POO

Sabloane

Visitor

Cuprins

- sablonul Visitor (prezentare bazata pe GOF)
- aplicatie: interpretor pentru un limbaj simplu

Intentie

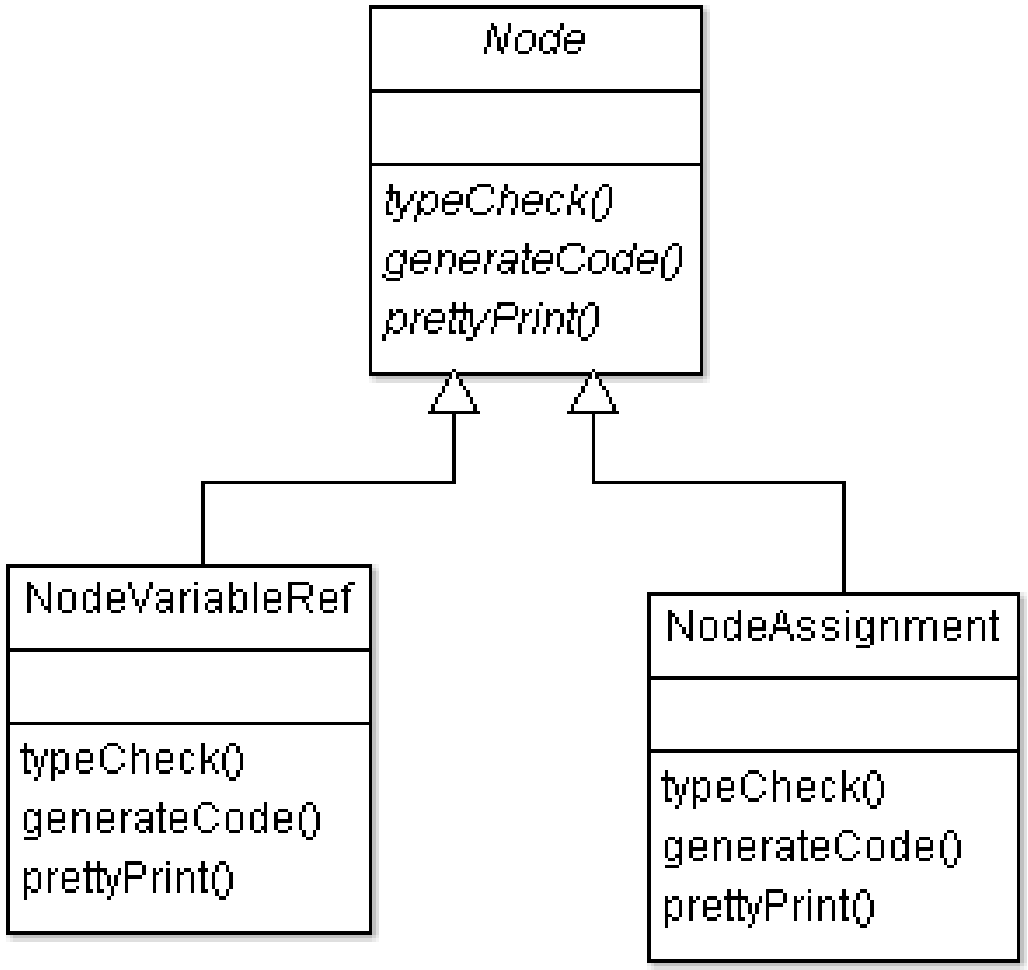
- reprezinta o operatie care se excuta peste elementele unei structuri de obiecte
- permite sa definirea de noi operatii fara a schimba clasele elementelor peste care lucreaza

