

POO

Sabloane de proiectare

Cuprins

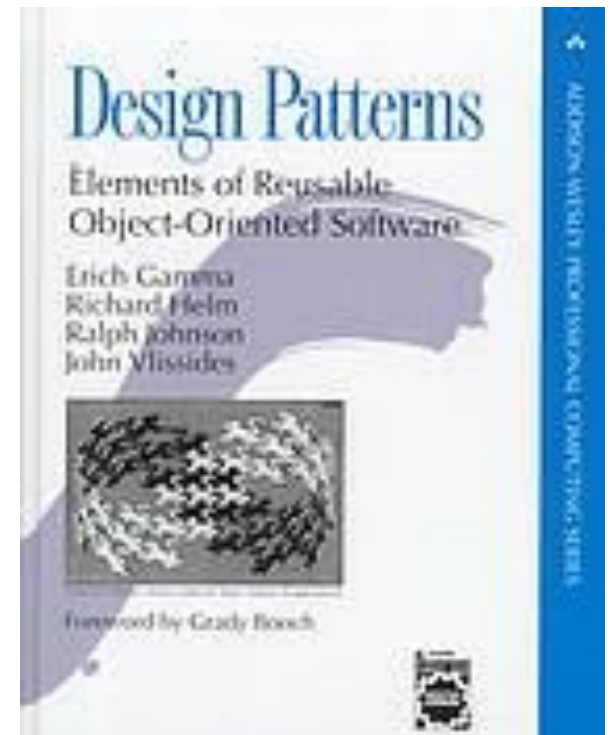
- Sabloane de proiectare (software design patterns)
 - singleton
 - Composite
 - Visitor
 - Object Factory

Sabloane de proiectare (Design Patterns)

- intai aplicate in proiectare urbanistica:
C. Alexander. A Pattern Language. 1977
- definitia originala a lui Alexander: *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"*
- prima contributie in software: 1987, Kent Beck (creatorul lui Extreme Programming) & Ward Cunningham (a scris primul wicki)

Aplicarea sabloanelor in POO

- contributia majora: Design Patterns:
Gamma et al. Elements of Reusable Object-Oriented Software was published, 1994
 - cunoscuta ca **GoF** (Gang of Four)
 - pot fi aplicate la nivel de clasa sau obiect
- GoF include 23 de sabloane



Formatul (template) complet al unui sablon

- nume si clasificare
- intentie
- cunoscut de asemenea ca
- motivatie
- aplicabilitate
- structura
- participanti
- colaborari
- consecinte
- implementare
- cod
- utilizari cunoscute
- sabloane cu care are legatura

Clasificarea sabloanelor GoF

- **creationale**
 - utilizate pentru crearea obiectelor
- **structurale**
 - utilizate pentru a defini structura (compunerea) claselor sau obiectelor
- **comportamentale**
 - descrie modul in care clasele si obiectele interactioneaza si isi distribuie responsabilitatile

POO

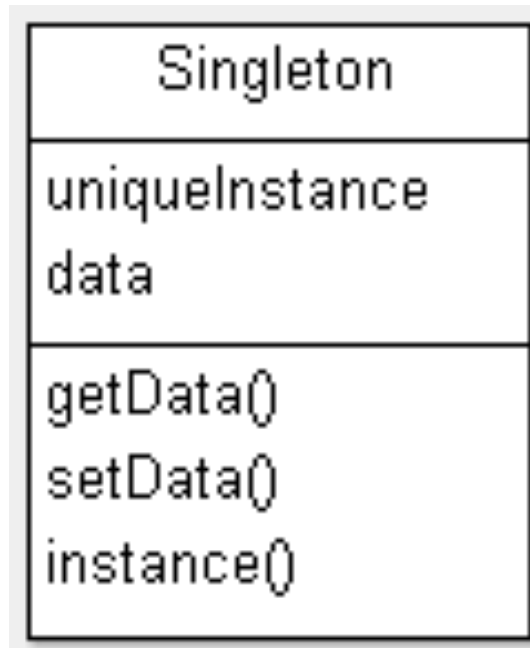
Singleton
(prezentare bazata pe GoF)

Singleton

- Clasificare
 - creational
- Intentia
 - proiectarea unei clase cu un singur obiect (o singura instanta)
- Motivatie
 - intr-un sistem de operare:
 - exista un sistem de fisiere
 - exista un singur manager de ferestre
 - intr-un sit web: exista un singur manager de pagini web
- Aplicabilitate
 - cand trebuie sa existe exact o instanta
 - clientii clasei trebuie sa aiba acces la instanta din orice punct bine definit

Singleton

- structura



- participant: Singleton
- collaborations: clients of the class

Clase cu o singura instanta (Singleton)

- Consecinte
 - acces controlat la instanta unica
 - reducerea spatiului de nume (eliminarea variab. globale)
 - permite rafinarea operatiilor si reprezentarii
 - permite un numar fix de instante
 - Doubleton
 - Tripleton
 - ...
 - mai flexibila decat operatiile la nivel de clasa (statice)
- Implementare

cum?

Clase cu o singura instanta

```
template <typename Data>
class Singleton {
public:
    static Singleton<Data>& instance() {
        return uniqueInstance;
    }
    Data getData() { return data; }
    void setData(Data x) { data = x; }
protected:
    Data data; // campurile care descriu starea instantei
    Singleton() { }
    void operator=(Singleton&);
    Singleton(const Singleton&);
private:
    static Singleton<Data> uniqueInstance;
};
```

manerul cu care se are acces la instanta

metode care opereaza peste instanta unica

constructor

operator atribuire

constructor de copiere

instanta unica

Clase cu o singura instanta

```
template <typename Data>
Singleton<Data> Singleton<Data>::uniqueInstance;

int main() {
    Singleton<int> &s1 = Singleton<int>::instance();
    cout << s1.getData() << endl; // 0
    Singleton<int> &s2 = Singleton<int>::instance();
    s2.setData(9);
    cout << s1.getData() << endl; // 9
    Singleton<int> &s3 = s2;
    cout << s3.getData() << endl; // 9
}
```

initializare

refera aceeași instanță
(pe cea unică)

Clase cu o singura instanta

- daca se comenteaza constructorul de copiere, atunci se poate executa urmatorul cod:

```
Singleton<int> s4 = s2;
```

```
s4.setData(23);
```

```
cout << s4.getData() << endl; // 23
```

```
cout << s2.getData() << endl; // 29
```

- operatorul de atribuire se declara privat doar pentru a evita operatii fara sens ca:

```
s4 = s2;
```

- in lipsa accesului la constructori, operatorul de atribuire nu poate fi utilizat pentru doua instante diferite

Demo

- interzicerea utilizarii unor functii/metode se poate face printr-o declaratie **delete**
- din manual (8.4.3):

“1 A function definition of the form:

attrib-specifier-seqopt decl-specifier-seqopt declarator = delete ;

is called a **deleted definition**. A function with a deleted definition is also called a deleted function.

2 A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed.

[Note: This includes calling the function implicitly or explicitly and forming a pointer or pointer-to-member to the function. It applies even for references in expressions that are not potentially-evaluated. If a function is overloaded, it is referenced only if the function is selected by overload resolution. — end note]”

C++ 2011

- inhibarea utilizarii constructorului de copiere si operatorului de atribuire

```
class Singleton {  
public:  
    ...  
    void operator=(Singleton&) = delete;  
    Singleton(const Singleton&) = delete;  
    ...  
private:  
    ...  
}
```


C++ 2011

- C++ 2011 include si constructorul de mutare
- din manual (12.8):

“3 A non-template constructor for class X is a move constructor if its first parameter is of type X&&, **const** X&&, **volatile** X&&, or **const volatile** X&&, and either there are no other parameters or else all other parameters have default arguments (8.3.6).

(continuate pe slide-ul urmator)

C++ 2011

[Example: `Y::Y(Y&&)` is a move constructor.

```
struct Y {  
    Y(const Y&);  
    Y(Y&&);  
};  
extern Y f(int);  
Y d(f(1)); // calls Y(Y&&)  
Y e = d; // calls Y(const Y&)  
— end example ]"
```

- Ar trebui ascuns si constructorul de mutare?

C++ 2011:

- Nu intotdeauna
- Din manual (12.8):

“10 If the definition of a class X does not explicitly declare a move constructor, one will be implicitly declared as defaulted if and only if

- X does not have a user-declared copy constructor,
- X does not have a user-declared copy assignment operator,
- X does not have a user-declared move assignment operator,
- X does not have a user-declared destructor, and
- the move constructor would not be implicitly defined as deleted.”

Instanta unica dinamica 1/2

```
template <typename Data>
class Singleton {
public:
    static Singleton* instance() {
        if (uniqueInstance == 0) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
    // metode care opereaza peste instanta
    // unica
    Data getData() { return data; }
    void setData(Data x) { data = x; }
```

Dynamic unique instance 2/2

protected:

```
Data data;
```

```
Singleton() { }
```

```
// void operator=(Singleton&);
```

```
// Singleton(const Singleton&);
```

Nu mai e necesar
de ascuns
De ce?

private:

```
static Singleton<Data>* uniqueInstance;
```

```
};
```

Pointer la
instanta unica

Demo

```
Singleton<int>* s1 = Singleton<int>::instance();  
s1->setData(47);  
cout << s1->getData() << endl; // 47  
Singleton<int>* s2 = Singleton<int>::instance();  
s2->setData(9);  
cout << s1->getData() << endl; // 9  
cout << s2->getData() << endl; // 9
```

Diferenta dintre pointer si referinta 1/2

- urmatoarele instructiuni se executa, indiferent cum declaram constructorul de copiere si/sau operatorul de atribuire

```
Singleton<int>* s4 = s2;
```

```
s4->setData(23);
```

```
cout << s4->getData() << endl;
```

```
cout << s2->getData() << endl;
```

```
s4 = s1;
```

```
s4->setData(43);
```

```
cout << s4->getData() << endl;
```

```
cout << s2->getData() << endl; de ce?
```

Diferenta dintre pointer si referinta 2/2

- dar urmatoarea instructiune nu se compileaza daca se decommenteaza constructorul de copiere

