

***P00***

Sabloane  
comportamentale

# Cuprins

---

- Visitor
- Observer
- Iterator

# ***P00***

Sablonul  
*Visitor*  
(prezentare bazata pe GoF)

# Intentie

---

- reprezinta o operatie care se executa peste elementele unei structuri de obiecte
- permite sa definirea de noi operatii fara a schimba clasele elementelor peste care lucreaza

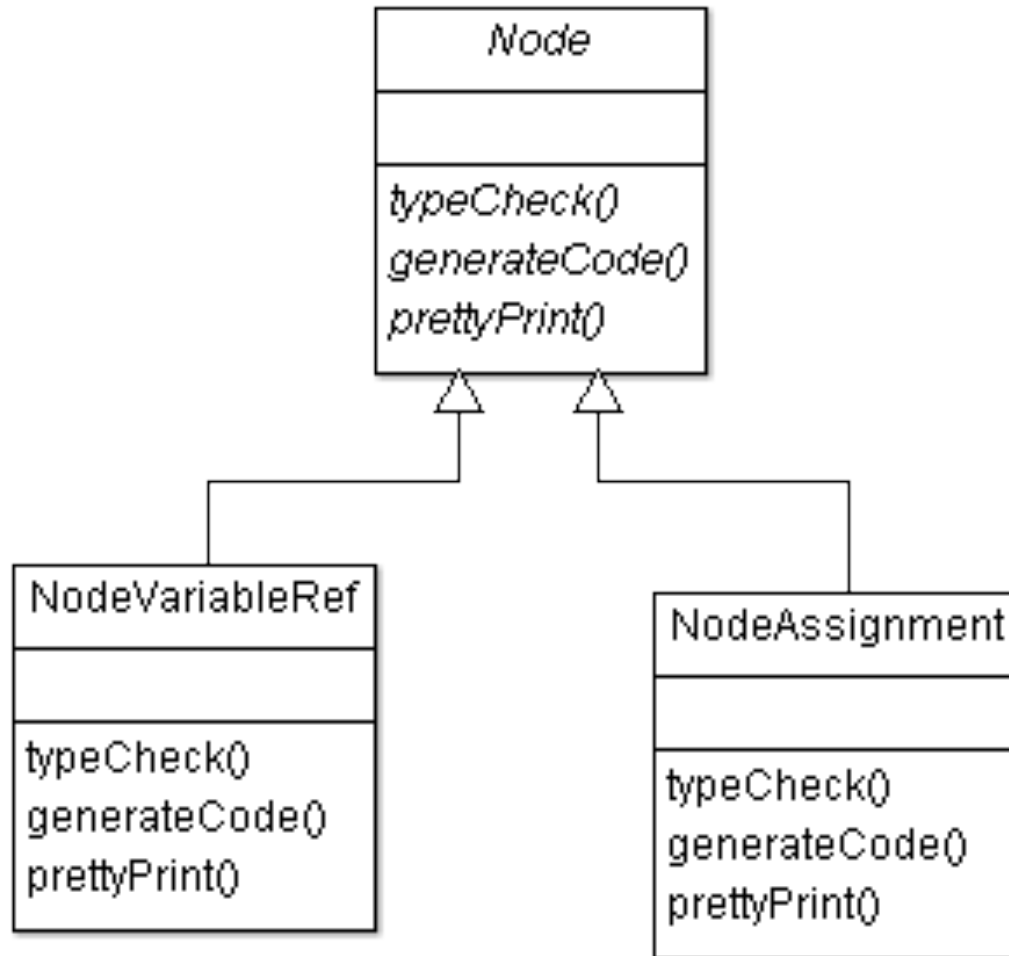
# Motivatie

---

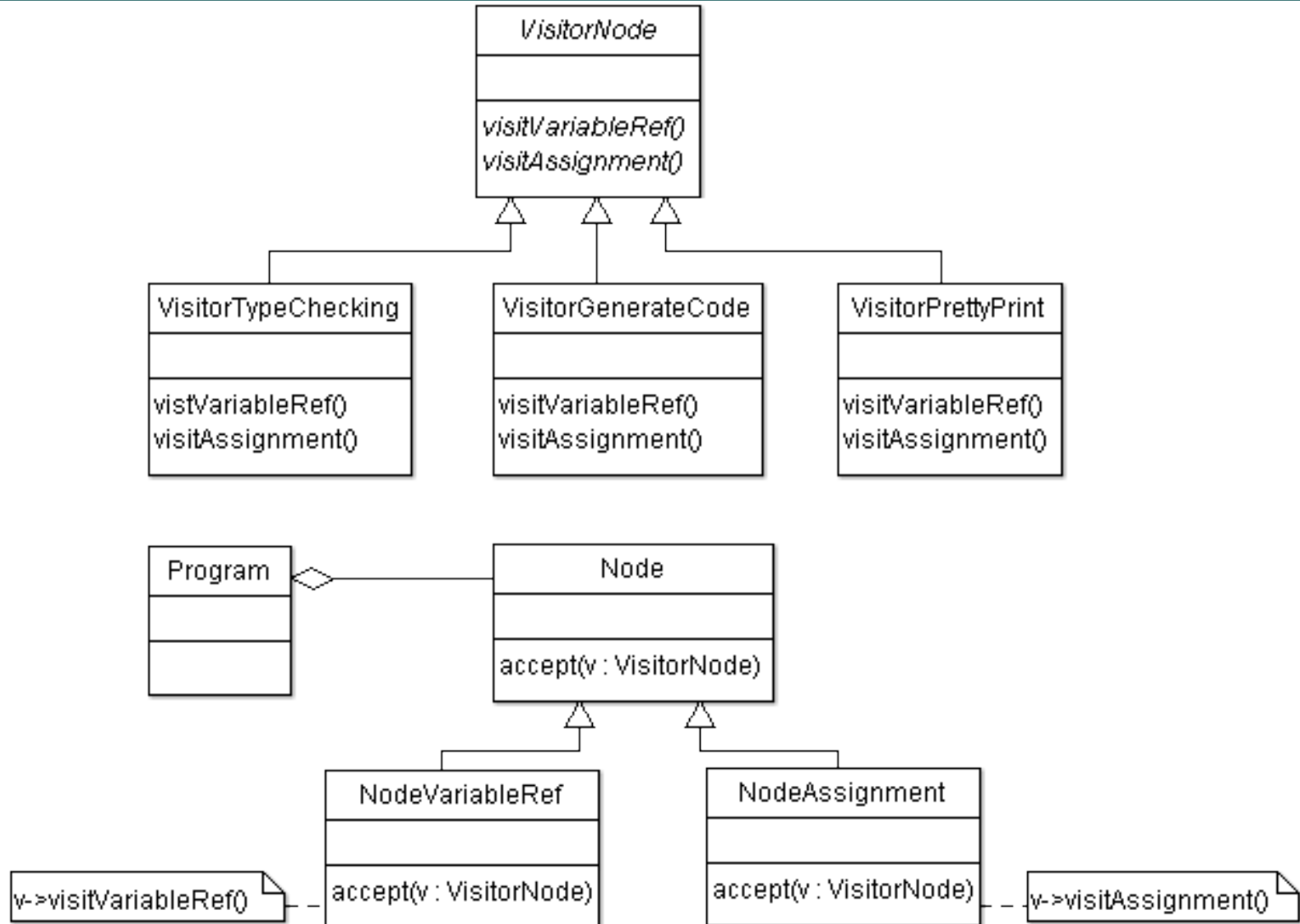
- Un compilator reprezinta un program ca un arbore sintactic abstract (AST). Acest arbore sintactic este utilizat atat pentru semantica statica (e.g., verificarea tipurilor) cat si pentru generarea de cod, optimizare de cod, afisare.
- Aceste operatii difera de la un tip de instructiune la altul. De exemplu, un nod ce reprezinta o atribuire difera de un nod ce reprezinta o expresie si in consecintele operatiile asupra lor vor fi diferite.
- Aceste operatii ar trebui sa se execute fara sa schimbe structura ASTului.
- Chiar daca structura ASTului difera de la un limbaj la altul, modurile in care se realizeaza operatiile sunt similare

# Solutie necorespunzatoare

- “polueaza” structura de clase cu operatii care nu au legatura cu structura



# Solutia cu vizitatori



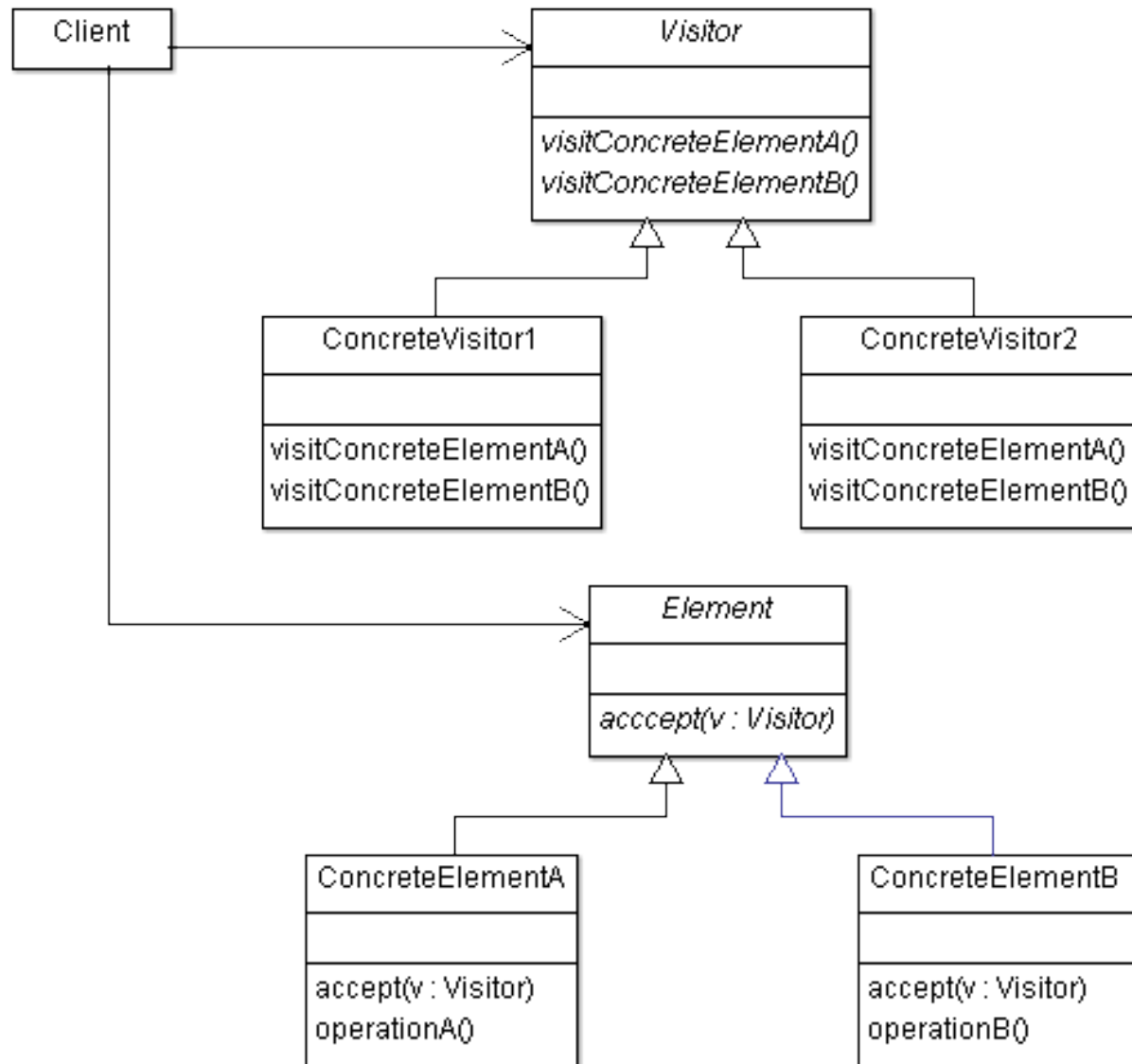
# Aplicabilitate

---

- O structura de obiecte contine mai multe clase cu interfete diferite si se doreste realizarea unor operatii care depind de aceste clase
- Operatiile care se executa nu au legatura cu clasele din structura si se doreste evitarea “poluarii” acestor clase.
- Sablonul Visitor pune toate aceste operatii intr-o singura clasa.
- Cand structura este utilizata in mai multe aplicatii, in Visitor se pun exact acele operatii de care e nevoie.
- Clasele din structura se schimba foarte rar dar se doreste adaugarea de operatii noi peste structura.
- Schimbarea structurii necesita schimbarea interfetelor tuturor vizitatorilor.



