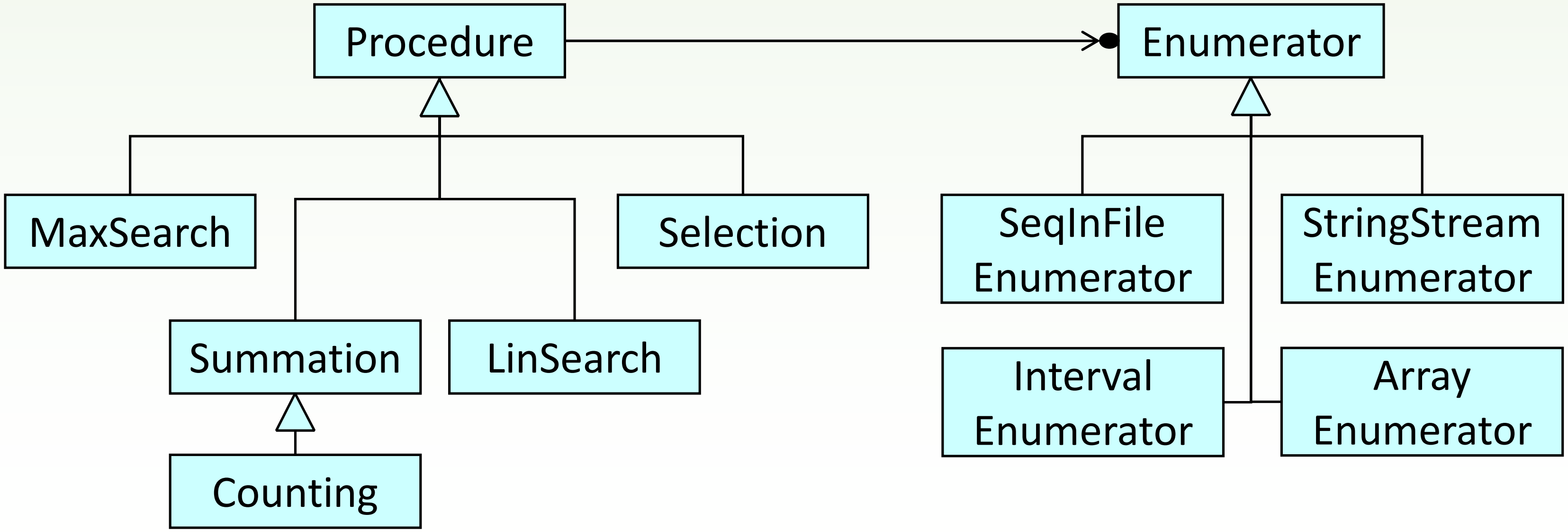
Library of class templates of the algorithmic patterns

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 paramters 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());
     }
}
```

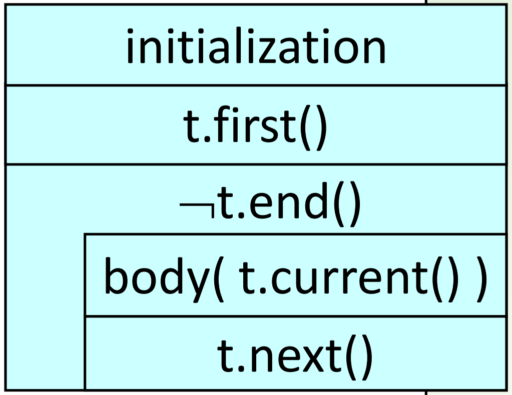
type of the enumerated items

pointer of the enumerator object

default: `_enor->first()`

default: `!_enor->end() && whileCond(_enor->current())`

default: `true`



initialization

t.first()

¬t.end()

body(t.current())

t.next()

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

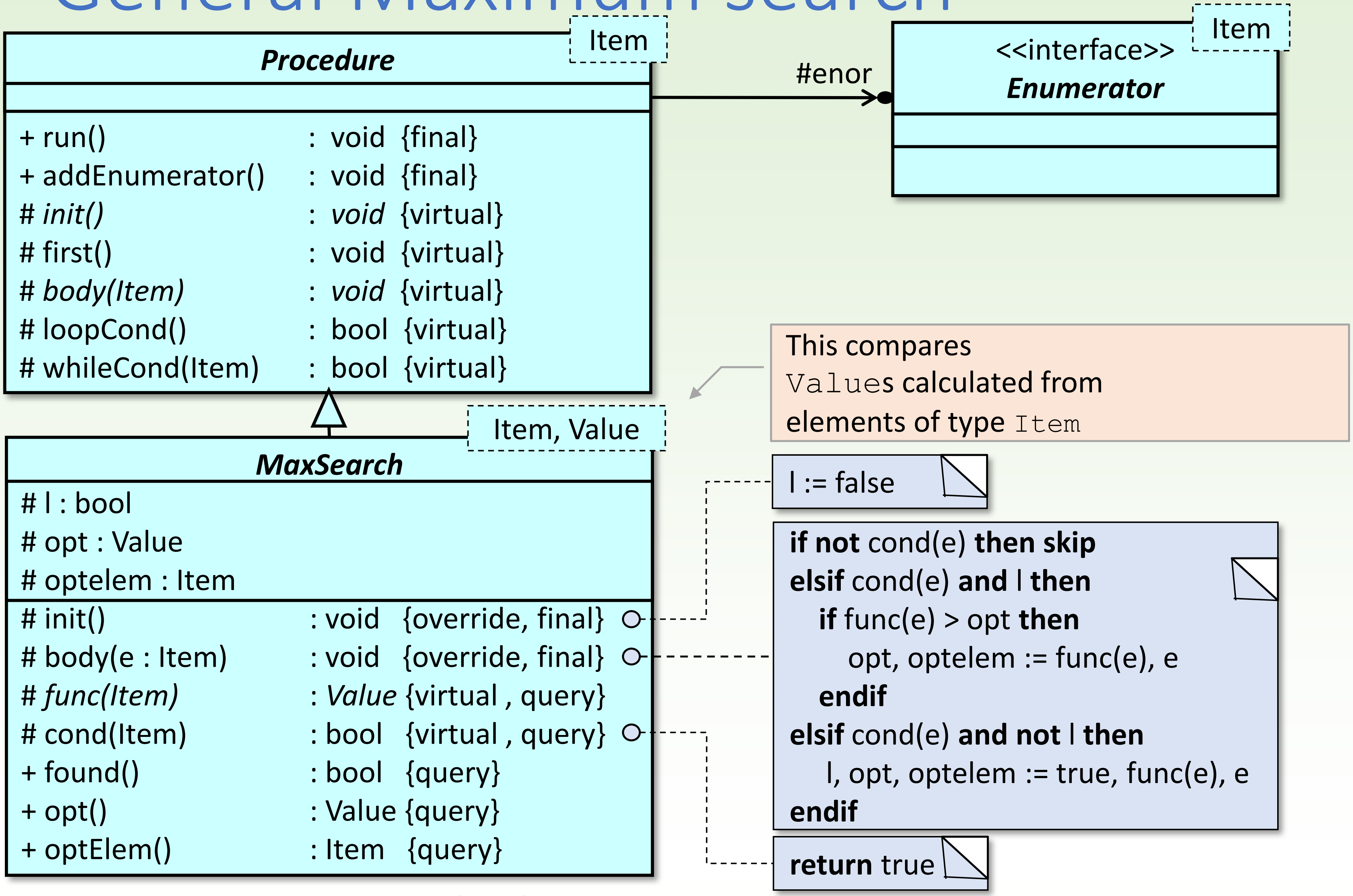
prime loop

cannot be overridden in the children

adding a concrete enumerator object

procedure.hpp

General Maximum search



- + run() : void {final}
- + addEnumerator() : void {final}
- # *init()* : *void* {virtual}
- # first() : void {virtual}
- # *body(Item)* : *void* {virtual}
- # loopCond() : bool {virtual}
- # whileCond(Item) : bool {virtual}

This compares `Values` calculated from elements of type `Item`

- # l : bool
- # opt : Value
- # optelem : Item

- # init() : void {override, final}
- # body(e : Item) : void {override, final}
- # *func(Item)* : *Value* {virtual , query}
- # cond(Item) : bool {virtual , query}
- + found() : bool {query}
- + opt() : Value {query}
- + optElem() : Item {query}

