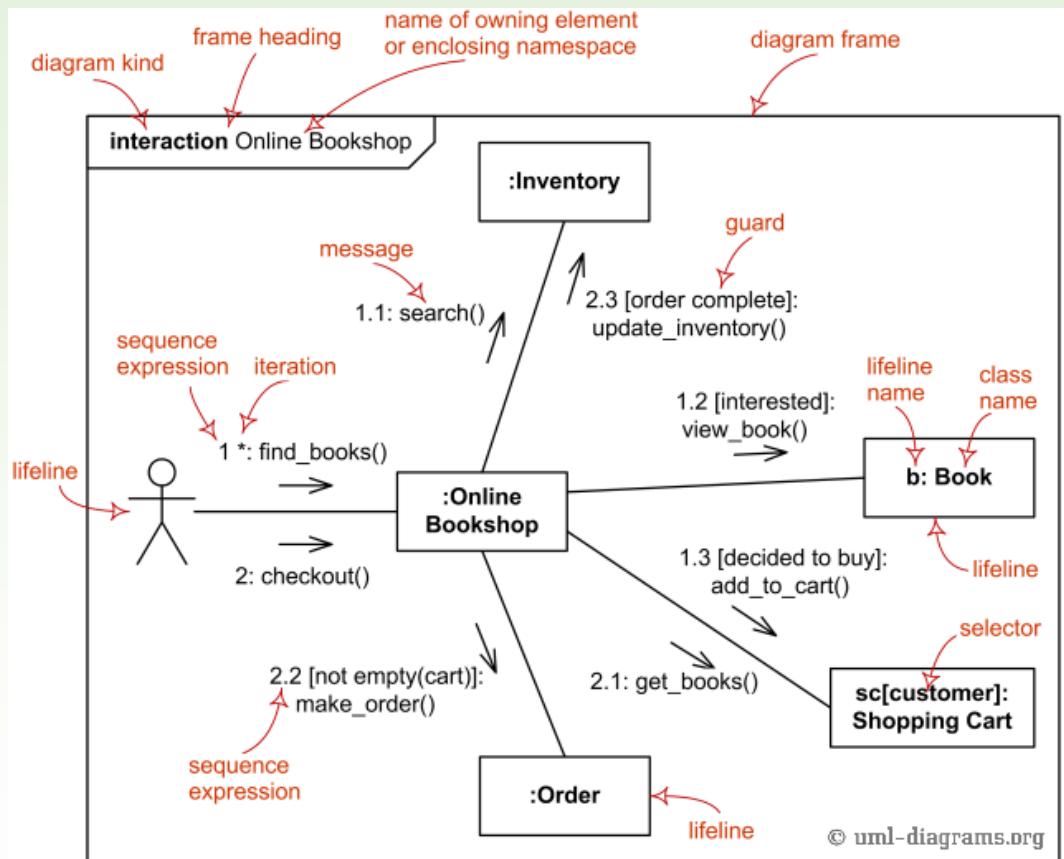
Example: Enumeration



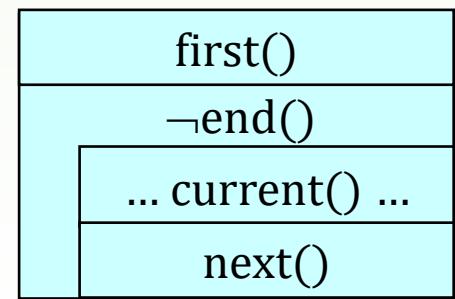
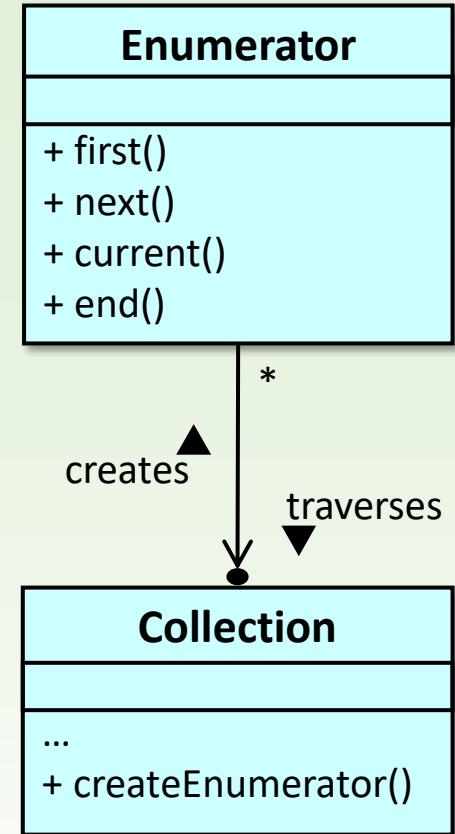
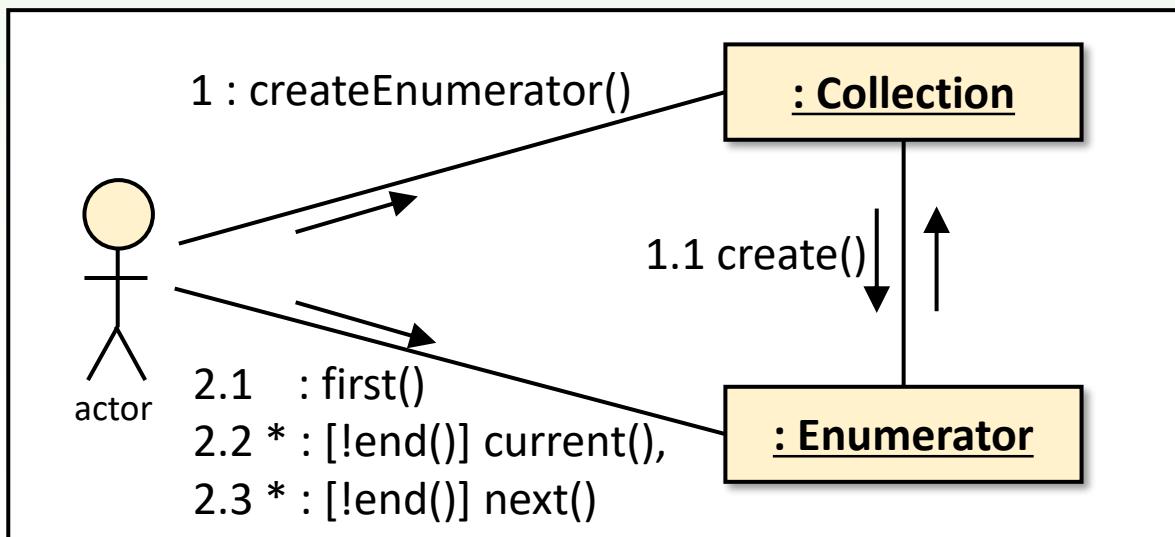
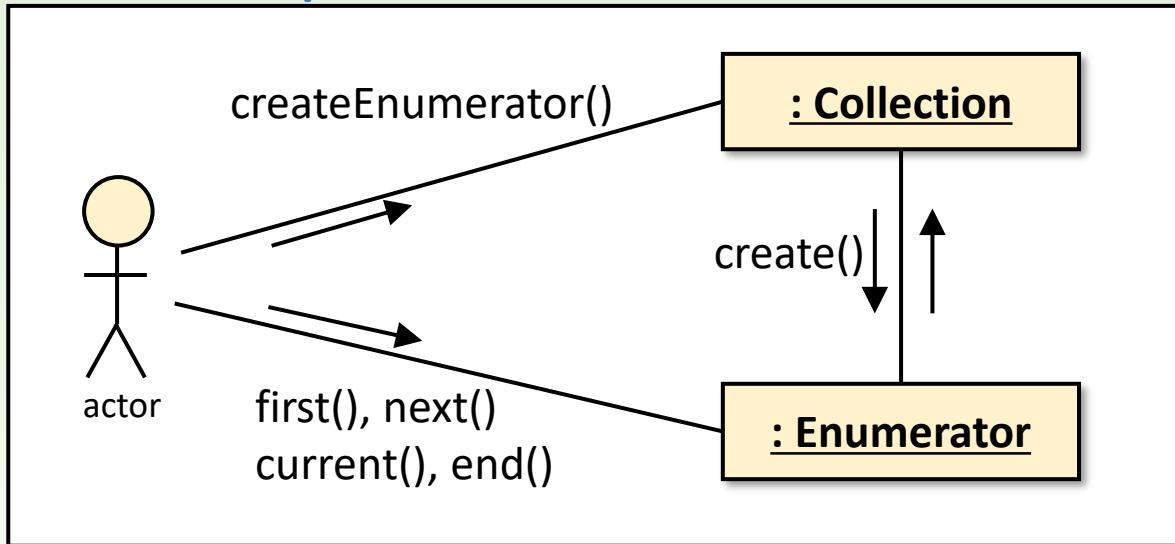
case		description
Preparation in normal case	GIVEN	Given a collection to be enumerated.
	WHEN	Instantiation of the enumerator.
	THEN	Enumerator is created.
Preparation in abnormal case	GIVEN	There is no collection to be enumerated.
	WHEN	Instantiation of the enumerator.
	THEN	Error, enumerator object is not created.
Start in normal case	GIVEN	Given an enumerator object in state <i>pre-start</i> .
	WHEN	Starting the enumeration with operation first() .
	THEN	The enumerator gets to state <i>in-process</i> .
Start in abnormal case	GIVEN	Given an enumerator object in state <i>in-process</i> or <i>finished</i> .
	WHEN	Starting the enumeration with operation first() .
	THEN	Error, and the enumerator preserves its state.
...		

Communication diagram

- ❑ A communication diagram shows with what kind of **messages** (method calls, signal sends) the **objects** communicate with each other.
- ❑ It is possible to indicate the **order** of the messages with numbers and to give **guards** (condition that allows the message) between square brackets.

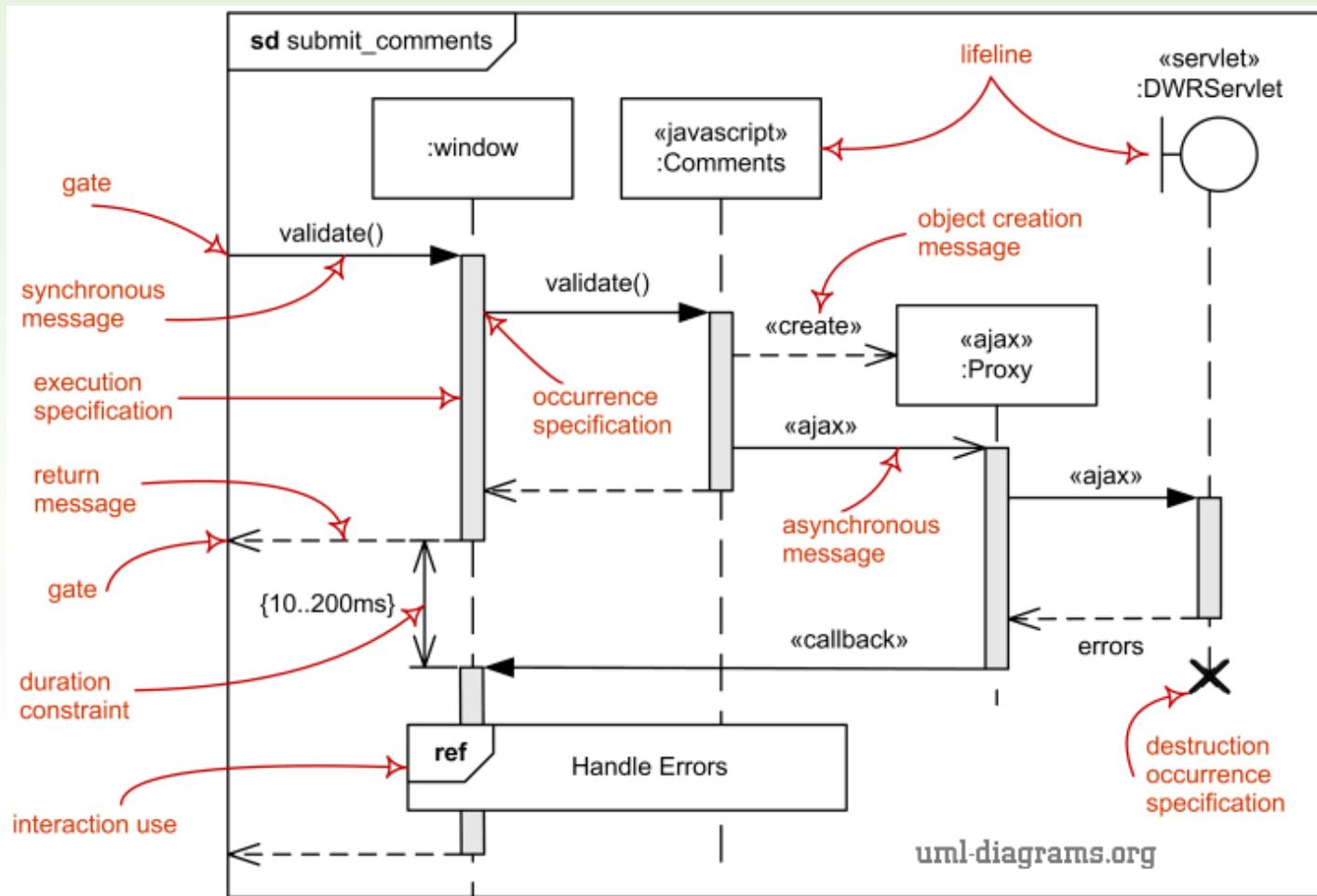


Example: Enumeration



Sequence diagram

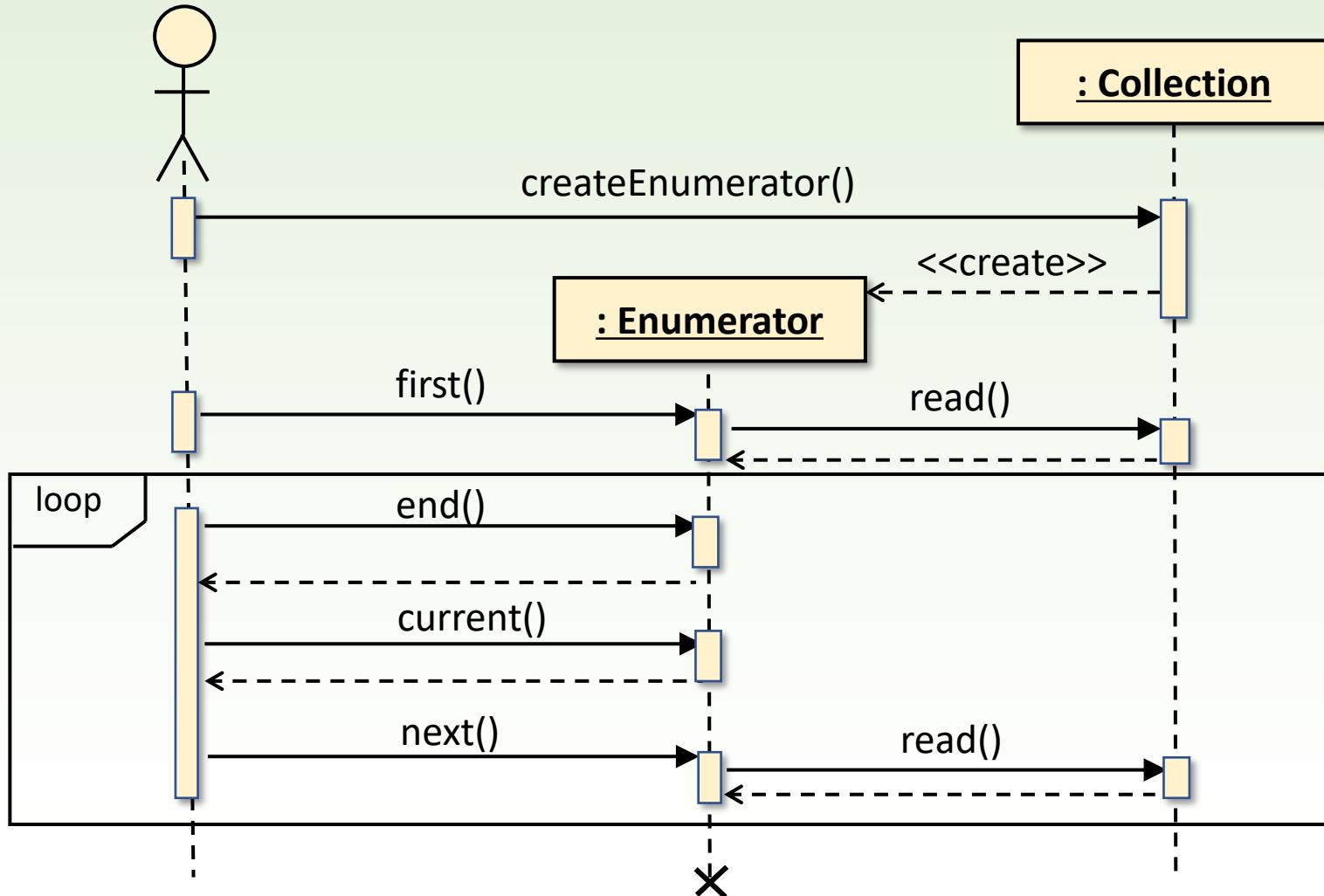
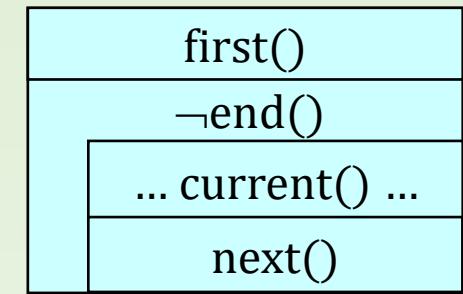
- It shows the order of the messages in the communication.