# Structura



# Participanti 1/2

---

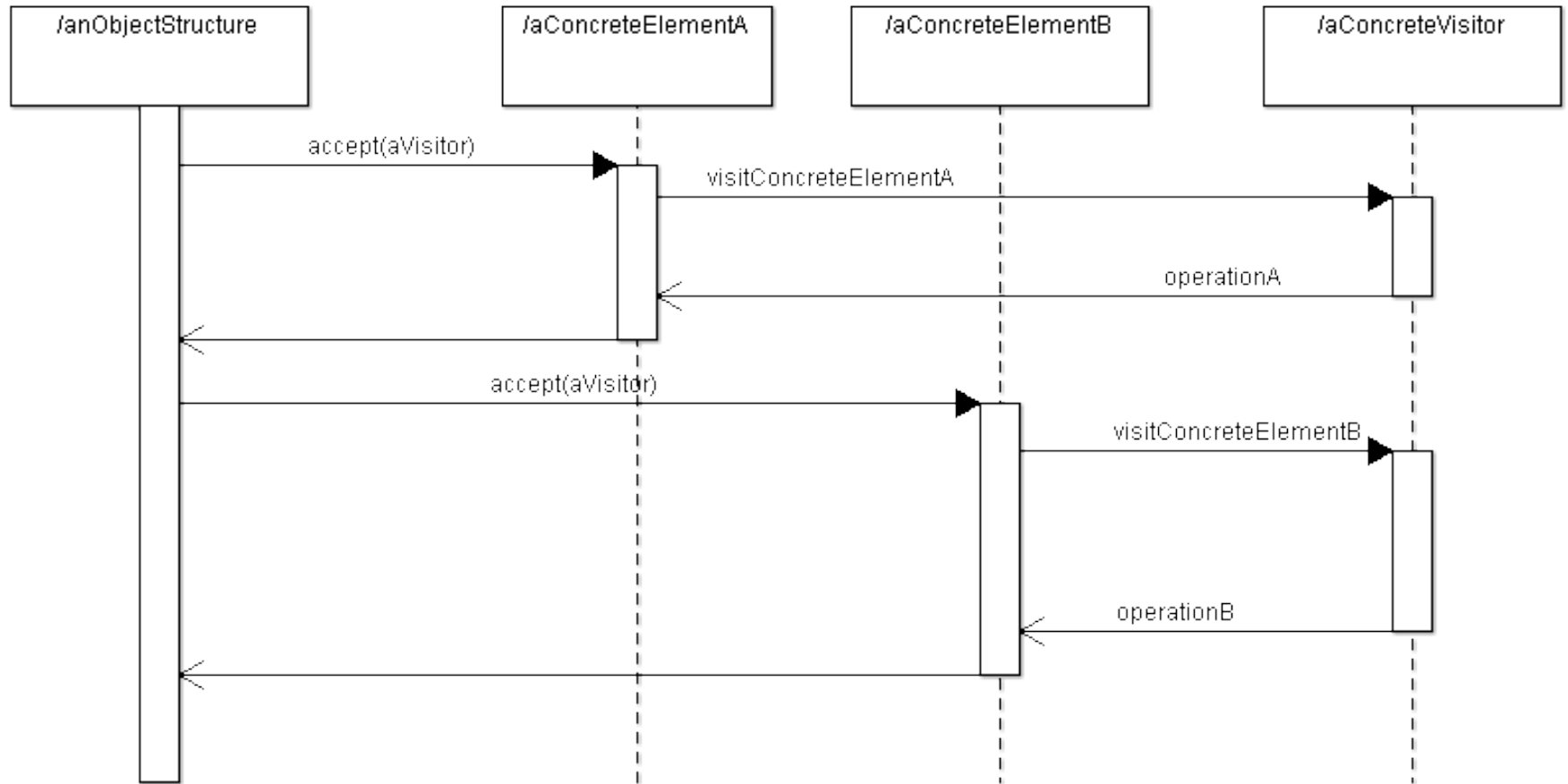
- **Visitor** (NodeVisitor)
  - declara cate o operatie de vizitare pentru fiecare clasa ConcreteElement din structura. Numele operatiei si signatura identifica clasa care trimite cererea de vizitare catre vizitator. Aceasta permite vizitatorului sa identifice elementul concret pe care il viziteaza. Apoi, vizitatorul poate vizita elementul prin intermediul interfetei sale.
- **ConcreteVisitor** (TypeCheckingVisitor)
  - implementeaza fiecare operatie declarata de vizitator. Fiecare operatie implementeaza un fragment din algoritmul de vizitare care corespunde elementului din structura vizitat. Memoreaza starea algoritmului de vizitare, care de multe ori acumuleaza rezultatele obtinute in timpul vizitarii elementelor din structura.

# Participanti 2/2

---

- **Element** (Node)
  - definește operații de acceptare, care au ca argument un vizitator
- **ConcreteElement** (AssignmentNode, VariableRefNode)
  - implementează operația de acceptare
- **ObjectStructure** (Program)
  - poate enumera elementele sale
  - poate furniza o interfață la nivel înalt pentru un vizitator care vizitează elementele sale
  - poate fi un “composite”

# Colaborari



## Consecinte 1/2

---

- *Visitor face adaugarea de noi operatii usoara*
- *Un vizitator aduna operatiile care au legatura intre ele si le separa pe cele care nu au legatura*
- *Adaugarea de noi clase ConcreteElement la structura este dificila. Provoaca schimbarea interfetelor tuturor vizitatorilor. Cateodata o implementare implicita in clasa abstracta Visitor poate usura munca.*
- *Spre deosebire de iteratori, un vizitator poate traversa mai multe ierarhii de clase*
- *Permite calcularea de stari cumulative. Altfel, starea cumulativa trebuie transmisa ca parametru*
- *S-ar putea sa distruga incapsularea. Elementele concrete trebuie sa aiba o interfata puternica capabila sa ofere toate informatiile cerute de vizitator*

# Implementare 1/3

---

```
class Visitor {
public:
    virtual void VisitElementA(ElementA*) ;
    virtual void VisitElementB(ElementB*) ;
    // and so on for other concrete elements
protected:
    Visitor() ;
};

class Element {
public:
    virtual ~Element() ;
    virtual void Accept(Visitor&) = 0;
protected:
    Element() ;
};
```

## Implementare 2/3

---

```
class ElementA : public Element {
public:
    ElementA();
    virtual void Accept(Visitor& v) {
        v.VisitElementA(this);
    }
};

class ElementB : public Element {
public:
    ElementB();
    virtual void Accept(Visitor& v) {
        v.VisitElementB(this);
    }
};
```

## Implementare 3/3

---

- *Simple dispatch.* Operatia care realizeaza o cerere depinde de doua criterii: numele cererii si tipul receptorului. De exemplu, *generateCode* depinde de tipul nodului.
- *Double dispatch.* Operatia care realizeaza cererea depinde de tipurile a doi receptori. De exemplu, un apel *accept()* depinde atat de element cat si de vizitator.
- *Cine este responsabil de traversarea structurii de obiecte?*
  - structura de obiecte
  - vizitatorul
  - un iterator

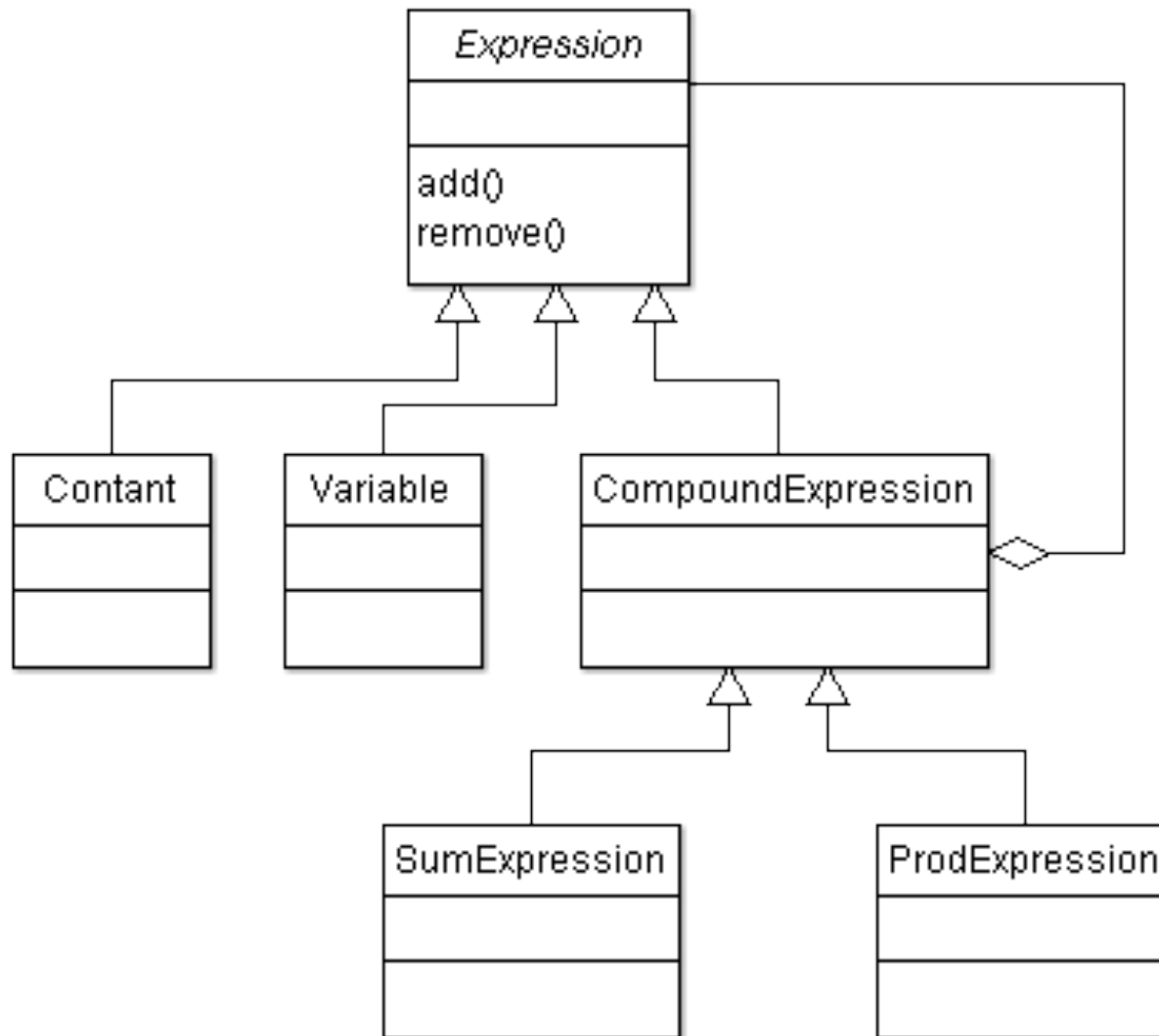


# Aplicatie

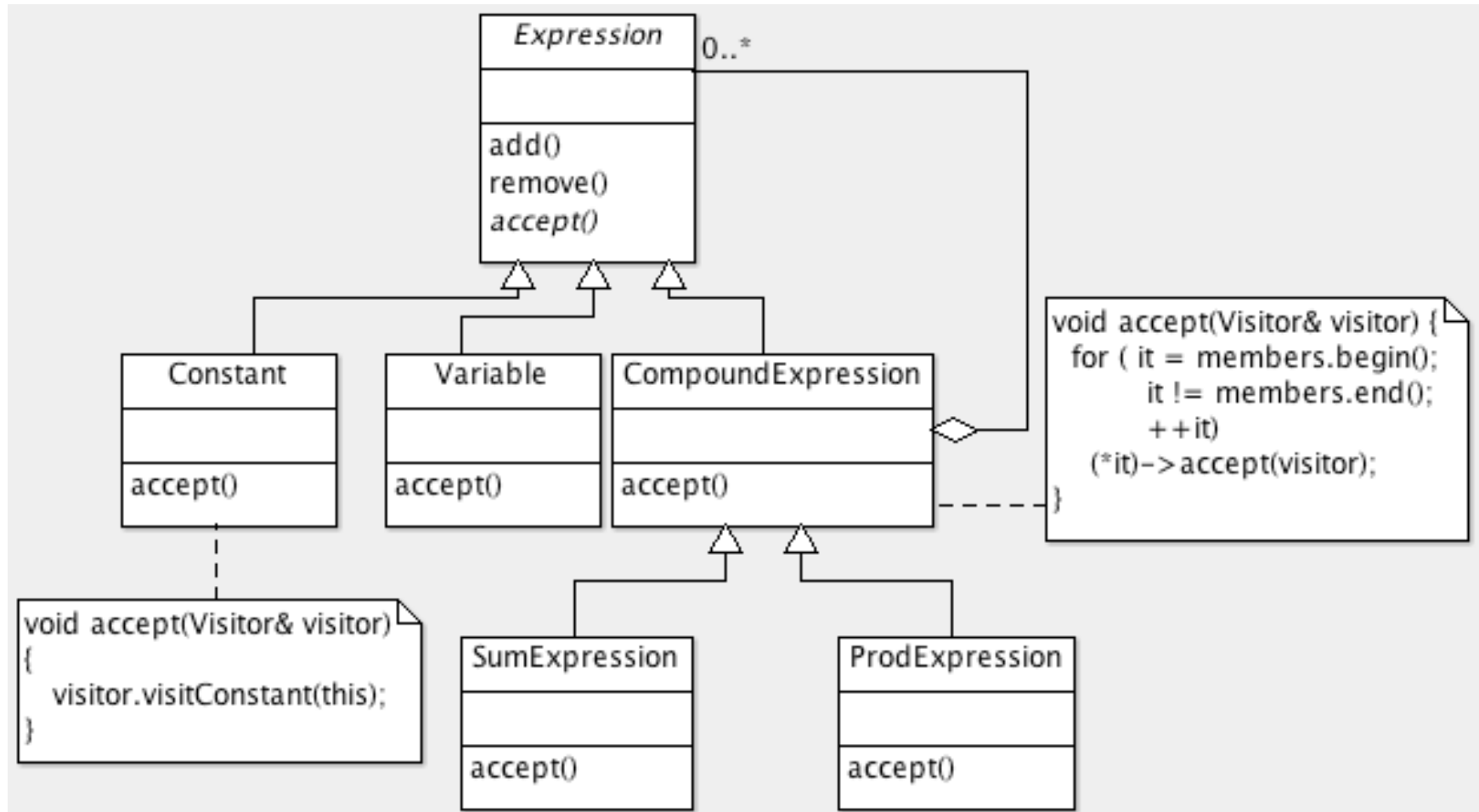
---

- vizitatori pentru expresii
  - afisare
  - evaluare
- vizitatori pentru programe
  - afisare
  - executie (interpretare)