```
l := false
```

```
if not cond(e) then skip
elsif cond(e) and l then
    if func(e) > opt then
        opt, optelem := func(e), e
    endif
elsif cond(e) and not l then
    l, opt, optelem := true, func(e), e
endif
```

```
return true
```

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

condition of the Maximum 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;
    }
}
```

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

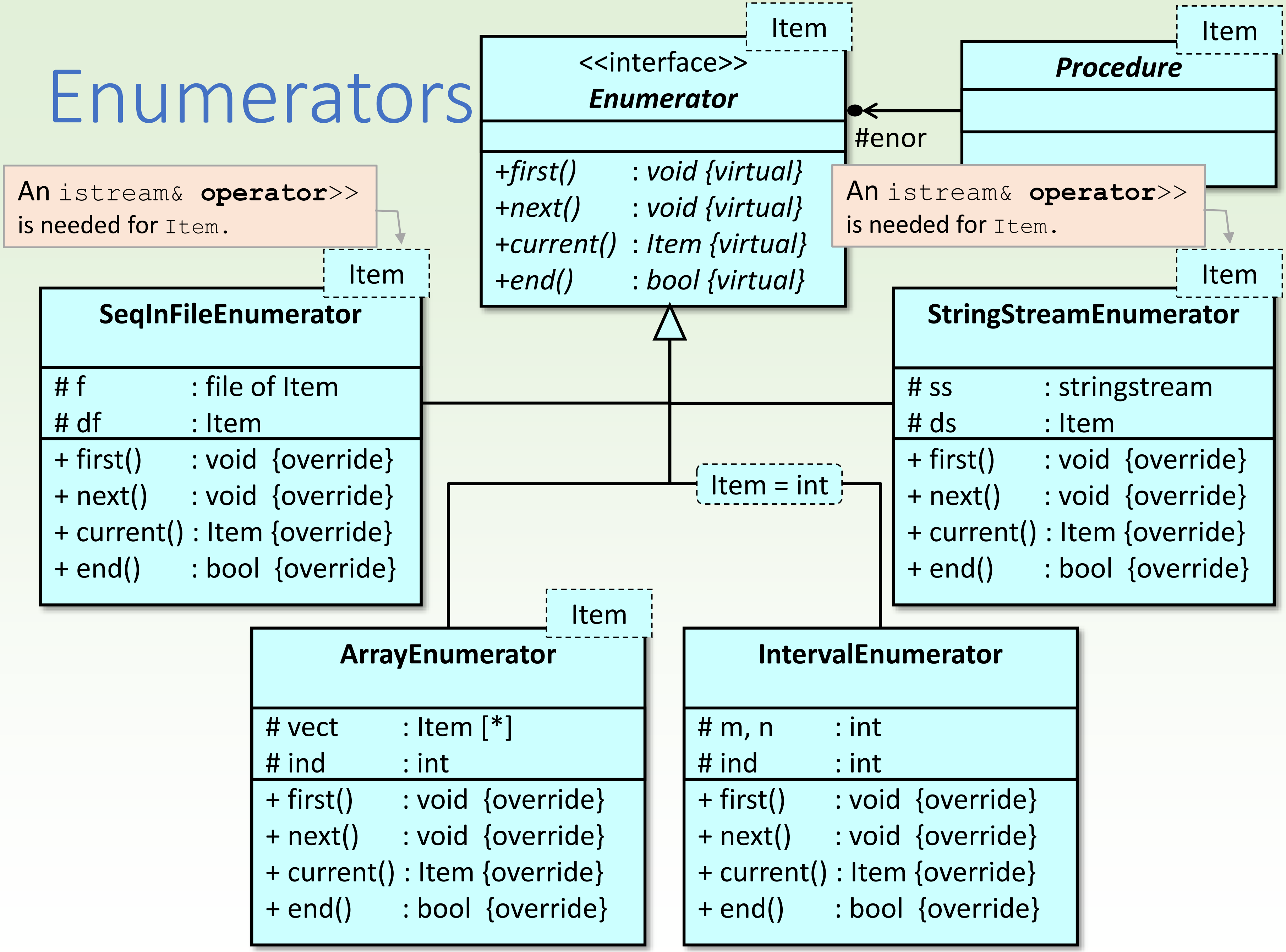
```
template <typename Value>
class Greater{
public:
    bool operator()(const Value& l, const Value& r)
    {
        return l > r;
    }
};

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



Item

<<interface>>

Enumerator

+first()   : void {virtual}
+next()   : void {virtual}
+current() : Item {virtual}
+end()   : bool {virtual}

Item

Procedure

#enor

An `istream& operator>>` is needed for `Item`.

Item

SeqInFileEnumerator

# f   : file of Item
# df   : Item

+ first()   : void {override}
+ next()   : void {override}
+ current() : Item {override}
+ end()   : bool {override}

An `istream& operator>>` is needed for `Item`.

Item

StringStreamEnumerator

# ss   : stringstream
# ds   : Item

+ first()   : void {override}
+ next()   : void {override}
+ current() : Item {override}
+ end()   : bool {override}

Item = int

Item

ArrayEnumerator

# vect   : Item [*]
# ind   : int

+ first()   : void {override}
+ next()   : void {override}
+ current() : Item {override}
+ end()   : bool {override}

IntervalEnumerator

# m, n   : int
# ind   : int

+ first()   : void {override}
+ next()   : void {override}
+ current() : Item {override}
+ end()   : bool {override}

Interval- and array enumerator

intervalenumerator.hpp

```
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; }
};
```

arrayenumerator.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]; }
};
```

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`.

seqinfileenumerator.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:infile($\mathbb{N}$) , l:$\mathbb{L}$, max:$\mathbb{N}$