Messages

- ❑ **Synchronous message**: the sender gives the control to the receiver. It means a synchronous method call. →
- ❑ **Reply message**: the receiver of a previous message sends it back to the sender when it has finished and wants to give back the control. Many times it is not represented in the diagram as it can be seen easily based on the environment. ←
- ❑ The following messages count only in case of concurrent messages:
→ , previously →
 - **Asynchronous message**: activity of the sender object does not stop, it is not important when the receiver receives the message.
 - **Synchronization message**: it blocks the activity of the sender as long as the receiver has not received the message.
 - **Timeout waiting message**: the sender waits for the receiver to receive message up to some fixed period of time.
 - **Rendezvous message**: the receiver waits for the sender to get a message from it.

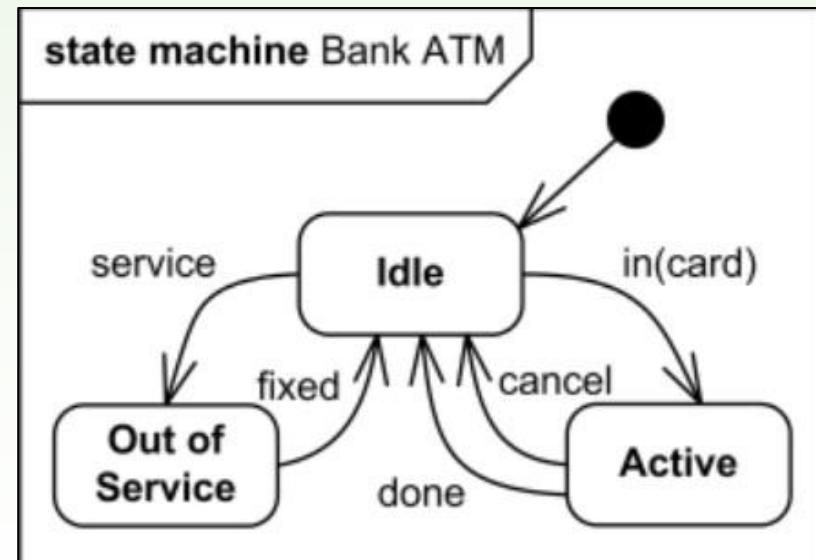
Example: Enumeration



State machine

- ❑ A state machine diagram illustrates the **lifecycle** and the behavior **of an object**. It shows how the logical state changes of an objects when it gets a message (method call or signal).

- ❑ A state machine is a directed graph the nodes of which are the logical states, the edges of which are the transitions between them.
- ❑ Executable actions may belong to both the states and the transitions.

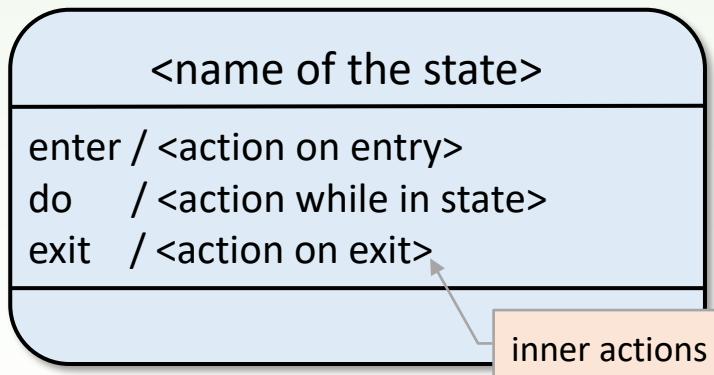
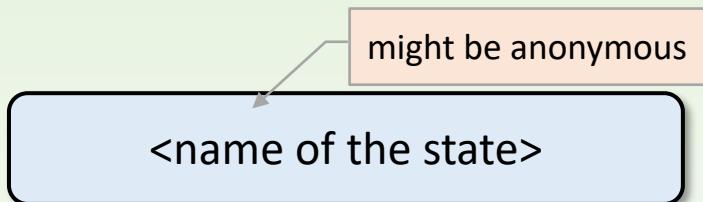


Lifecycle of an object

- ❑ During the **lifecycle** of an object:
 - it is created: by its **constructor**,
 - it works: **communicates** with other objects, which means they call each other's methods synchronously or asynchronously, or they react to the other's signal asynchronously and during that **their properties may change**, and
 - it is destroyed by its **destructor**.
- ❑ An object may have different **physical states**: a physical state means the current values of each of its attributes that may change during its lifetime.
- ❑ Often, one **logical state** includes several physical states with similar or common properties.

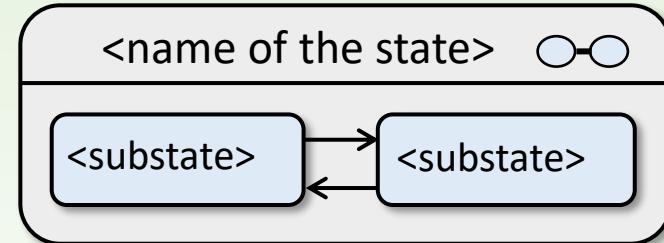
Notations of the states

- Simple state

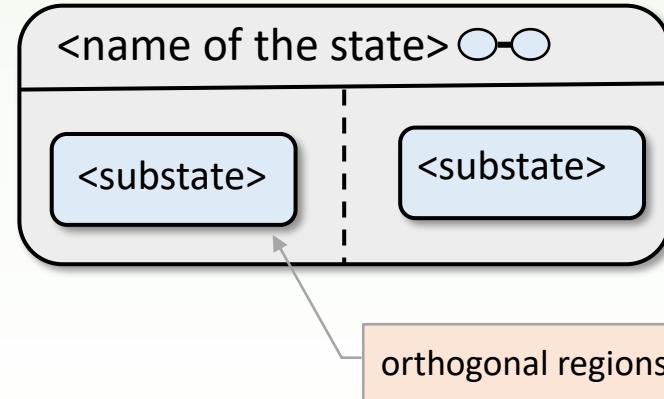


- Complex state

Sequential:



Concurrent:



Pseudo states

- Start state



- Final state



- Entry point



- Exit point



- Terminate state



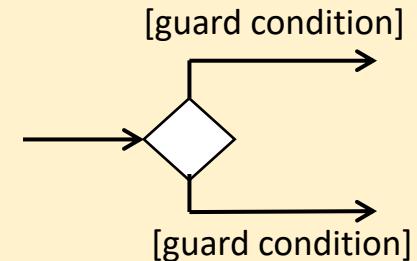
- Shallow history



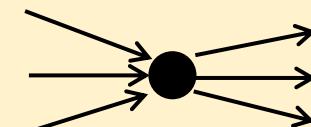
- Deep history



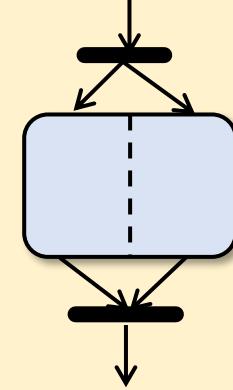
- Choice



- Junction



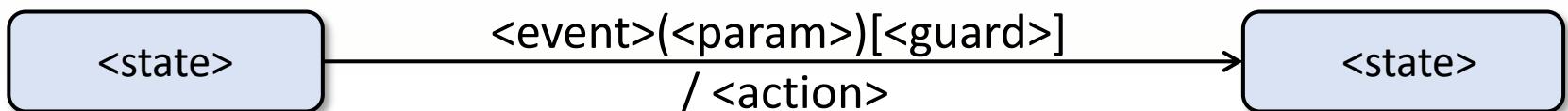
- Fork



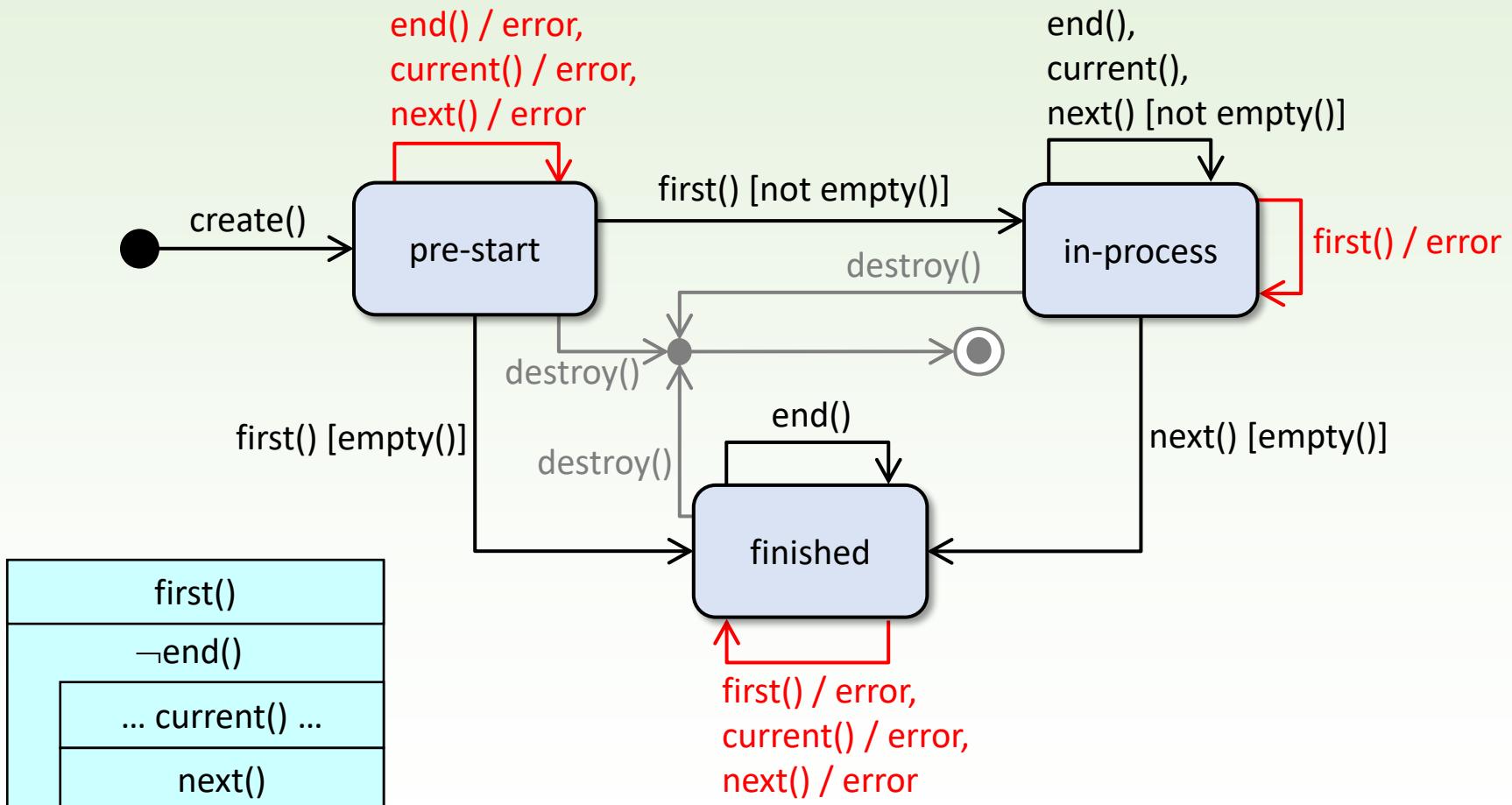
- Join

Notations of the transitions

- ❑ Properties of the transitions (any of them may be skipped):
 - trigger (*event, trigger*) of the transition with parameters
 - either a synchronous method call of the object
 - or an asynchronously processed signal which is sent to it
 - a guard condition, which necessarily precedes it
 - either a logical statement which depends on the parameters (*when*)
 - or a time-bound waiting condition (*after*)
 - an action assigned to the transition (a program operating with the attributes of the object and the parameters of the triggering event)
 - short explanatory description (often missing)
- ❑ A transition may be reflexive (inner) where the state does not change and enter and exit actions are not executed.



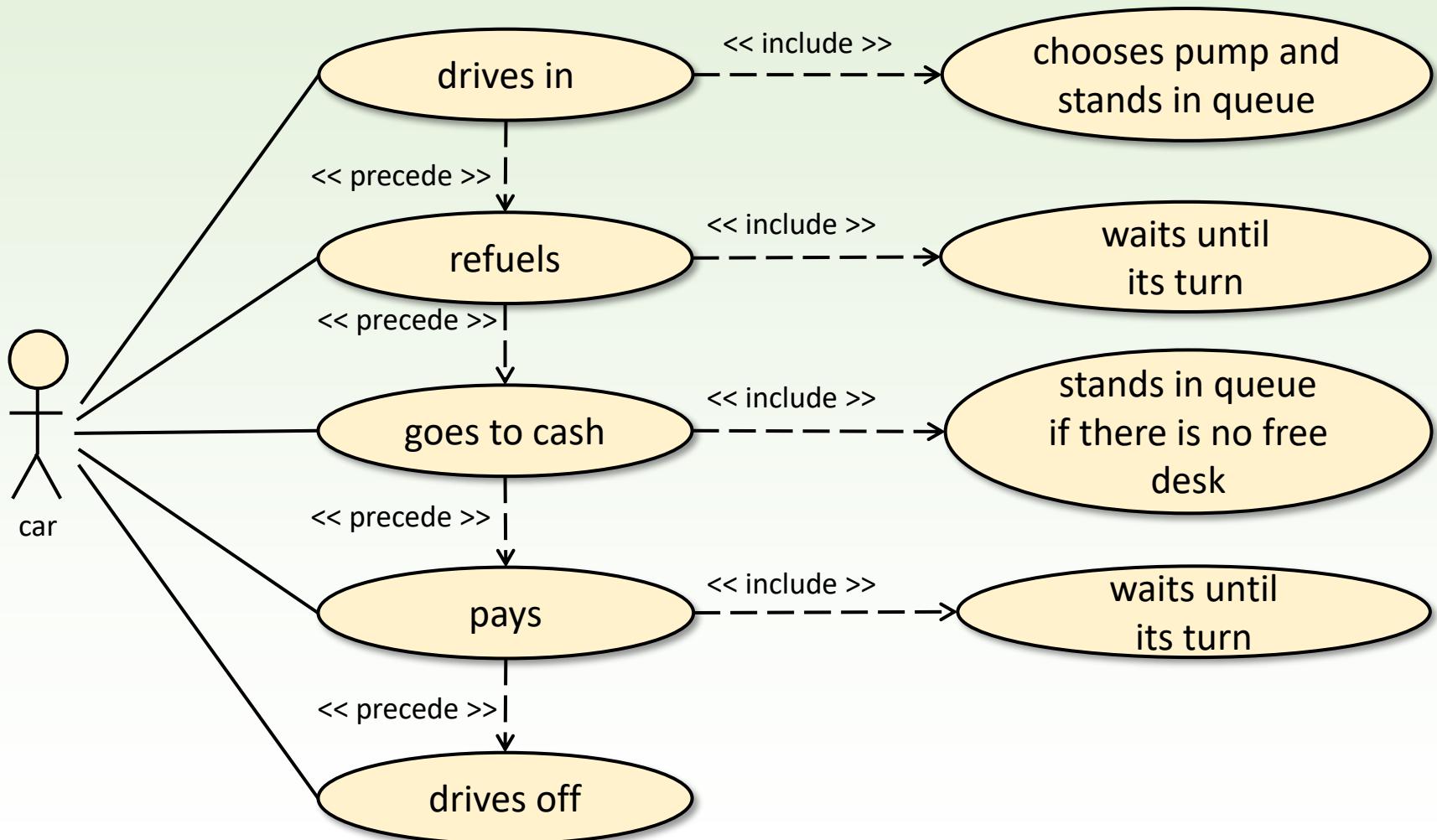
Example: Enumeration



Task

- ❑ On a petrol station, there are some pumps and some cash desks. The cars drive in and queue for a concrete pump. When it is their turn, they fill by a previously decided amount of fuel. After that, they go to pay. There is one queue for all the cash desks. When its their turn, the cash desk calculates the money to pay based on the amount of fuel. After paying, the cars drive off.
- ❑ Model this process for arbitrary number of cars acting concurrently.