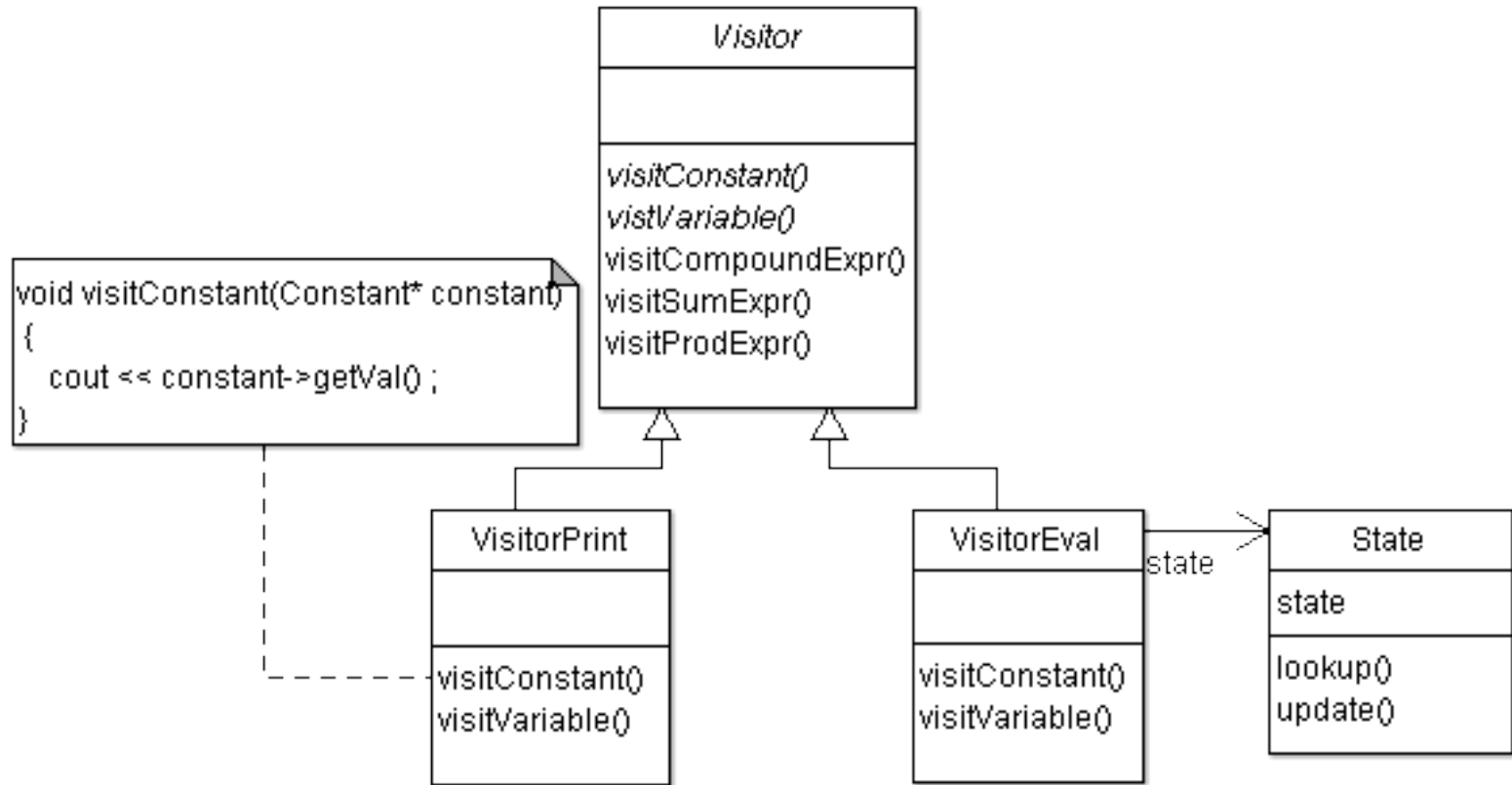
# Expresii



# Adaugarea operatiei accept()



# Vizitatori pentru expresii



## VisitorEval 1/2

---

```
class VisitorEval : public Visitor {
public:
    ...
    void visitConstant(Constant* constant) {
        tempVals.push(constant->getVal());
    }

    void visitVariable(Variable* variable) {
        tempVals.push(
            state->lookup(variable->getName())
        );
    }
}
```

## VisitorEval 2/2

---

```
void visitProdExpression
    (ProdExpression* prod) {
    int temp = 1;
    while (!tempVals.empty()) {
        temp *= tempVals.top();
        tempVals.pop();
    }
    cumulateVal += temp;
}

int getCumulateVal() {return cumulateVal;}

private:
    State *state;
    stack<int> tempVals;
    int cumulateVal;
};
```

## Clientul 1/2

---

```
Constant* one = new Constant(1);  
Constant* two = new Constant(2);  
Variable *a = new Variable("a");  
Variable *b = new Variable("b");
```

```
ProdExpression* e1 = new  
ProdExpression();  
e1->add(one);  
e1->add(a);  
SumExpression* e2 = new SumExpression();  
e2->add(e1);  
e2->add(two);  
e2->add(b);
```

## Clientul 2/2

```
VisitorPrint visitorPrint;  
e1.accept(visitorPrint);
```

1 a \*

scriere postfixata (de ce?)

```
State st;  
st.update("a", 10);
```

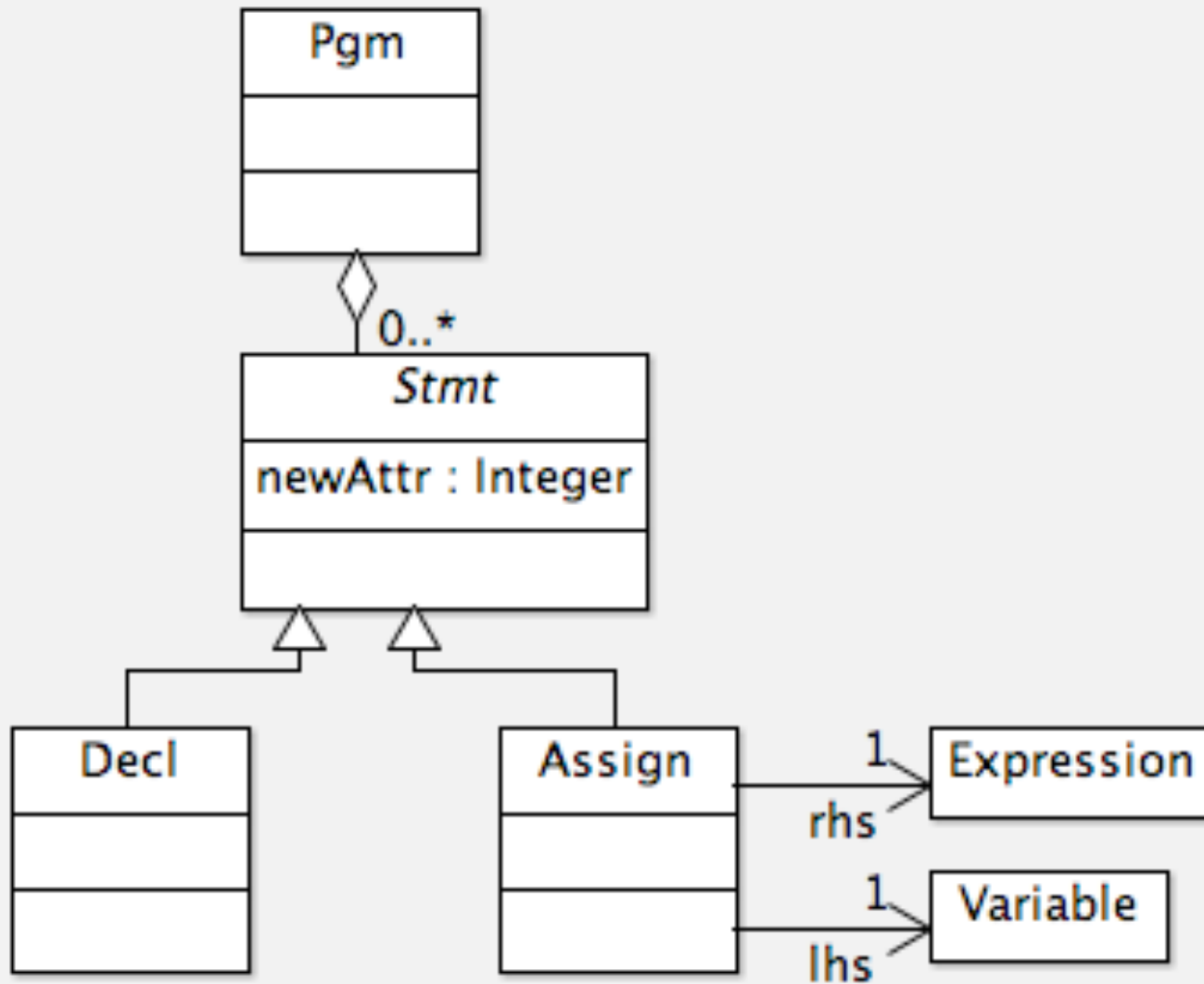
state = (... a |-> 10 ...)

```
VisitorEval visitorEval1(0, &st);  
e1->accept(visitorEval1);  
cout << "e1 = "  
    << visitorEval1.getCumulateVal()  
    << endl;
```

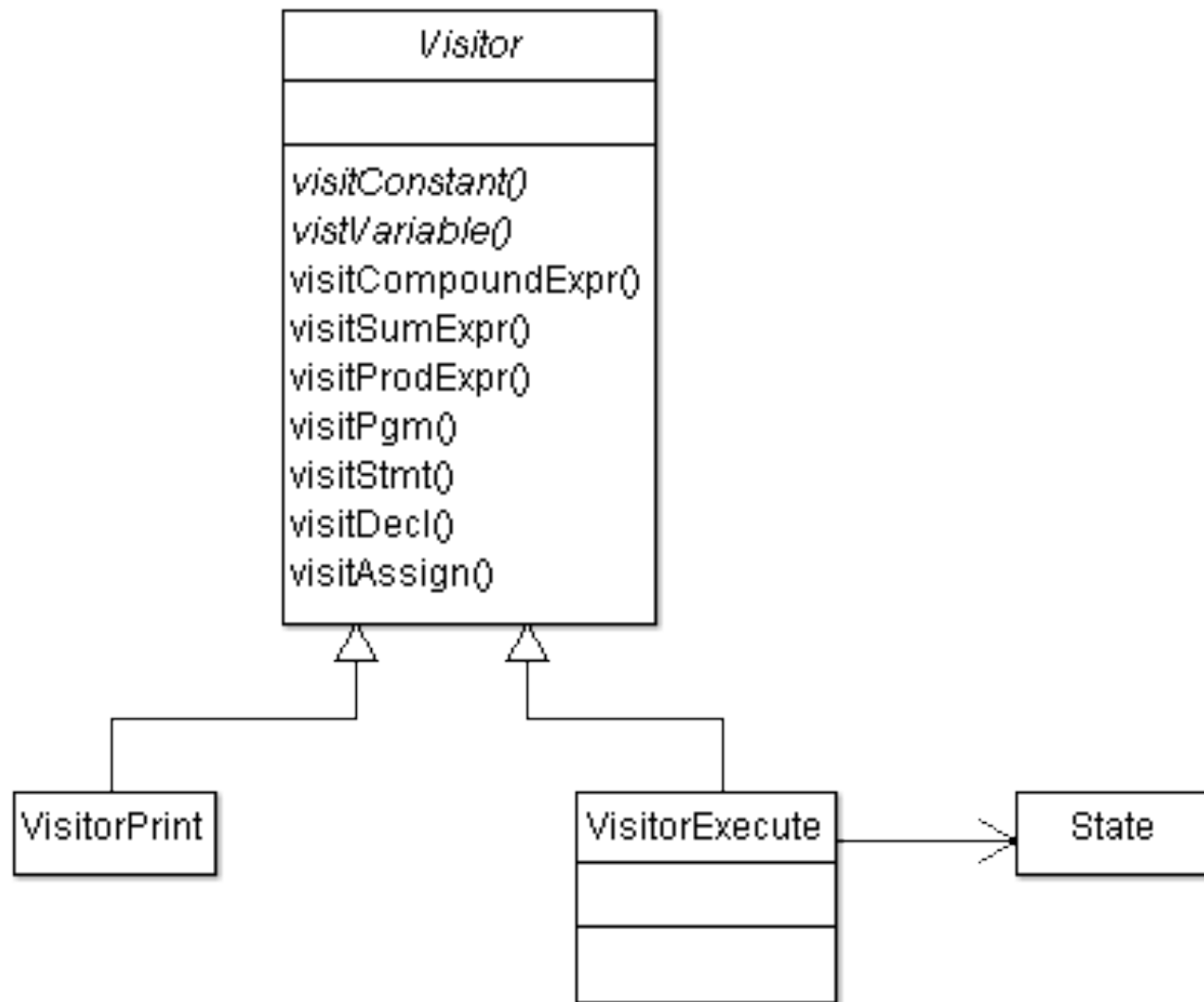
e1 = 10



# Programe



# Vizitator pentru programe



## VisitorExec 1/2

---

```
class VisitorExec : public Visitor {
public:
    void visitDecl(Decl* decl) {
        if (decl->getType() == "int") {
            state->update(decl->getName(), 0);
        }
    }
    void visitAssign(Assign* assign) {
        VisitorEval evalExpr(0, state);
        (assign->getRhs()).accept(evalExpr);
        state->update(assign->getLhs(),
                      evalExpr.getCumulateVal());
    }
}
```