Pre : $f = f_0$

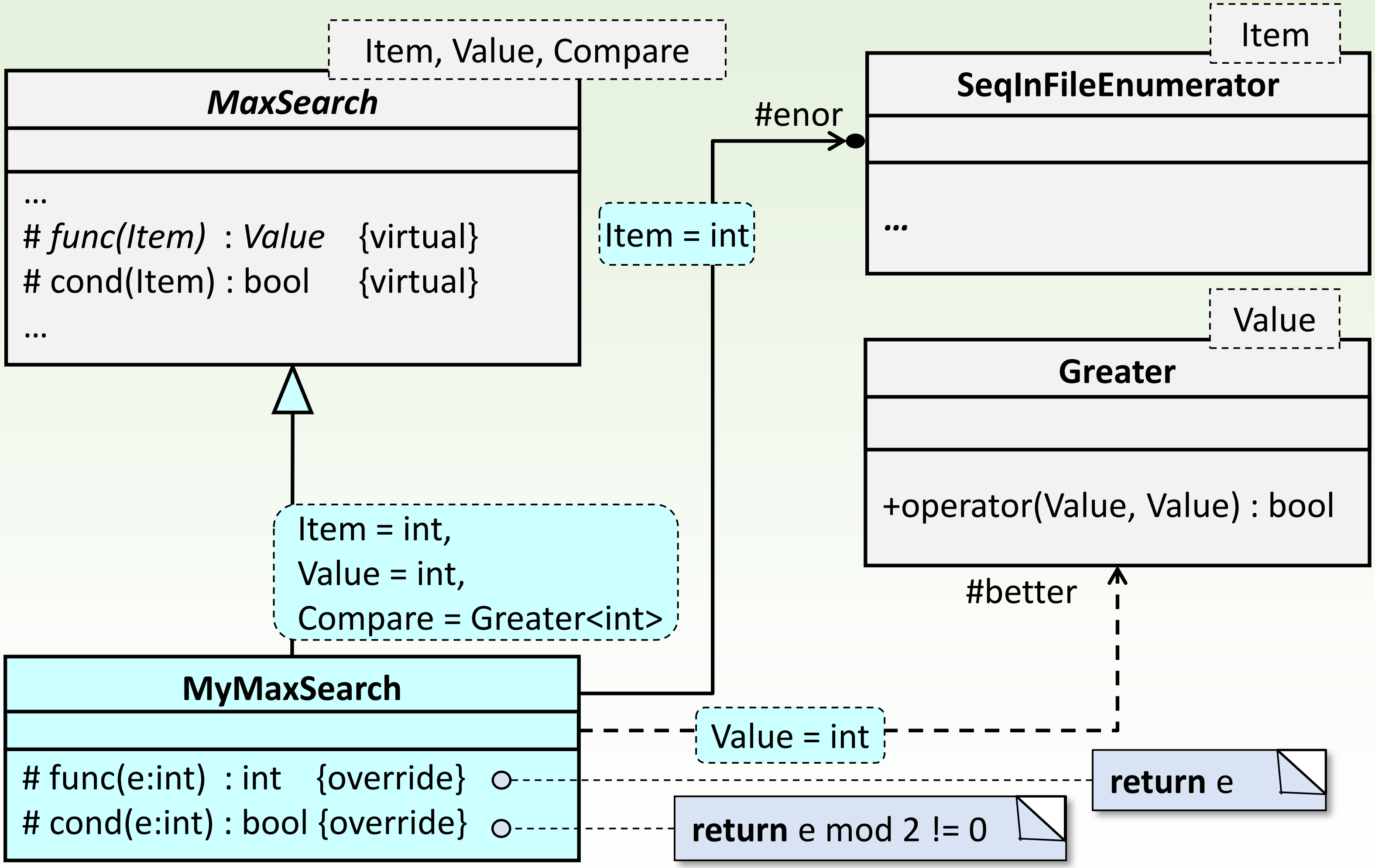
$Post$: $l, max = \mathbf{MAX}_{\substack{e \in f_0 \\ 2 \nmid e}} \; e$

<u>Conditional maximum search</u>

t:enor(E)	~	f:infile($\mathbb{N}$)
f(e)	~	e
H, >	~	$\mathbb{N}$, >
cond(e)	~	$2 \nmid e$

E / Item	~ $\mathbb{N}$ / int
H / Value	~ $\mathbb{N}$ / int
f(e) / func(e)	~ e
cond(e) / cond(e)	~ $2 \nmid e$ / 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;}
};
```

activity object

enumerator object

```
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;
}
```

input.txt

```
12 -5  23
44 130 56 3
-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?

input.txt

```
0108 23
0112 44
0116 130
0207 120
…
```

Specification

A : f:infile(Pair) , l:$\mathbb{L}$, min:$\mathbb{N}$, elem:Pair
Pair = rec(timestamp : $\mathbb{N}$, number : $\mathbb{N}$)

Pre : $f = f_0$

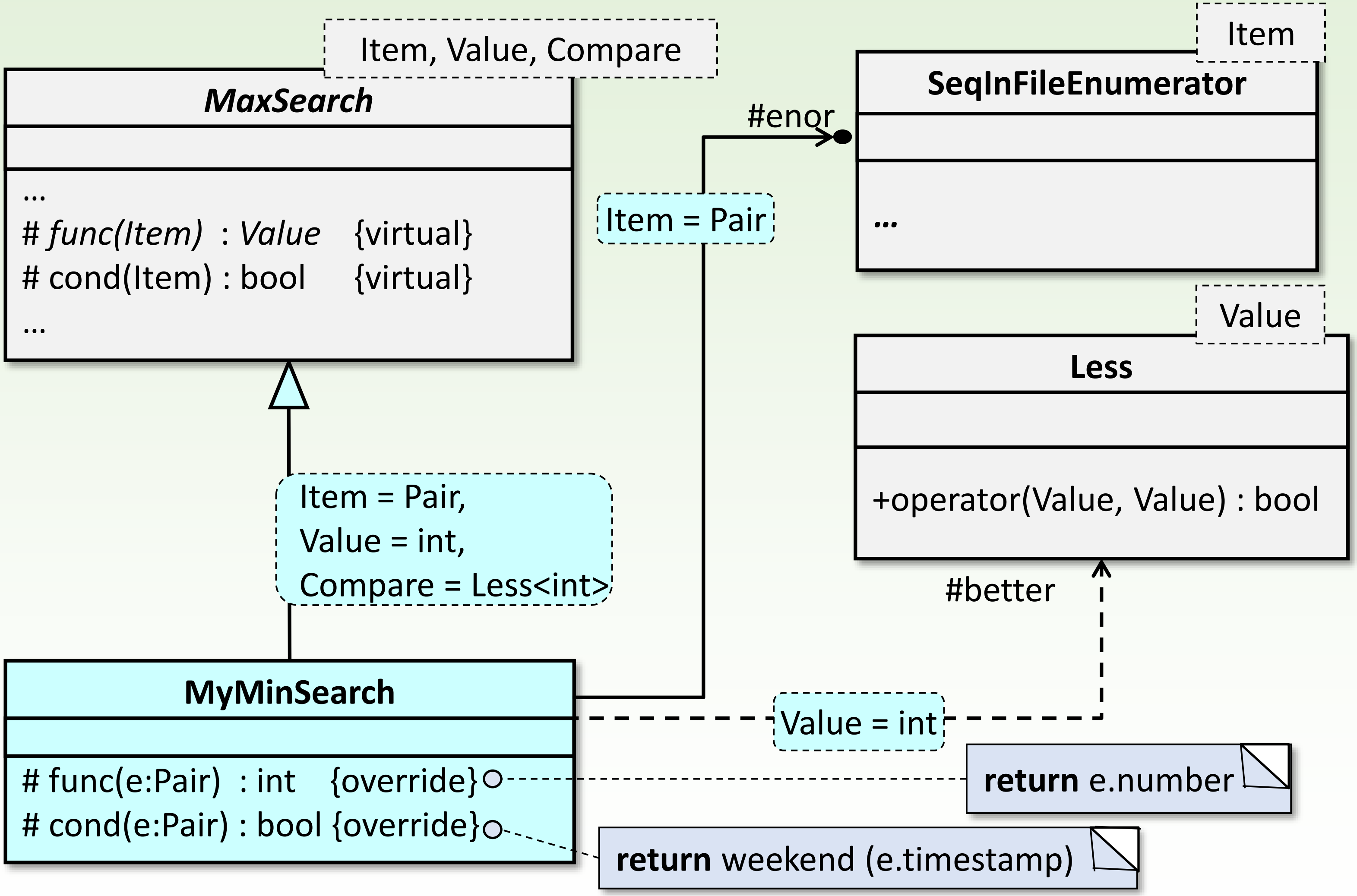
Post : $l, min, elem = \mathbf{MIN}_{e \in f_0}$ e.number
weekend (e.timestamp)

