

P00

Patternul Iterator

Cuprins

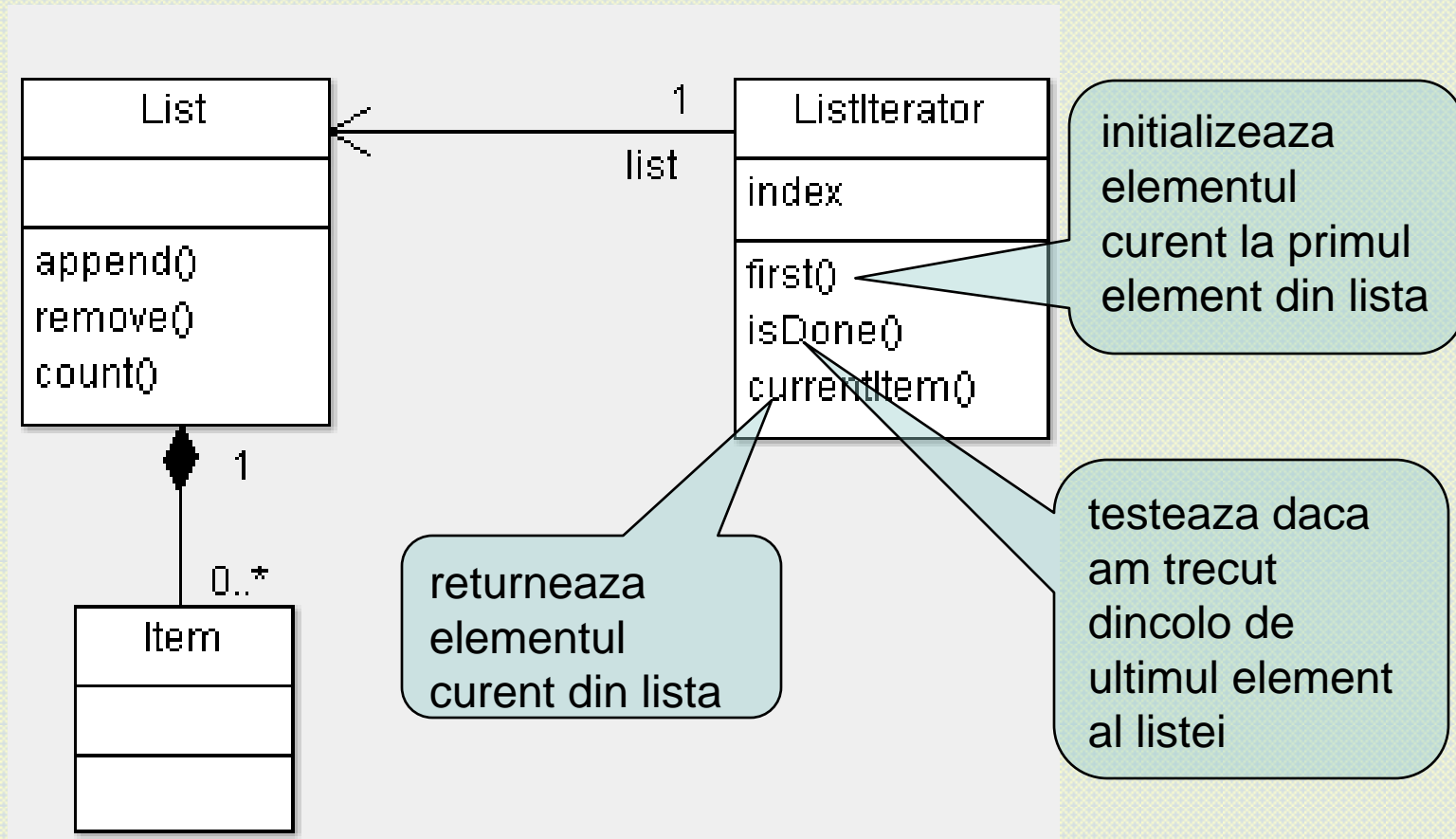
- patternul *Iterator*
 - comparatie cu iteratorii din STL