## VisitorExec 2/2

---

```
void visitConstant(Constant* constant) { }
```

```
void visitVariable(Variable* variable) { }
```

```
State& getState() {return state;}
```

```
private:
```

```
    State *state;
```

```
};
```

## Clientul 1/2

---

```
Decl* decl1 = new Decl("int", "a");  
Decl* decl2= new Decl("int", "b");  
Assign* assign1 = new Assign("a", e1);  
Assign* assign2= new Assign("b", e2);
```

```
Pgm pgm;  
pgm.insert(decl1);  
pgm.insert(decl2);  
pgm.insert(assign1);  
pgm.insert(assign2);
```

## Clientul 2/2

---

```
State st2;  
VisitorExec visitorExec(&st2);  
pgm.accept(visitorExec);  
visitorExec.getState().print();
```

```
a |-> 0  
b |-> 2
```

# ***POO***

Patternul  
*Observer*  
(prezentare bazata pe GoF)

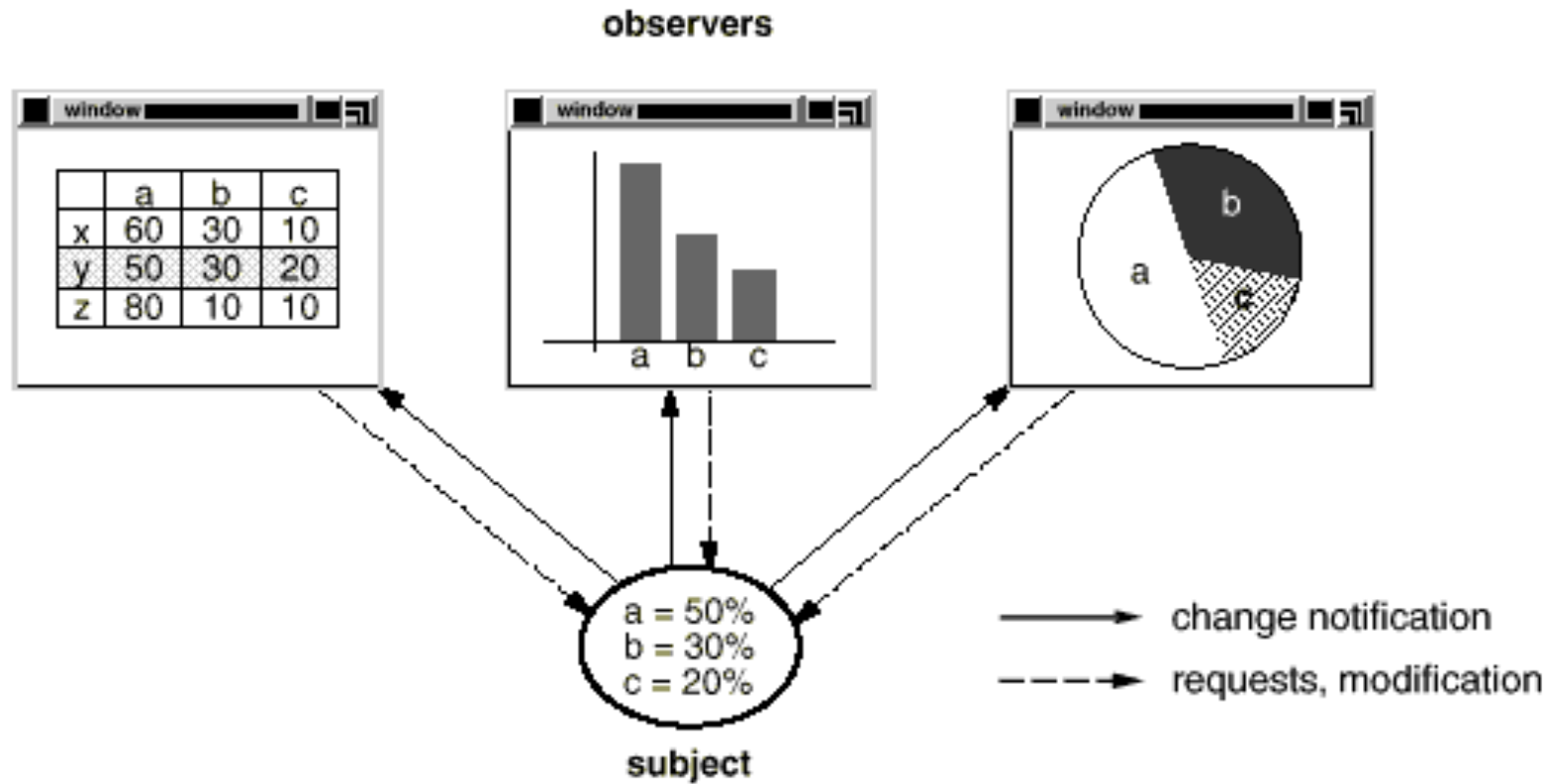
# Observator: :intentie

---

- Defineste o relatie de dependenta 1..\* intre obiecte astfel incat cand un obiect isi schimba starea, toti dependentii lui sunt notificati si actualizati automat



# Observer :: motivatie

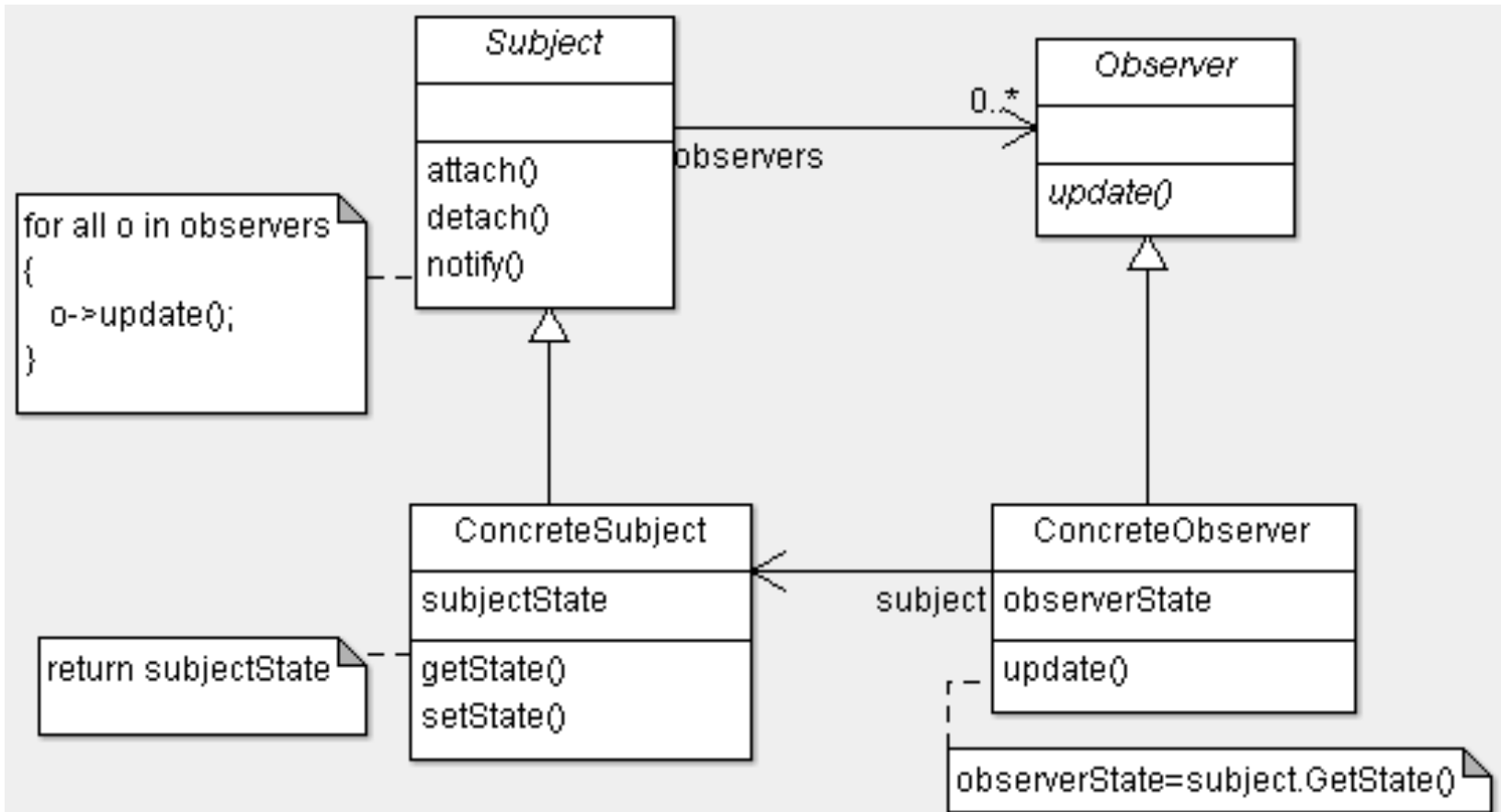


# Observator :: aplicabilitate

---

- cand o abstractie are doua aspecte, unul depinzand de celalalt. Incapsuland aceste aspecte in obiecte separate, permitem reutilizarea lor in mod independent
- cand un obiect necesita schimbarea altor obiecte si nu stie cat de multe trebuie schimbate
- cand un obiect ar trebui sa notifice pe altele, fara sa stie cine sunt acestea
- in alte cuvinte, nu dorim ca aceste obiecte sa fie cuplate strans (a se compara cu relatia de asociere)

# Observer :: structura

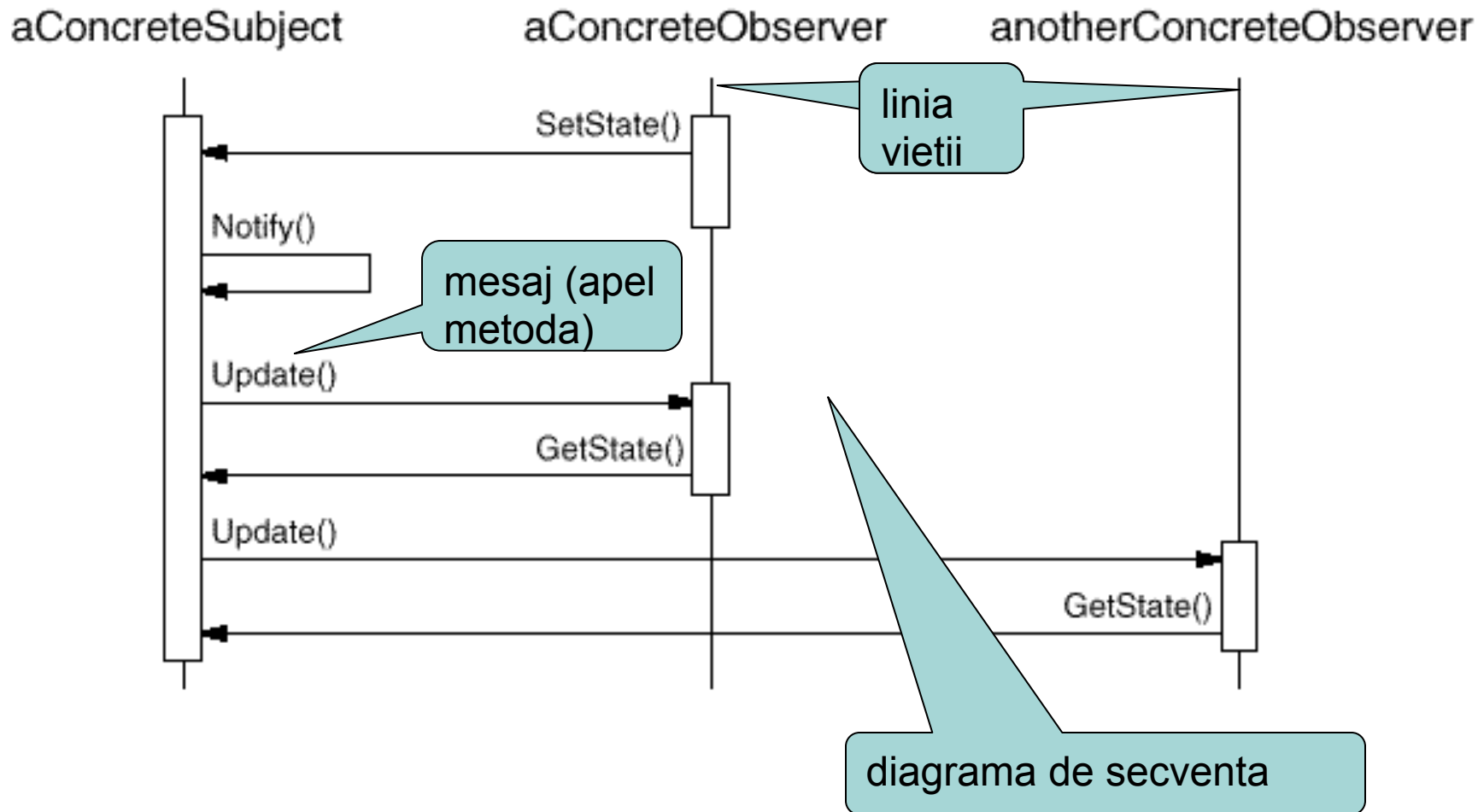


# Observator :: participant

---

- **Subject**
  - cunoaste observatorii (numar arbitrar)
- **Observer**
  - defineste o interfata de actualizare a obiectelor ce trebuie notificate de schimbarea subiectelor
- **ConcreteSubject**
  - memoreaza starea de interes pentru observatori
  - trimite notificari observatorilor privind o schimbare
- **ConcreteObserver**
  - mentine o referinta la un obiect ConcreteSubject
  - memoreaza starea care ar trebui sa fie consistenta cu subiectii