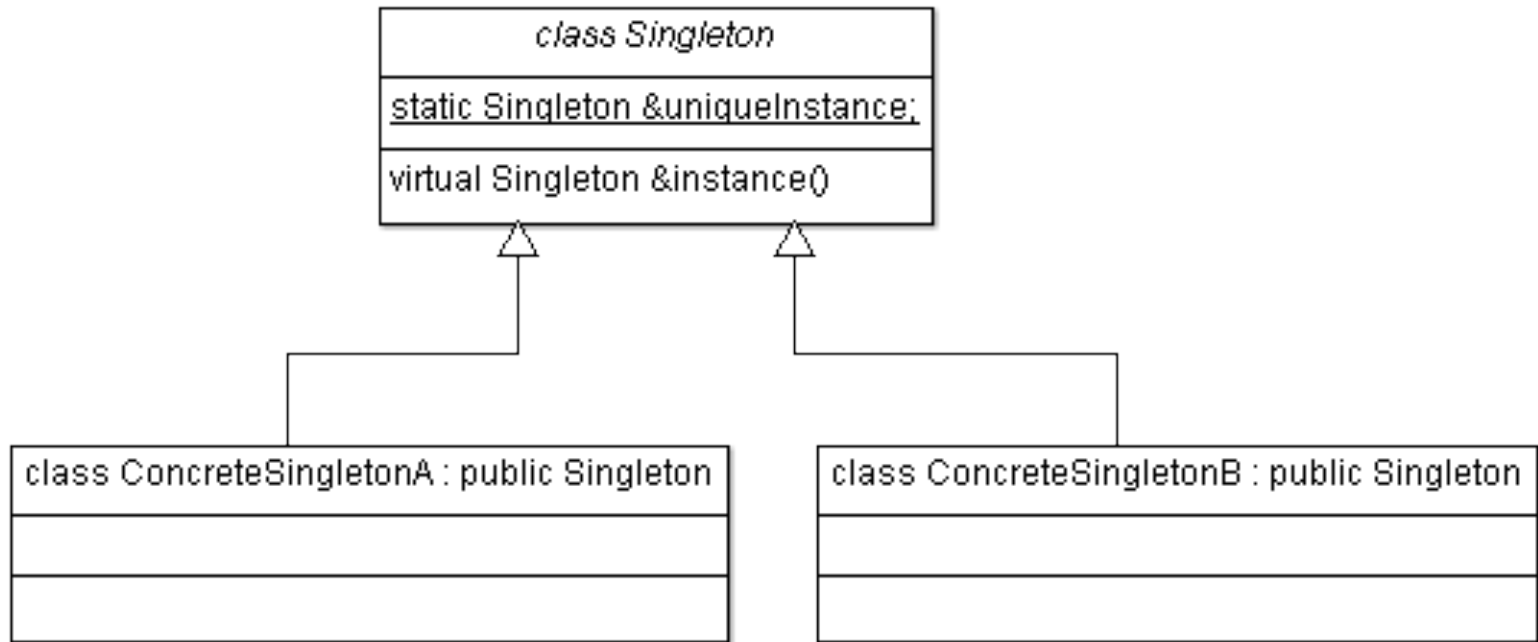
```
Singleton s5 = (*s2);
```

- si urmatoarea instructiune nu se compileaza daca se decommenteaza si operatorul de atribuire

```
s5 = *s1;
```

Demo

Clase singleton derivate



- pot fi probleme la crearea instantelor claselor concrete din ierarhie => repository de clase Singleton

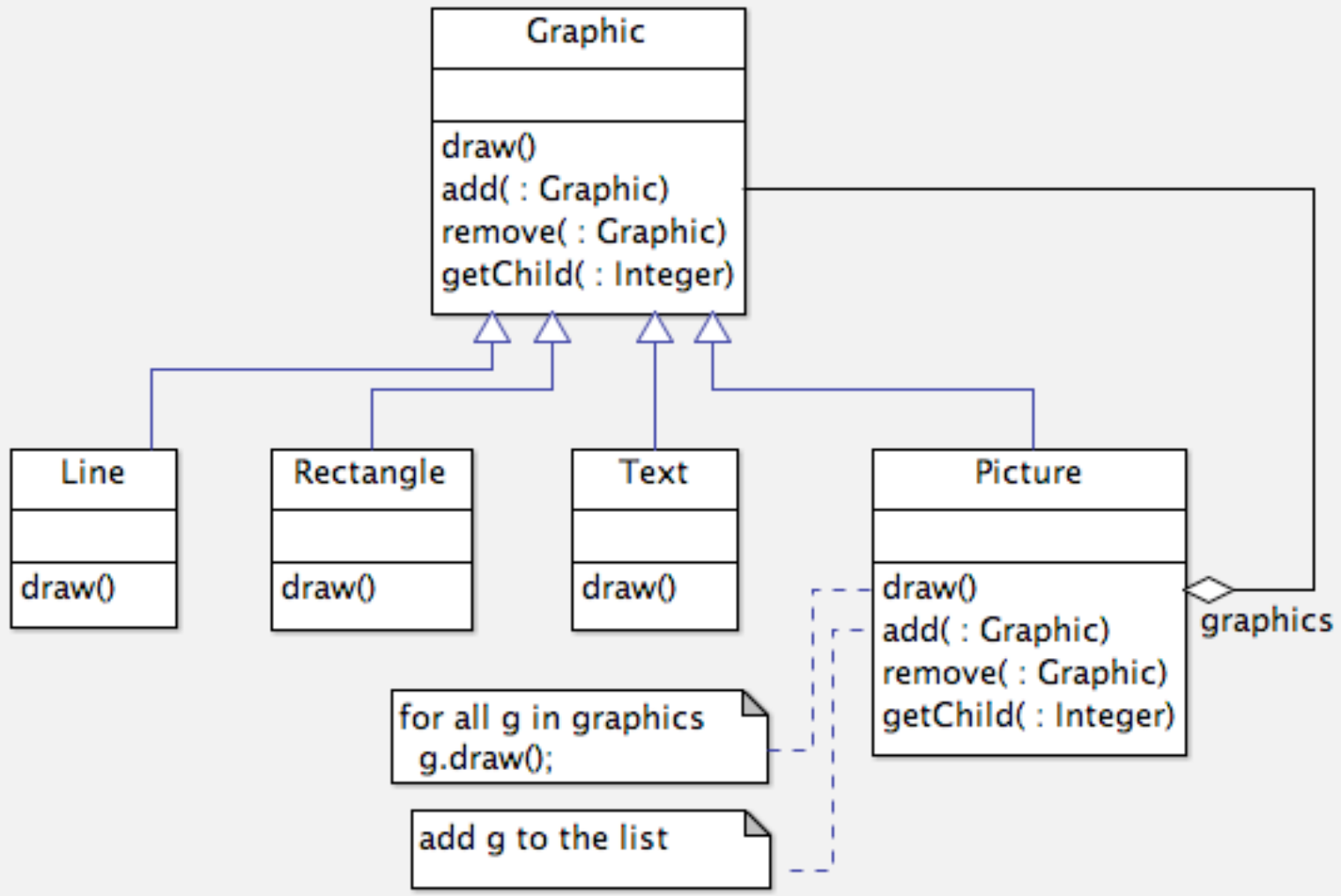
POO

Composite
(prezentare bazata pe GoF)

Composite::intentie

- este un pattern structural
- compune obiectele intr-o structura arborescenta pentru a reprezenta o ierarhie parte-intreg
- lasa clientii (structurii) sa trateze obiectele individuale si compuse intr-un mod uniform

Composite:: motivatie



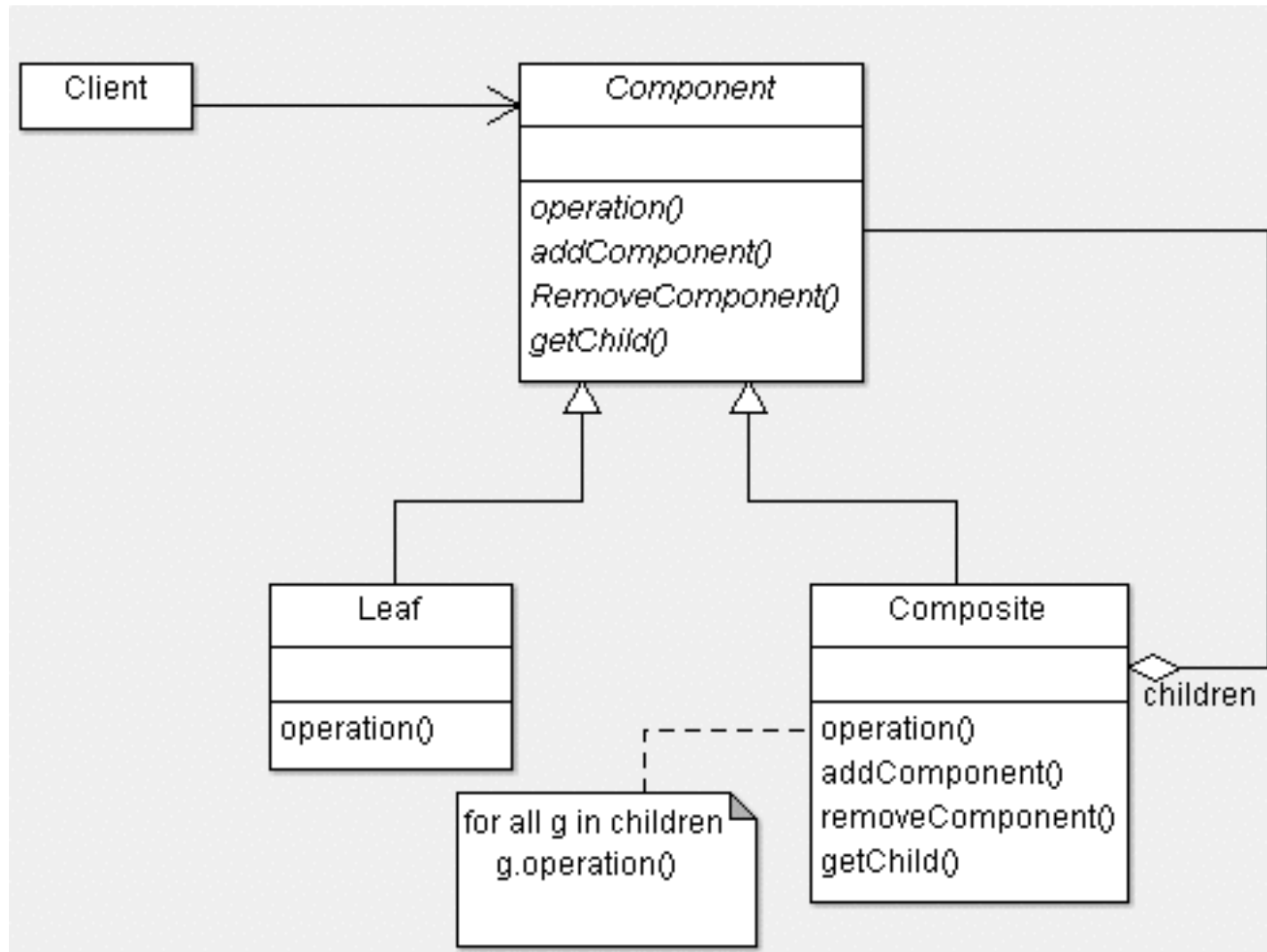
Composite:: caracterul recursiv al str.