Patternul *Iterator* **(prezentare bazata** **pe GoF)**

Iterator::intentie

- furnizeaza o modalitate de a accesa componentele unui obiect agregat fara a le expune reprezentarea

Iterator::motivatie

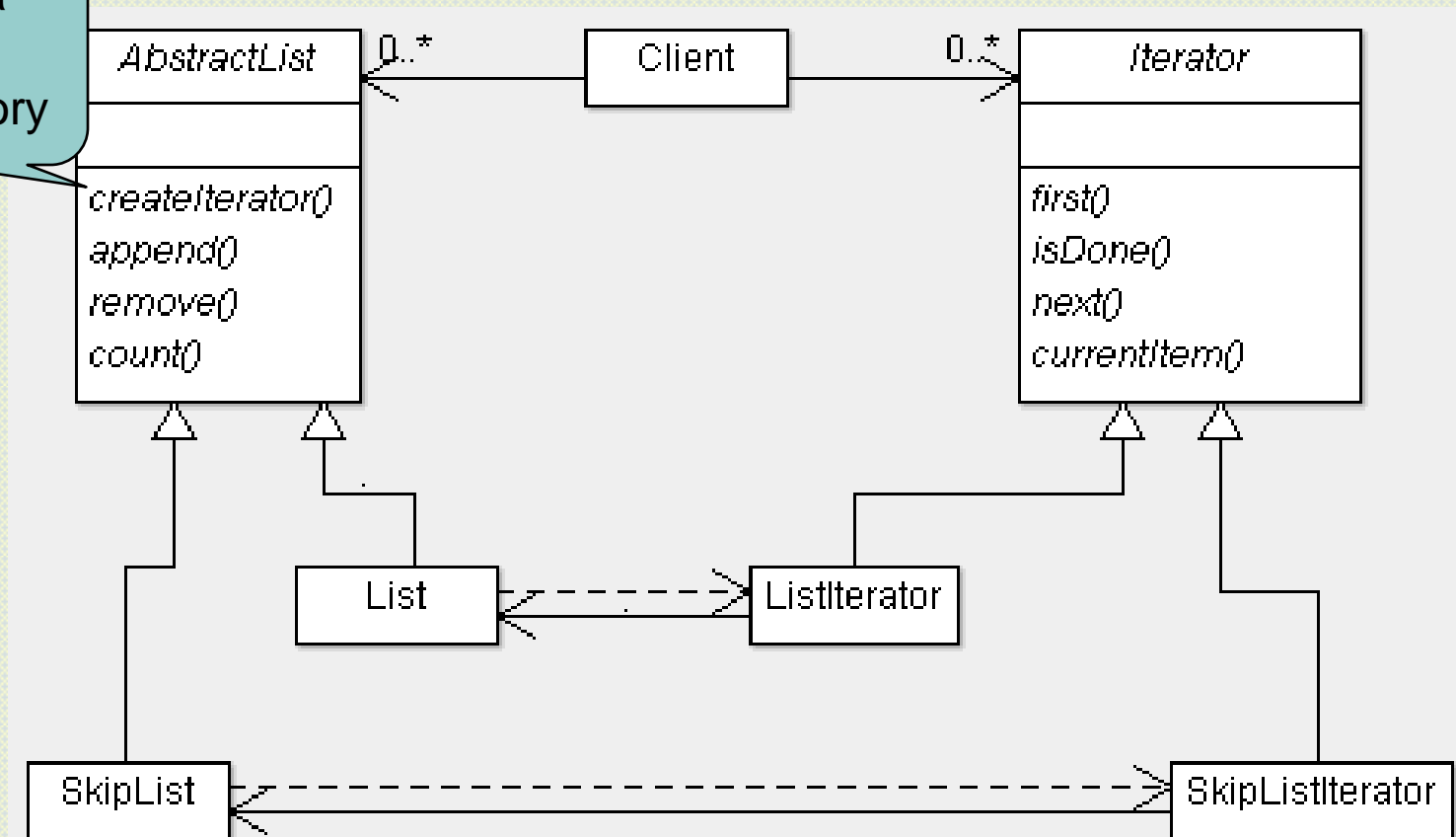


Iterator::motivatie

- Inainte de a instantia ListIterator, trebuie precizat obiectul agregat List care urmeaza a fi traversat
- odata ce avem o instanta ListIterator, putem accesa elementele listei secvential
- separand mecanismul de traversare de obiectele listei, avem libertatea de a defini iteratori pentru diferite politici de traversare
- de exemplu, am putea defini FilteringListIterator care sa acceseze (viziteze) numai acele elemente care satisfac un anumit criteriu de filtrare