# Observer :: colaborari



# Observator :: consecinte

---

- abstractizeaza cuplarea dintre subiect si observator
- suporta o comunicare de tip “broadcast”
  - notificarea ca un subiect si-a schimbat starea nu necesita cunoasterea destinatarului
- schimbari “neasteptate”
  - o schimbare la prima vedere inocenta poate provoca schimbarea in cascada a starilor obiectelor

# Observator :: implementare

---

- maparea subiectilor la observatori
  - memorarea de referinte la observatori
- observarea mai multor subiecti
- cine declanseaza o actualizare
  1. subiectul apeleaza o metoda Notify() dupa fiecare schimbare
  2. clientii sunt responsabili de apela Notify()
    - fiecare solutie are avantaje si dezavantaje (care?)
- evitarea de referinte la subiecti stersi
  - subiectii ar trebui sa notifice despre stergerea lor (?)
  - ce se intampla cu un observator la primirea vestii?

# Observer :: implementare

---

- fii sigur ca starea subiectului este consistenta inainte de notificare

```
void MySubject::Operation (int newValue) {  
    BaseClassSubject::Operation(newValue);  
    // trigger notification  
    _myInstVar += newValue;  
    // update subclass state (too late!)  
}
```

- evita protocole de actualizare specifice observatorilor
  - modelul push: subiectul trimite notificari detaliate tot timpul, chiar si cand observatorul nu doreste
  - modelul pop: subiectul trimite notificari minimale si observatorul cere detalii atunci cand are nevoie



# Observer :: implementare

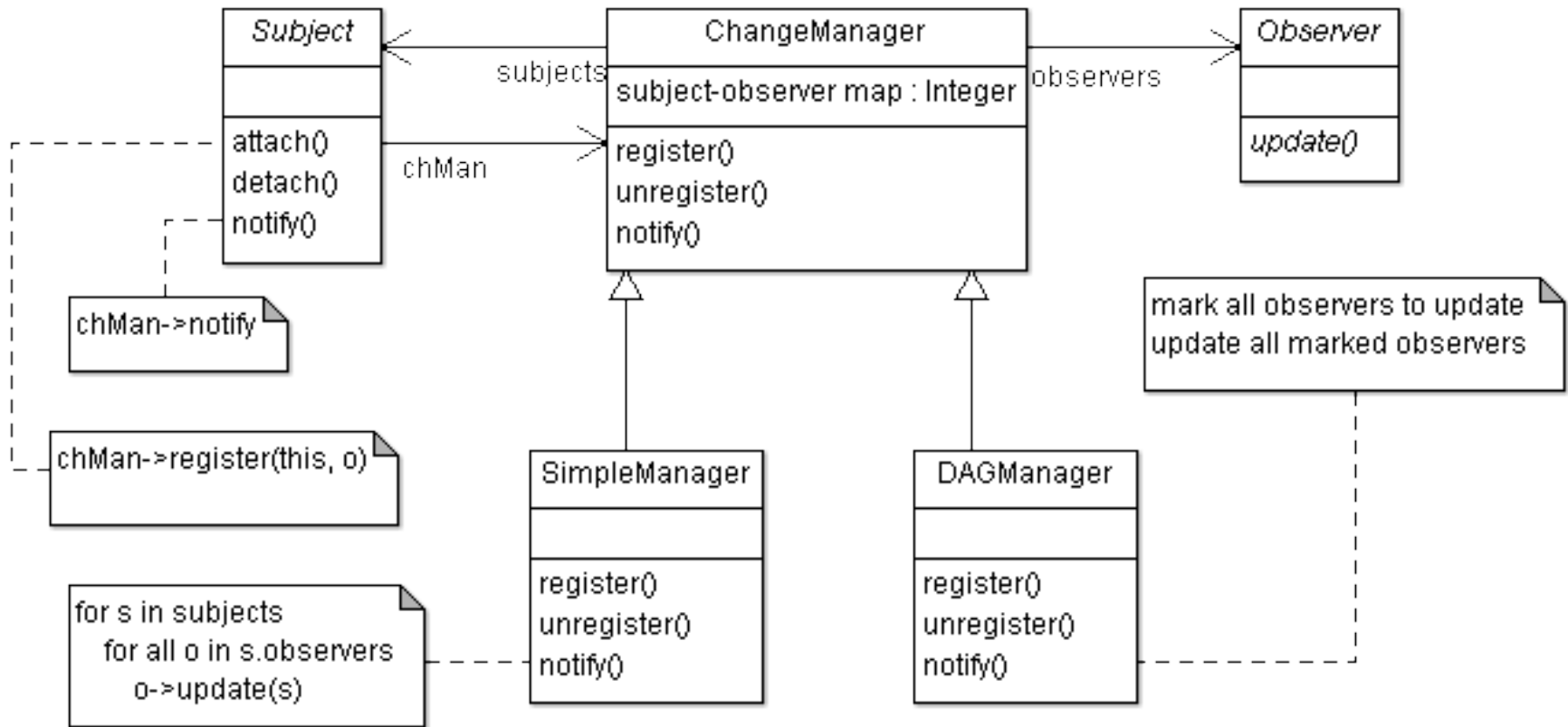
---

- specificarea explicita a modificarilor de interes

```
void Subject::attach(Observer*, Aspect& interest);  
void Observer::update(Subject*, Aspect& interest);
```

- incapsularea actualizarilor complexe
  - relatia dintre subiect si observator este gestionata de un obiect de tip ChangeManager
  - este o situatie frecventa ca o relatie de asociere sa fie implementata prin intermediul unei clase

# Observer :: implementare



# Observer :: cod

---

- clasa abstracta Observer

```
class Subject;  
class Observer {  
public:  
    virtual ~Observer();  
    virtual void update(Subject* theChangedSubject)  
        = 0;  
protected:  
    Observer();  
};
```

metoda abstracta

constructor ascuns (de ce?)

# Observer :: cod

---

- clasa abstracta Subject

```
class Subject {  
public:  
    virtual ~Subject();  
    virtual void attach(Observer*);  
    virtual void detach(Observer*);  
    virtual void notify();  
protected:  
    Subject();  
private:  
    List<Observer*> *_observers;  
};
```

constructor ascuns (de ce?)

Relatia de asociere cu *Observer*

# Observer :: cod

---

- metodele clasei Subject

```
void Subject::attach(Observer* o) {
    _observers->append(o);
}

void Subject::detach(Observer* o) {
    _observers->remove(o);
}

void Subject::notify() {
    ListIterator<Observer*> i(_observers);
    for (i.first(); !i.isDone(); i.next()) {
        i.currentItem()->update(this);
    }
}
```

# Observer :: cod

---

- un subiect concret

```
class ClockTimer : public Subject {
public:
    ClockTimer();
    virtual int getHour();
    virtual int getMinute();
    virtual int getSecond();
    void tick();
};

void ClockTimer::tick() {
    // update internal time-keeping state
    // ...
    notify();
}
```

# Observer :: cod

- un observator concret care mosteneste in plus o interfata grafica

```
class DigitalClock: public Widget, public Observer
{
public:
```

mostenire multipla

```
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();
    virtual void update(Subject*);
    // overrides Observer operation
    virtual void draw();
    // overrides Widget operation;
    // defines how to draw the digital clock
```

```
private:
```

```
    ClockTimer* _subject;
```

```
};
```

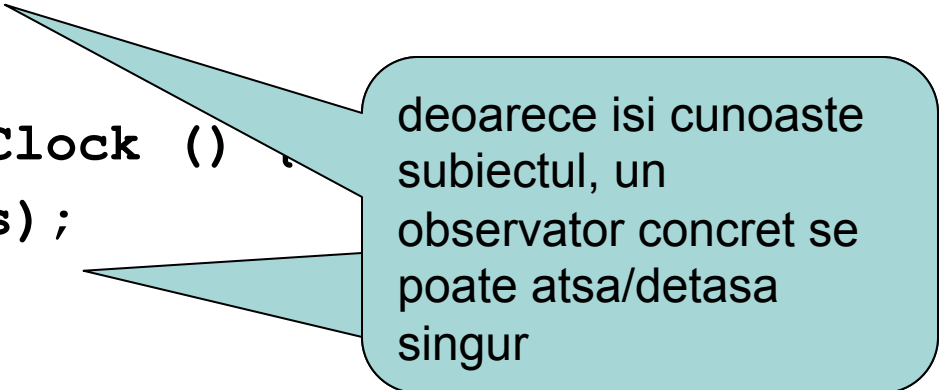
relatia de asociere cu  
"ConcreteSubject"

# Observer :: cod

- constructorul si destructorul observatorului concret

```
DigitalClock::DigitalClock (ClockTimer* s) {  
    _subject = s;  
    _subject->attach(this);  
}
```

```
DigitalClock::~~DigitalClock () {  
    _subject->detach(this);  
}
```



deoarece isi cunoaste  
subiectul, un  
observator concret se  
poate atasa/detasa  
singur



# Observer :: cod

---

- operatia de actualizare

```
void DigitalClock::update
    (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        draw();
    }
}
```

de ce se face  
aceasta verificare?

```
void DigitalClock::draw () {
    // get the new values from the subject
    int hour = _subject->getHour();
    int minute = _subject->getMinute();
    // etc.
    // draw the digital clock
}
```

## Observer :: cod

---

- un alt observator

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void update(Subject*);
    virtual void draw();
    // ...
};
```

- crearea unui AnalogClock si unui DigitalClock care arata acelasi timp:

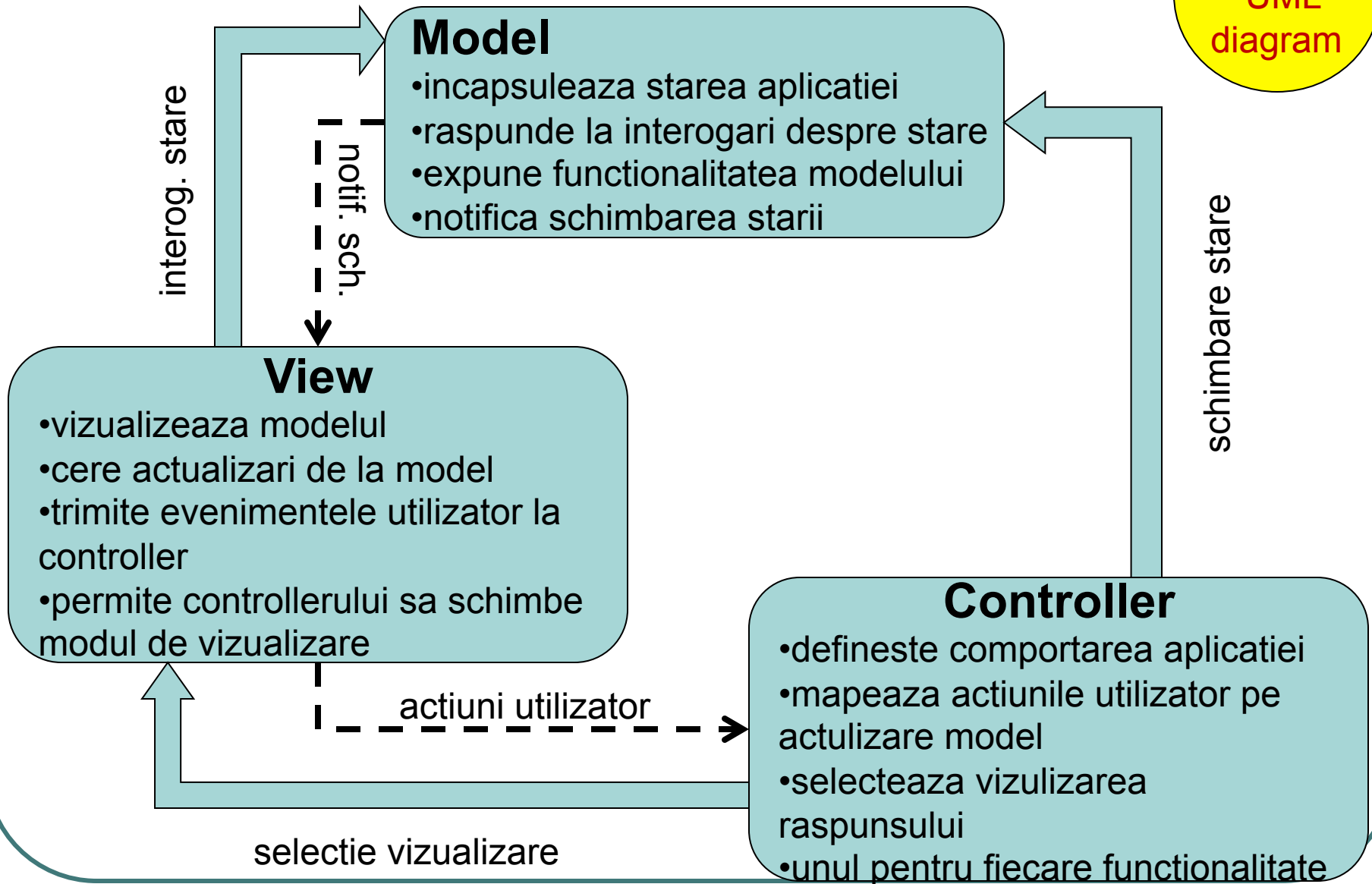
```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