Use case diagram

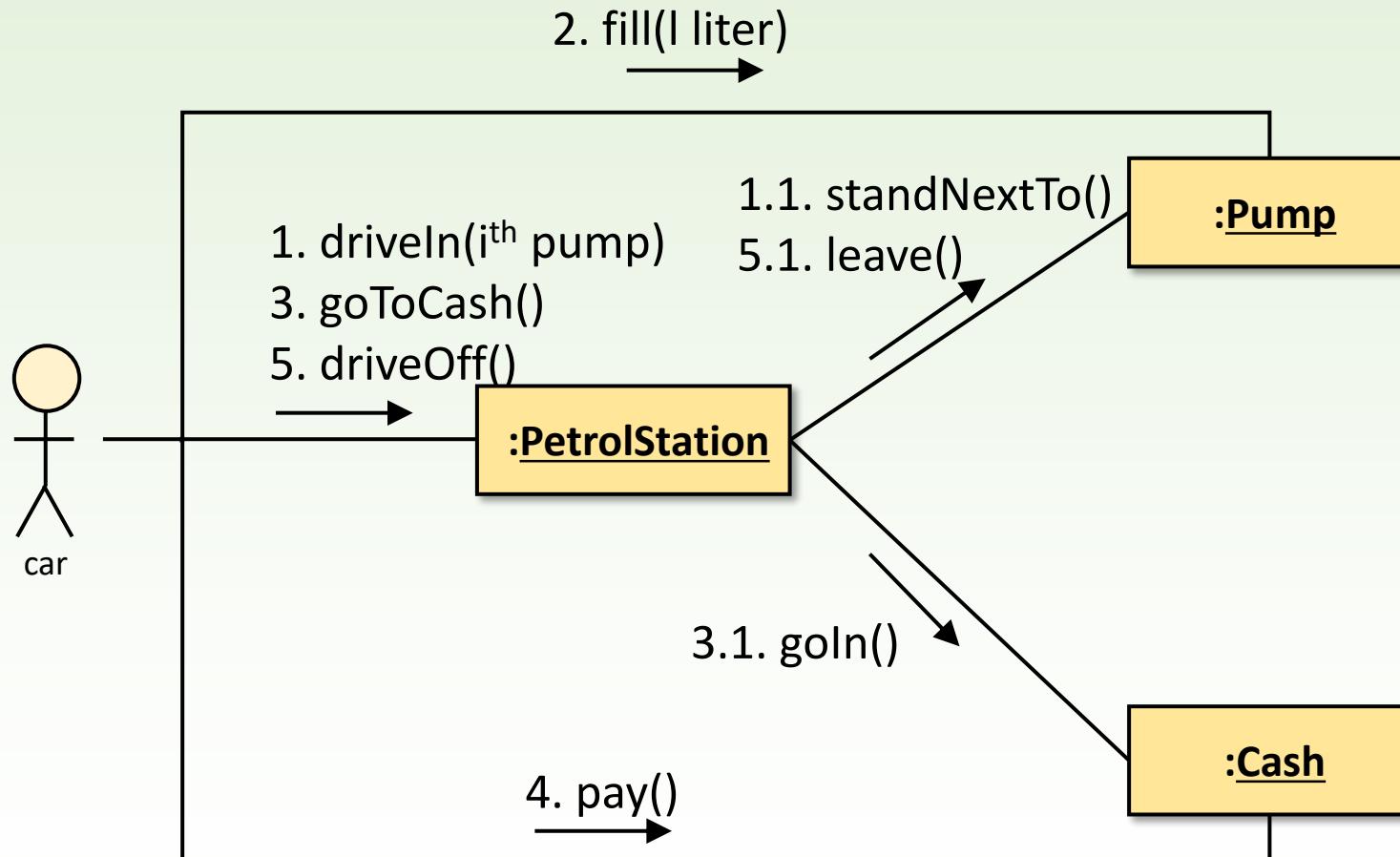


User story

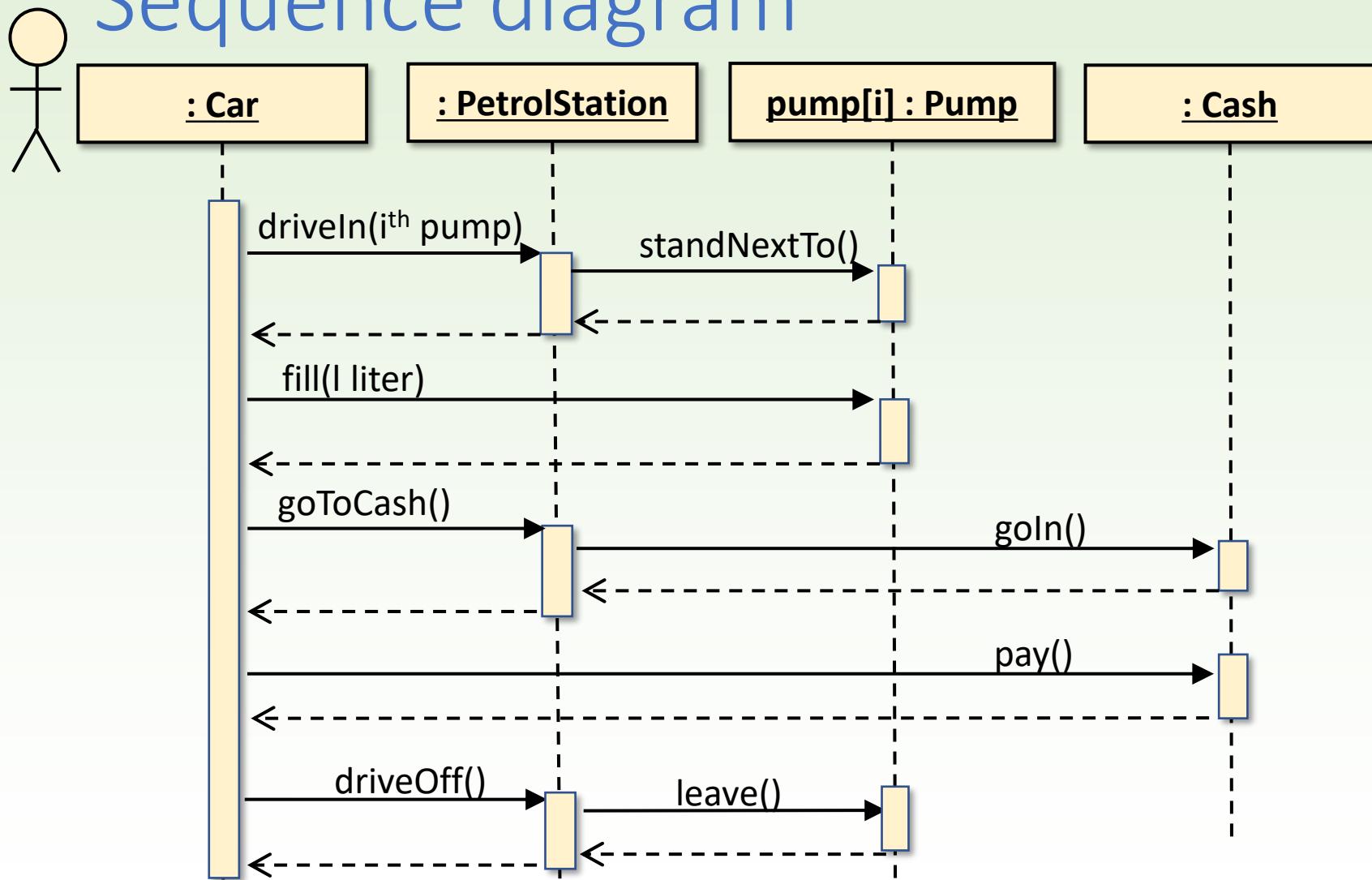
case		description
drives in	GIVEN	there is a petrol station with pumps
	WHEN	drives to an existing pump
	THEN	stands in the queue
refuels	GIVEN	in the queue of a pump
	WHEN	wants to get a given amount of fuels
	THEN	waits for its turns, then fills
goes to pay	GIVEN	there is a petrol station with cash desks
	WHEN	goes to the cash desks
	THEN	goes to a free cash desk or stands in the queue
pays	GIVEN	there is a petrol station with a cash desk, it stands next to a pump and it is at a free cash desk or is waiting in a queue to pay
	WHEN	pays
	THEN	if it is in a queue waiting for its turn, steps out of the queue, the cash computes the money to be paid and the display of the pump is reset
leaves	GIVEN	It stands next to a pump
	WHEN	drives off
	THEN	the queue at the pump becomes shorter

case		description
drives in	GIVEN	there is no petrol station
	WHEN	drives in
	THEN	error
drives in	GIVEN	there is petrol station
	WHEN	drives in to a nonexisting pump
	THEN	error
refuels	GIVEN	it is not at the given pump
	WHEN	fills
	THEN	error
goes to cash desk	GIVEN	there is no petrol station
	WHEN	goes in to the cash desk
	THEN	error
pays	GIVEN	there is no petrol station, or it is not at a pump
	WHEN	pays
	THEN	error
leaves	GIVEN	it is not at a pump
	WHEN	drives off
	THEN	error

Communication diagram



Sequence diagram



Result of the analysis

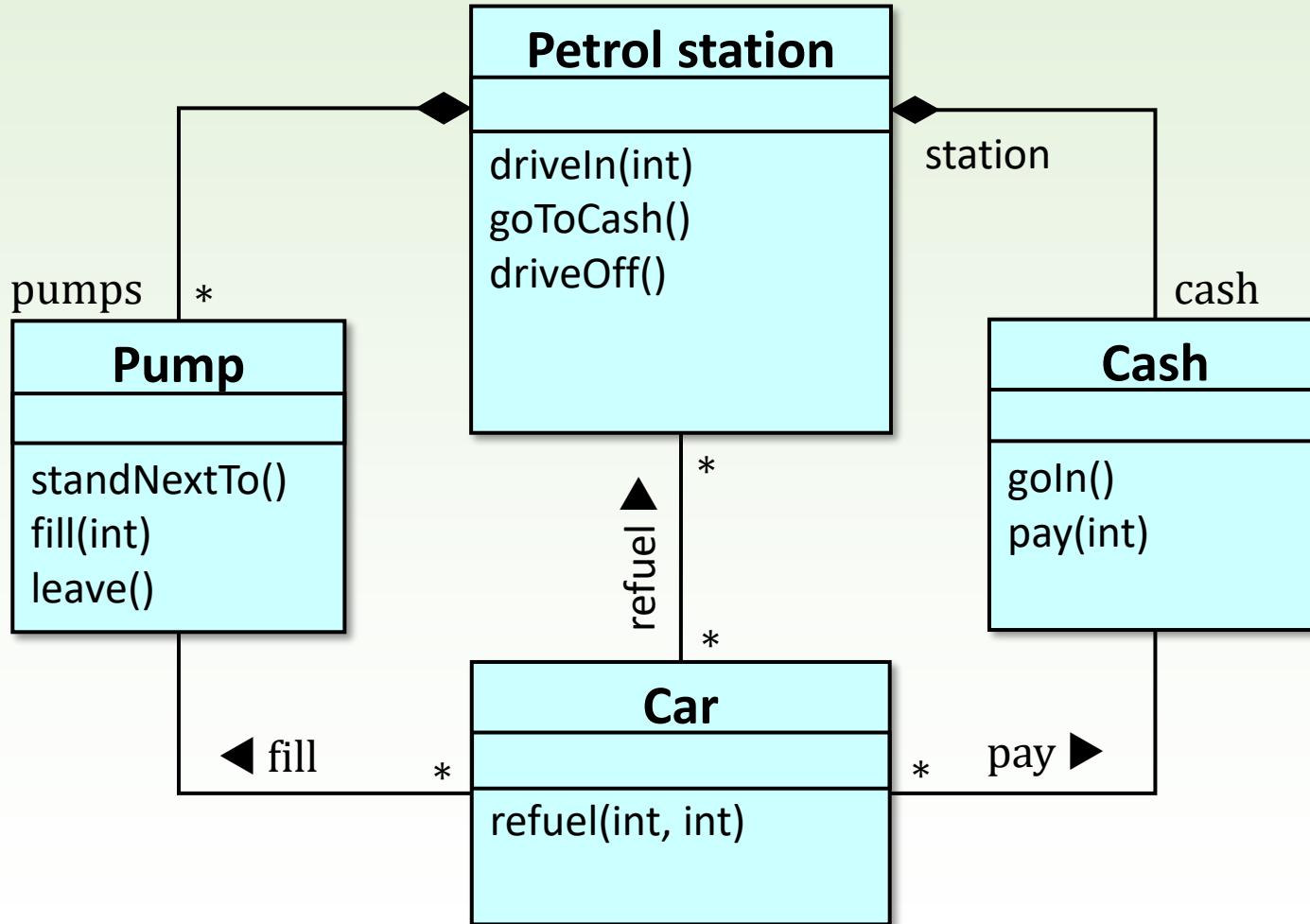
❑ Objects and their activities:

- **cars** (they refuel)
- **petrol station** (where the cars drive in, refuel, go to cash, and pay and from where they drive off)
- **pumps** (where the car stands next to and fills and from where it leaves)
- **cash with more desks** (where the driver goes in and pays)

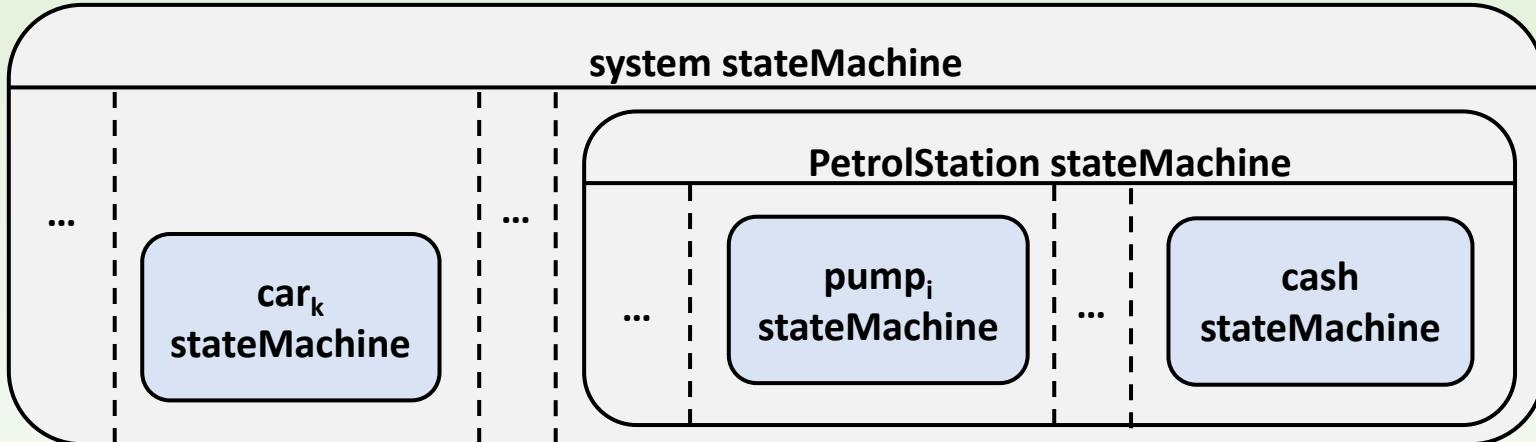
❑ Relationships between objects:

- parts of the petrol station are the pumps and the cash
- a car temporarily gets in touch with a pump and a cash