Motivatie

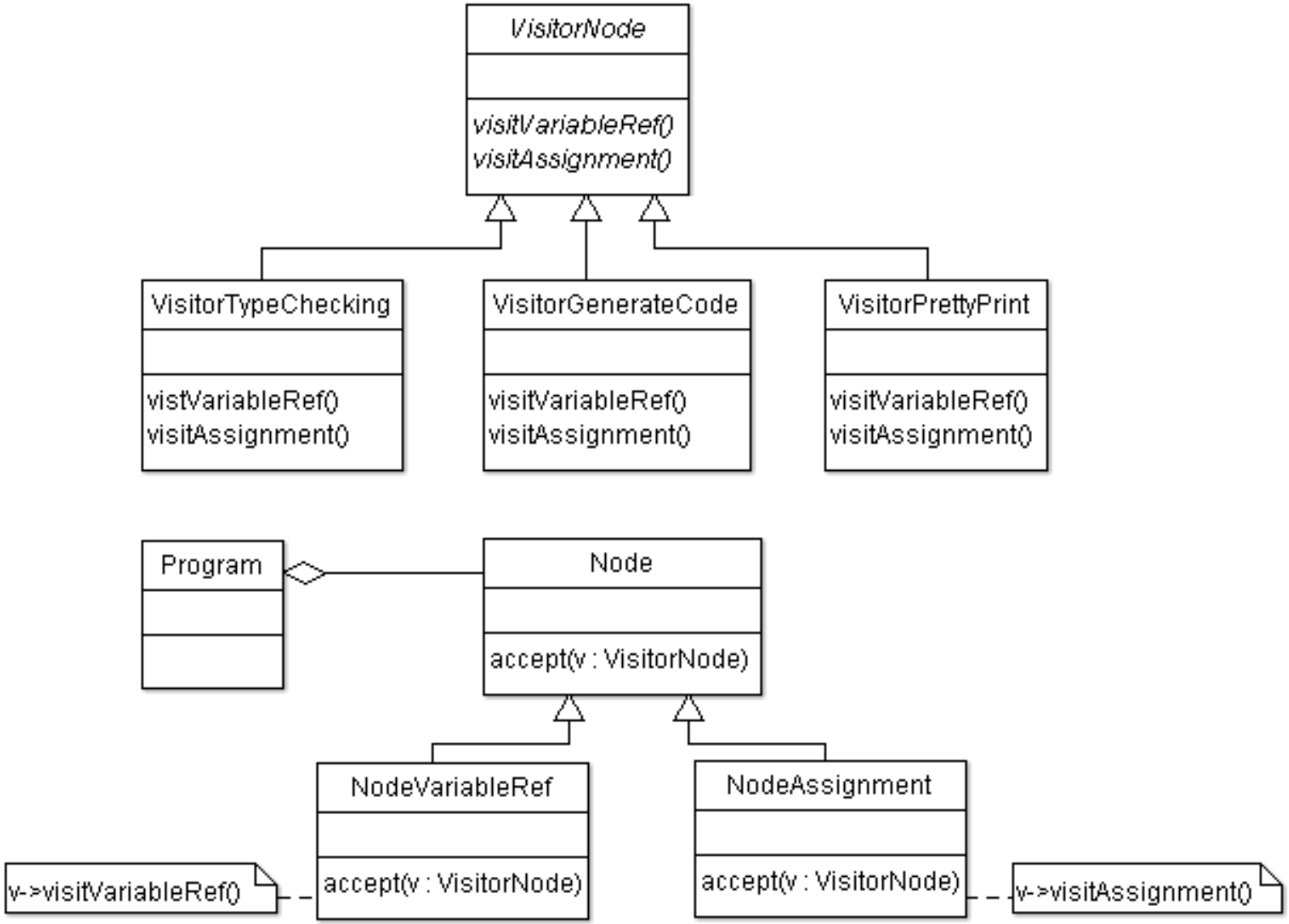
- Un compilator reprezinta un program ca un arbore sintactic abstract (AST). Acest arbore sintactic este utilizat atat pentru semantica statica (e.g., verificarea tipurilor) cat si pentru generarea de cod, optimizare de cod, afisare.
- Aceste operatii difera de la un tip de instructiune la altul. De 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