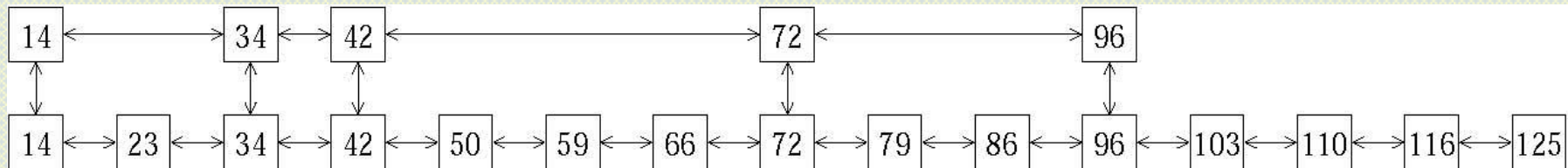
Iterator polimorfic::motivatie

vom discuta
mai mult la
ObjectFactory

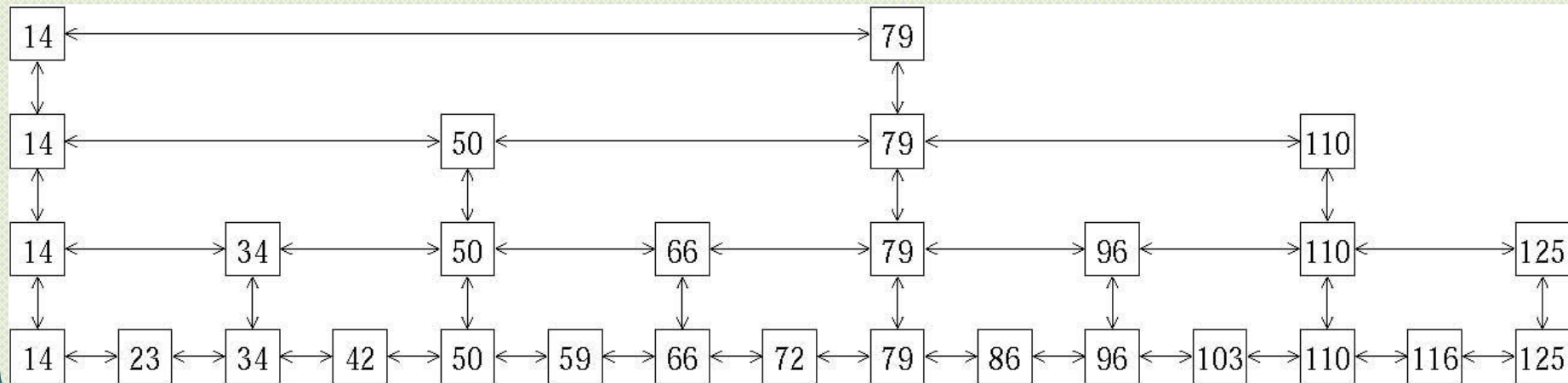


intermezzo structuri de date – skip list

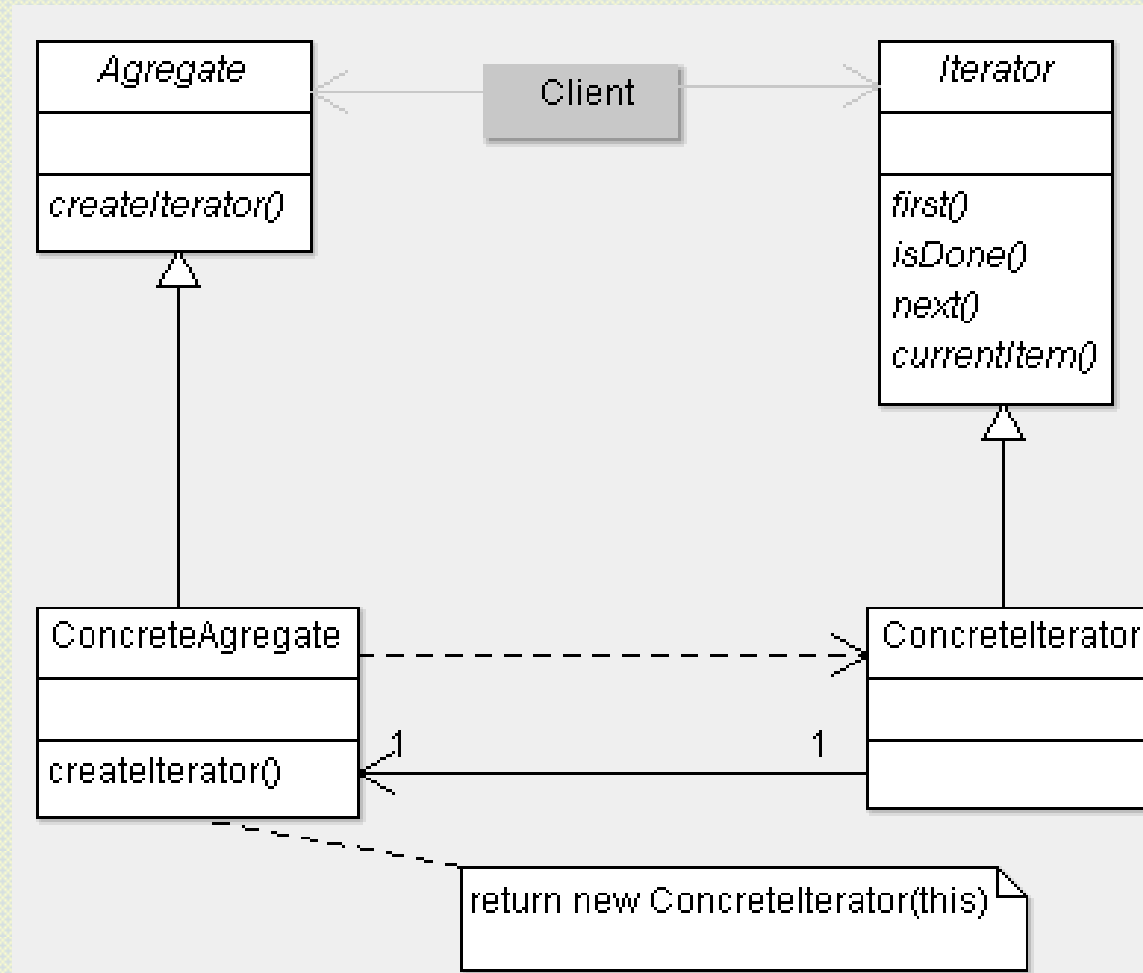
- structuri de date aleatoare simple si eficiente pentru cautare
- structura pe 2 nivele (cost operatie de cautare: $2 \sqrt{n}$)



- structura pe 4 nivele (similara unui arbore binar)



Iterator::structura



Iterator::participanti

- *Iterator*
 - definește interfata de accesare și traversare a componentelor
- *ConcreteIterator*
 - implementează interfata *Iterator*.
 - memorează poziția curentă în traversarea agregatului
- *Aggregate*
 - definește interfata pentru crearea unui obiect *Iterator*
- *ConcreteAggregate*
 - implementează interfata de creare a unui *Iterator* pentru a întoarce o instanță proprie *ConcreteIterator*.