Class diagram



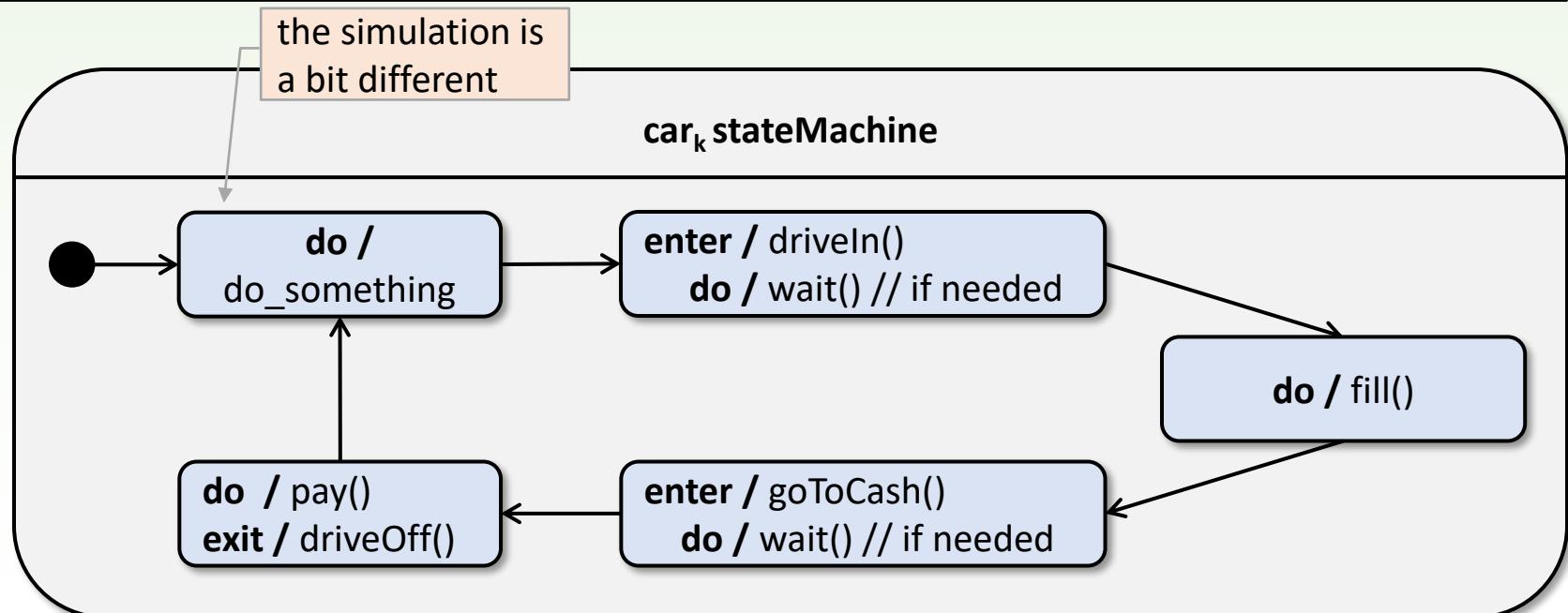
State machine of the system



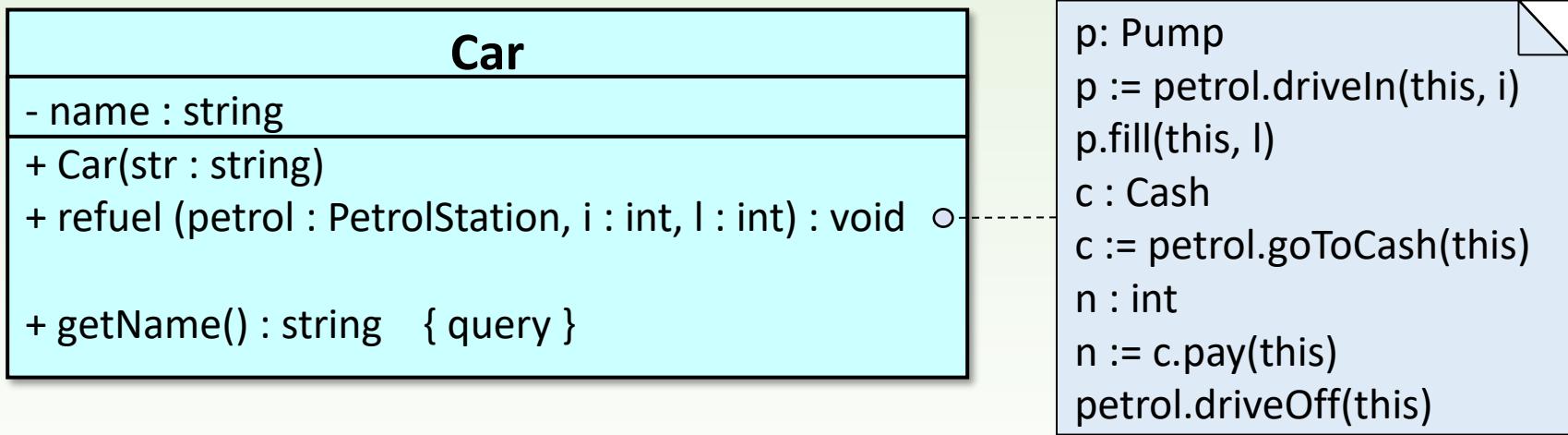
- ❑ State of the system is determined by the state of the cars and the petrol station. State of the petrol station is determined by the state of the pumps and the cash.
- ❑ Cars are the so-called **active objects**: they perform actions concurrently, their state machines run on different threads.
- ❑ Petrol station is a **passive object**: its state machine runs synchronously (by calling its methods) with the state machine of other objects. It does not need other thread.

State machine of the cars

- ❑ A car may be in five states that may change cyclically due to the methods of the petrol station:
 - does something else; drives in (to a pump) and waits; fills; goes to the cash and waits; pays and drives off
- ❑ Transitions are triggered by the end of the actions of the states.



Class Car



Class Car

```
class PetrolStation;
class Car {
public:
    Car(const std::string &str) : _name(str) {}
    ~Car() { _fuel.join(); }
    std::string getName() const { return _name; }
    void refuel(PetrolStation* petrol, unsigned int i, int l) {
        _fuel = new std::thread(activity, this, petrol, i, l);
    }
private:
    std::string _name;
    std::thread _fuel;
    void activity(PetrolStation* petrol, unsigned int i, int l);
};
```

waits for the end of
the extra thread

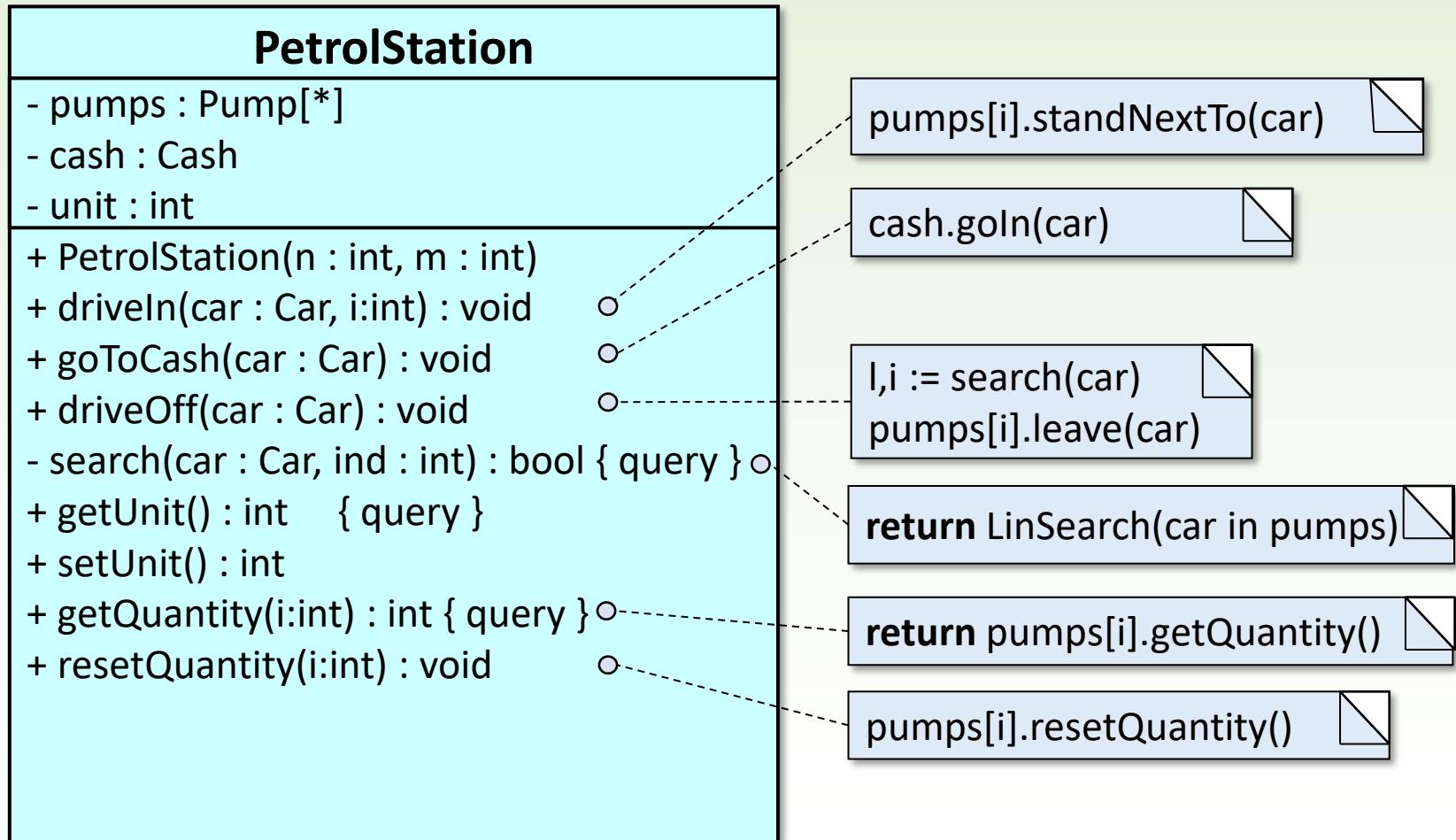
car.h

```
void Car::activity(PetrolStation* petrol, unsigned int i, int l) {
    if ( nullptr == petrol ) return;           // if there is no petrol station
    Pump *pump = petrol ->driveIn(this, i);   // goes to the ith pump
    if ( nullptr == pump ) return;             // if there is no ith pump
    pump->fill(this, l);                    // fills l liter fuel
    Cash *cash = petrol->goToCash(this);
    if ( nullptr == cash ) return;           // if there is no cash
    int n = cash->pay(this);
    petrol->driveOff(this);
}
```

it runs on an extra thread
#include <thread>

car.cpp

Class PetrolStation



Class PetrolStation

```
class PetrolStation {  
public:  
    PetrolStation(int n, int m) {  
        for(int i = 0; i < n; ++i) _pumps.push_back( new Pump() );  
        _cash = new Cash(this, m);  
    }  
    ~PetrolStation() { for( Pump* p : _pumps ) delete p; delete _cash; }  
  
    bool driveIn(Car* car, unsigned int i);  
    void goToCash(Car* car);  
    bool driveOff(Car* car);  
  
    int getUnit() const { return _unit; }  
    void setUnit(int u) { _unit = u; }  
    void resetQuantity(unsigned int i) { _pumps[i]->resetQuantity(); }  
    int getQuantity(unsigned int i) const { return _pumps[i]->getQuantity(); }  
private:  
    std::vector<Pump*> _pumps;  
    Cash* _cash;  
    int _unit;  
  
    bool search(Car* car, unsigned int &ind) const;  
};
```

petrol.h

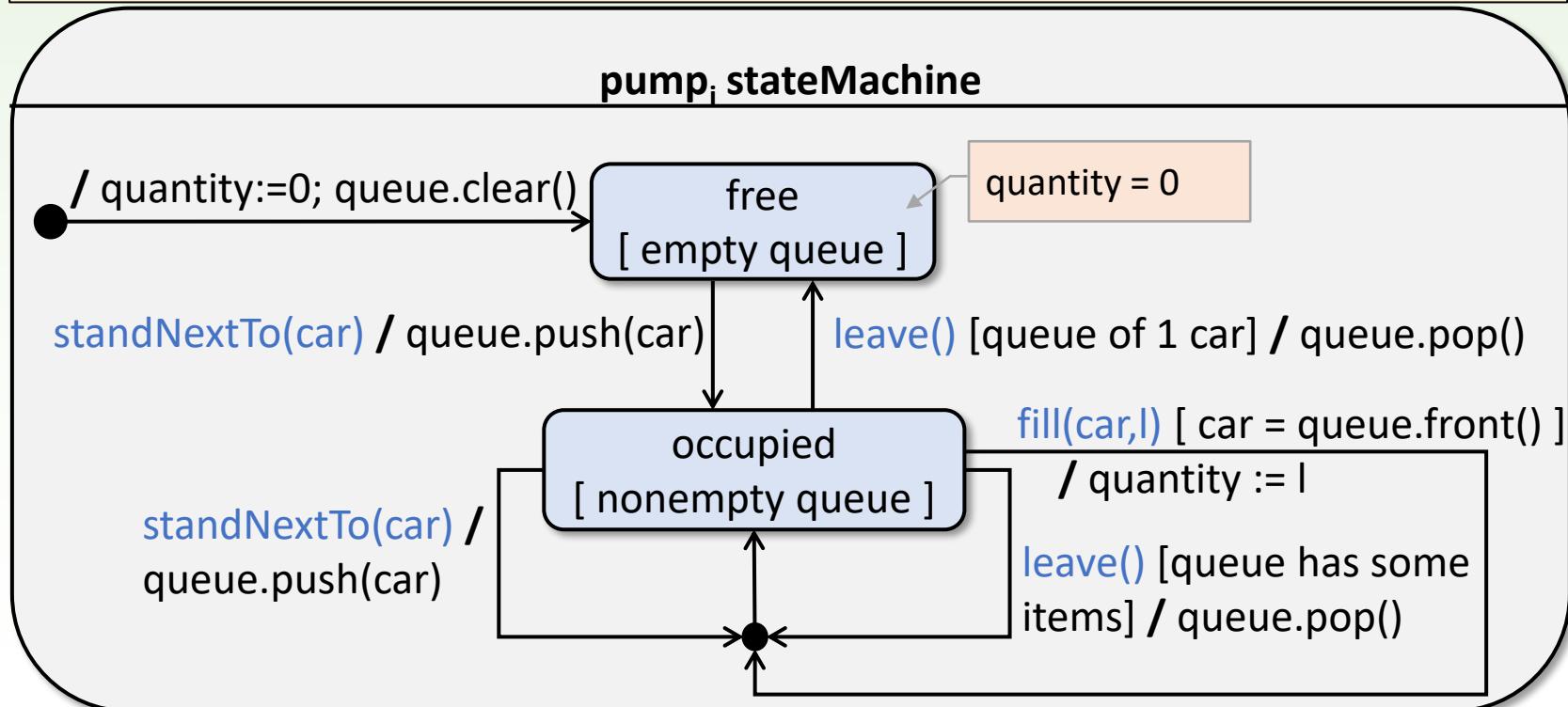
Methods of PetrolStation

```
Pump* PetrolStation::driveIn(Car* car, unsigned int i) {
    if ( i >= _pumps.size() ) return nullptr;
    _pumps[i]->standNextTo(car);
    return _pumps[i];
}
Cash* PetrolStation::goToCash(Car* car) {
    if (nullptr == _cash ) return nullptr;
    _cash->goIn(car);
    return _cash;
}
bool PetrolStation::driveOff(Car* car) {
    unsigned int i;
    if ( !search(car, i) ) return false;
    _pumps[i]->leave();
    return true;
}
bool PetrolStation::search(Car* car, unsigned int &i) const {
    bool l = false;
    for ( i = 0; i < _pumps.size(); ++i) {
        if ( (l = _pumps[i]->getCurrent() == car) ) break;
    }
    return l;
}
```