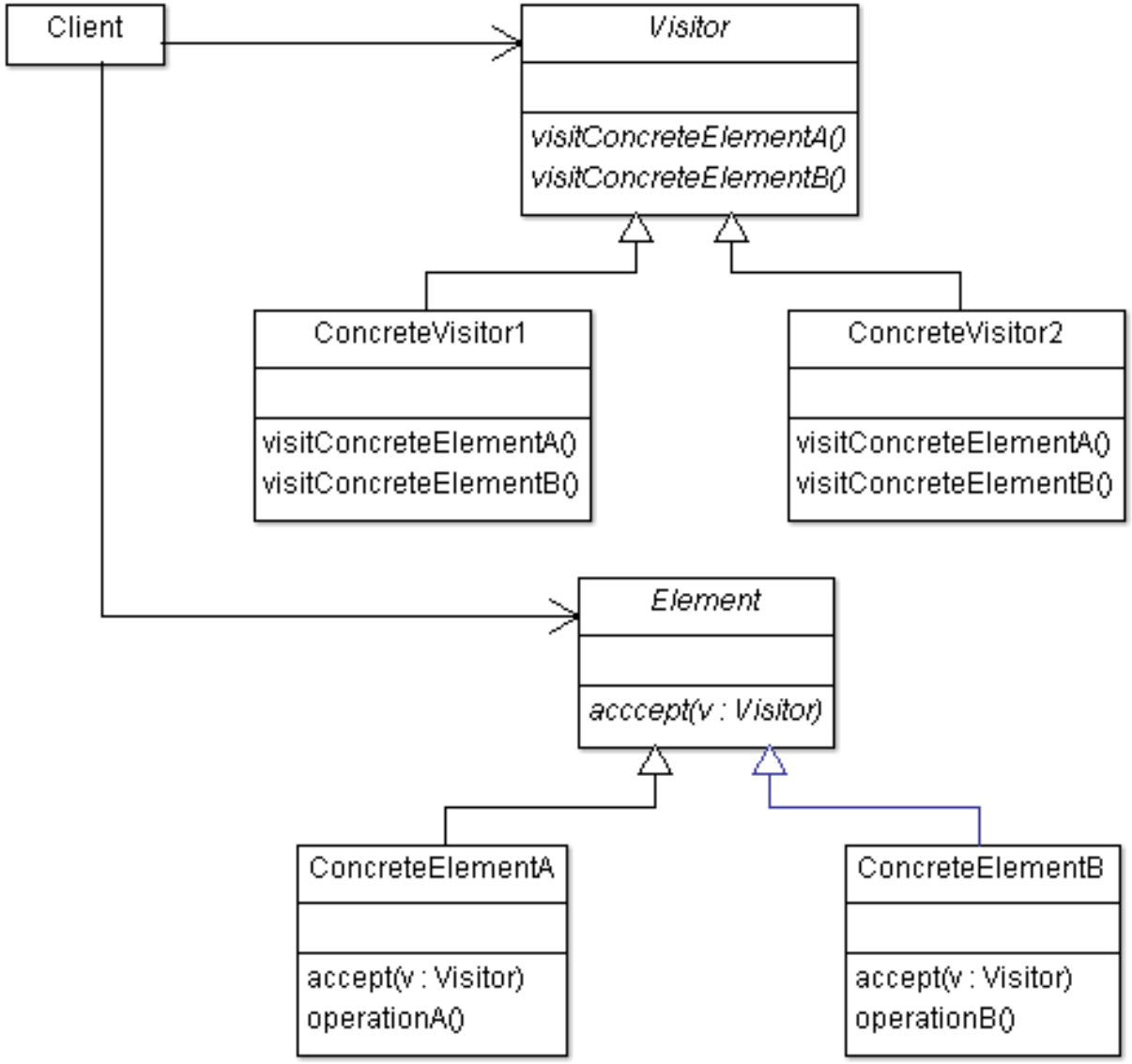
Solutia cu vizitatori



Aplicabilitate

- O structura de obiecte contine mai multe clase cu interfete diferite si se doreste realizarea unor operatii care depind de aceste calse
- Operatiile care se executa nu au legatura cu clasele din structura si se doreste eviatrea “poluarii” acestor clase. Sablonul Visitor pune toate aceste operatii intr-o singura clasa. Cand structura este utilizata in mai multe aplicatii, in Vistor 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 schimabrea intefetelor tuturor vizitatorilor.

Structura


Client
Visitor
visitConcreteElementA()
visitConcreteElementB()
ConcreteVisitor1
visitConcreteElementA()
visitConcreteElementB()
ConcreteVisitor2
visitConcreteElementA()
visitConcreteElementB()
Element
acccept(v : Visitor)
ConcreteElementA
accept(v : Visitor)
operationA()
ConcreteElementB
accept(v : Visitor)
operationB()