<u>Conditional maximum search</u>

t:enor(Item)	~	f:infile(Pair)
func(e)	~	e.number
H, >	~	$\mathbb{N}$, <
cond(e)	~	weekend (e.timestamp)

e.time / 100 % 7 == 6 || e.time / 100 % 7 == 0

Class diagram of the solution


Item, Value, Compare
MaxSearch
…
# func(Item) : Value {virtual}
# cond(Item) : bool {virtual}
…
Item
SeqInFileEnumerator
…
#enor
Item = Pair
Value
Less
+operator(Value, Value) : bool
Item = Pair,
Value = int,
Compare = Less<int>
#better
MyMinSearch
# func(e:Pair) : int {override}
# cond(e:Pair) : bool {override}
Value = int
return e.number
return weekend (e.timestamp)

Type Pair

input.txt

```
0108 23
0112 44
0116 130
0207 120
…
```

```
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;
}
```

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; }
};
```

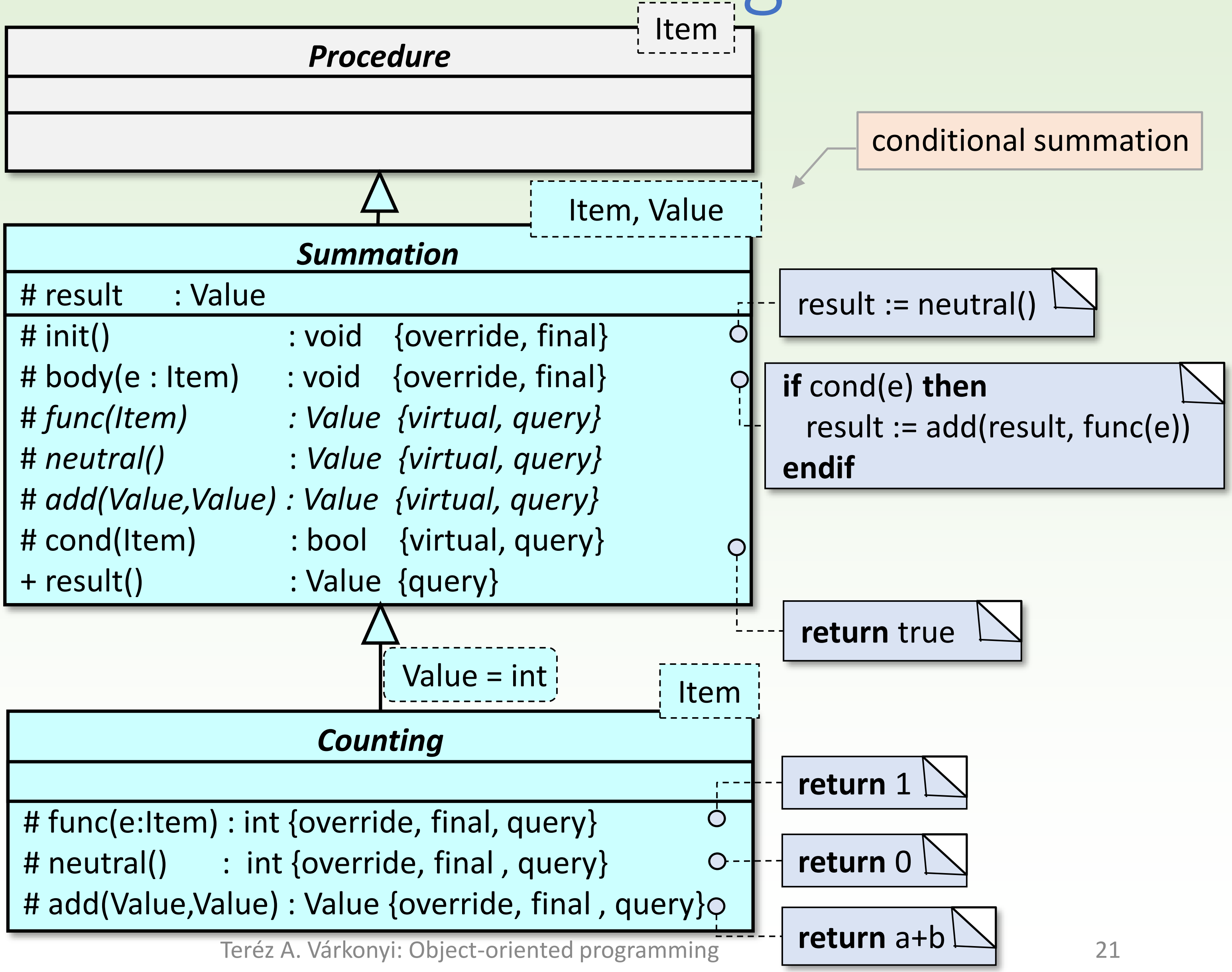
minimum search

according to numbers

with this condition

```
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;
}
```

Summation and Counting



Item

Procedure

conditional summation

Item, Value

Summation

# result : Value

# init() : void {override, final}

# body(e : Item) : void {override, final}

# *func(Item) : Value {virtual, query}*

# *neutral() : Value {virtual, query}*

# *add(Value,Value) : Value {virtual, query}*

# cond(Item) : bool {virtual, query}

+ result() : Value {query}

result := neutral()

```
if cond(e) then
    result := add(result, func(e))
endif
```

return true

Value = int

Item

Counting

# func(e:Item) : int {override, final, query}

# neutral() : int {override, final , query}

# add(Value,Value) : Value {override, final , query}

return 1

return 0

return a+b

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

counting.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;
    }
};
```

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.