Iterator::consecinte

- suporta diferite moduri de traversare a unui agregat
- simplifica interfata Aggregate
- pot fi executate concurent mai multe traversari (pot exista mai multe traversari in progres la un moment dat); un iterator pastreaza urma numai a propriei sale stari de travesare

Iterator::implementare

- cine controleaza iteratia? clientul (*iterator extern*) sau iteratorul (*iterator intern*)?
- cine defineste algoritmul de traversare?
 - agregatul (iterator = cursor)
 - iteratorul (mai flexibil)
 - s-ar putea sa necesite violarea incapsularii
- cat de robust este iteratorul?
 - operatiile de inserare/eliminare nu ar trebui sa interfereze cu cele de traversare
- operatii aditionale cu iteratori
- operatii aditionale peste iteratori

Iterator::implementare

- iteratori polimorfici
 - trebuie utilizati cu grija
 - clientul trebuie sa-i stearga (ihm ...)
- iteratorii pot avea acces privilegiat (C++ permite)
- iteratori pentru componente compuse recursiv (a se vedea patternul *Composite*)
 - external versus internal
- iteratori nuli
 - pot usura traversarea obiectelor agregate cu structuri mai complexe (de ex. arborescente)
 - prin definitie, un isDone() intoarce totdeauna true pentru un iterator nul

Iterator::cod::interfete

- un agregat concret - lista (parametrizata)

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);
    long count() const;
    Item& get(long index) const;
    // ...
};
```

constanta ce reprezinta
valoarea implicita a
capacitatii unei liste

marimea listei

intoarce elementul de la o
anumita pozitie

Iterator::cod::interfete

- interfata Iterator

```
template <class Item>
class Iterator {
public:
    virtual void first() = 0;
    virtual void next() = 0;
    virtual bool isDone() const = 0;
    virtual Item currentItem() const = 0;
protected:
    Iterator();
};
```

metode
abstracte

Constructorul implicit este
ascuns (de ce?)

Iterator::cod::implementare subclasa

- iterator concret pentru liste

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void first();
    virtual void next();
    virtual bool isDone() const;
    virtual Item currentItem() const;
private:
    const List<Item>* _list;
    long _current;
};
```

constructorul are intotdeauna
parametru (agregatul asociat)

implementarea
operatiilor din
interfata

referinta la agregatul asociat

elementul curent

Iterator::cod::implementare subclasa

```
template <class Item>
ListIterator<Item>::ListIterator
    ( const List<Item>* aList)
    : _list(aList), _current(0) {
    //nothing
}
```

agregatul asociat

initializare

```
template <class Item>
Item ListIterator<Item>::currentItem () const {
    if (isDone()) {
        throw IteratorOutOfBounds;
    }
    return _list->get(_current);
}
```

ietratorul curent in
afara marginilor

Iterator::cod::implementare subclasa

```
template <class Item>
void ListIterator<Item>::First () {
    _current = 0;
}
```

pozitionarea pe
primul

```
template <class Item>
void ListIterator<Item>::next () {
    _current++;
}
```

trecerea la
urmatorul

```
template <class Item>
bool ListIterator<Item>::isDone () const {
    return _current >= _list->count();
}
```

complet?