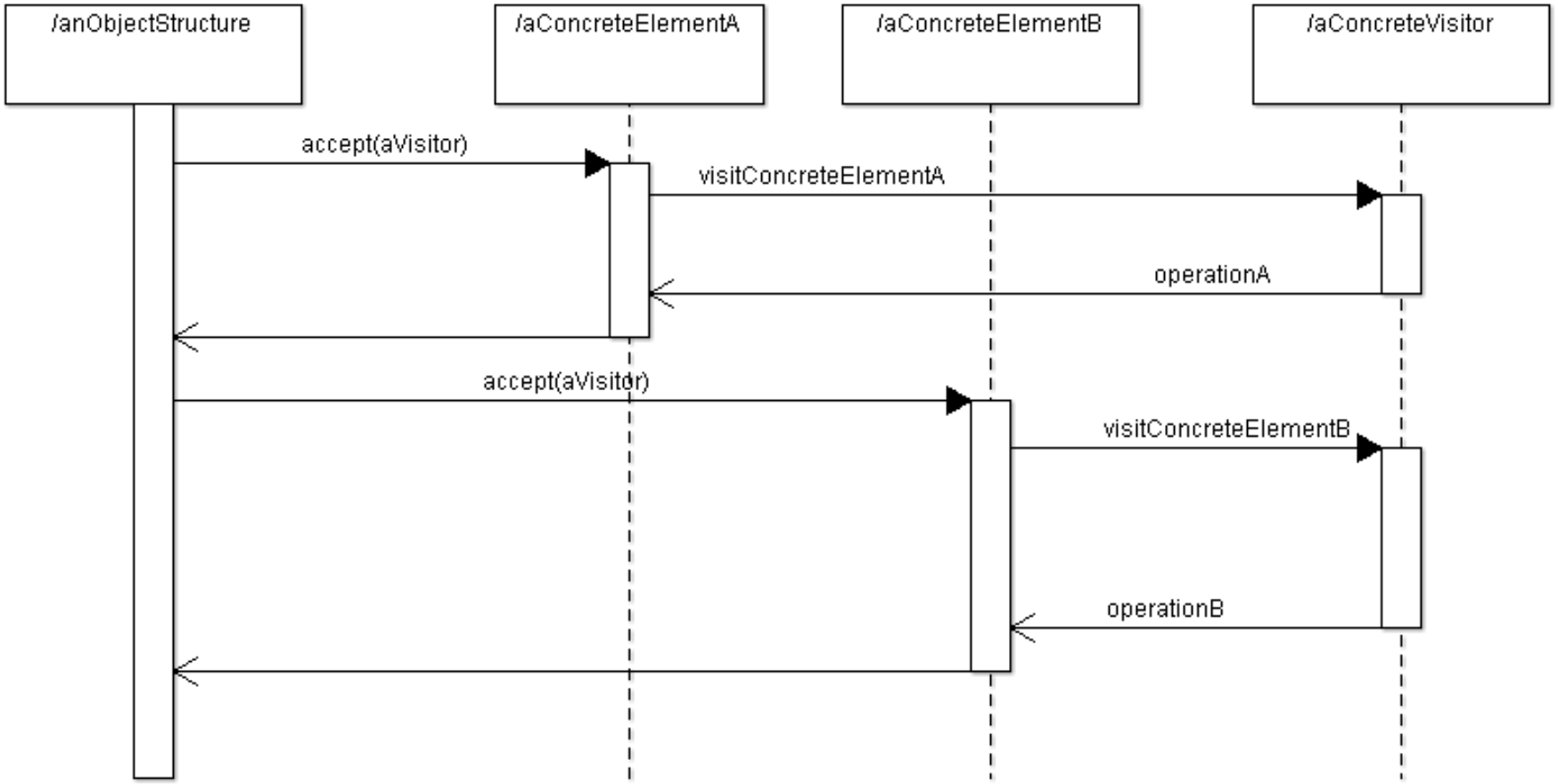
Participanti 1/2

- **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 vizitatoruli sa identifice elemntul concret pe care il viziteaza. Apoi, vizitatorul poate vizita elemntul prin intermediul interfetei sale.
- **ConcreteVisitor** (TypeCheckingVisitor)
  - implementeaza fiacre opetaie declara de vizitator. Fiecare operatie implementeaza un fragment din algoritmul de vzitare care corespunde elementului din structura vizitat. Memoreaza starea algoritmului de vzitare, care de multe ori acumuleaza rezultatele obtinute in timpul vzitarii elementelor din structura.

Participanti 2/2

- **Element** (Node)
  - defineste operati de acceptare, care are ca argument un vizitator
- **ConcreteElement** (AssignmentNode,VariableRefNode)
  - implementeaza operatia de acceptare
- **ObjectStructure** (Program)
  - poate enumera elementele sale
  - poate furniza o interfata la nivel inalt pentru un vizitator care viziteaza elementele sale
  - poate fi un “composite”

Colaborari


/anObjectStructure
/aConcreteElementA
/aConcreteElementB
/aConcreteVisitor
accept(aVisitor)
visitConcreteElementA
operationA
accept(aVisitor)
visitConcreteElementB
operationB