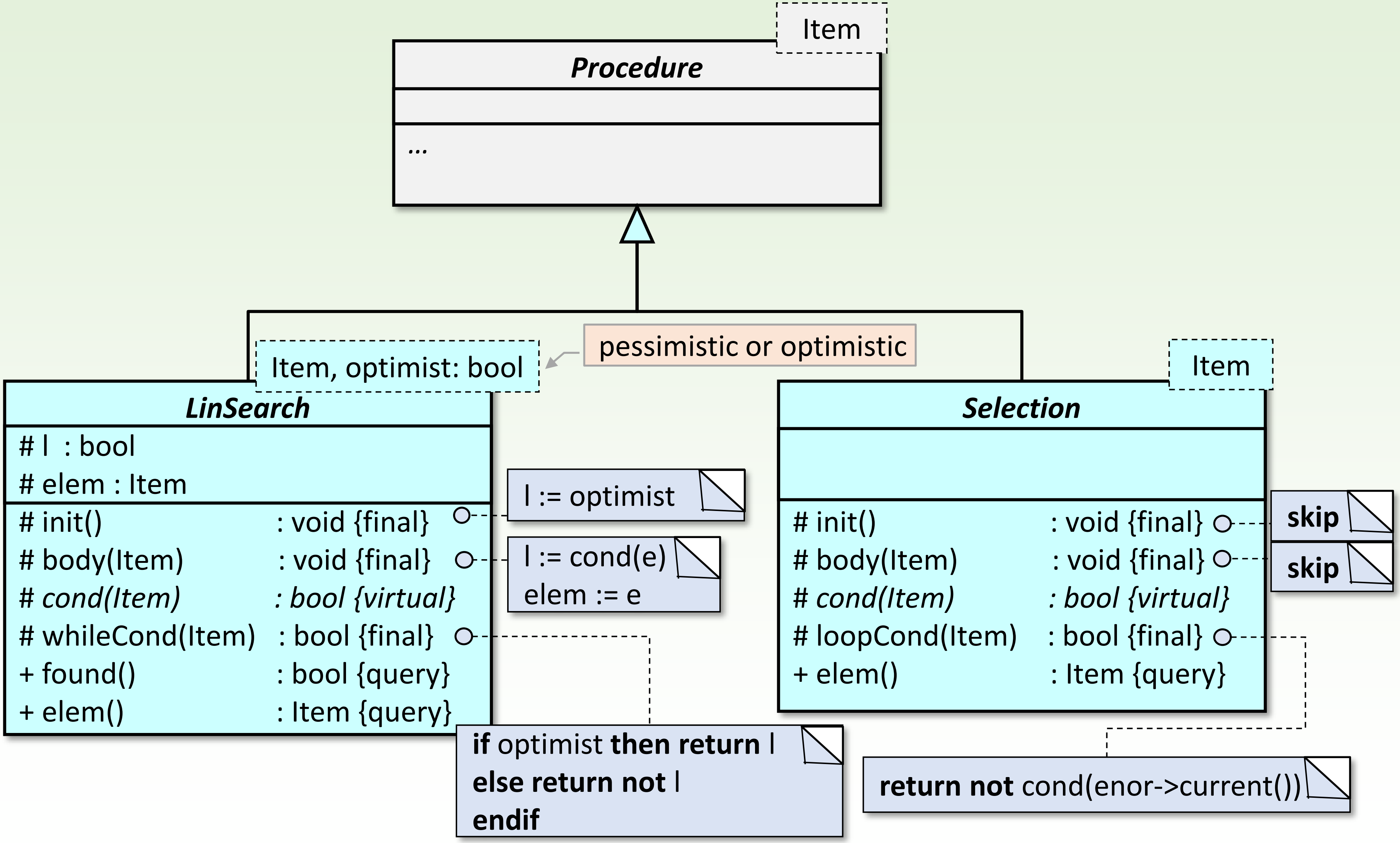
3rd task – only C++

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");
pr.addEnumerator(&enor1);
pr1.run();

Write pr2(&cout);
ArrayEnumerator<int> enor2(&pr1.result());
pr2.addEnumerator(&enor2);
pr2.run();
```

Linear search and Selection



Item

Procedure

…

Item, optimist: bool

pessimistic or optimistic

LinSearch

# l : bool
# elem : Item

# init() : void {final}
# body(Item) : void {final}
# *cond(Item) : bool {virtual}*
# whileCond(Item) : bool {final}
+ found() : bool {query}
+ elem() : Item {query}

l := optimist

l := cond(e)
elem := e

```
if optimist then return l
else return not l
endif
```

Item

Selection

# init() : void {final}
# body(Item) : void {final}
# *cond(Item) : bool {virtual}*
# loopCond(Item) : bool {final}
+ elem() : Item {query}

skip

skip

return not cond(enor->current())

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

selection.hpp

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

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:infile(Recipe) , c:$\mathbb{N}$

Recipe = rec(name : String, ingredients : Ingredient*)

Ingredient = rec(substance : String, quantity: $\mathbb{R}$, unit : String)

Pre : $f = f_0$

Post : $c = \sum_{\substack{e \in f_0 \\ has\_sugar(e)}} \mathbf{1}$, where

subtask: is there sugar in the ingredients?

$$has\_sugar(e) = \mathbf{SEARCH}_{i=1}^{|e.ingredients|} \; e.ingredients[i].substance=\text{``sugar''}$$

<u>Counting</u>

t:enor(Item) ~ f:infile(Recipe)

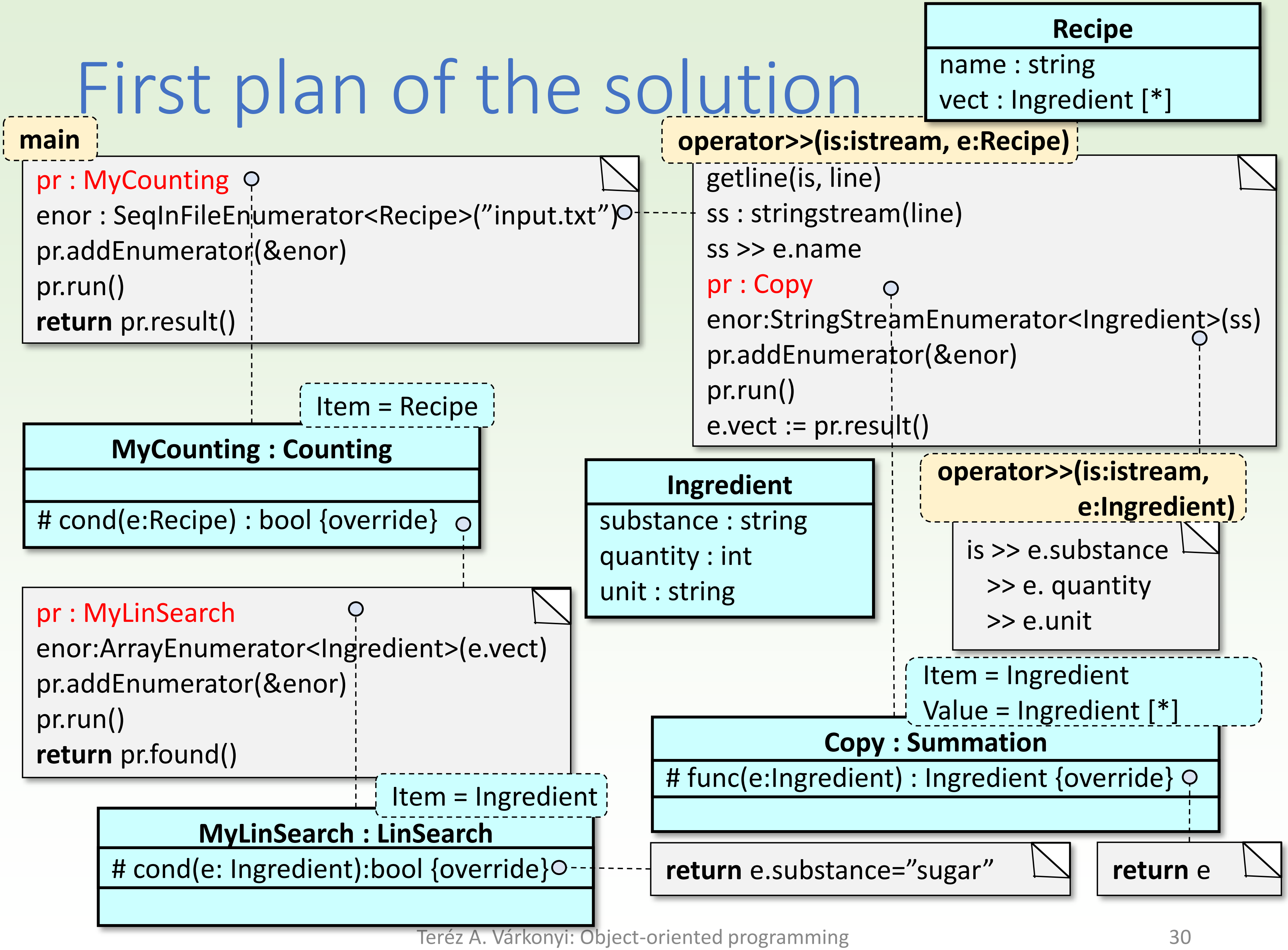
cond(e) ~ has_sugar(e)

<u>Linear search</u>

t:enor(Item) ~ Ingredient* (i=1..*)

cond(e) ~ e.ingredients[i].substance=“sugar”

First plan of the solution



Recipe
name : string
vect : Ingredient [*]

main