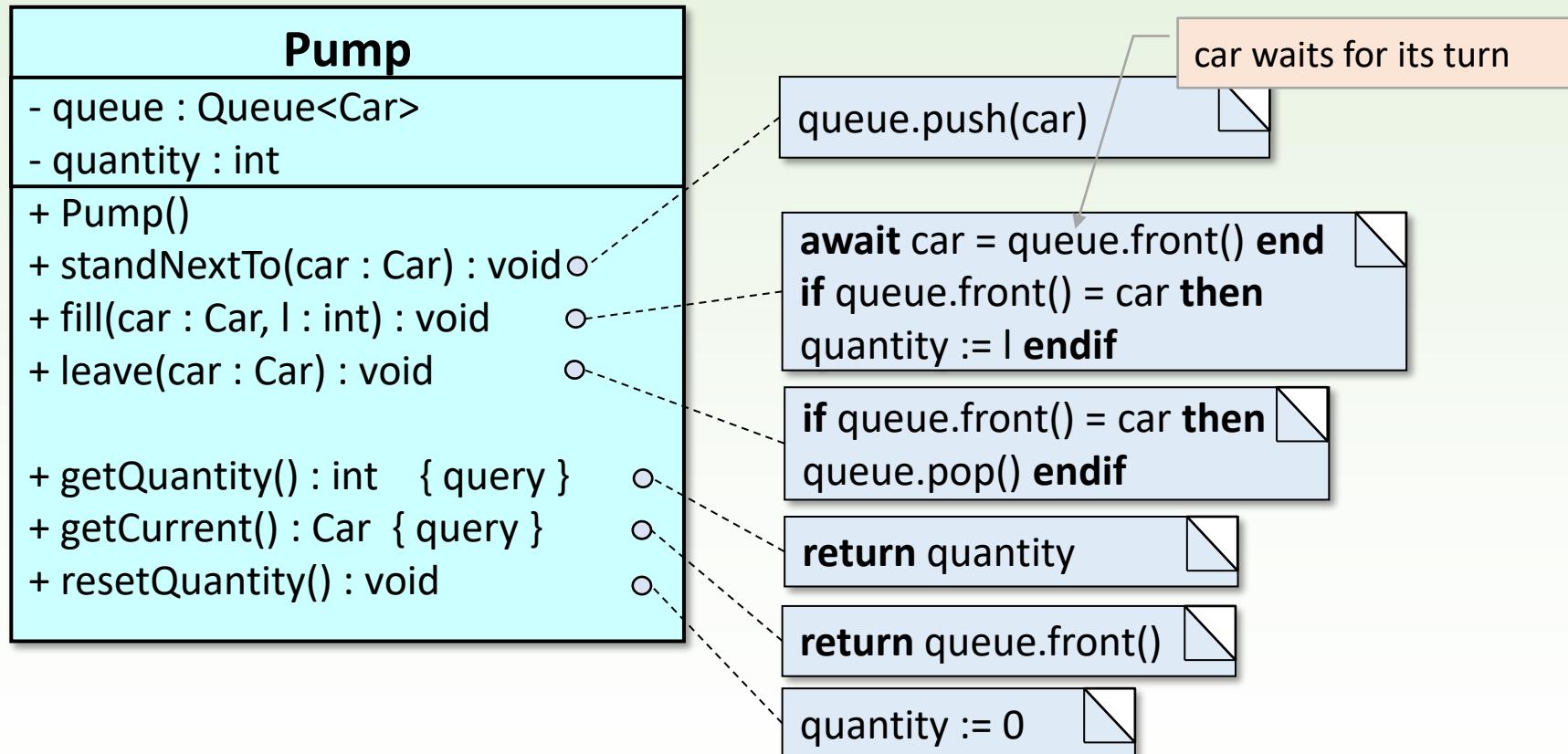
petrol.cpp

State machine of a pump

- ❑ A pump may be **free** or **occupied**.
- ❑ Methods standNextTo() and leave() effect the queue at the pump.
- ❑ Method fill() can be executed only in state occupied, when the car is in the front. In this case, the **quantity** of the fuel to be filled is given which can be seen on the display of the pump until the payment.



Class Pump



Class Pump

```
class Car;

class Pump {
public:
    Pump() : _quantity(0) { }

    void standNextTo(Car* car);
    void fill(Car* car, int l);
    void leave(Car* car);

    Car* getCurrent() const { return _queue.front(); }
    int getQuantity() const { return _quantity; }
    void resetQuantity() { _quantity = 0; }

private:
    int _quantity;
    std::queue<Car*> _queue;
    std::mutex _mu;
    std::condition_variable _cond;
};

#include <mutex>
#include <condition>
```

pump.h

Methods of Pump

```
void Pump::standNextTo(Car* car)
{
    std::unique_lock<std::mutex> lock(_mu);
    _queue.push(car);
}
```

to avoid simultaneous
acces to `_queue`

```
void Pump::fill(Car* car, int l)
{
    std::unique_lock<std::mutex> lock(_mu);
    while( car != _queue.front() ) _cond.wait(lock);
    _quantity = l;
}
```

thread is waiting:
it "falls asleep"

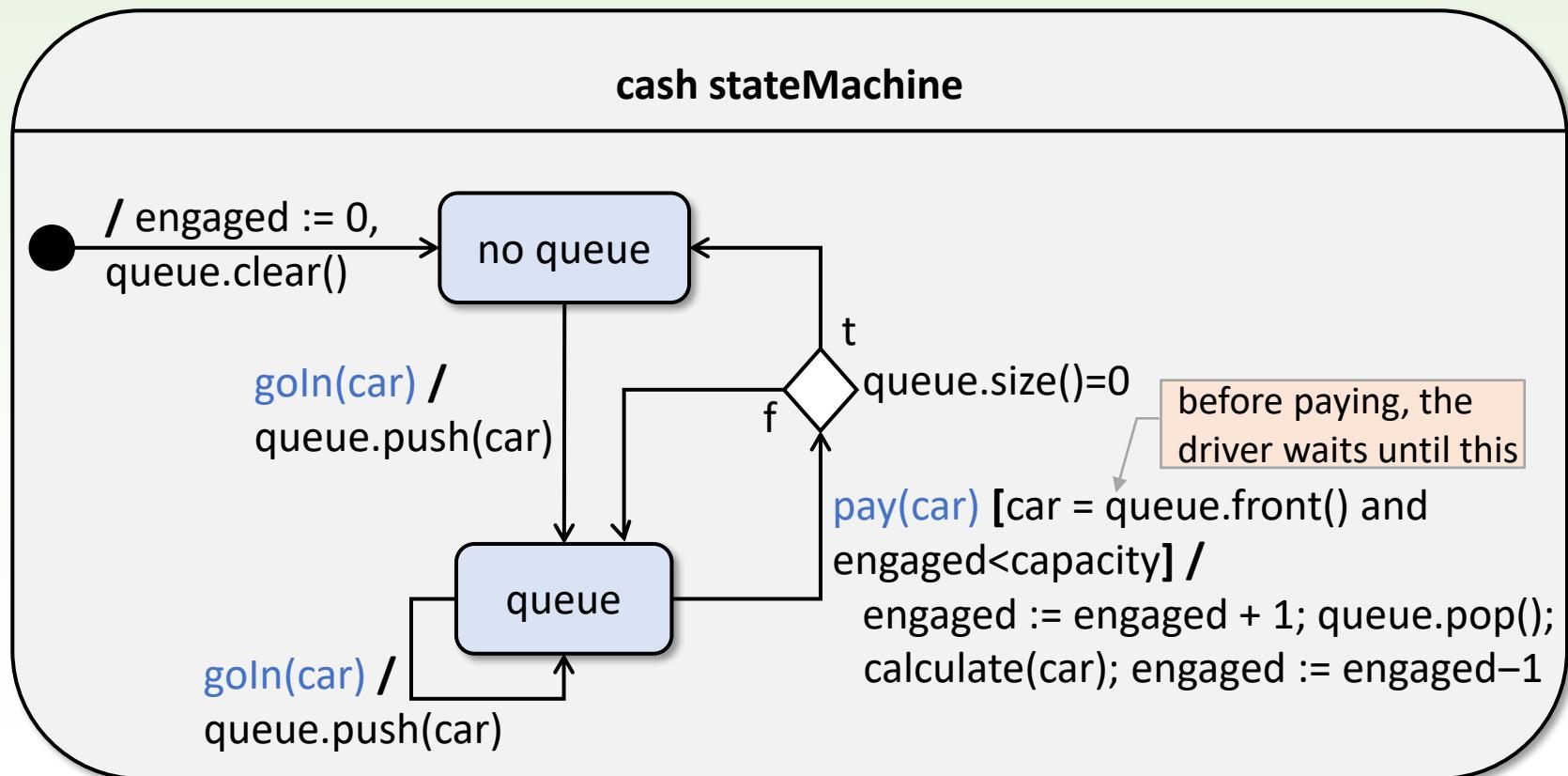
```
void Pump::leave(Car* car)
{
    std::unique_lock<std::mutex> lock(_mu);
    if( car == _queue.front() ) _queue.pop();
    _cond.notify_all();
}
```

"wakes up" all those threads
that are waiting at cond

pump.cpp

State machine of Cash

- ❑ In the cash, the drivers (cars) stand in queue.
- ❑ The states are effected by methods `goIn()` and `pay()`. Only that driver (car) can pay which is at the front of the queue and there is a free desk.



Cash

- ❑ A cash may be “used” by more cars. As many drivers (cars) may pay (**engaged**) as the number of the cash desks (**capacity**). The rest is waiting in the **queue**.

Cash	
- queue : Queue<Car>	
- engaged : int	
- capacity : int	
- station : PetrolStation	
+ Cash(station : PetrolStation, cp : int)	
+ goIn(car : Car) : void	○
+ pay(car : Car) : int	○

queue.push(car)

```
await queue.front() = car and  
engaged < capacity end  
engaged := engaged +1  
queue.pop()  
l := station.search(car,i)  
if not l then return nil endif  
amount := station.getQuantity(i) *  
                  station.getUnit()  
station.resetQuantity(i)  
engaged := engaged - 1  
return amount
```

waits for its turn

index of the pump

Class Cash

```
class PetrolStation;
class Car;

class Cash {
public:
    Cash(PetrolStation* station, int cp): _station(station),
        _capacity(cp) {}
    void goIn(Car* car);
    int pay(Car* car);
private:
    PetrolStation* _station;

    std::atomic_int _engaged;
    int _capacity;
    std::queue<Car*> _cashQueue;

    std::mutex _mu;
    std::condition_variable _cond;
};
```

cash.h

Methods of Cash

```
int Cash::pay(Car* car)
{
    std::unique_lock<std::mutex> lock(_mu);
    while( _cashQueue.front() != car || _engaged == _capacity ) {
        _cond.wait(lock);
    }
    ++_engaged;
    _cashQueue.pop();
    _cond.notify_all();
    _mu.unlock();
}

unsigned int i;
if ( !_station->search(car, i) ) return nullptr;
int amount = _station->getQuantity(i) * _station->getUnit();
_station->resetQuantity(i); // resets the display of the ith pump
--_engaged;
_cond.notify_all();
return amount;
}
```

```
void Cash::goIn(Car* car)
{
    std::unique_lock<std::mutex> lock(_mu);
    _cashQueue.push(car);
}
```

thread is waiting

starts those threads that
are waiting at cond

cash.cpp

Event handling

Asynchronous communication

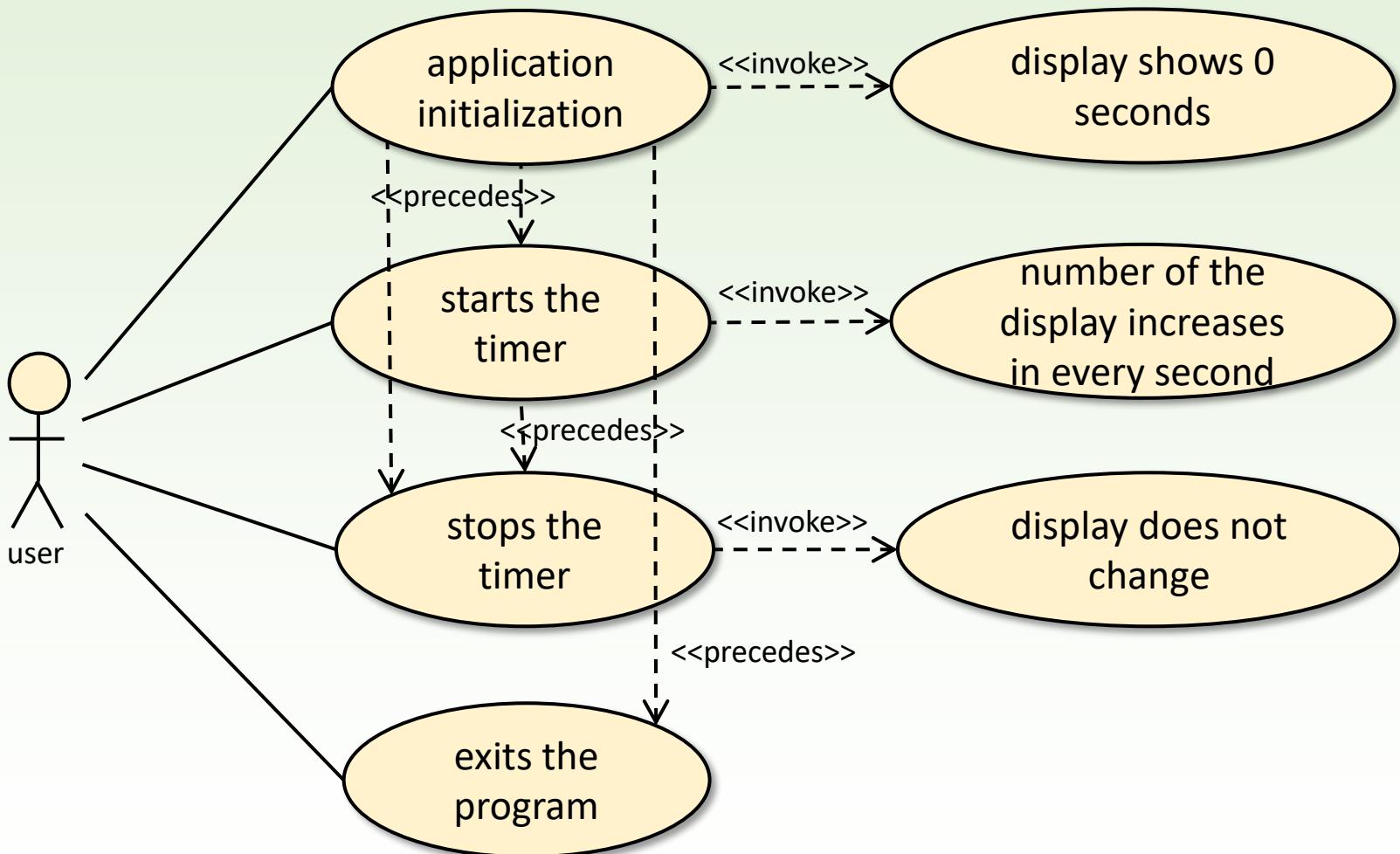
Task

Create a timer which displays the time passed in every second. The process

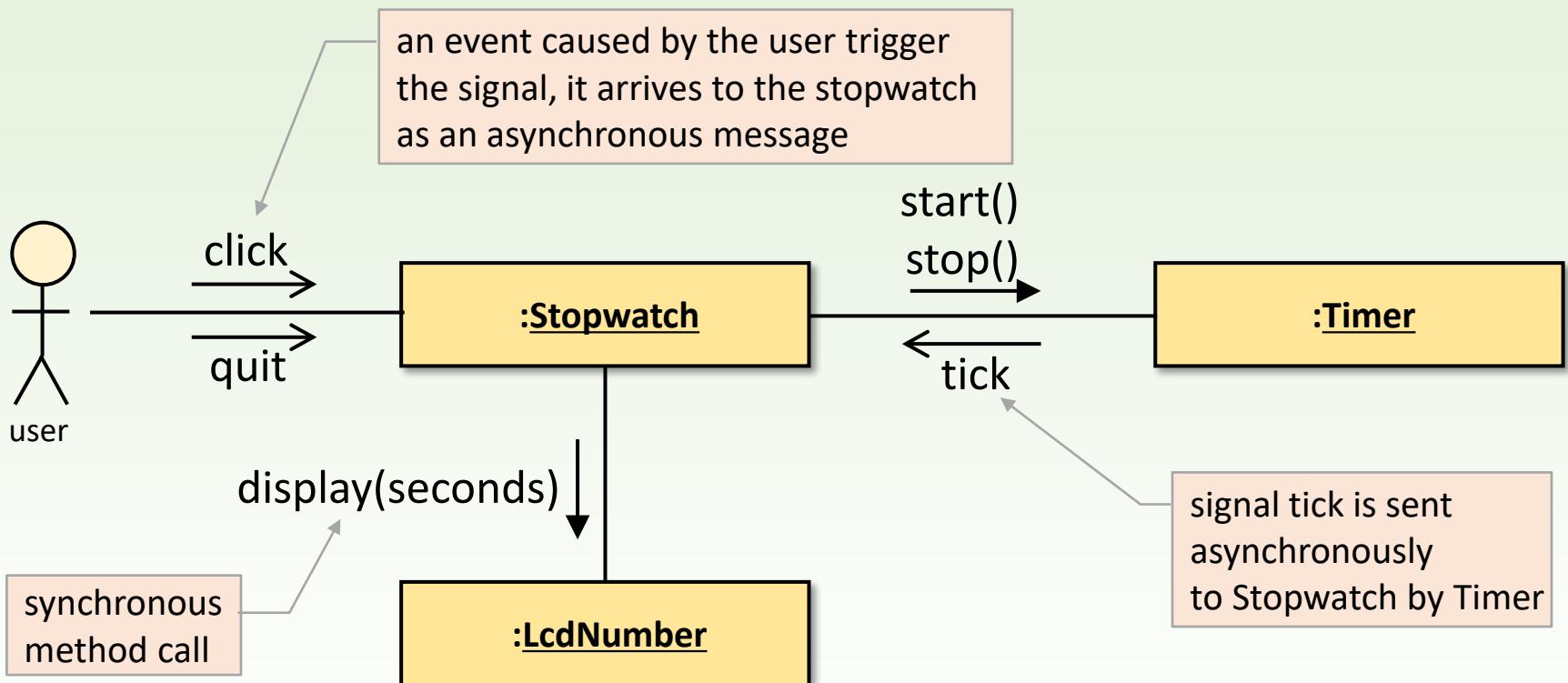
- starts due to a signal,
- pauses and continues due to the same signal, and
- due to another signal, it stops.