Consecinte 1/2

- *Visitor face adaugarea de noi operatii usoara*
- *Un vizitator aduna opratiile care au legatura intre ele si le separa pe cele care nu au legatura*
- *Adaugarea de noi clase ConcreteElement la structura este dificila.* Provoaca scimbarea interfetelor tutoror 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.* Elementel 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);
  }
};
```

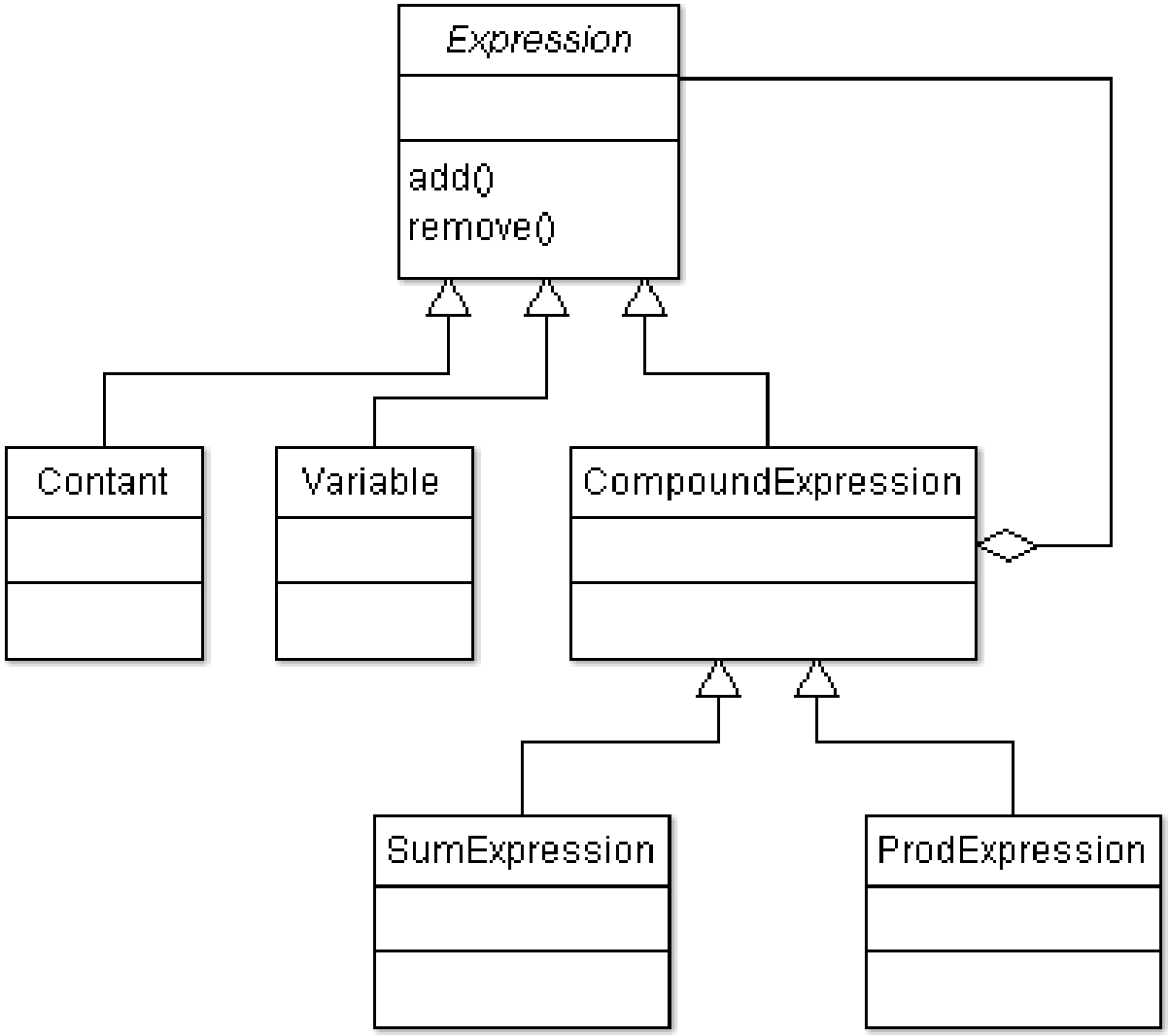
Implementare 3/3

- *Simple dispatch.* Operatia care realizeaza o cerere depinde de doua criterii: numele cererii si tipul receptorului. De exemplu, operatia o cerere *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

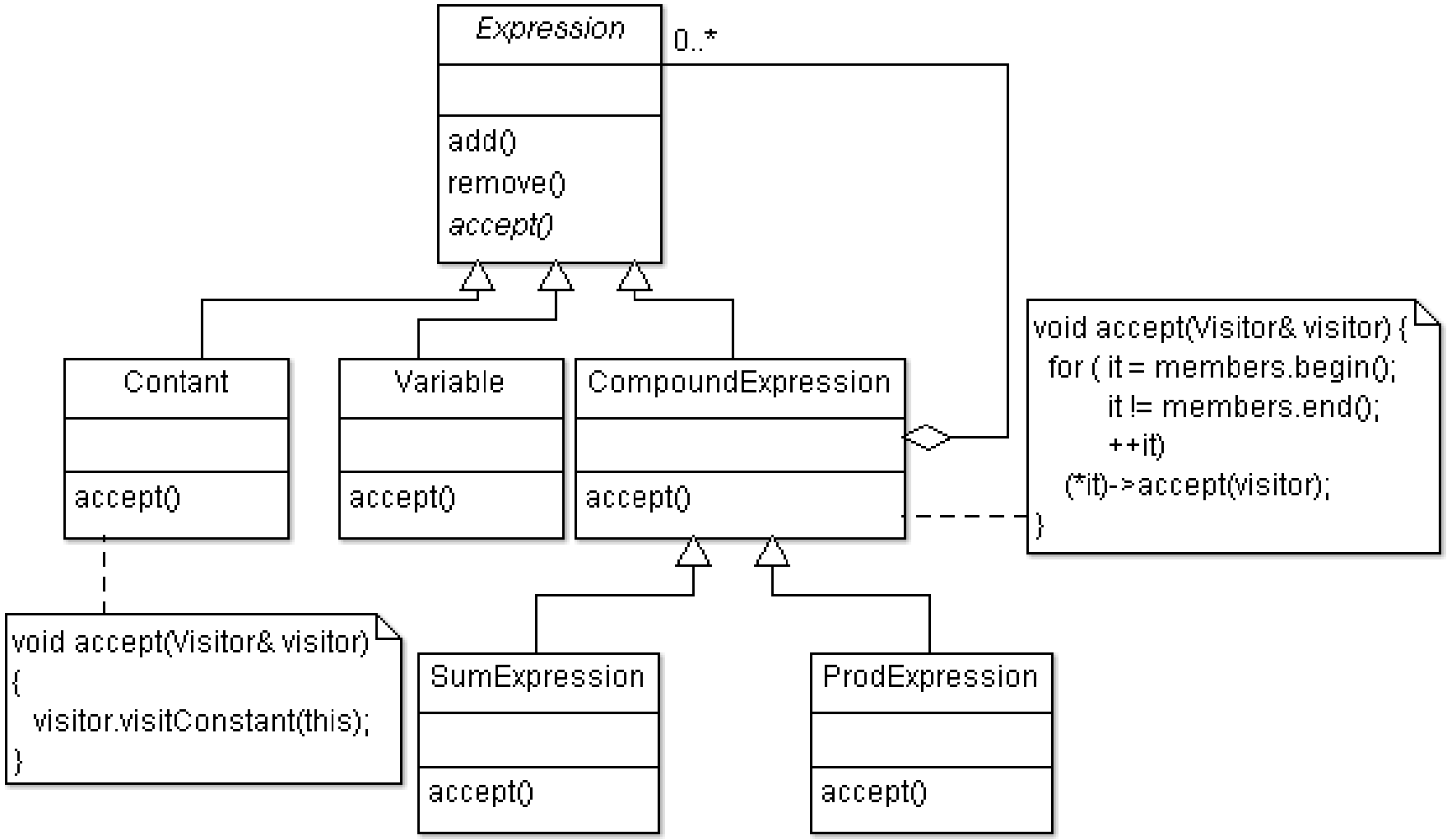
Aplicatie