- orice (obiect) linie este un obiect grafic
- orice (obiect) dreptunghi este un obiect grafic
- orice (obiect) text este un un obiect grafic
- o pictura formata din mai multe obiecte grafice este un obiect grafic

Composite::aplicabilitate

- pentru a reprezenta ierarhii parte-intreg
- clientii (structurii) sa poata ignora diferentele dintre obiectele individuale si cele compuse
- obiectele structurii posibil sa fie tratate uniform (cu un anumit pret)

Composite::structura



Composite::participanti

- **Component (Graphic)**
 - declara interfata pentru obiectele din compozitie
 - implementeaza comportarea implicita pentru interfata comuna a tuturor claselor
 - declara o interfata pentru accesarea si managementul componentelor-copii
 - (optional) defineste o interfata pentru accesarea componentelor-parinte in structura recursiva
- **Leaf (Rectangle, Line, Text, etc.)**
 - reprezinta obiectele primitive; o frunza nu are copii
 - defineste comportarea obiectelor primitive

Composite::participanti

- **Composite (Picture)**
 - defineste comportarea componentelor cu copii
 - memoreaza componentele-copil
 - implementeaza operatiile relative la copii din interfata *Component*
- **Client**
 - manipuleaza obiectele din compozitie prin intermediul interfetei *Component*

Composite::colaborari

- clientii utilizeaza clasa de interfata *Component* pentru a interactiona cu obiectele din structura
- daca recipientul este o instanta *Leaf*, atunci cererea este rezolvata direct
- daca recipientul este o instanta *Composite*, atunci cererea este transmisa mai departe componentelor-copil; alte operatii aditionale sunt posibile inainte sau dupa transmitere

Composite::consecinte

- definește o ierarhie de clase constând din obiecte primitive și compuse
- obiectele primitive pot fi compuse în obiecte mai complexe, care la rândul lor pot fi compuse în alte obiecte mai complexe și așa mai departe (recursiv)
- ori de câte ori un client așteaptă un obiect primitiv, el poate lua de asemenea și un obiect compus
- pentru client este foarte simplu; el tratează obiectele primitive și compuse în mod uniform
- clientului nu-i pasă dacă are de-a face cu un obiect primitiv sau compus (evitarea utilizării structurilor de tip *switch-case*)

Composite:: consecinte

- este usor de adaugat noi tipuri de componente *Leaf* sau *Composite*; noile subclase functioneaza automat cu structura existenta si codul clientului. Clientul nu schimba nimic.
- face designul foarte general
- dezavantaj: e dificil de restrictionat ce componente pot sa apara intr-un obiect compus (o solutie ar putea fi verificarea in timpul executiei)

Composite::implementare

- *Referinte explicite la parinte.*
 - simplifica traversarea si managementul structurii arborescente
 - permite travesarea bottom-up si stergerea unei componente
 - referinta parinte se pune in clasa *Component*
 - usureaza mentinerea urmatorului invariant: parintele unui copil este un obiect compus si-l are pe acesta ca si copil (metodele `add()` si `remove()` sunt scrise o singura data si apoi mostenite)

Composite::implementare

- *Componente partajate.*
 - cateodata este util sa partajam componente
 - ... dar daca o componenta are mai mult decat un parinte, atunci managementul devine dificil
 - o solutie posibila: parinti multipli (?)
 - exista alte patternuri care se ocupa de astfel de probleme ([Flyweight](#))

Composite::implementare

- *Maximizarea interfetei Component*
 - *Component* ar trebui sa implementeze cat mai multe operatii comune (pt *Leaf* si *Composite*)
 - aceste operatii vor descrie comportarea implicita si pot fi rescrise de *Leaf* si *Composite* (sau subclasele lor)
 - totusi aceasta incalca principiul “o clasa trebuie sa implementeze numai ce are sens pentru subclase”; unele operatii pentru *Composite* nu au sens pt. *Leaf* (sau invers)
 - de ex. *getChild()*
 - solutie: comportarea default = nu intoarce niciodata vreun copil

Composite:: implementare

- *Operatiile de management a copiilor* (*cele mai dificile*)
 - unde le declaram?
 - daca le declaram in *Component*, atunci avem *transparenta* (datorita uniformitatii) dare ne costa la siguranta (safety) deoarece clientii pot incerca op fara sens (ex. eliminarea copiilor unei frunze)
 - daca le declaram in *Composite*, atunci avem *siguranta* dar nu mai avem transparenta (avem interfete diferite pt comp. primitive si compuse)
 - patternul opteaza pentru transparenta

Composite:: implementare

- ce se intampla daca optam pentru siguranta?
- se pierde informatia despre tip si trebuie convertita o instanta *Component* intr-o instanta *Composite*
- cum se poate face?
- o posibila solutie: declara o operatie
 Composite getComposite()*
 in clasa *Component*
- *Component* furnizeaza comportarea implicita intorcand un pointer NULL
- *Composite* rafineaza operatia intorcandu-se pe sine insasi prin intermediul pointerului *this*

Implementarea metodei getComposite()

```
class Composite;
```

problema de tip “oul sau gaina”

```
class Component {
```

```
public:
```

```
    //...
```

```
    virtual Composite* getComposite() { return 0; }
```

```
};
```

```
class Composite : public Component {
```

```
public:
```

```
    void Add(Component*);
```

```
    // ...
```

```
    virtual Composite* getComposite() { return this; }
```

```
};
```

```
class Leaf : public Component {
```

```
    // ...
```

```
};
```

implementarea implicita

implementarea pt. un compus

Exemplu utilizare a metodei getComposite()

```
Composite* aComposite = new Composite;  
Leaf* aLeaf = new Leaf;  
Component* aComponent;  
Composite* test;  
cin >> option;  
if (option == 1)  
    aComponent = aComposite;  
else  
    aComponent = aLeaf;  
if (test = aComponent->getComposite()) {  
    test->Add(new Leaf);  
}
```

crearea unui obiect compus si a unei frunze

option == 1:
adauga, pentru ca test va fi diferit de zero
option != 1:
NU adauga, pentru ca test va fi zero

Composite:: implementare

- evident, componentele nu sunt tratate uniform
- singura posibilitate de a avea transparenta este includerea operatiile relativ la copii in *Component*
- este imposibil de a implementa *Component:add()* fara a intoarce o exceptie (esec)
- ar fi ok sa nu intoarca nimic?
- ce se poate spune despre *Component:remove()* ?

Composite:: implementare

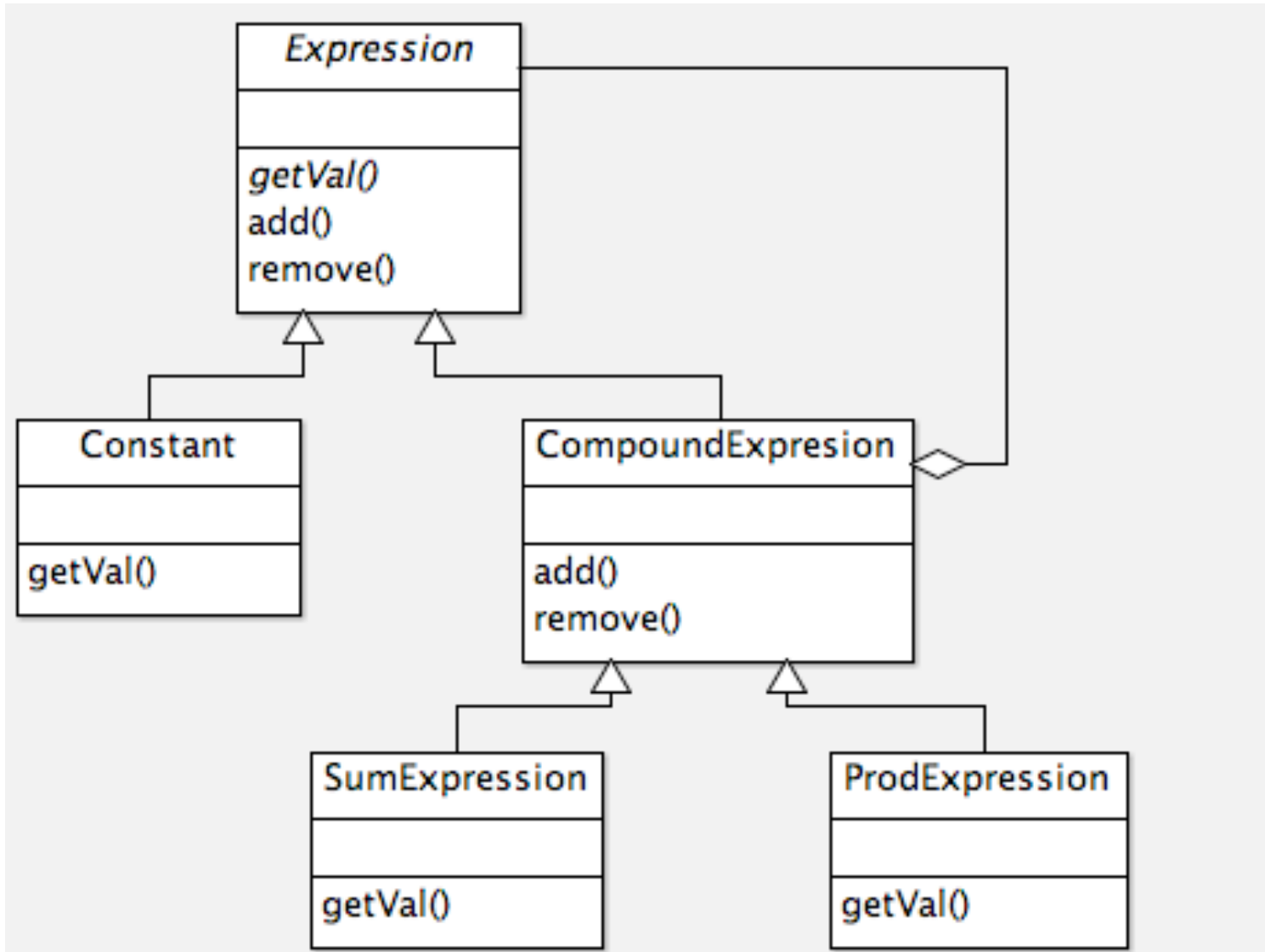
- *Ar trebui implementata o lista de fii in Component?*
 - ar fi tentant
 - dar ...
 - ... ar fi irosire de spatiu
- *Ordinea copiilor*
 - sunt aplicatii in care conteaza
 - daca da, atunci accesul si managementul copiilor trebuie facut cu grija
- *Cine sterge componentele?*
 - fara GC, responsabilitatea este a lui *Composite*
 - atentie la componentele partajate
- *Care structura e cea mai potrivita pentru lista copiilor?*