Use case diagram



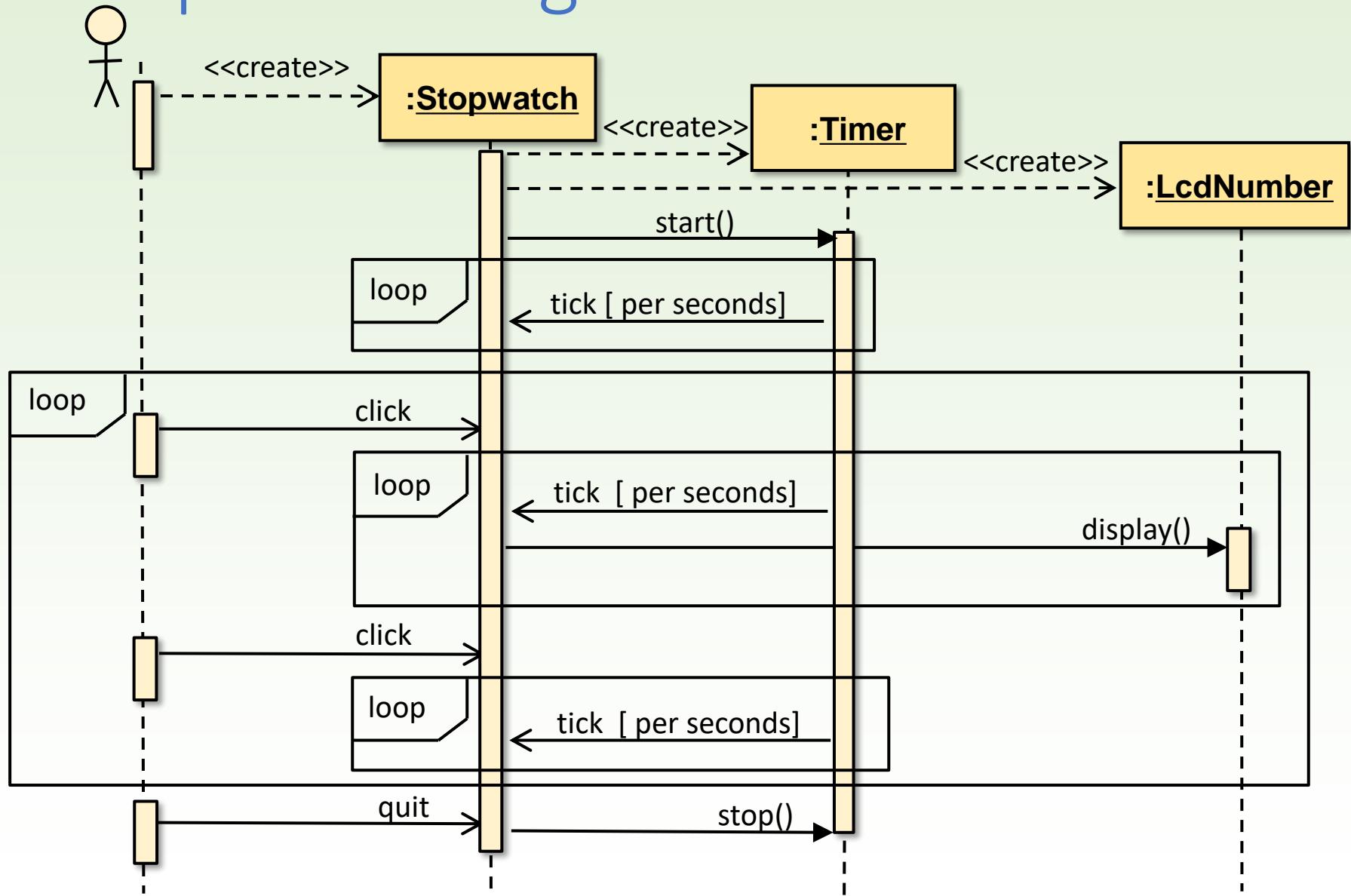
Communication diagram: planning



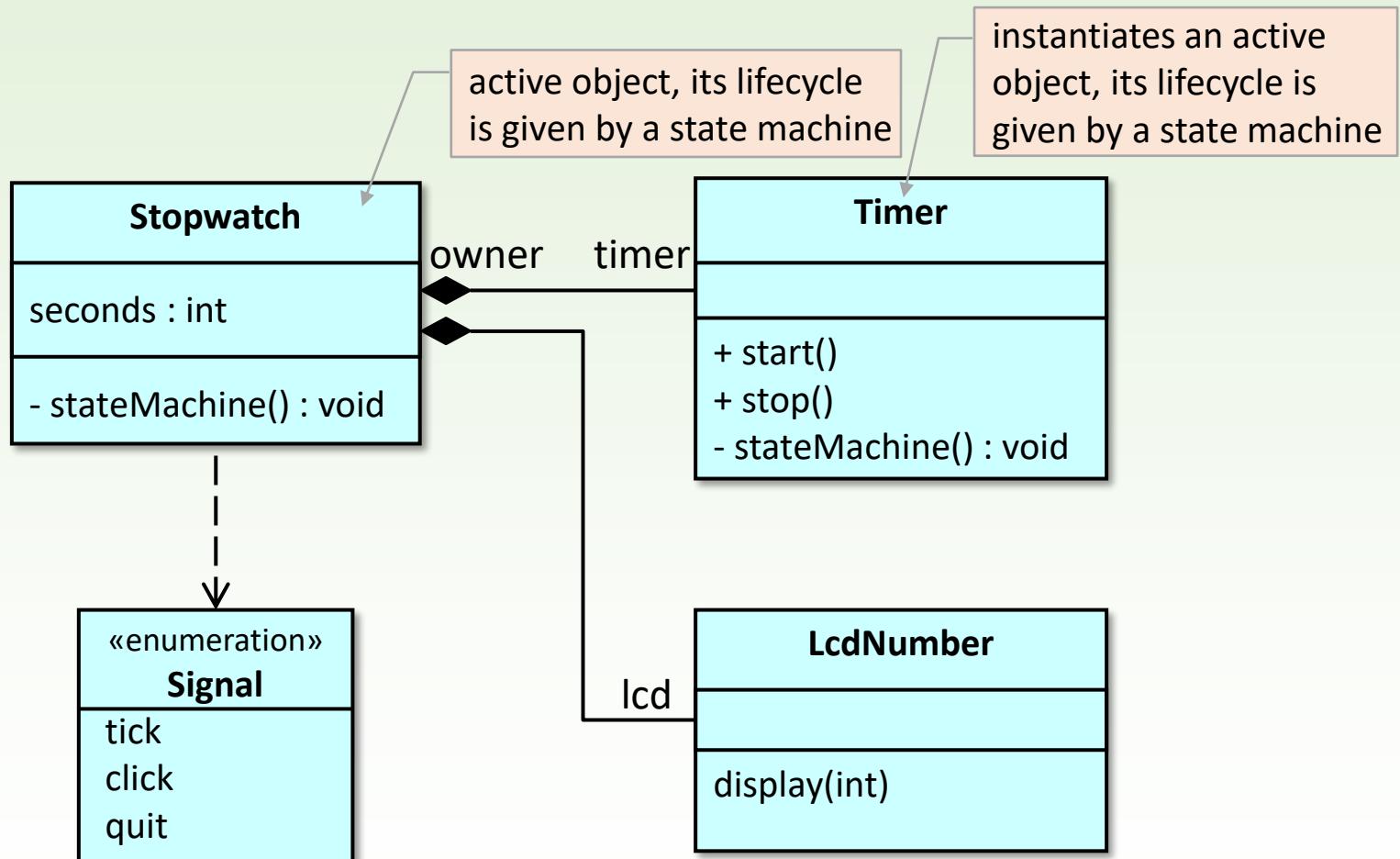
☐ Stopwatch and Timer are the so-called **active objects**.

- Stopwatch has to process the signals while the new ones arrive.
- Timer has to send a signal in every second no matter what.