# MVC cu Observer

---

# MVC

not quite  
UML  
diagram

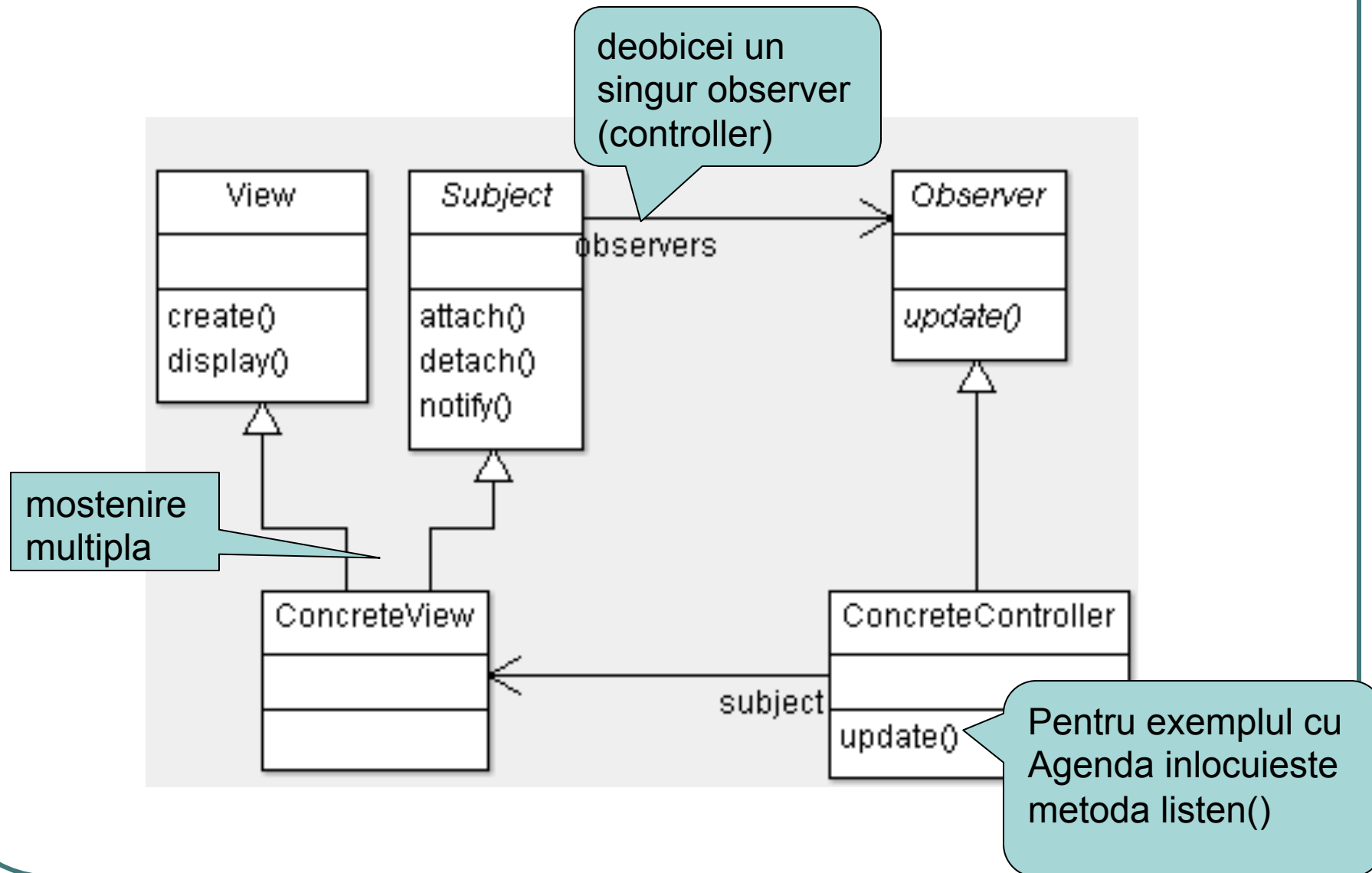


## ***View – Controller modelat cu Observer***

---

- un *Controller* “observa” un *View*
  - un *View* notifica *Controllerul* asociat despre actiunile utilizator
- ⇒ *View* joaca rolul de subiect
- ⇒ *Controller* joaca rolul de observator

# View – Controller modelat cu *Observer*



## ***Model – View cu Observer***

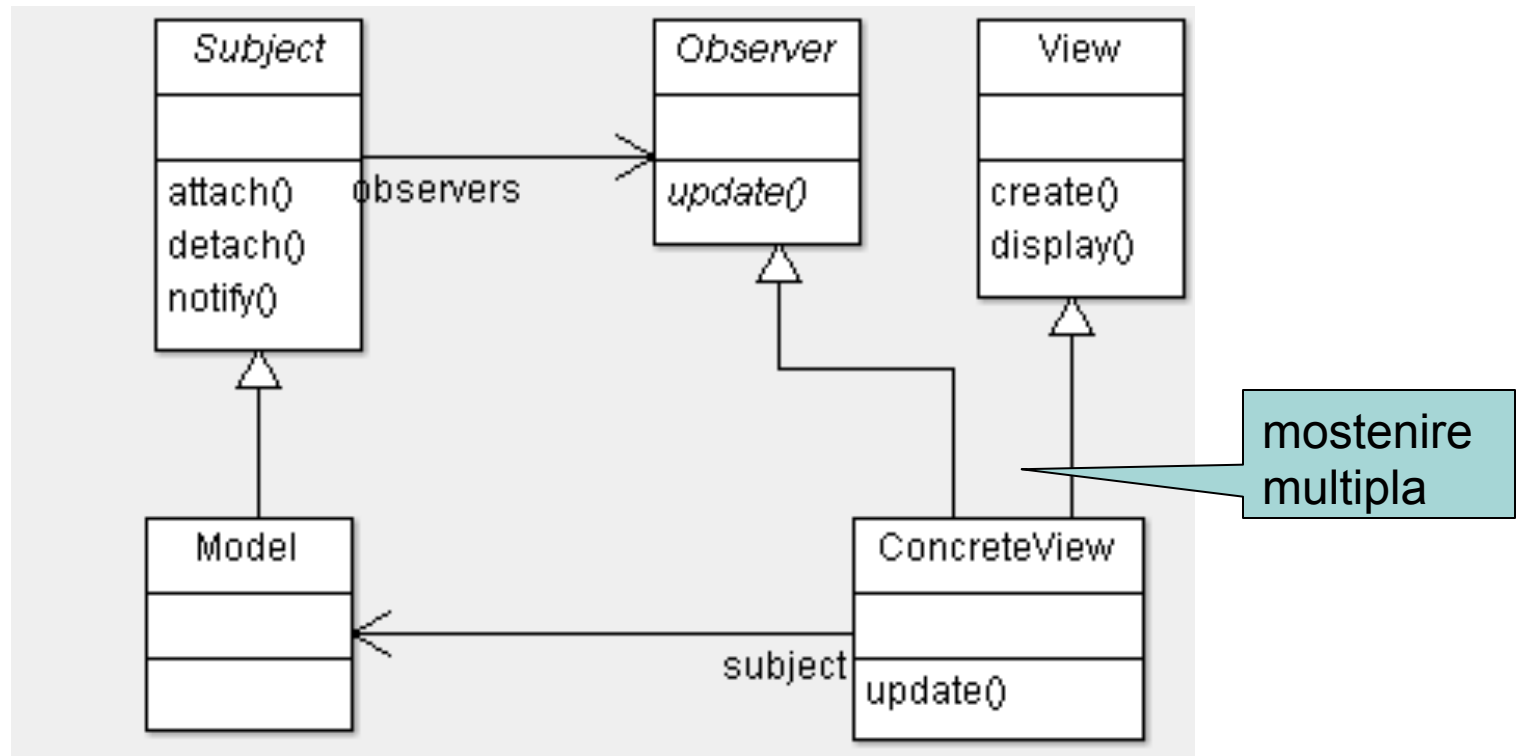
---

- un *View* “observa” un *Model*
- un *Model* notifica *View-urile* asociate despre schimbarea starii