```
pr : MyCounting
enor : SeqInFileEnumerator<Recipe>(”input.txt”)
pr.addEnumerator(&enor)
pr.run()
return pr.result()
```

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()
```

Item = Recipe

MyCounting : Counting
# cond(e:Recipe) : bool {override}

Ingredient
substance : string
quantity : int
unit : string

operator>>(is:istream, e:Ingredient)

```
is >> e.substance
   >> e. quantity
   >> e.unit
```

```
pr : MyLinSearch
enor:ArrayEnumerator<Ingredient>(e.vect)
pr.addEnumerator(&enor)
pr.run()
return pr.found()
```

Item = Ingredient
Value = Ingredient [*]

Copy : Summation
# func(e:Ingredient) : Ingredient {override}

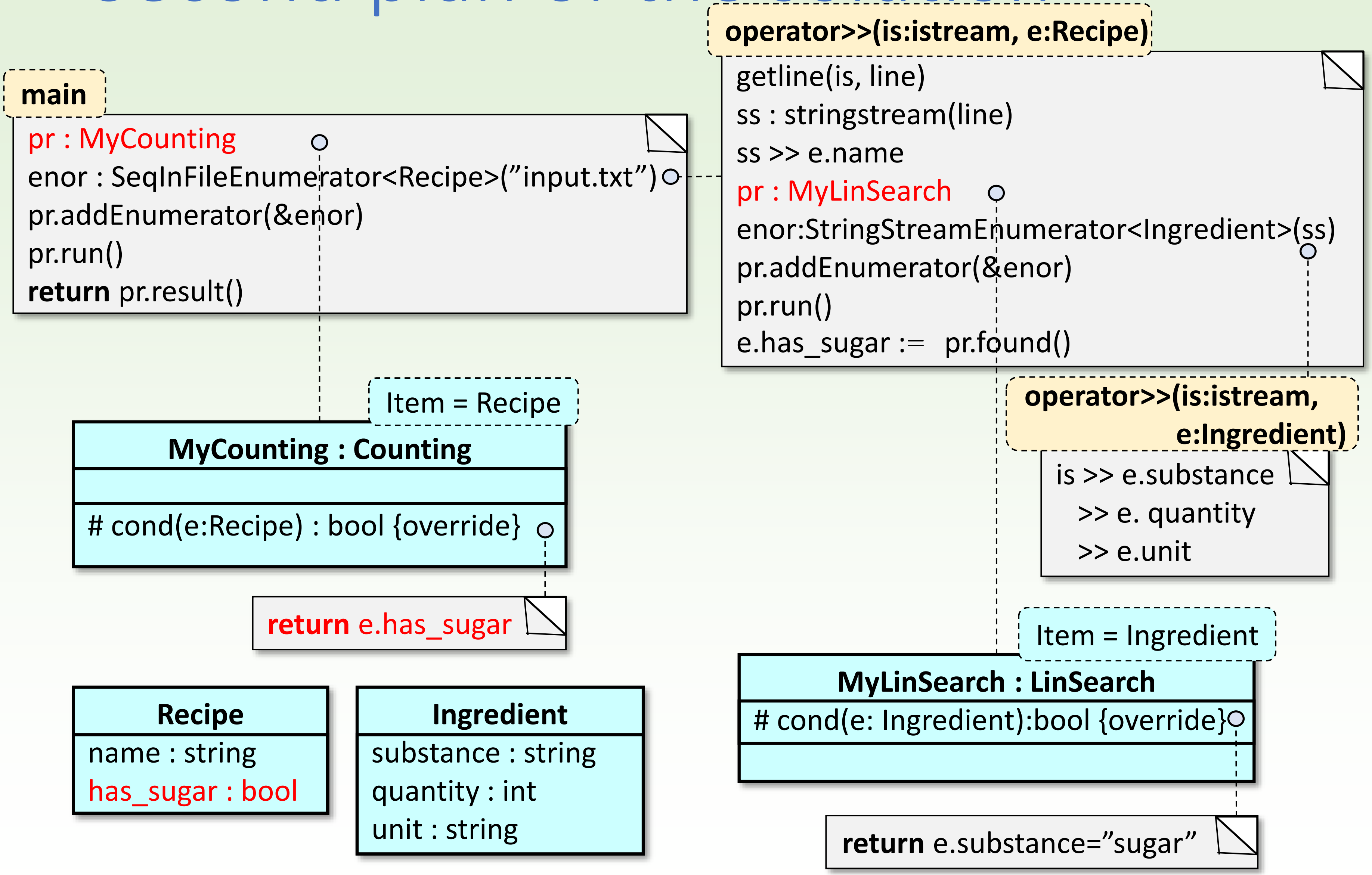
Item = Ingredient

MyLinSearch : LinSearch
# cond(e: Ingredient):bool {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()
```

operator>>(is:istream, e:Recipe)

```
getline(is, line)
ss : stringstream(line)
ss >> e.name
pr : MyLinSearch
enor:StringStreamEnumerator<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 = 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