Studiu de caz

Problema

- expresii
 - orice numar intreg este o expresie
 - daca e_1, e_2, e_3, \dots sunt expresii atunci suma lor $e_1 + e_2 + e_3 + \dots$ este expresie
 - daca e_1, e_2, e_3, \dots sunt expresii atunci produsul lor $e_1 * e_2 * e_3 * \dots$ este expresie

Expresii : : structura



Interfata

```
class Expression
{
public:
    virtual int getVal() = 0;
    virtual void add(Expression* exp) = 0;
    virtual void remove() = 0;
};
```

Constant (Leaf)

```
class Constant : public Expression
{
public:
    Constant(int x = 0) { val = x; }
    int getVal() { return val; }
    void add(Expression*) {}
    void remove() {}
private:
    int val;
};
```



implementare vida

Expresie compusa

```
class CompoundExpression : public Expression
{
public:
    void add(Expression* exp)
    {
        members.push_back(exp);
    }
    void remove()
    {
        members.erase(members.end());
    }
protected:
    list<Expression*> members;
};
```

indicele/referinta componentei care se
sterge ar putea fi data ca parametru

listele din STL

Expresie de tip suma

```
class SumExpression : public CompoundExpression
{
public:
    SumExpression() {}
    int getVal()
    {
        list<Expression*>::iterator i;
        int valTemp = 0;
        for ( i = members.begin();
              i != members.end(); ++i)
            valTemp += (*i)->getVal();
        return valTemp;
    }
};
```

declarare iterator pentru parcurgere componente

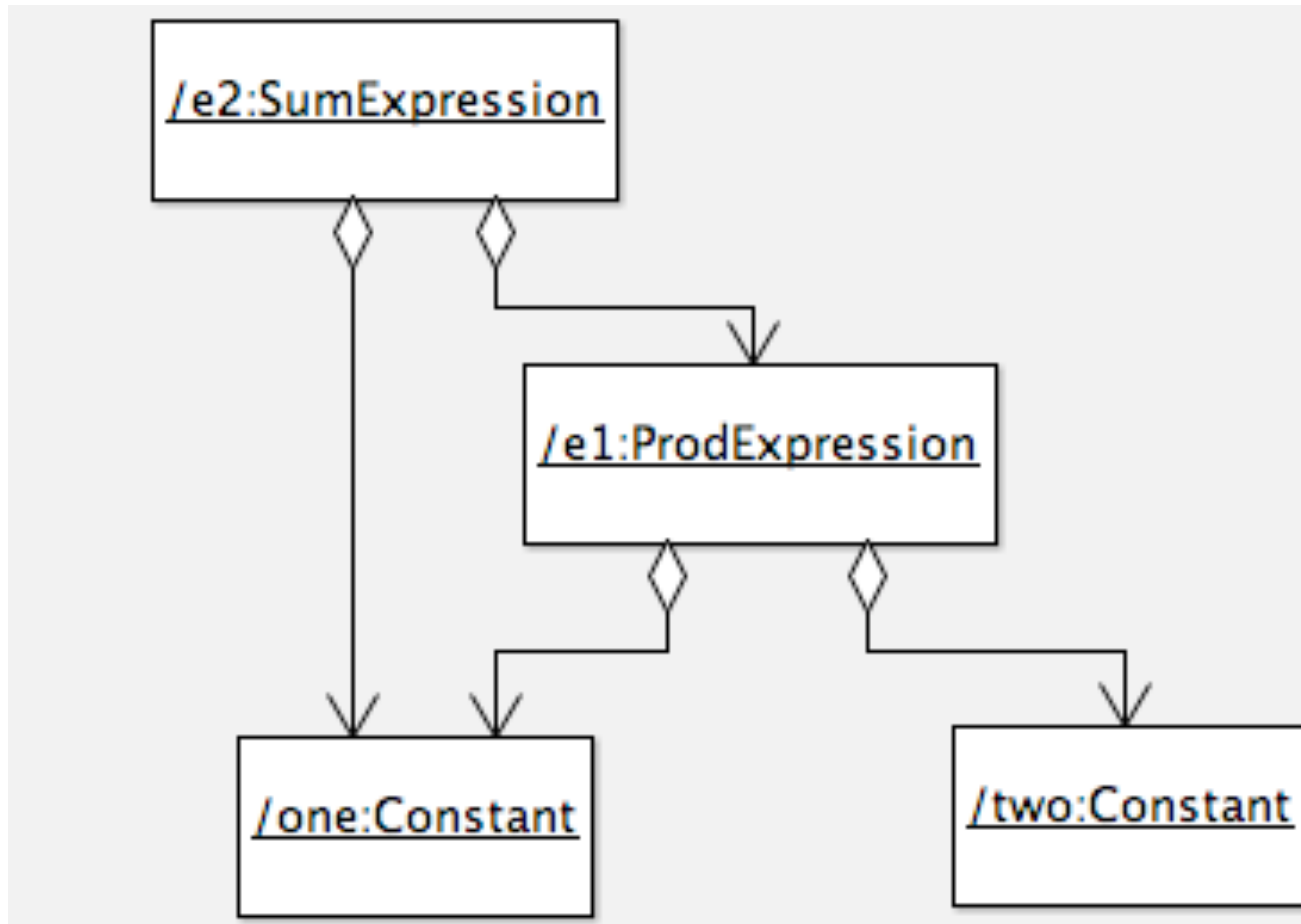
valoarea unei expresii suma este suma valorilor componentelor

Expresie de tip produs



Demo 1/2

$1 + 1 * 2$



Demo 2/2

```
Constant* one = new Constant(1);  
Constant* two = new Constant(2);
```

creare doua
constante

```
ProdExpression* e1 = new ProdExpression();  
e1->add(one);  
e1->add(two);  
cout << e1->getVal() << endl;
```

creare expresie
compusa produs

```
SumExpression* e2 = new SumExpression();  
e2->add(e1);  
e2->add(two);  
cout << e2->getVal() << endl;
```

creare expresie
compusa suma

POO

Sablonul

Visitor

(prezentare bazata pe GoF)

Intentie

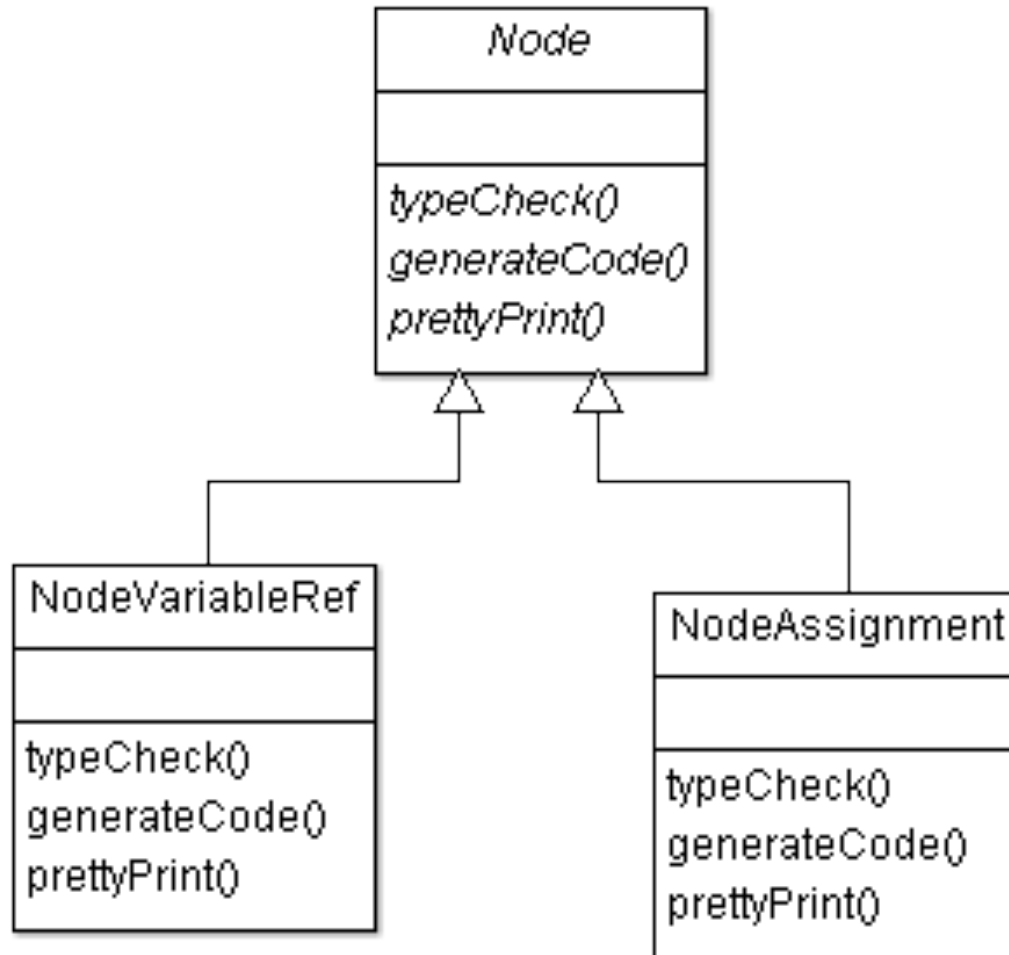
- reprezinta o operatie care se executa peste elementele unei structuri de obiecte
- permite sa definirea de noi operatii fara a schimba clasele elementelor peste care lucreaza