⇒ *Model* joaca rolul de subiect

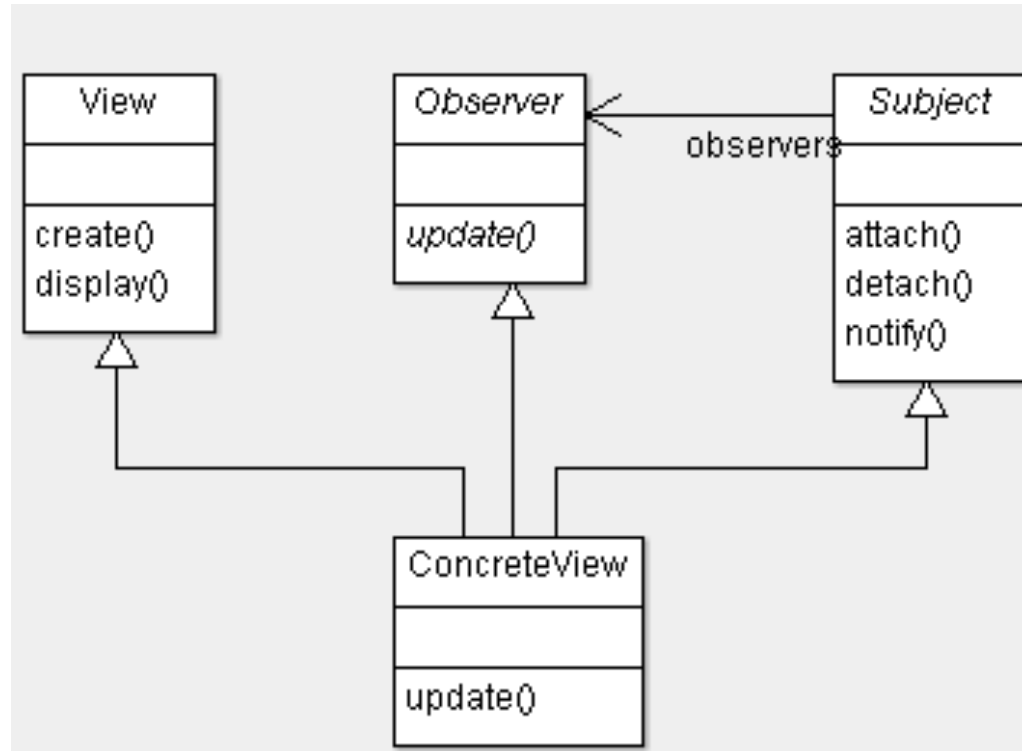
⇒ *View* joaca rolul de observator

# Model – View cu Observer





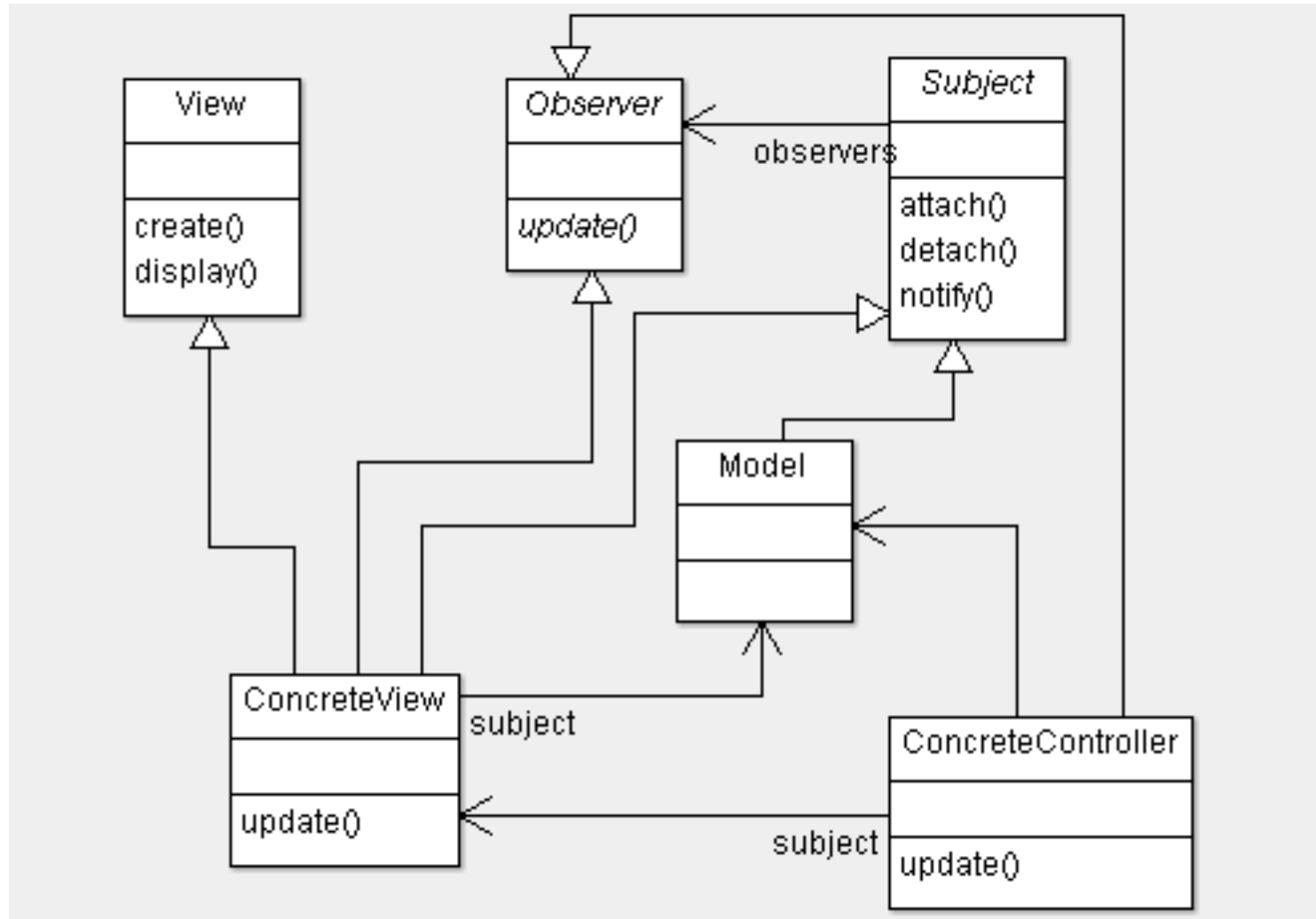
# Oops, View = subject + observator



O fi  
corect?

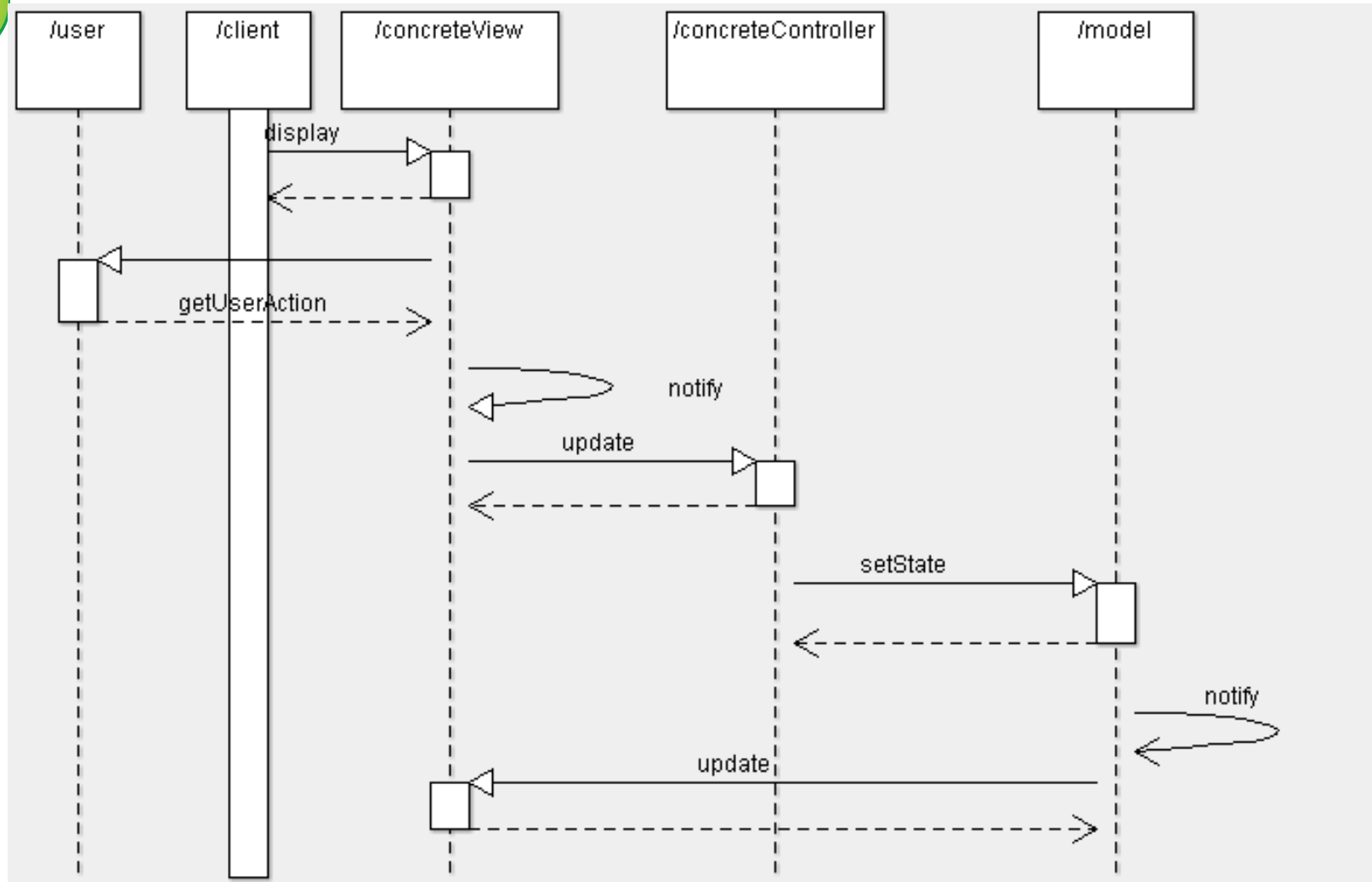


# Diagrama cu toate clasele nu ajuta prea mult





# O diagrama de secventa ma poate lamuri



# ***P00***

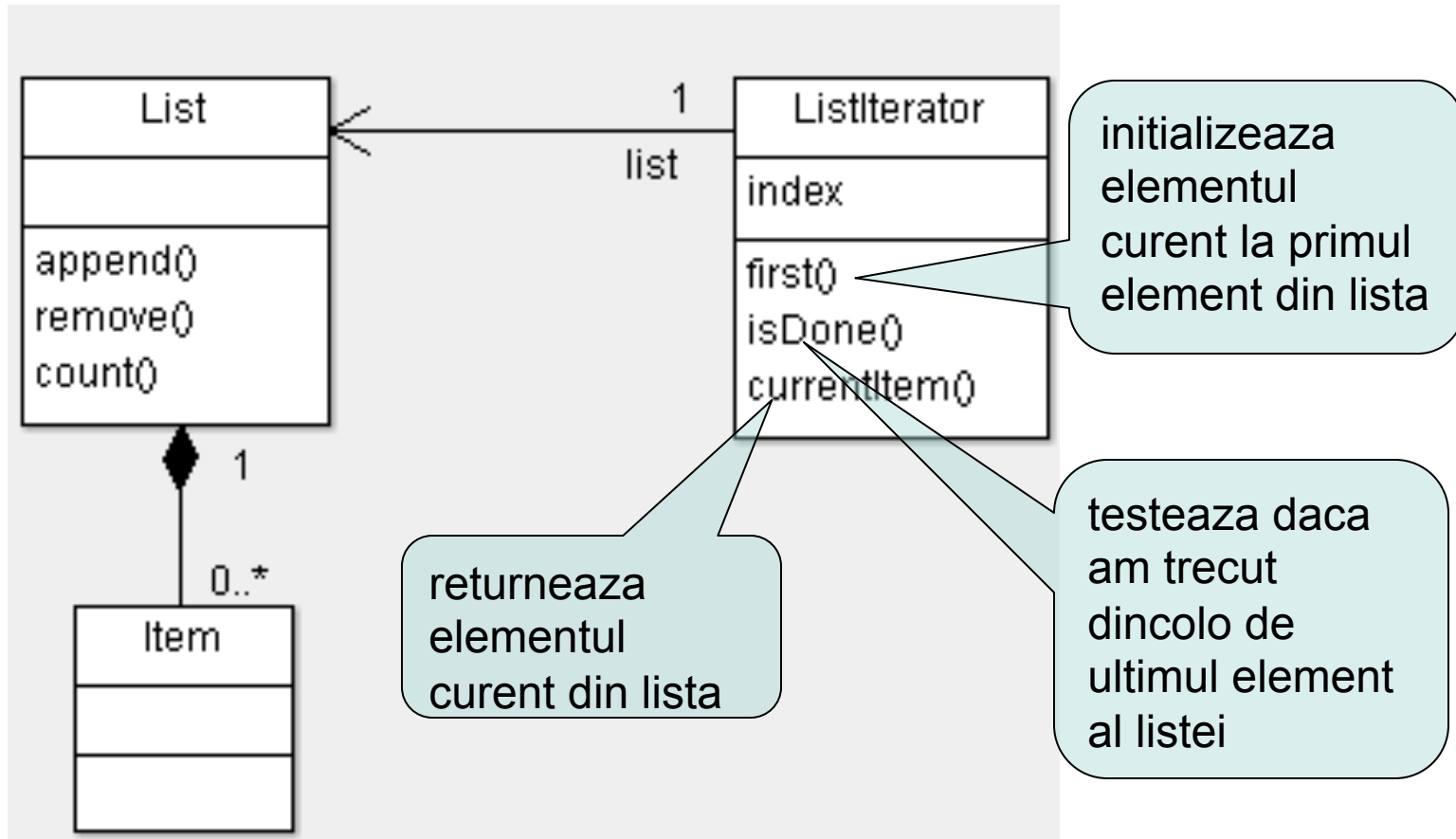
Patternul  
*Iterator*  
(prezentare bazata pe GoF)