Iterator::cod::utilizare

```
void PrintEmployees (Iterator<Employee*>& i) {  
    for (i.first(); !i.isDone(); i.next()) {  
        i.currentItem()->print();  
    }  
}
```

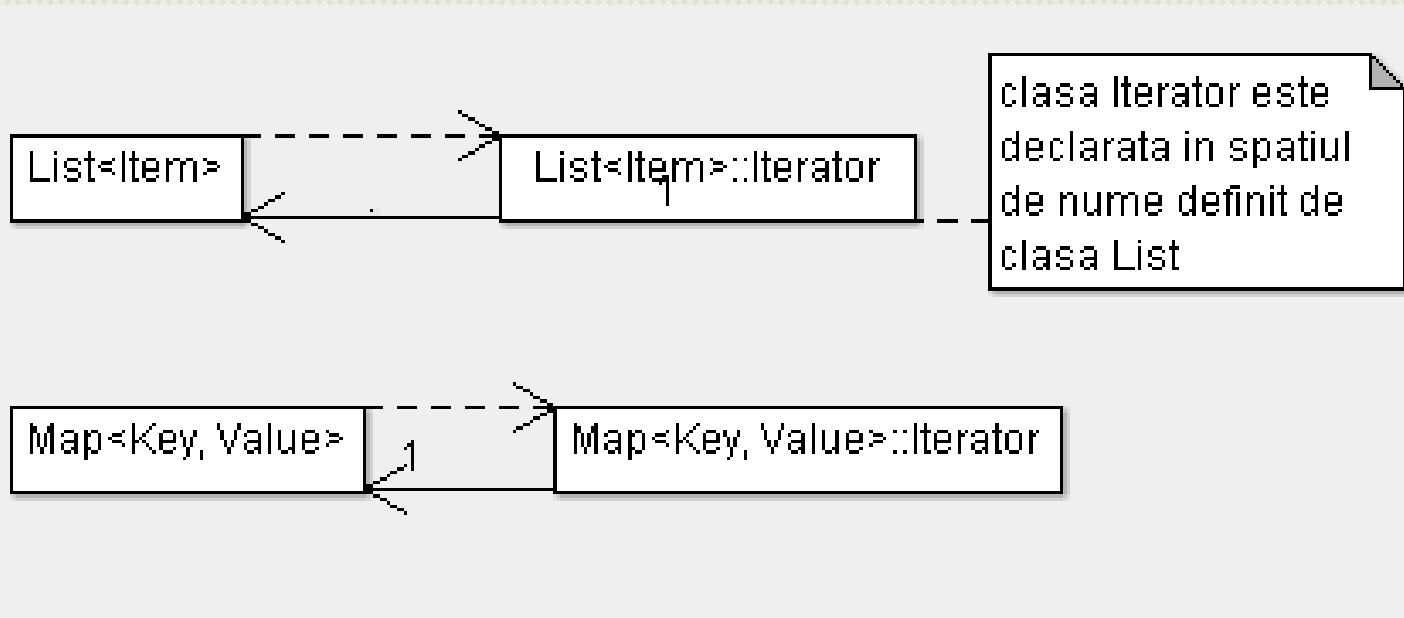
schema de
parcurgere a unei
liste cu iteratori

```
List<Employee*>* employees;  
// ...  
ListIterator<Employee*> forward(employees);  
ReverseListIterator<Employee*> backward(employees);  
printEmployees(forward);  
printEmployees(backward);
```

Iterator care parcurge lista invers;
Este asemanator cu ListIterator cu
exceptia lui first() si next()

Iteratorii in STL

- nu respecta intocmai patternul *Iterator*
- fiecare tip container isi are asociatul propriul tip de iterator



Iteratorii in STL

- Functii membre in Container care se refera la iteratori
 - `iterator begin()`
intoarce un iterator ca refera prima componenta
 - `iterator end()`
intoarce un iterator ca refera sfarsitul containerului
(dincolo de ultima componenta)
 - `iterator insert(iterator pos, const T& x)`
insereaza x inaintea lui pos
 - `iterator erase(iterator pos)`
elimina componenta de la pozitia pos

numai pentru containere de tip secventa

Iteratorii in STL

- Exista mai multe tipuri de iteratori
 - reverse_iterator
 - reverse_bidirectional_iterator
 - insert_iterator
 - front_insert_iterator
 - back_insert_iterator
 - input_iterator
 - output_iterator
 - forward_iterator
 - bidirectional_iterator
 - random_access_iterator
 - ...