Motivatie

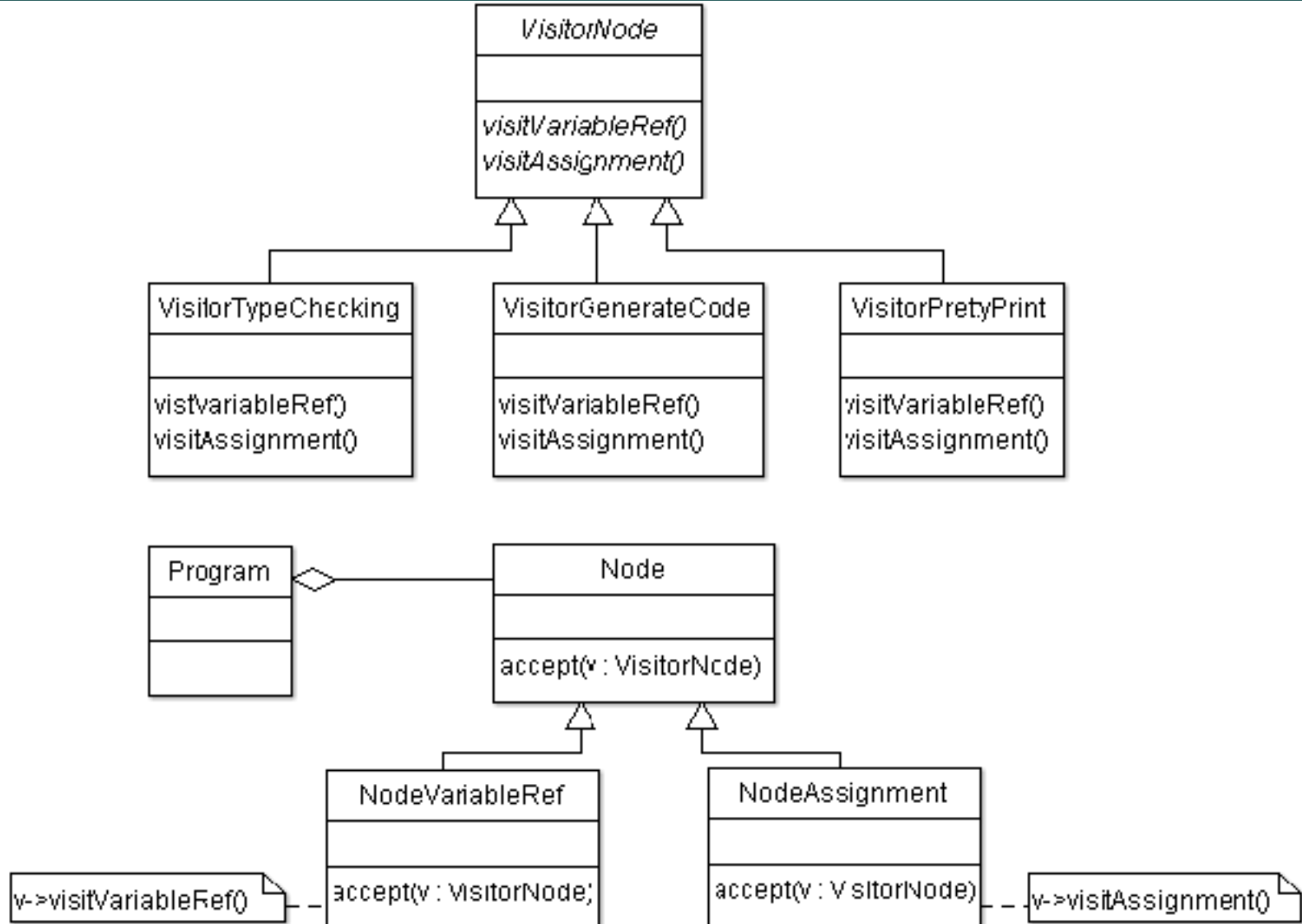
- Un compilator reprezinta un program ca un arbore sintactic abstract (AST). Acest arbore sintactic este utilizat atat pentru semantica statica (e.g., verificarea tipurilor) cat si pentru generarea de cod, optimizare de cod, afisare.
- Aceste operatii difera de la un tip de instructiune la altul. De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