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:inFile(Observation) , cout:outfile(String × $\mathbb{N}$)

Observation = rec(id:String, date:String, mass:$\mathbb{N}$, distance:$\mathbb{N}$)

Pre : $f = f' \wedge f\nearrow_{id}$

A' : t:enor(Asteroid) , cout:outfile(String × $\mathbb{N}$)

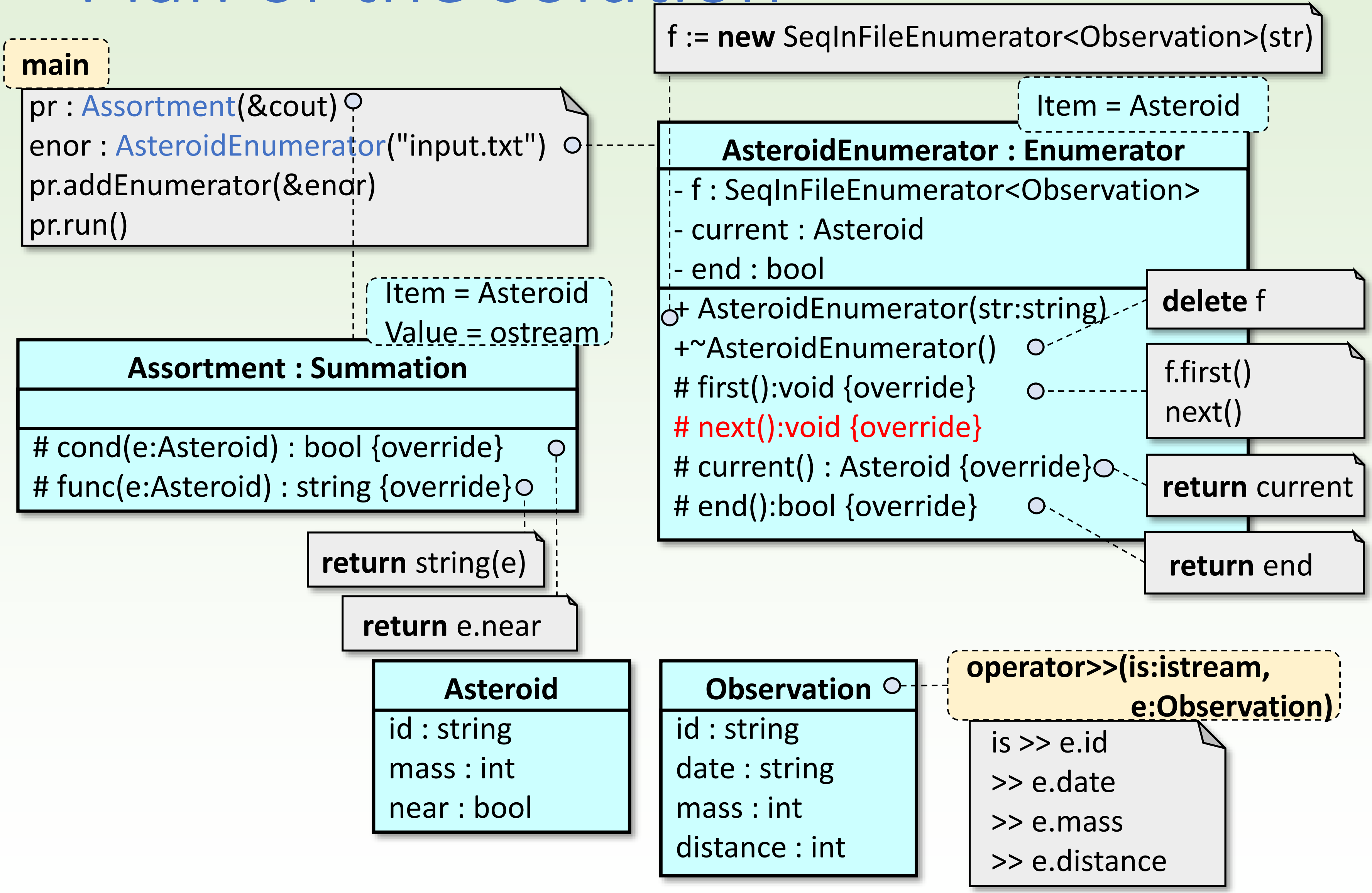
Asteroid = rec(id:String, mass:$\mathbb{N}$, near:$\mathbb{L}$)

Pre : $t = t_0$

Post : $cout = \bigoplus_{\substack{e \in t_0 \\ e.near}} <(e.id, e.mass)>$

<u>Summation</u> (assortment)
t:enor(Item) ~ t:enor(Asteroid)
func(e) ~ (e.id, e.mass)
H, +, 0 ~ (String × $\mathbb{N}$)*, $\bigoplus$, <>)
cond(e) ~ e.near

Plan of the solution



f := **new** SeqInFileEnumerator<Observation>(str)

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

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

Asteroid

id : string
mass : int
near : bool

Observation

id : string
date : string
mass : int
distance : int

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:$\mathbb{L}$

Observation = rec(id:String, date:String, mass:$\mathbb{N}$, distance:$\mathbb{N}$)
Asteroid = rec(id:String, mass:$\mathbb{N}$, near:$\mathbb{L}$)

Pre : $f = f' \wedge e = e' \wedge st = st' \wedge f\nearrow_{id}$

Post : $end = (st'=abnorm) \wedge (\neg end \rightarrow current.id = e'.id \wedge$

$$(current.mass, st, e, f) = \mathbf{MAX}_{e\in(e', f')}^{e.id\ =\ current.id}\ e.mass \wedge$$