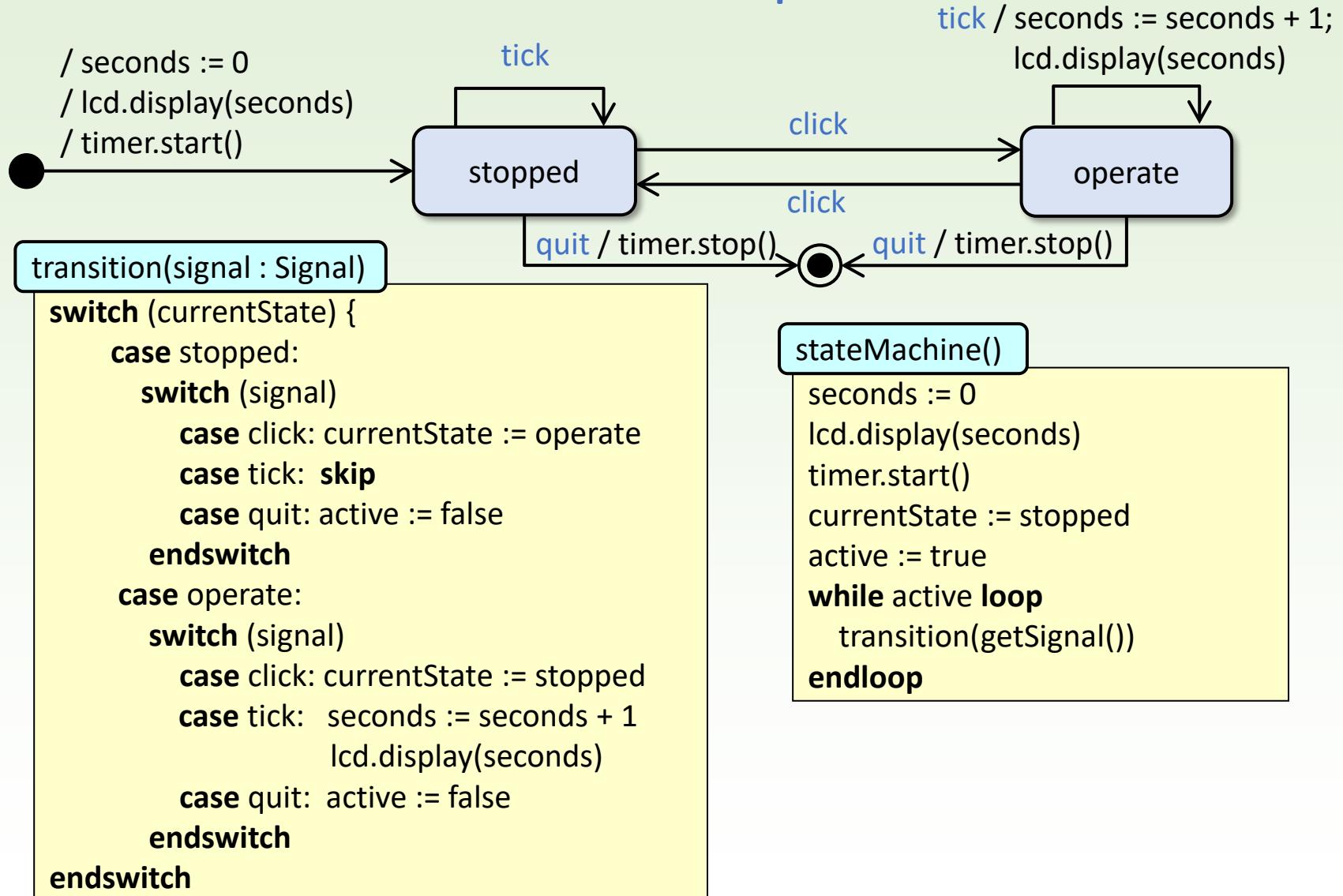
Sequence diagram



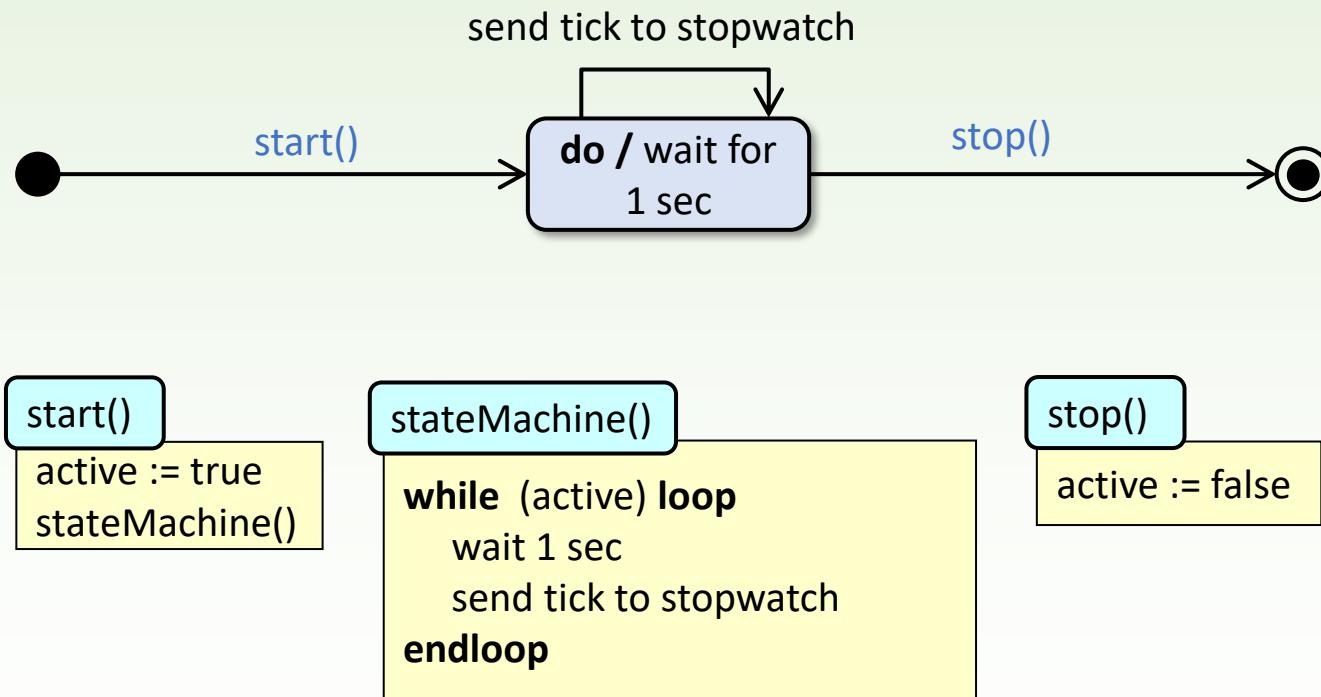
Class diagram: analysis



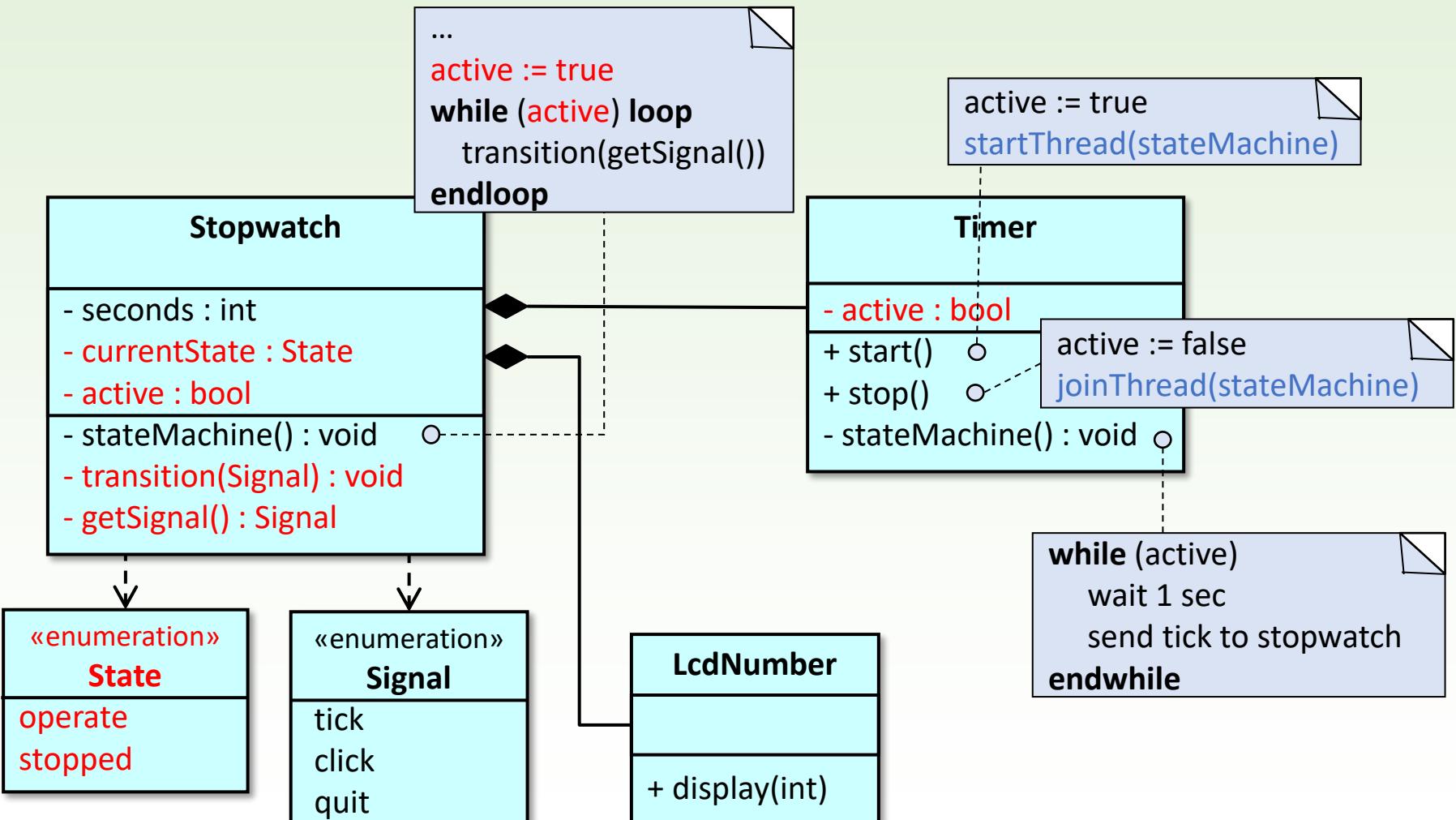
State machine of Stopwatch



State machine of Timer



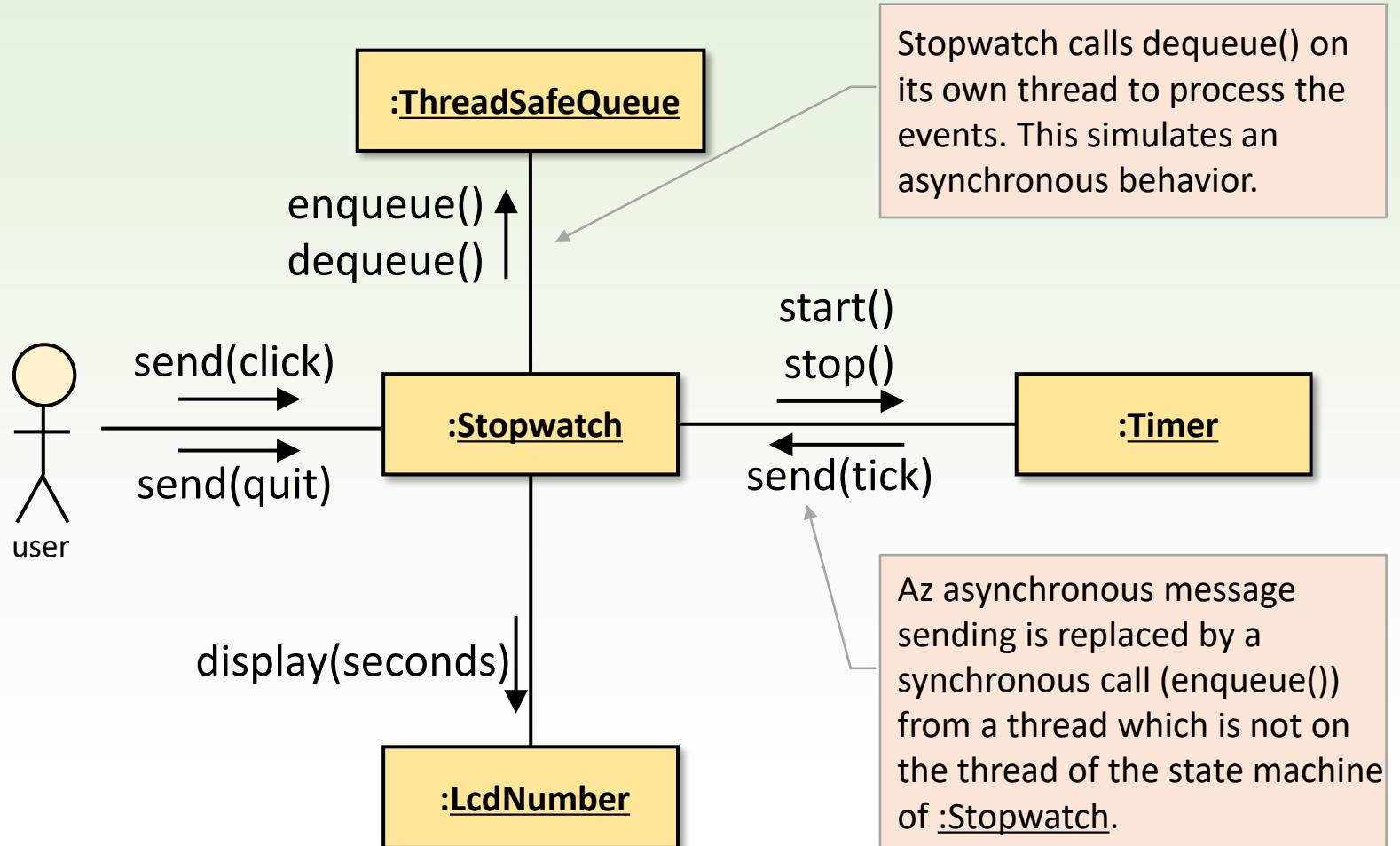
Class diagram: planning



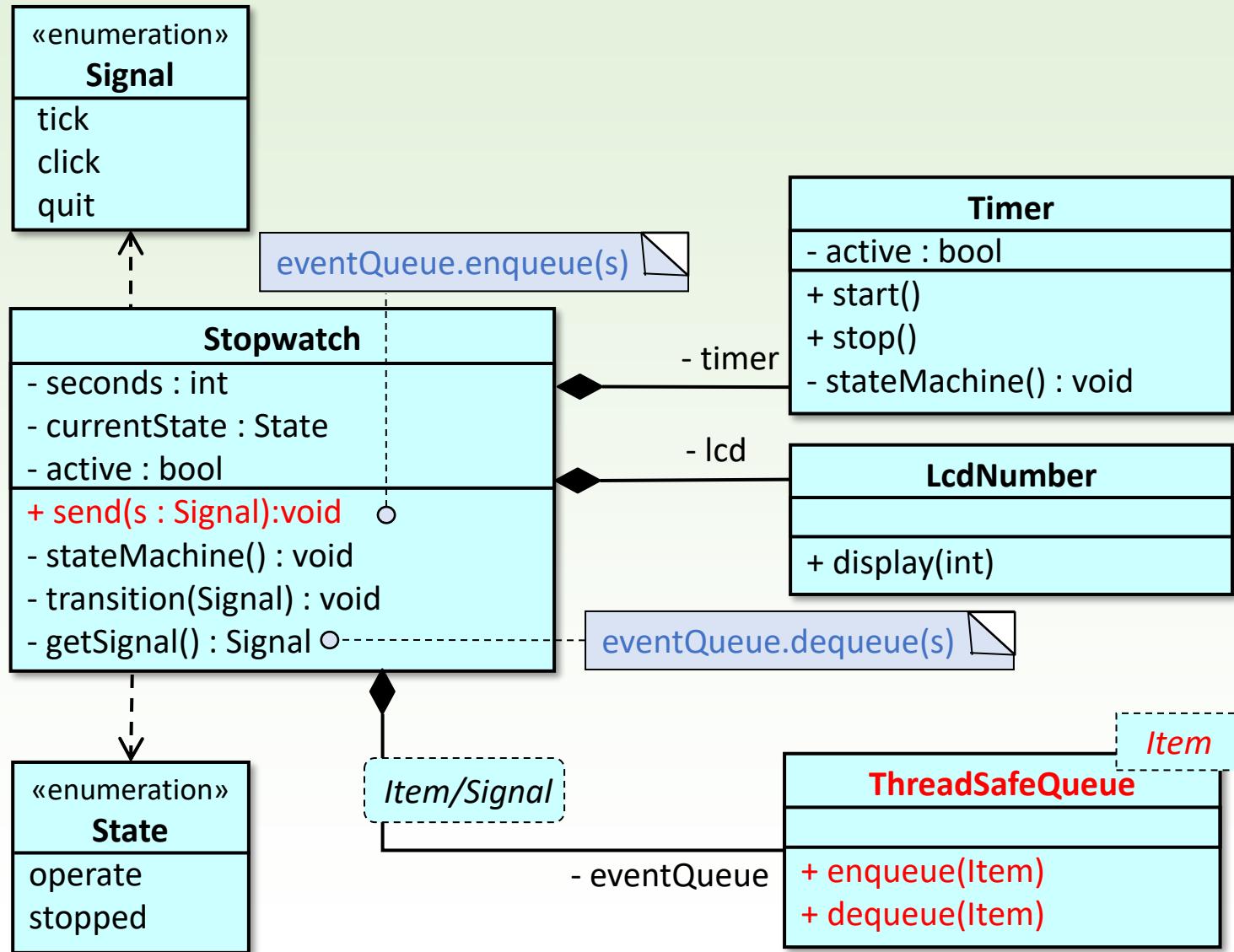
Realization

- ❑ Multithread application is needed, as there are several active objects.
 - Different threads are needed for the state machines of stopwatch and timer.
- ❑ The stopwatch object collects the asynchronous messages (signals) arriving from different sources into one event queue.
 - Method `send()` belonging to object stopwatch pushes the signals into the event queue. This method is called on the own thread of the sender.
 - From the event queue, signals are got by the state machine of the stopwatch (`getSignal()`) which runs on its own thread.
 - Operations of the queue, `send()` and `getSignal()`, have to be synchronized: they have to work in a mutually exclusive way. In addition, in case of empty queue, `getSignal()` has to be blocked with a waiting command.

Communication diagram: realization



Class diagram: realization



Class Stopwatch

```
enum Signal {tick, click, quit};  
  
class Stopwatch {  
public:  
    Stopwatch();  
    ~Stopwatch();  
    void send(Signal event) { _eventQueue.enqueue(event); }  
private:  
    enum State { operate, stopped };  
  
    void stateMachine();  
    void transition(Signal event);  
    Signal getSignal();  
  
    Timer _timer;  
    LcdNumber _lcd;  
    ThreadSafeQueue<Signal> _eventQueue;  
  
    State _currentState;  
    int _seconds;  
    bool _active;  
    std::thread _thread;  
};
```

New thread for Stopwatch::stateMachine()
#include <thread>

stopwatch.h

Class Stopwatch

```
Stopwatch::Stopwatch() : _timer(this)
{
    _thread = new std::thread(&Stopwatch::stateMachine, this);
    _eventQueue.startQueue();
}
Stopwatch::~Stopwatch()
{
    _thread->join();
    _eventQueue.stopQueue();
}
```

state machine of Stopwatch
is run on separate thread

waits for the ending of the state machine

```
void Stopwatch::stateMachine()
{
    _seconds = 0;
    _lcd.display(_seconds);
    _timer.start();
    _currentState = stopped;
    _active = true;
    while (_active) {
        transition(getSignal());
    }
    _timer.stop();
}
Signal getSignal()
{
    Signal s;
    _eventQueue.dequeue(s);
    return s;
}
```

stopwatch.cpp

Event handler of Stopwatch

```
void Stopwatch::transition(Signal signal)
{
    switch (_currentState) {
        case stopped:
            switch (signal) {
                case click: _currentState = operate; break;
                case tick : break;
                case quit : _active = false; break;
            }
            break;
        case operate:
            switch (signal) {
                case click: _currentState = stopped; break;
                case tick : _lcd.display(++_seconds); break;
                case quit : _active = false; break;
            }
            break;
    }
}
```

stopwatch.cpp

Class Timer

```
class Stopwatch;

class Timer{
    typedef std::chrono::milliseconds milliseconds;
public:
    Timer(Stopwatch *t) : _owner(t), _active(false)
    {}

    void start();
    void stop();
private:
    void stateMachine();

    Stopwatch* _owner;
    bool _active;
    std::thread _thread;
};
```

timer.h is already included by stopwatch.h,
and Stopwatch is needed here.

new thread for Timer::stateMachine()
#include <thread>

timer.h

Class Timer

```
void Timer::start() {
    _active = true;
    _thread = std::thread(&Timer::stateMachine, this);
}

void Timer::stop() {
    _active = false;
    _thread.join();
}

void Timer::stateMachine() {
    std::condition_variable _cond;
    std::mutex mu;
    while (_active) {
        std::unique_lock<std::mutex> lock(mu);
        _cond.wait_for(lock, milliseconds(1000));
        _owner->send(tick);
    }
}
```

new thread for the state machine of Timer

semaphore for waiting
#include <condition_variable>
#include <mutex>

blocks the thread for 1 second

timer.cpp

Class LcdNumber

```
class LcdNumber
{
public:
    void display(int seconds)
    {
        std::cout << extend((seconds % 3600) / 60) + ":" +
            extend((seconds % 3600) % 60) << std::endl;
    }
private:
    std::string extend(int n) const
    {
        std::ostringstream os;
        os << n;
        return (n < 10 ? "0" : "") + os.str();
    }
};
```

Lcdnumber.h

Template ThreadSafeQueue

```
template <typename Item>
class ThreadSafeQueue
{
public:
    ThreadSafeQueue() { _active = false; }

    void enqueue(const Item& e) ;
    void dequeue(Item& e);

    void startQueue() { _active = true; }
    void stopQueue() { _active = false; _cond.notify_all(); }
    bool empty() const { return _queue.empty(); }

private:
    std::queue<Item> _queue;
    bool _active;

    std::mutex _mu;
    std::condition_variable _cond;
};
```

all threads blocked by `_cond` (state machine of stopwatch) are permitted to continue

threadsafequeue.h

Template ThreadSafeQueue

```
template <typename Item>
void ThreadSafeQueue::enqueue(const Item& e)
{
    std::unique_lock<std::mutex> lock(_mu);
    _queue.push(e);
    _cond.notify_one();
}

template <typename Item>
void ThreadSafeQueue::dequeue(Item& e)
```

one of the threads blocked at `_cond` is allowed to continue (there is only one)

waits as long as the queue is empty and active

enqueue() and dequeue() may be called mutually exclusively

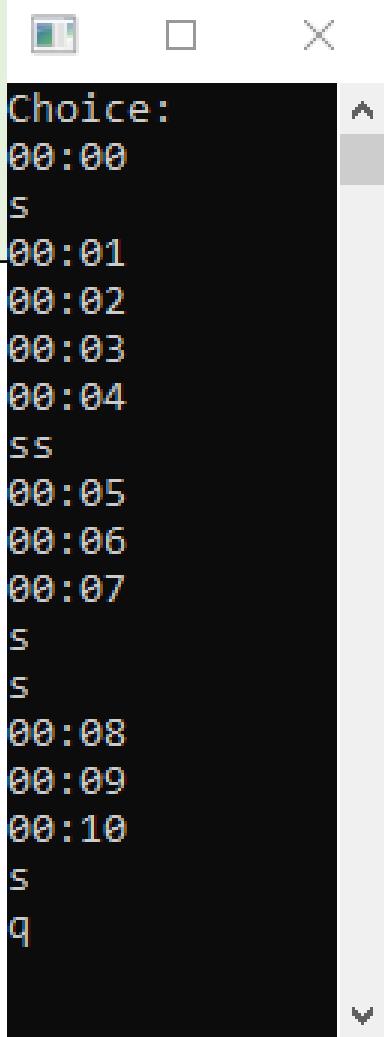
threadsafequeue.hpp

main()

```
int main()
{
    Stopwatch stopwatch;
    std::cout << "Choice:" << std::endl;
    char o;

    do {
        std::cin >> o;
        if(o == 's') {
            stopwatch.send(click);
        }
    } while(o != 'q');
    stopwatch.send(quit);

    return 0;
}
```



```
Choice:
00:00
s
00:01
00:02
00:03
00:04
ss
00:05
00:06
00:07
s
s
00:08
00:09
00:10
s
q
```

main.cpp

Unique and general elements of an event-driven application

unique

- ❑ creating the necessary objects for the application and for this
 - (inheritance of) unique classes
 - plan their position on the user interface
- ❑ creating event handler functions
- ❑ connecting signals and event handler functions

general

- ❑ objects with typical appearance and signal sending habits (window, LCD display, timer)
- ❑ plan of the user interface
- ❑ event handling mechanism
 - asynchronous signal sending,
(sender and receiver on different threads)
 - safe handling of the event queue

Stopwatch user interface

User interface is usually drawn by a visual planner, by which

- the stopwatch may be created on an individual window
- components of the stopwatch may be defined and instantiated (LCD display, timer)
- visible components may be arranged



The stopwatch is a window-like object the close of which triggers signal stop. It contains an LCD display, an invisible timer sending tick signals and a push button sending click signals.