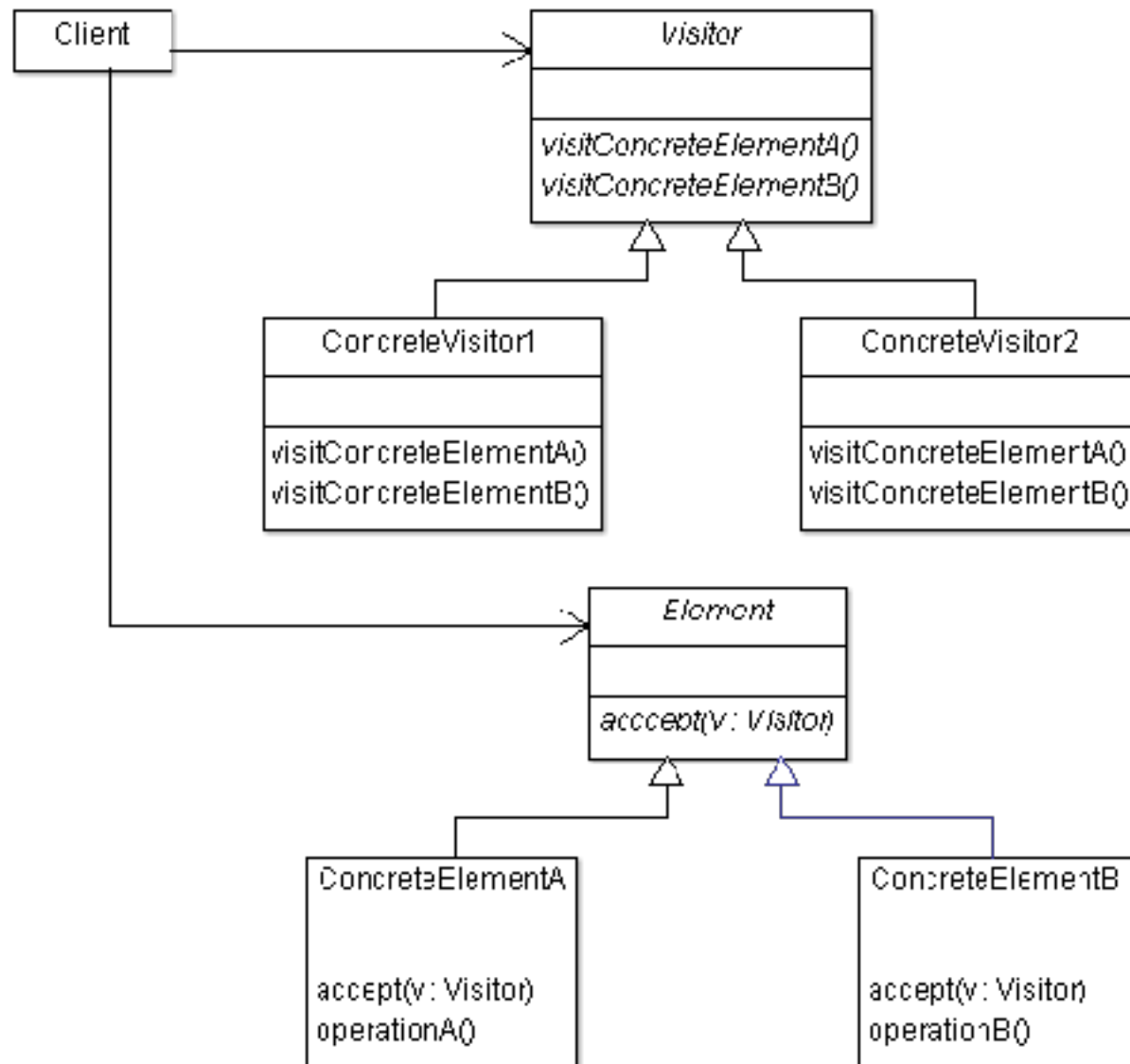
Solutia cu vizitatori



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



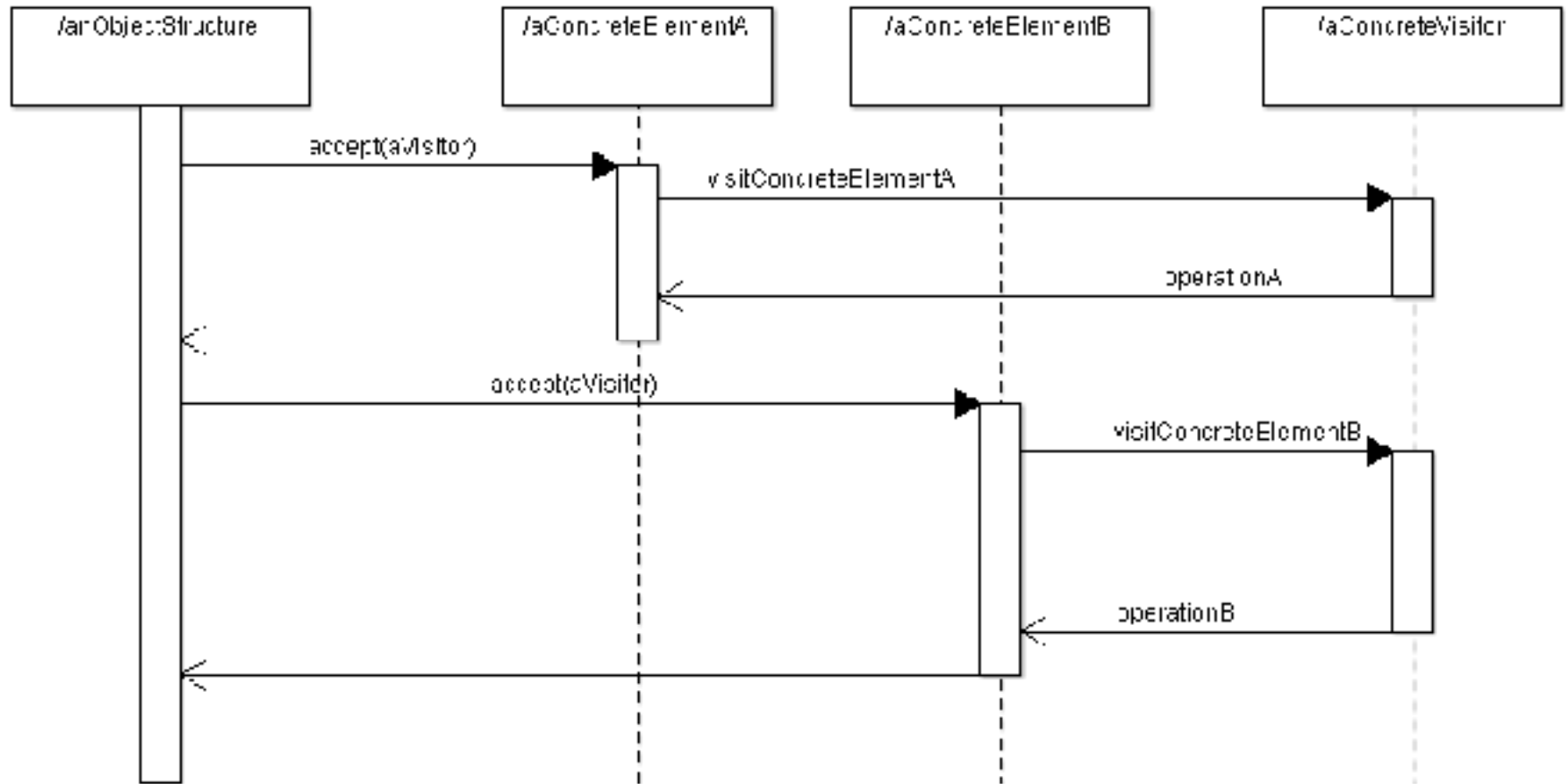
Participanti 1/2

- **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() ;  
};
```

usual abstract methods

abstract class

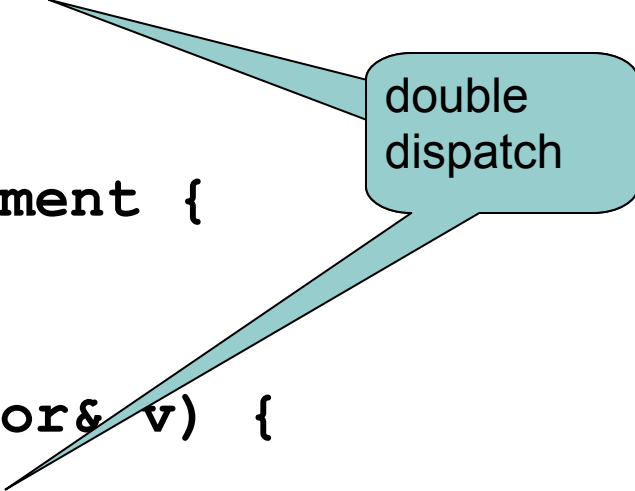
abstract method

abstract class

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);
    }
};
```