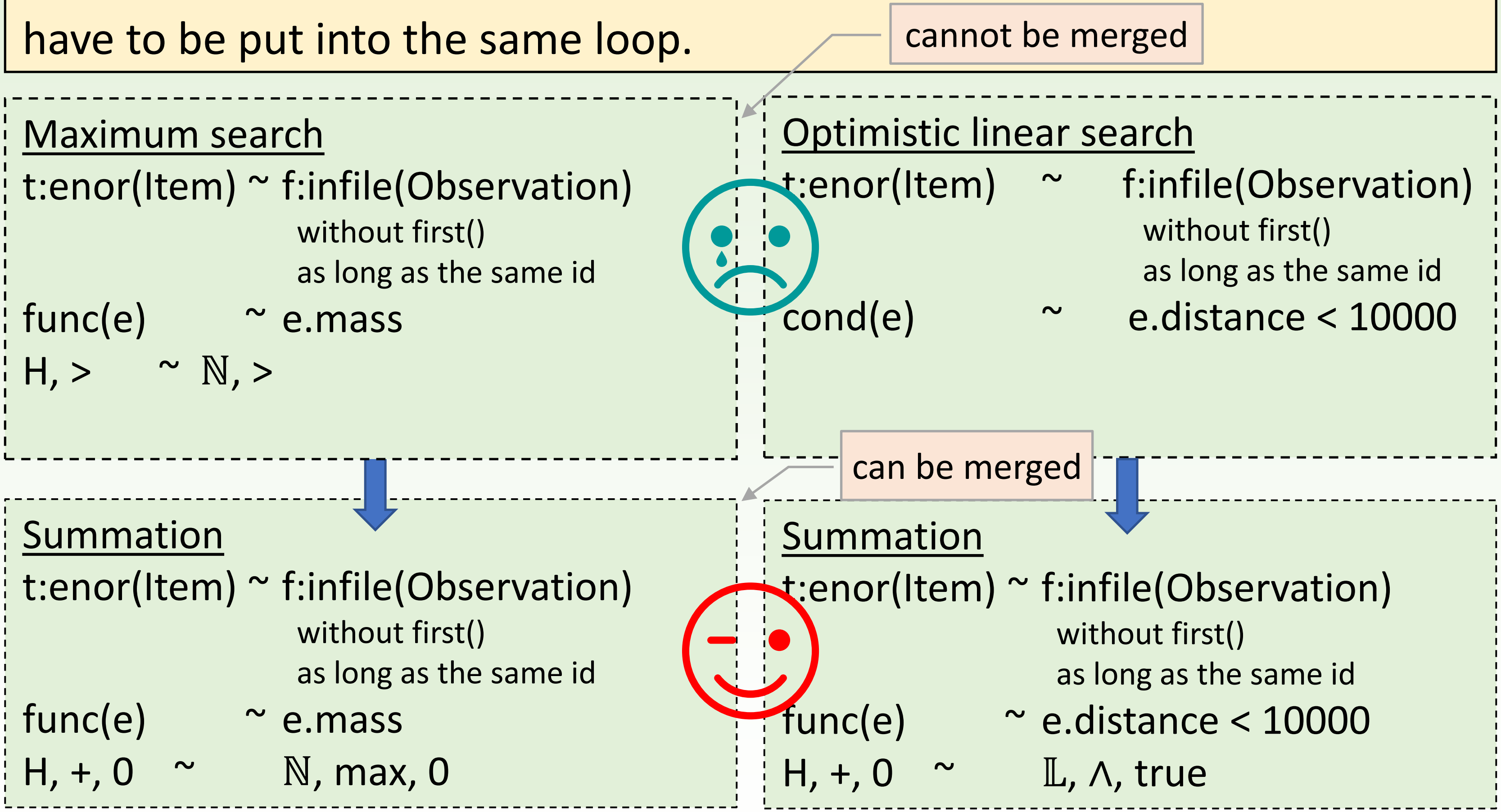
$$(current.near, st'', e'', f'') = \mathbf{\forall SEARCH}_{e\in(e', f')}^{e.id\ =\ current.id}\ e.distance<10000\ )\ )$$

the two enumerations cannot be done in sequence, they have to be merged into one loop

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:enor(Item) ~ f:infile(Observation)
without first()
as long as the same id
func(e) ~ e.mass
H, > ~ ℕ, >

Optimistic linear search

t:enor(Item) ~ f:infile(Observation)
without first()
as long as the same id
cond(e) ~ e.distance < 10000

can be merged

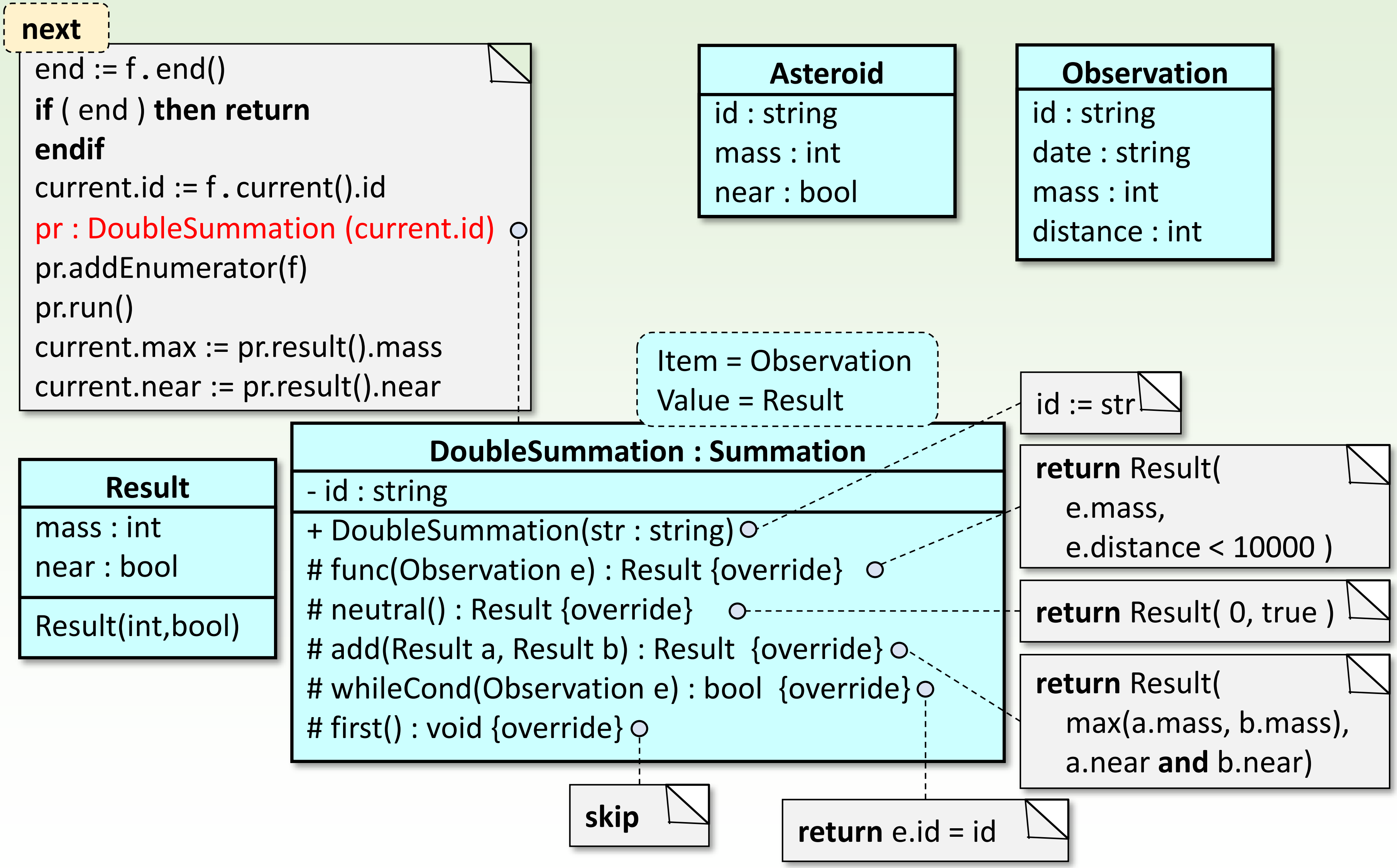
Summation

t:enor(Item) ~ f:infile(Observation)
without first()
as long as the same id
func(e) ~ e.mass
H, +, 0 ~ ℕ, max, 0

Summation

t:enor(Item) ~ f:infile(Observation)
without first()
as long as the same id
func(e) ~ e.distance < 10000
H, +, 0 ~ 𝕃, ∧, 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

Item = Observation
Value = Result

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}

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)
```

skip

return e.id = id