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

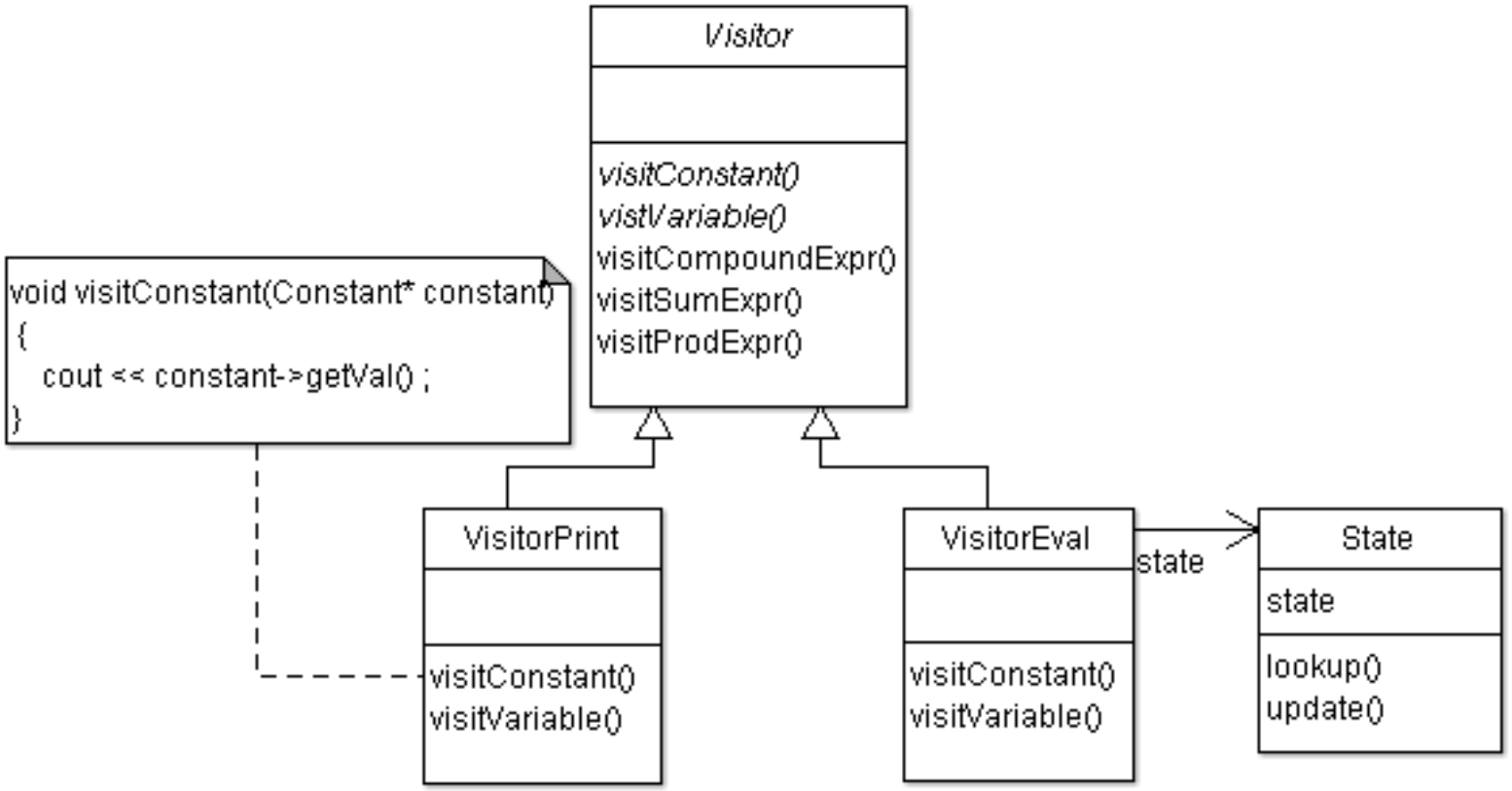
Expresii


Expression
add()
remove()
Contant
Variable
CompoundExpression
SumExpression
ProdExpression

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?)

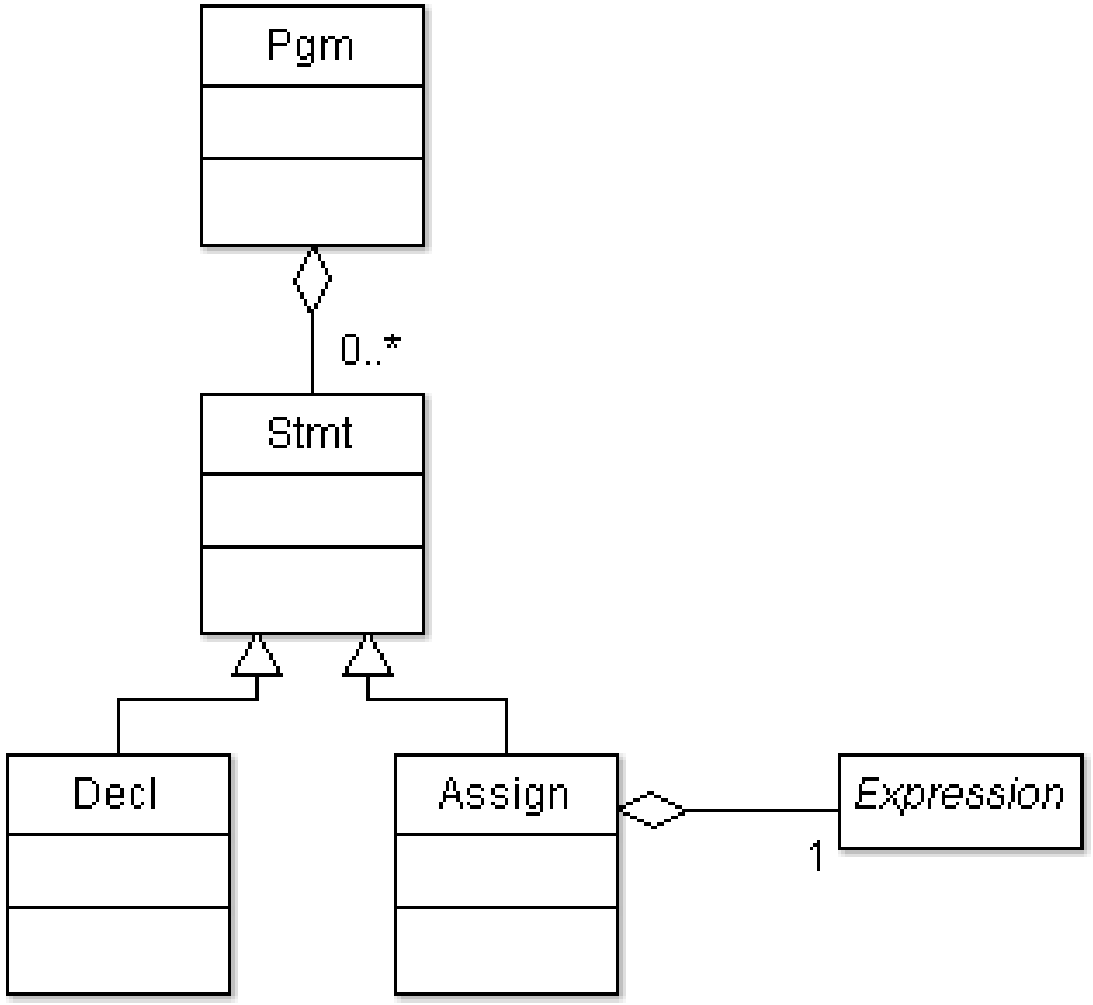
```
State st;
st.update("a", 10);
```

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