double
dispatch

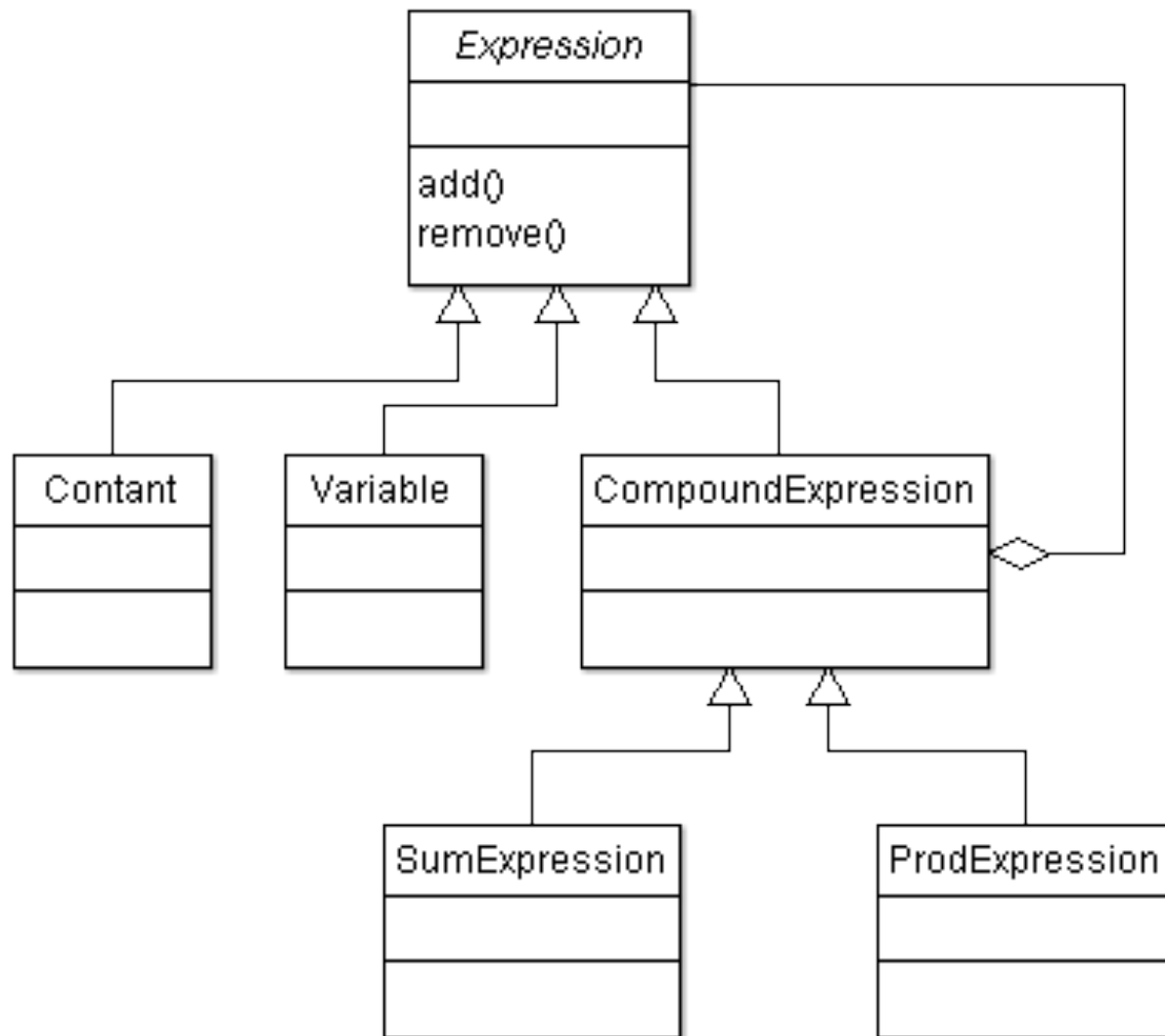
Implementare 3/3

- *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

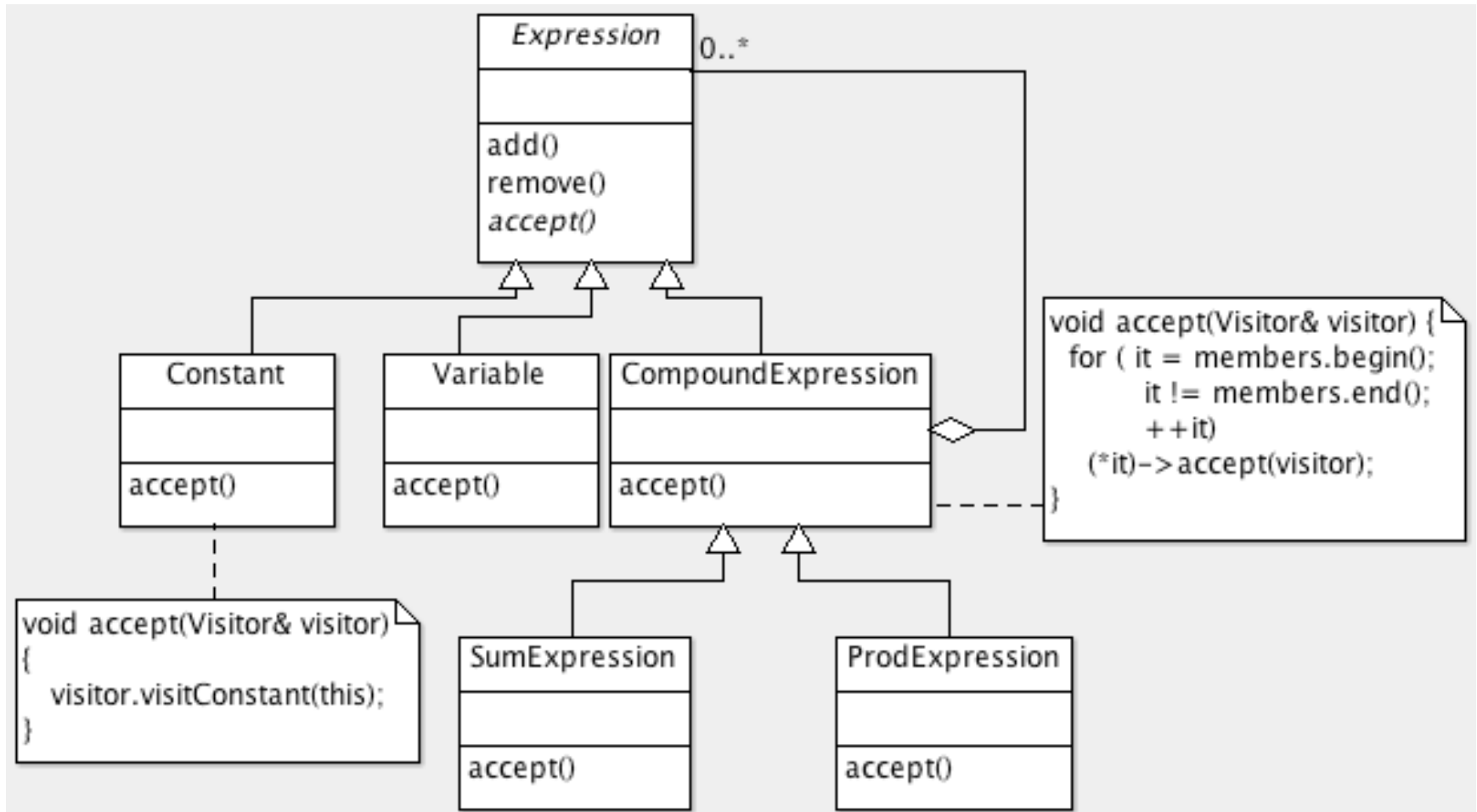
Aplicatie

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

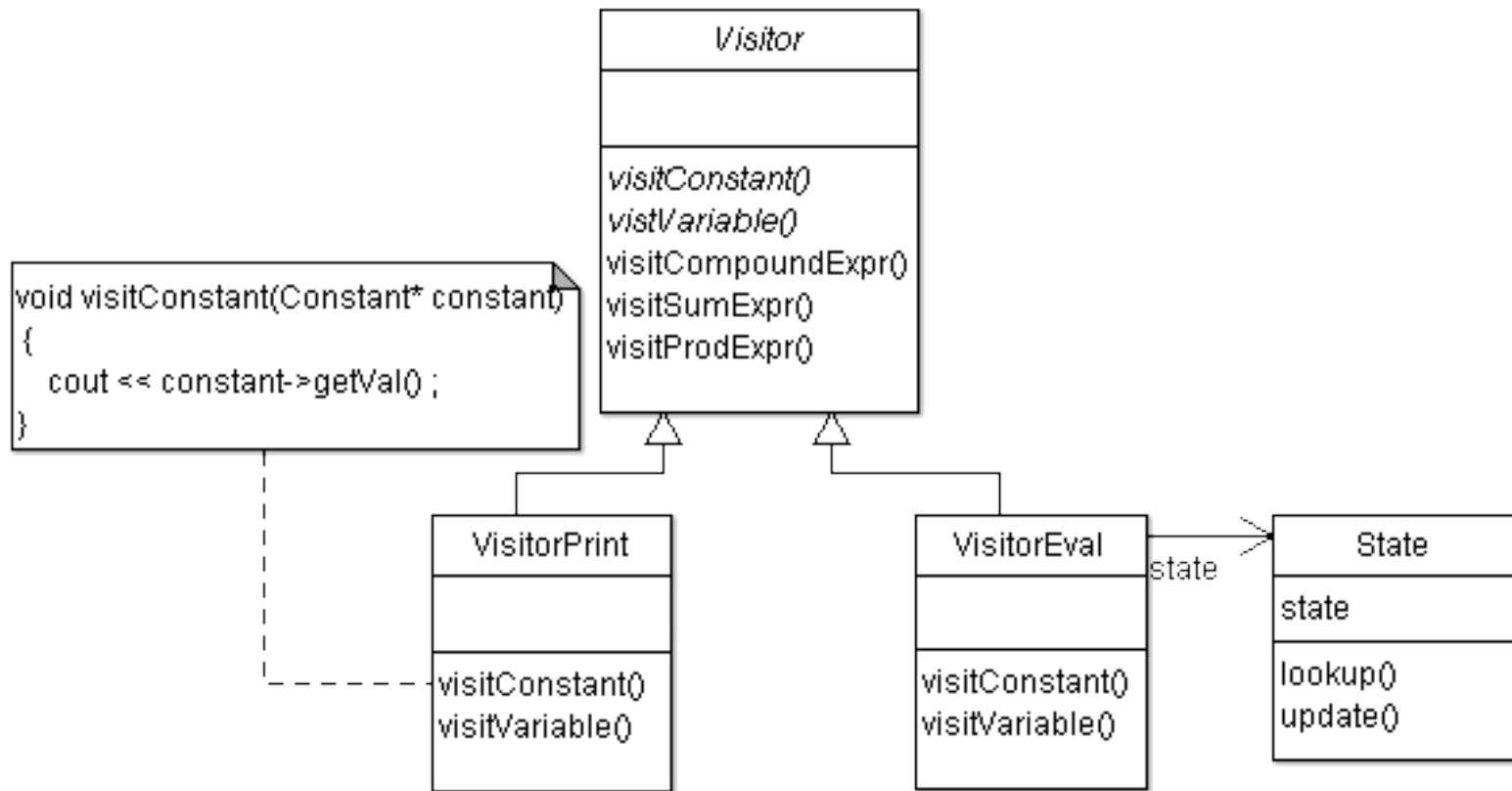
Expresii



Adaugarea operatiei accept()



Vizitatori pentru expresii



Evaluarea

- reamintim ca vizitatorii pentru expresii compuse sunt apelati in post-ordine
- asta este OK pentru evaluare, unde subexpresiile trebui evaluate mai intai
- vom considera o stare cumulativa format dintr-o stiva
- fiecare sub-expresie va pune valoarea sa in stiva
- o (sub-)expresie compusa va gasi valorile sub-expresiilor sale in partea de sus a stivei
 - vizitarea va consta in adunare/inmultirea ... acestor valori

VisitorEval 1/2

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

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

VisitorEval 2/2

```
void visitProdExpression
    (ProdExpression* prod) {
    int temp = 1;
    for (int i = 0; i < prod->size(); ++i) {
        temp *= valStack.top();
        valStack.pop();
    }
    valStack.push(temp);
}

int getCumulateVal() {
    return valStack.top();
}

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

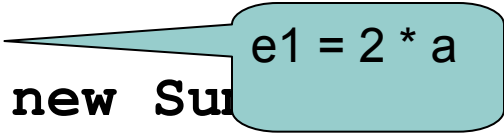
valorile sub-
expresiilor se gasesc
la inceputul stivei

cumulative visitor's
state

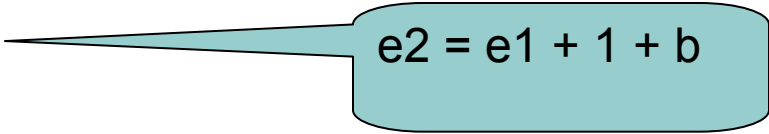
Clientul 1/2

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



$e1 = 2 * a$



$e2 = e1 + 1 + b$

Clientul 2/2

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

2 a *

scriere postfixata (de ce?)

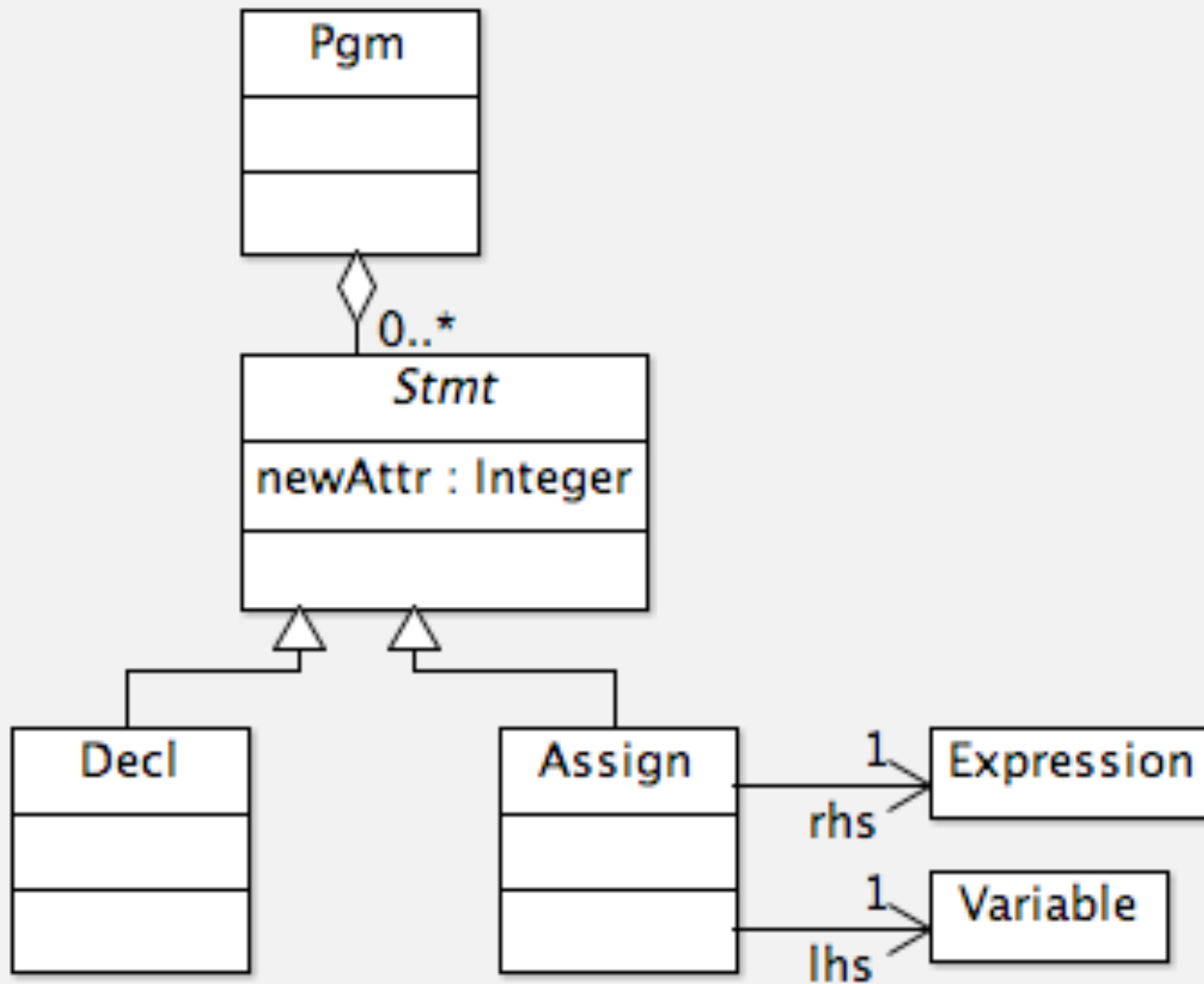
```
State st;  
st.update("a", 10);
```

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

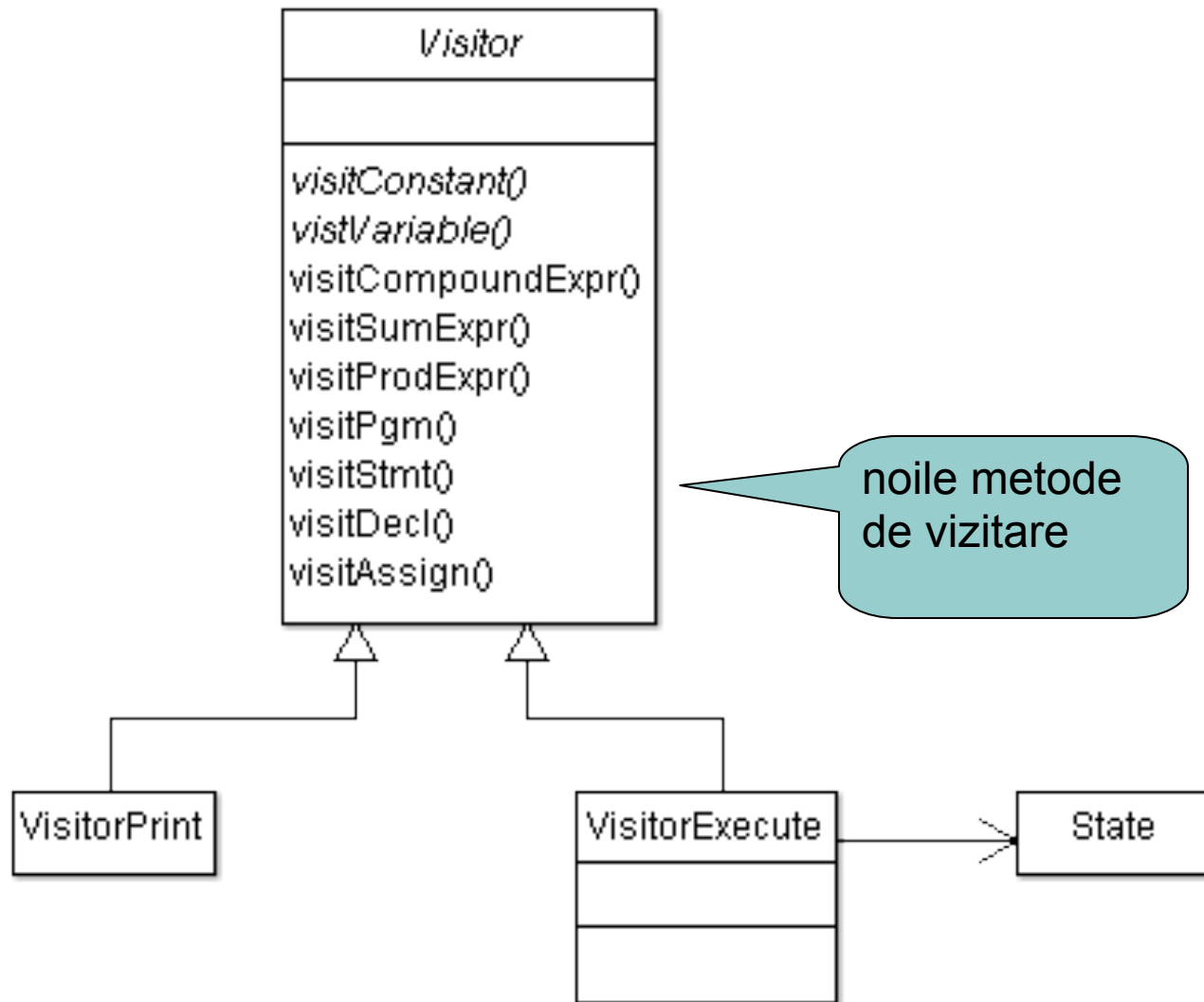
```
VisitorEval visitorEval1(&st);  
e1->accept(visitorEval1);  
cout << "e1 = "  
      << visitorEval1.getCumulateVal()  
      << endl;
```

e1 = 20

Programe

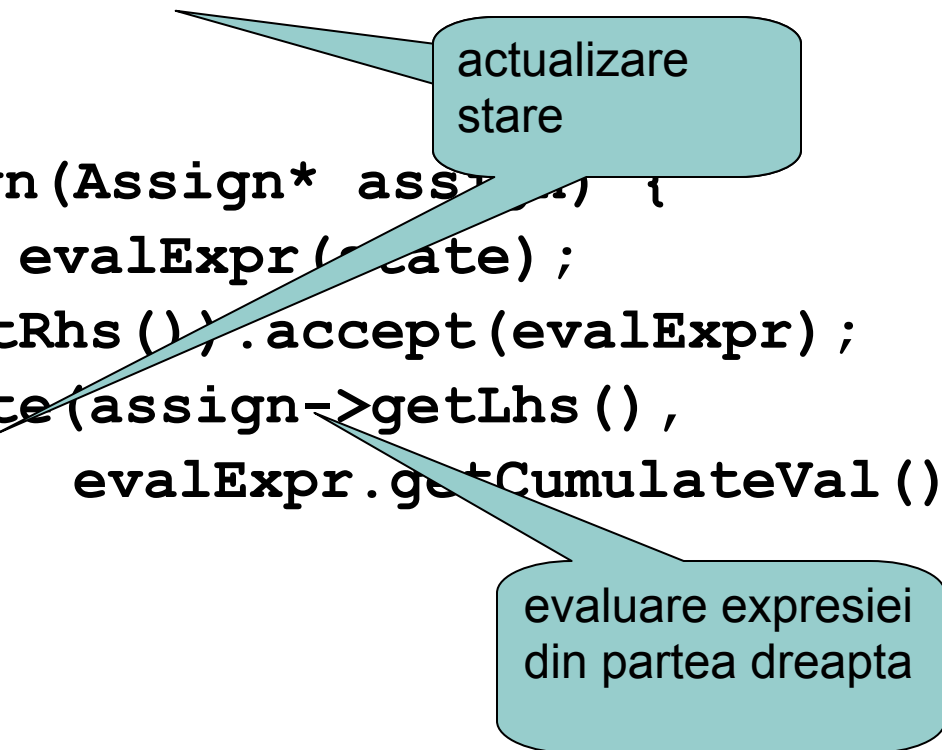


Vizitator pentru programe



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(state);
        (assign->getRhs()).accept(evalExpr);
        state->update(assign->getLhs(),
                      evalExpr.getCumulateVal());
    }
}
```