Iteratorii in STL

- exemplu de utilizare a unui iterator de inserare

```
list<int> L;  
L.push_front(3);  
insert_iterator<list<int> > ii(L, L.begin());  
*ii++ = 0;  
*ii++ = 1;  
*ii++ = 2;  
copy(L.begin(), L.end(),  
      ostream_iterator<int>(cout, " "));
```

declarare

insereaza pe o si apoi avanseaza

copierea listei in fluxul "cout" este echivalenta cu afisarea

0 1 2 3

Iteratorii in STL versus patternul *Iterator*

Iterator

```
ListIterator<Item> i(list);
```

```
i.first()
```

```
i.isDone()
```

```
i.next()
```

```
i.currentItem()
```

```
for (i.first();  
     !i.isDone();  
     i.next()) {...}
```

STL

```
List<Item>::Iterator<Item> i;  
List<Item>::Iterator<Item>  
    i(list.begin());
```

```
i = list.begin()
```

```
i == list.end()
```

```
++i      (i++)
```

```
*i
```

```
for (i = list.begin();  
     i != list.end();  
     ++i) {...}
```


Mai mult despre iteratori

material suplimentar

Iterator::cod::iteratorii polimorfici

- motivatie
- sa presupunem ca utilizam mai multe tipuri de liste

```
SkipList<Employee*>* employees;  
// ...  
SkipListIterator<Employee*> iterator(employees);  
PrintEmployees(iterator);
```

- cateodata e mai flexibil sa consideram o clasa abstracta pentru a standardiza accesul la diferite tipuri de lista

Iterator::cod::iteratorii polimorfici

```
template <class Item>
```

```
class AbstractList
```

interfata la lista

```
{
```

```
public:
```

```
    virtual Iterator<Item>* CreateIterator()
```

```
        const = 0;
```

```
    // ...
```

```
};
```

lista concreta

```
template <class Item>
```

```
Iterator<Item>* List<Item>::CreateIterator () const
```

```
{
```

```
    return new ListIterator<Item>(this)
```

implementeaza
met. din interfata

```
}
```

Iterator::cod::iteratorii polimorfici

```
// cunoastem numai AbstractList  
AbstractList<Employee*>* employees;
```

pointer

iteratorul este asociat
la o lista concreta

```
// ...  
Iterator<Employee*>* iterator =  
    employees->CreateIterator();  
PrintEmployees(*iterator);  
delete iterator; // noi suntem resp. pt. stergere!
```

- pentru a ne usura munca, cream o clasa `IteratorPtr` care joaca rol de “proxy” pentru iterator

Iterator::cod::stergere it. polim.

```
template <class Item>
class IteratorPtr {
public:
    IteratorPtr(Iterator<Item>* i) : _i(i)
    ~IteratorPtr() { delete _i; }
    Iterator<Item>* operator->() { return _i; }
    Iterator<Item>& operator*() { return *_i; }

private:
    IteratorPtr(const IteratorPtr&);
    IteratorPtr& operator=(const IteratorPtr&);
private:
    Iterator<Item>* _i;
};
```

destructorul este apelat automat

supraincarcare operatori de tip pointer

implementare inline

ascunde copierea si atribuirea pentru a nu permite stergeri multiple ale lui _i

Iterator::cod::stergere it. polim.