Event handling of Stopwatch

To the signals triggered by the events handling methods have to be connected and the methods have to be implemented.

```
switch (currentState) {  
    case stopped:  
        switch (signal)  
            case click: currentState := operate  
            case tick: skip  
            case quit: timer.stop()  
        endswitch  
    case operate:  
        switch (signal)  
            case click: currentState := stopped  
            case tick: seconds := seconds +1  
                lcd.display(seconds)  
            case quit: timer.stop()  
        endswitch  
    endswitch
```

```
switch (signal) {  
    case click:  
        switch (currentState) {  
            case stopped: currentState := operate  
            case operate: currentState := stopped  
        endswitch  
    case tick:  
        switch (currentState) {  
            case stopped:  
            case operate: seconds := seconds +1  
                lcd.display(seconds)  
        endswitch  
    case quit:  
        timer.stop()  
    endswitch
```

Stopwatch developed by Qt



Stopwatch : public QWidget

Window-like controller object containing other controllers (timer, display, push button).
Its closure triggers signal quit.

QLCDNumber with method display

QPushbutton-type object triggering signal click

```
#include < QApplication>
#include "stopwatch.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Stopwatch *stopwatch = new Stopwatch;
    stopwatch->show();
    return app.exec();
}
```

QApplication

Among others, it takes care of the events to get the proper controllers where they trigger signals.

main.cpp

Class Stopwatch as a QWidget

```
#include <QWidget>
class QTimer;
class QLCDNumber;
class QPushButton;
enum State {stopped, operate};
class Stopwatch: public QWidget
{
    Q_OBJECT
public:
    Stopwatch(QWidget *parent = 0);
protected:
    void closeEvent(QCloseEvent* event) { _timer->stop(); }
private:
    QTimer      *_timer;
    QLCDNumber  *_lcd;
    QPushButton *_button;
    State _currentState;
    int _seconds;
    QString Stopwatch::format(int n) const;
    QString Stopwatch::extend(int n) const;
private slots:
    void oneSecondPass(); // tick
    void buttonPressed(); // click
};
```

event handler of signal
quit triggered on exit

QTimer-type object
triggers signal tick

event handler of the other signals

stopwatch.h

Qt event handlers of class Stopwatch

```
Stopwatch::oneSecondPass() {
    switch (_currentState) {
        case operate: _lcd->display(format(++_seconds)); break;
        case stopped: break;
    }
}

Stopwatch::buttonPressed() {
    switch (_currentState) {
        case operate: _currentState = stopped; break;
        case stopped: _currentState = operate; break;
    }
}

QString Stopwatch::format(int n) const
{
    return extend((n % 3600) / 60) + ":" + extend((n % 3600) % 60);
}

QString Stopwatch::extend(int n) const
{
    return (n < 10 ? "0" : "") + QString::number(n);
```

event handler of signal tick
(same as the native C++ code
in void transition())

event handler of signal tick
(same as the native C++ code
in void transition())

same as the native C++ code
in class LcdNumber

stopwatch.cpp

Qt constructor of class Stopwatch

```
Stopwatch::Stopwatch(QWidget *parent) : QWidget(parent)
{
    setWindowTitle(tr("Stopwatch"));
    resize(150, 60);

    _timer = new QTimer;
    _lcd = new QLCDNumber;
    _button = new QPushButton("Start/Stop");

    ...
    connect(_timer, SIGNAL(timeout()), this, SLOT(oneSecondPass()));
    connect(_button, SIGNAL(clicked()), this, SLOT(buttonPressed()));

    _currentState = stopped;
    _seconds = 0;
    _lcd->display(_seconds);
    _timer->start(1000);
}
```

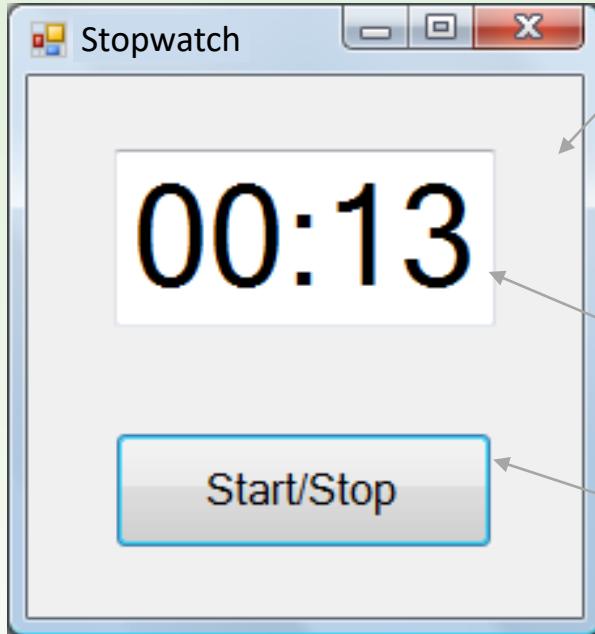
here the arrangement and the properties of the controllers are given

connecting signals and handlers:
tick (timeout()) ~ oneSecondPass()
click (clicked()) ~ buttonPressed()

part of the code may be generated automatically by using a visual designer (QtDesigner)

stopwatch.cpp

Stopwatch developed under .net



Stopwatch : Form

Window-like controller object containing other controllers (timer, display, push button).

Its closure triggers signal quit.

TextBox with method display

Button-type object triggering signal click

Application

Among others, it takes care of the events to get the proper controllers where they trigger signals.

```
static class Program
{
    [STAThread]
    static void Main() {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Stopwatch());
    }
}
```

program.cs

Class Stopwatch as a .net Form

```
public partial class Stopwatch : Form
{
    enum State { stopped, operate };
    State _currentState;
    DateTime _seconds = new DateTime(0);

    private System.Windows.Forms.Timer _timer;
    private System.Windows.Forms.Button _button;
    private System.Windows.Forms.TextBox _lcd;

    public Stopwatch() { ... }

    private void timer_Tick(object sender, EventArgs e){ ... }
    private void button_Click(object sender, EventArgs e){ ... }
    private void MainForm_FormClosed(object sender, FormClosedEventArgs e)
    { ... }

    private void display()
    {
        _lcd.Text = string.Format("{0}:{1}",
            seconds.Minute.ToString().PadLeft(2, '0'),
            seconds.Second.ToString().PadLeft(2, '0'));
    }
}
```

part of the code may be generated automatically by using a visual designer

event handlers of signals tick, click, and quit

this belongs to stopwatch instead of the LCD display

Stopwatch.cs

.net event handlers of Stopwatch

```
private void timer_Tick(object sender, EventArgs e)
{
    switch (currentState) {
        case State.operate:
            seconds = seconds.AddSeconds(1);
            display();
            break;
        case State.stopped: break;
    }
}

private void button_Click(object sender, EventArgs e)
{
    switch (currentState) {
        case State.operate:
            currentState = State.stopped;
            break;
        case State.stopped:
            currentState = State.operate;
            break;
    }
}

private void MainForm_FormClosed(object sender, FormClosedEventArgs e)
{
    _timer.Stop();
}
```

Both belong to method transition() in native C++.

Stopwatch.cs

.net constructor of Stopwatch

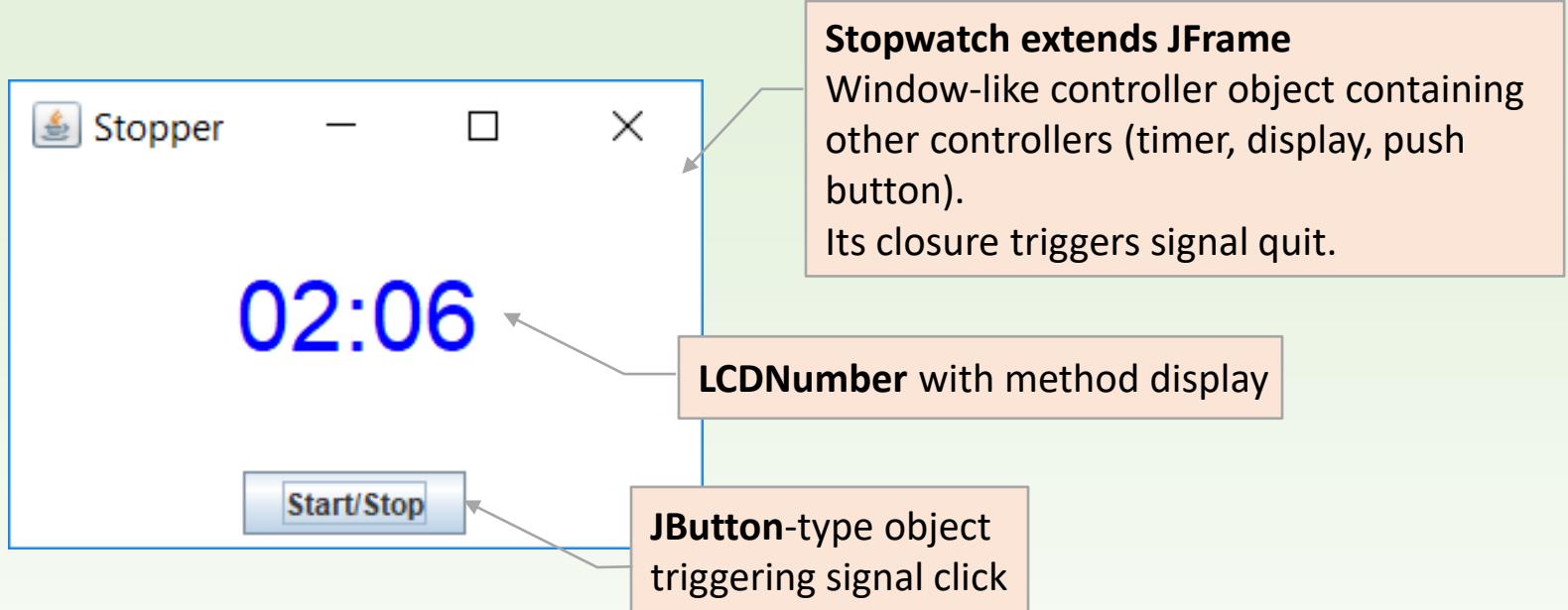
```
public Stopwatch() {
    this.components = new System.ComponentModel.Container();
    this.button = new System.Windows.Forms.Button();
    this.lcd = new System.Windows.Forms.TextBox();
    this.timer = new System.Windows.Forms.Timer(this.components);
    ...
    this.Text = "Stopwatch";
    this.button.Text = "Start/Stop";
    this.lcd.Text = "00:00";
    this.timer.Interval = 1000;
    ...
    this.Controls.Add(this.lcd);
    this.Controls.Add(this.button);
    ...
    ...
    this._timer.Tick += new System.EventHandler(this.timer_Tick);
    this._button.Click += new System.EventHandler(this.button_Click);
    this.FormClosed += new System.Windows.Forms.
        FormClosedEventHandler(MainForm_FormClosed);
    ...
    _currentState = State.stopped;
    display();
    _timer.Start();
}
```

Here, the arrangement and the properties of the controllers are given. It may be generated automatically with a visual designer.

connecting signals and their handlers

Stopwatch.cs

Stopwatch in Java



```
public class Stopwatch extends JFrame
{
    ...
    public static void main(String[] args) {
        new Stopwatch();
    }
}
```

Stopwatch.java

Stopwatch in Java

```
public class Stopwatch extends JFrame
{
    enum State { operate, stopped }
    private State currentState;
    private int seconds = 0;

    private final static int SECOND = 1000 /* milliseconds */;
    private Timer timer = new Timer(SECOND, null);
    private LCDNumber lcd = new LCDNumber("00:00");
    private JButton button = new JButton("Start/Stop");
    private JPanel buttonPanel = new JPanel();

    public Stopwatch() { ... }

    void click() { ... }

    void tick() { ... }

    protected void finalize() throws Throwable { ... }

    public static void main(String[] args) {
        new Stopwatch();
    }
}
```

Stopwatch.java

Event handlers of Stopwatch in Java

```
void click() {  
    switch (currentState) {  
        case operate :  
            currentState = State.stopped;  
            break;  
        case stopped :  
            currentState = State.operate;  
            break;  
    }  
}  
  
void tick() {  
    switch (currentState) {  
        case operate :  
            ++seconds;  
            lcd.display( format(seconds) );  
            break;  
        case stopped :    break;  
    }  
}  
  
protected void finalize() throws Throwable {  
    if (timer.isRunning()) timer.stop();  
    super.finalize();  
}
```

event handler of signal tick
(same as the native
C++ code in void transition())

event handler of signal click
(same as the native
C++ code in void transition())

signal quit implies
the stopping of the timer

Stopwatch.java

Constructor of Stopwatch in Java

```
public Stopwatch() {  
    super("Stopwatch");  
    setBounds(250, 250, 300, 200);  
    setDefaultCloseOperation(EXIT_ON_CLOSE);  
    buttonPanel.setBackground(Color.WHITE);  
    buttonPanel.add(button);  
    add(lcd);  
    add(buttonPanel, "South");  
  
    button.addActionListener(new ActionListener() {  
        @Override  
        public void actionPerformed(ActionEvent e) { click(); }  
    });  
  
    timer.addActionListener(new ActionListener() {  
        @Override  
        public void actionPerformed(ActionEvent e) { tick(); }  
    });  
  
    currentState = State.stopped;  
    timer.start();  
    setVisible(true);  
}
```

quit

connecting events
with handlers

Stopwatch.java

LCD display in Java

```
public class LcdNumber extends JLabel {  
    public LcdNumber(String text) {  
        super(text);  
        setHorizontalAlignment(JLabel.CENTER);  
        setOpaque(true);  
        setBackground(Color.WHITE);  
        setForeground(Color.BLUE);  
       setFont(new Font(Font.DIALOG, Font.PLAIN,  
        40));  
    }  
  
    public void display(String text) {  
        setText(String.format("%02d:%02d",  
            (seconds % 3600) / 60,           // minutes  
            (seconds % 3600) % 60));      // seconds  
    }  
}
```

formatting belongs to the display

LcdNumber.java

Object oriented program

- ❑ **Object:** vacation
- ❑ **Oriented?** YES!
- ❑ **Program:**
 - 10am Breakfast
 - 11am Beach
 - 1pm Lunch
 - 2pm Siesta
 - 5pm Beach
 - 7pm Dinner
 - 8pm Go out
 - 1am Bedtime



vacation