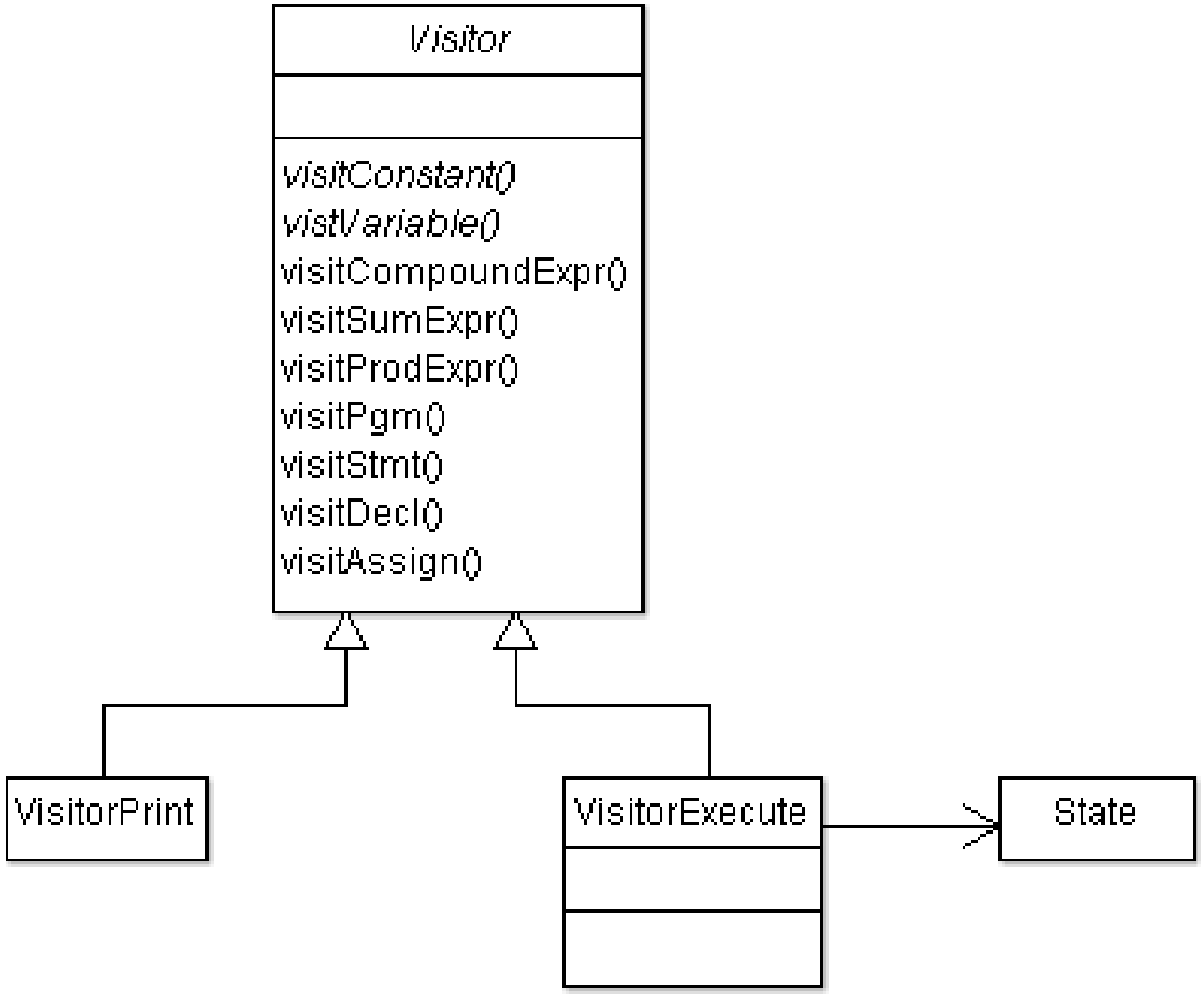
```
VisitorEval visitorEval1(0, st);
e1->accept(visitorEval1);
cout << "e1 = “
     << visitorEval1.getCumulateVal()
     << endl;
```

```
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

```
VisitorExec visitorExec;
pgm.accept(visitorExec);
visitorExec.getState().print();
```

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