- proxy-ul ne usureaza munca

```
AbstractList<Employee*>* employees;
```

```
// ...
```

```
IteratorPtr<Employee*>
```

```
    iterator(employees->CreateIterator());
```

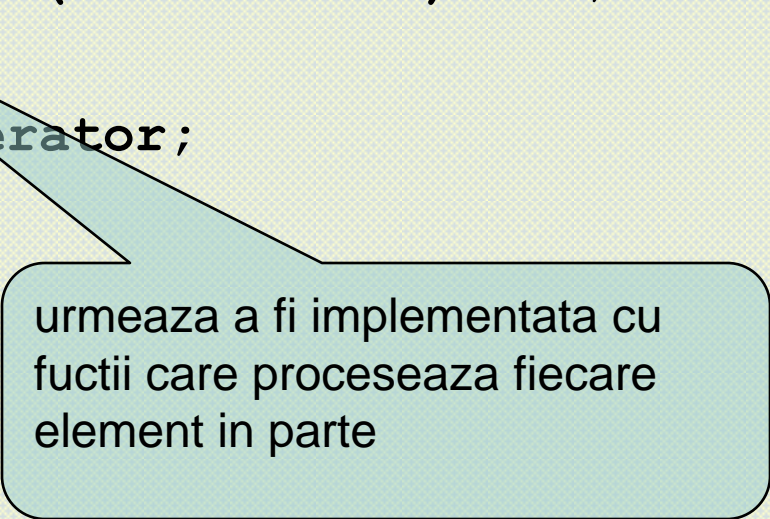
```
PrintEmployees(*iterator);
```

Iterator::cod::iterator intern

- mai este numit si iterator *pasiv*
- cum parametrizam un iterator cu operatia pe care dorim sa o executam peste fiecare element?
- o problema: C++ nu suporta functii anonime
- solutii posibile:
 - un parametru pointer la o functie
 - subclase care suprascriu functia cu comportarea dorita
- ambele au avantaje si dezavantaje
- optam pentru a doua

Iterator::cod::iterator intern

```
template <class Item>
class ListTraverser {
public:
    ListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```



urmeaza a fi implementata cu
fuctii care proceseaza fiecare
element in parte

Iterator::cod::iterator intern

```
template <class Item>
ListTraverser<Item>::ListTraverser
    ( List<Item>* aList ) : _iterator(aList) { }
template <class Item>
bool ListTraverser<Item>::Traverse () {
    bool result = false;
    for ( _iterator.First(); !_iterator.IsDone();
          _iterator.Next() ) {
        result = ProcessItem(_iterator.CurrentItem());
        if (result == false) {
            break;
        }
    }
    return result;
}
```

Iterator::cod::iterator intern

```
class PrintNEmployees
    : public ListTraverser<Employee*> {
public:
    PrintNEmployees(List<Employee*>* aList, int n) :
        ListTraverser<Employee*>(aList),
        _total(n), _count(0)
{ /* nothing */ }
protected:
    bool ProcessItem(Employee* const&);
private:
    int _total;
    int _count;
};
```

Iterator::cod::iterator intern

```
bool PrintNEmployees::ProcessItem
    (Employee* const& e) {
    _count++;
    e->Print();
    return _count < _total;
}
```

- utilizzare

```
List<Employee*>* employees;
// ...
PrintNEmployees pa(employees, 10);
pa.Traverse();
```

Iterator::cod::iterator intern

- diferenta fata de iteratori externi

```
ListIterator<Employee*> i(employees);  
int count = 0;  
for (i.First(); !i.IsDone(); i.Next()) {  
    count++;  
    i.CurrentItem()->Print();  
    if (count >= 10) {  
        break;  
    }  
}
```

Iterator::cod::iterator intern

- incapsularea diferitelor iteratii

```
template <class Item>
class FilteringListTraverser {
public:
    FilteringListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
    virtual bool TestItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```

Iterator::code::iterator intern

```
template <class Item>
void FilteringListTraverser<Item>::Traverse () {
    bool result = false;
    for ( _iterator.First(); !_iterator.IsDone();
          _iterator.Next() ) {
        if (TestItem(_iterator.CurrentItem())) {
            result = ProcessItem(_iterator.CurrentItem());
            if (result == false) {
                break;
            }
        }
    }
    return result;
}
```