# Iterator::intentie

---

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

# Iterator::motivatie



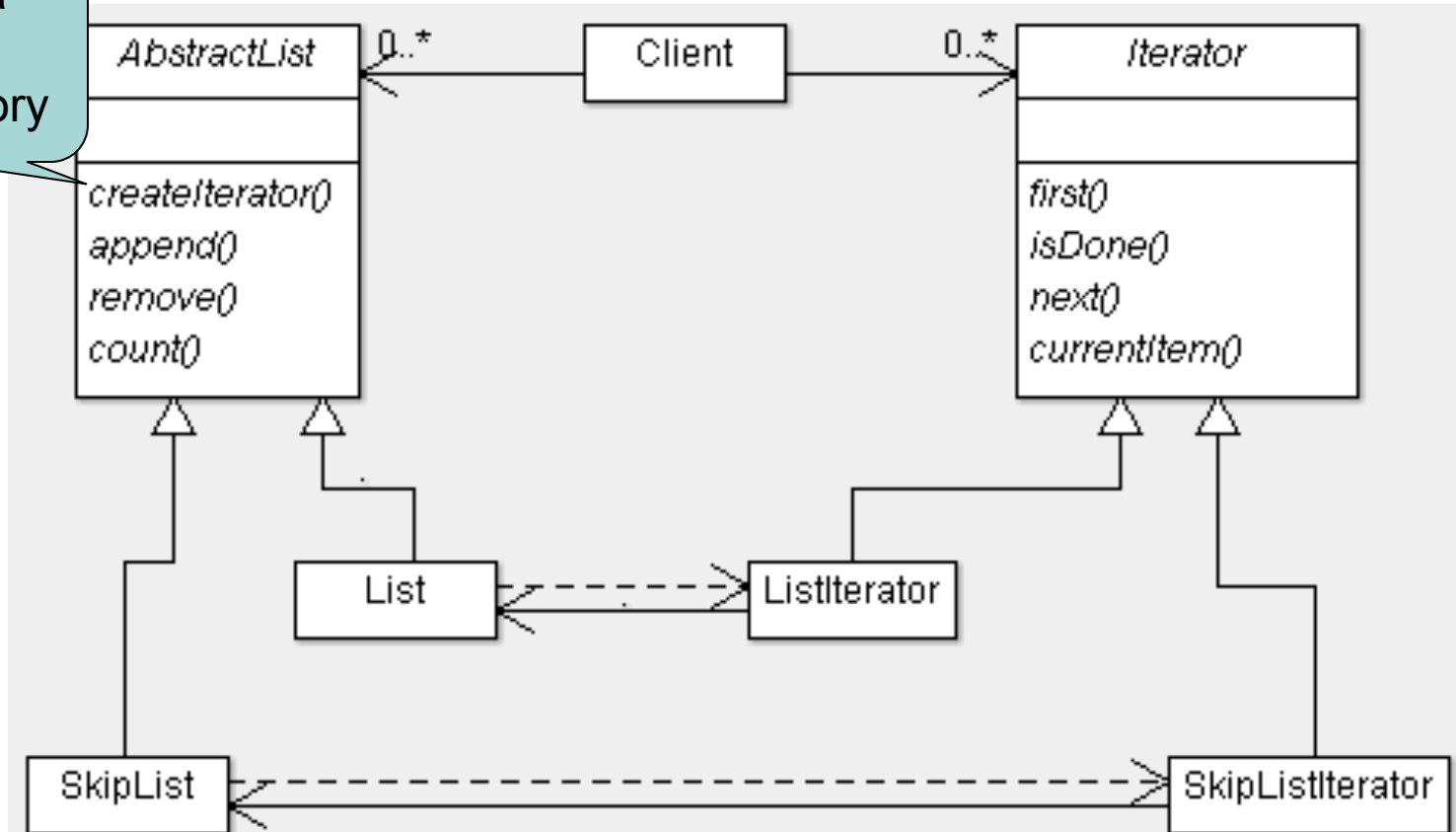
# Iterator::motivatie

---

- 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

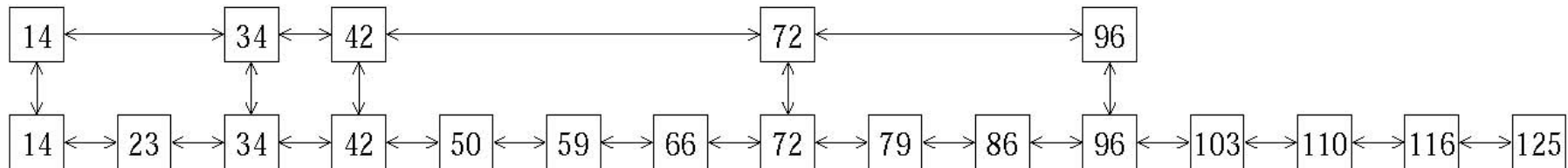
vom discuta  
mai mult la  
ObjectFactory



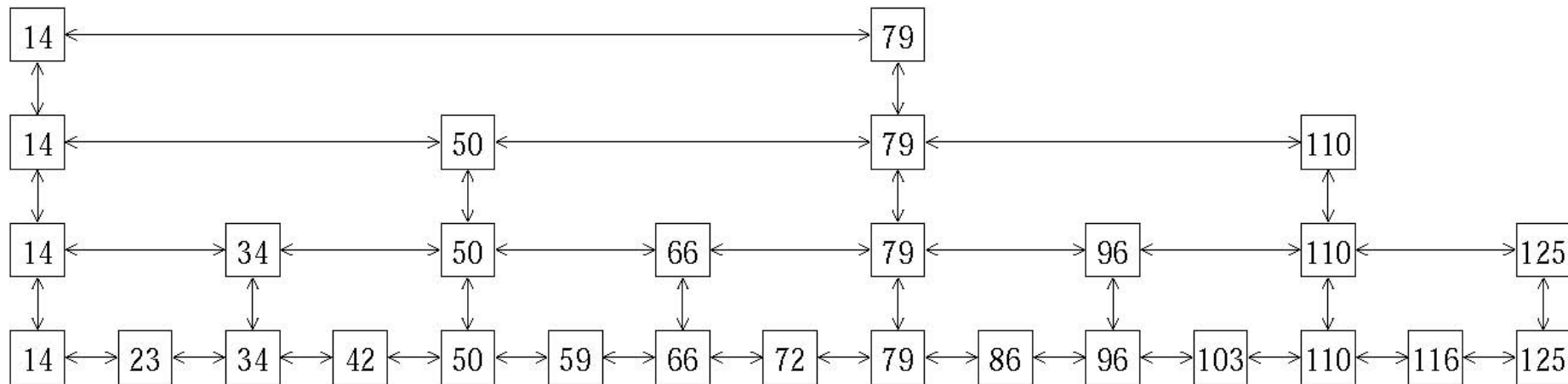


# intermezzo structuri de date – skip list

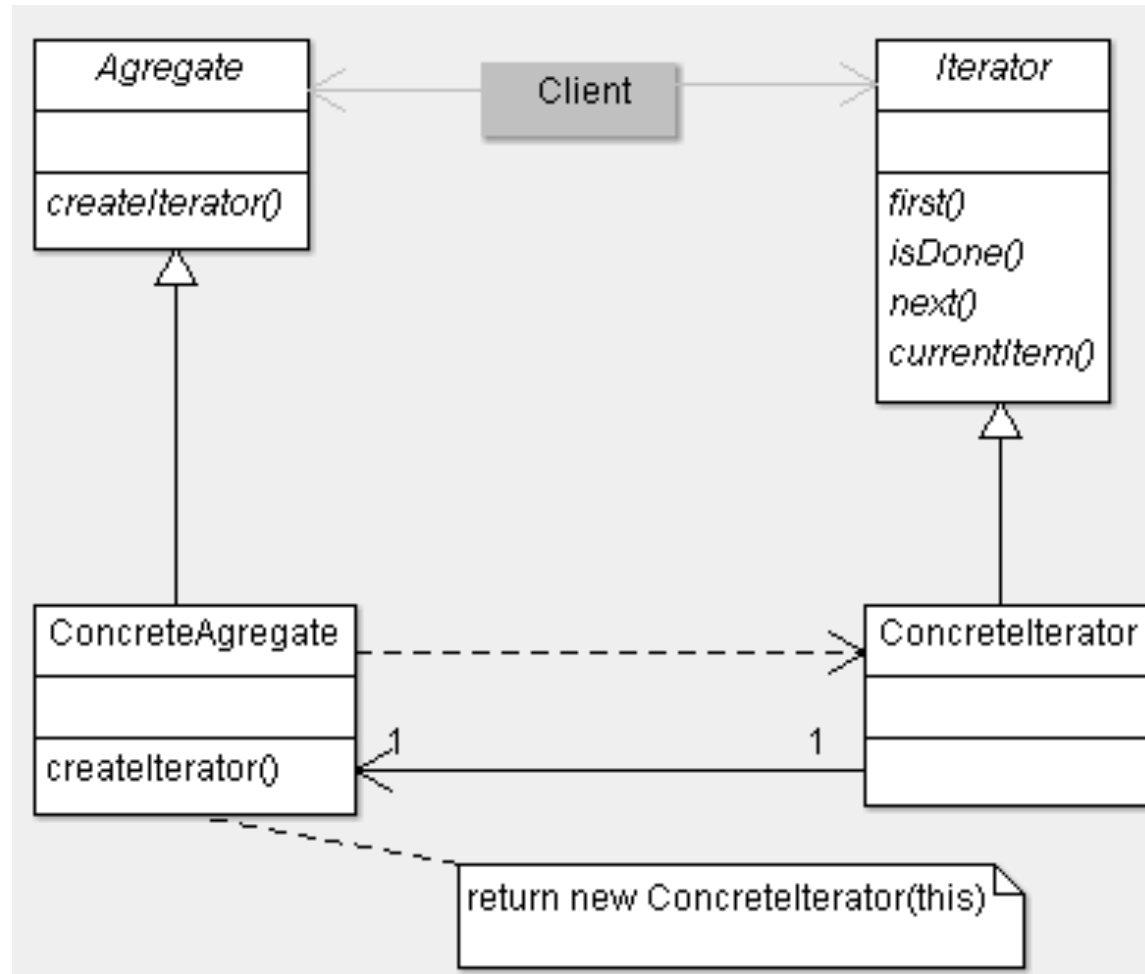
- structuri de date aleatoare simple si eficiente pentru cautare
- structura pe 2 nivele (cost operatie de cautare:  $2 \sqrt{n}$ )



- structura pe 4 nivele (similara unui arbore binar)



# Iterator::structura



# Iterator::participanti

---

- *Iterator*
  - definește interfata de accesare și traversare a componentelor
- *ConcretIterator*
  - 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 *ConcretIterator*.

# 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();  
    }  
}
```

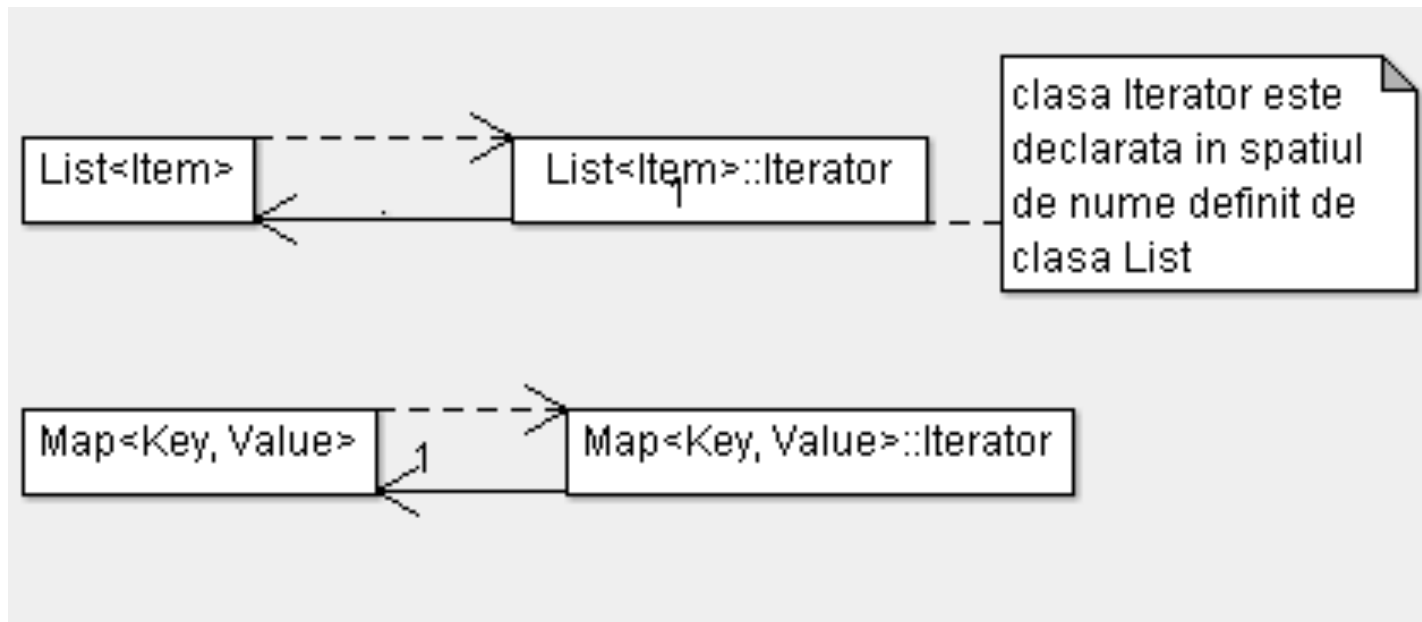
schema de  
parcure a unei  
liste cu iteratori

```
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::iterator 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::iterator 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

implemen  
tare 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

---

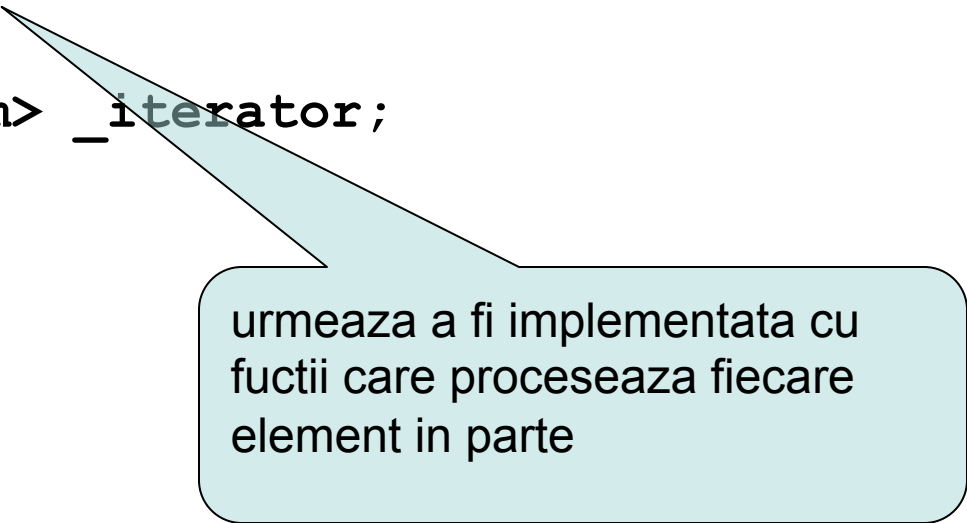
- mai este numit si iterator *pasiv*
- cum parametrizam un iterator cu operatia pe care dorim sa o executam peste fiecare element?
- o problema: C++ nu suporta functii anonime
- solutii posibile:
  - un parametru pointer la o functie
  - subclase care suprascriu functia cu comportarea dorita
- ambele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

- utilizezare

```
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::cod::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;
}
```