actualizare
stare

evaluare expresiei
din partea dreapta

VisitorExec 2/2

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

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

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

```
private:
```

```
    State *state;
```

```
};
```

Clientul 1/3

```
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 2/3

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

```
VisitorPrint:  
int a;  
int b;  
a = 2 a * ;  
b = 2 a * 1 b + ;
```

Clientul 2/3

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

```
a |-> 0  
b |-> 1
```

POO

Sablonul
Object Factory

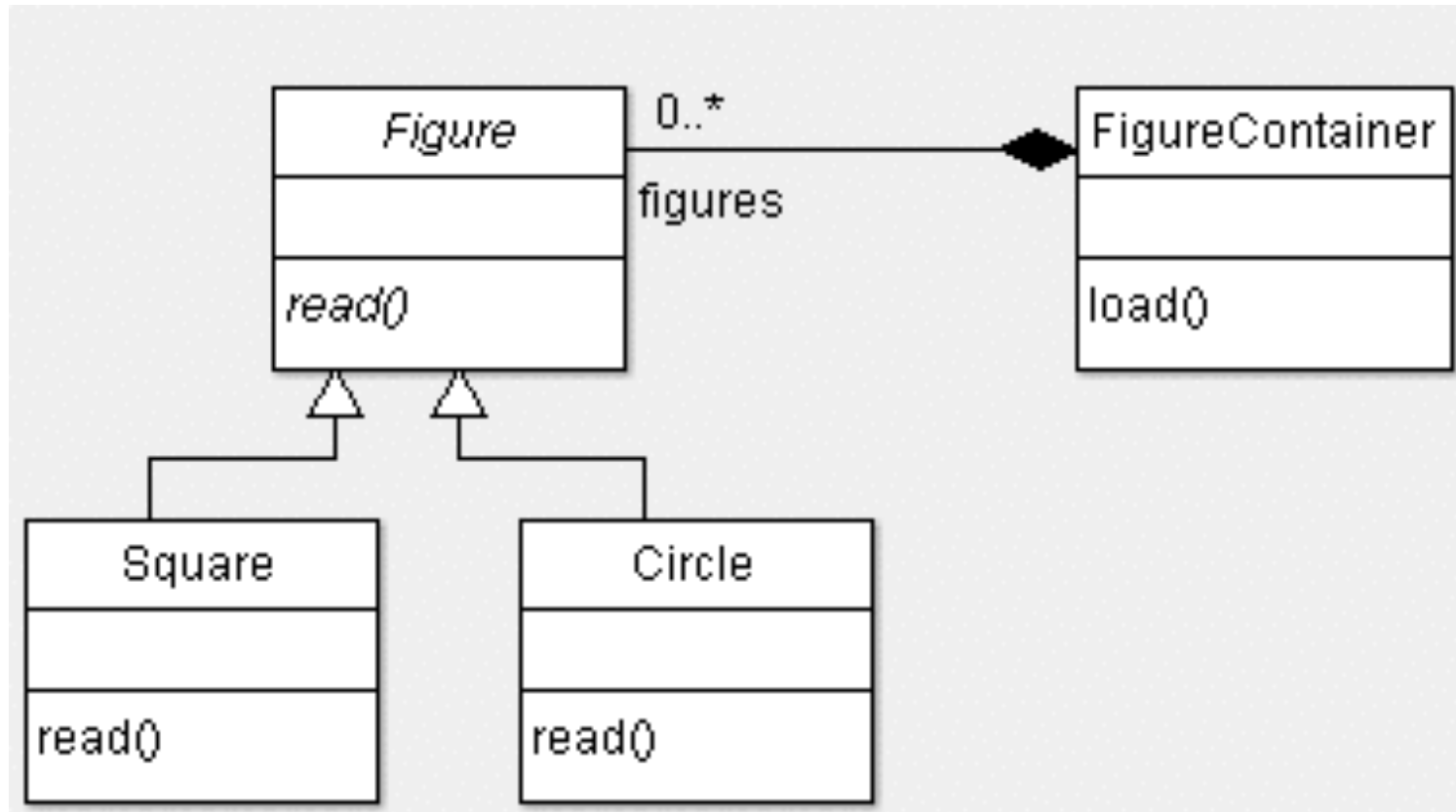
Cuprins

- principiul inchis-deschis
- fabrica de obiecte (Abstract Object Factory)
(prezentare bazata pe GoF)
- studii de caz:
 - *expression factory*

Principiul “inchis-deschis”

- “Entitatile software (module, clase, functii etc.) trebuie sa fie **deschise la extensii** si **inchise la modificare**” (Bertrand Meyer, 1988)
- “deschis la extensii” = comportarea modulului poate fi extinsa pentru a satisface noile cerinte
- “inchis la modificare” = nu este permisa modificarea codului sursa

Principiul “inchis-deschis” : exemplu



Principiul “inchis-deschis”: neconformare

```
void FigureContainer::load(std::ifstream& inp)
{
    while (inp)
    {
        int tag;
        Figura* pfig;
        inp >> tag;
        switch (tag)
        {
            case SQUAREID:
                ...
            case CIRCLEID:
                ...
        }
    }
}
```

eticheta figura

citeste tipul figurii ce urmeaza a fi incarcate

adaugarea unui nou tip de figura presupune modificarea acestui cod

`pfig = new Square;`
`pfig.read(inp);` `Square::read()`

`pfig = new Circle;`
`pfig.read(inp);` `Circle::read()`

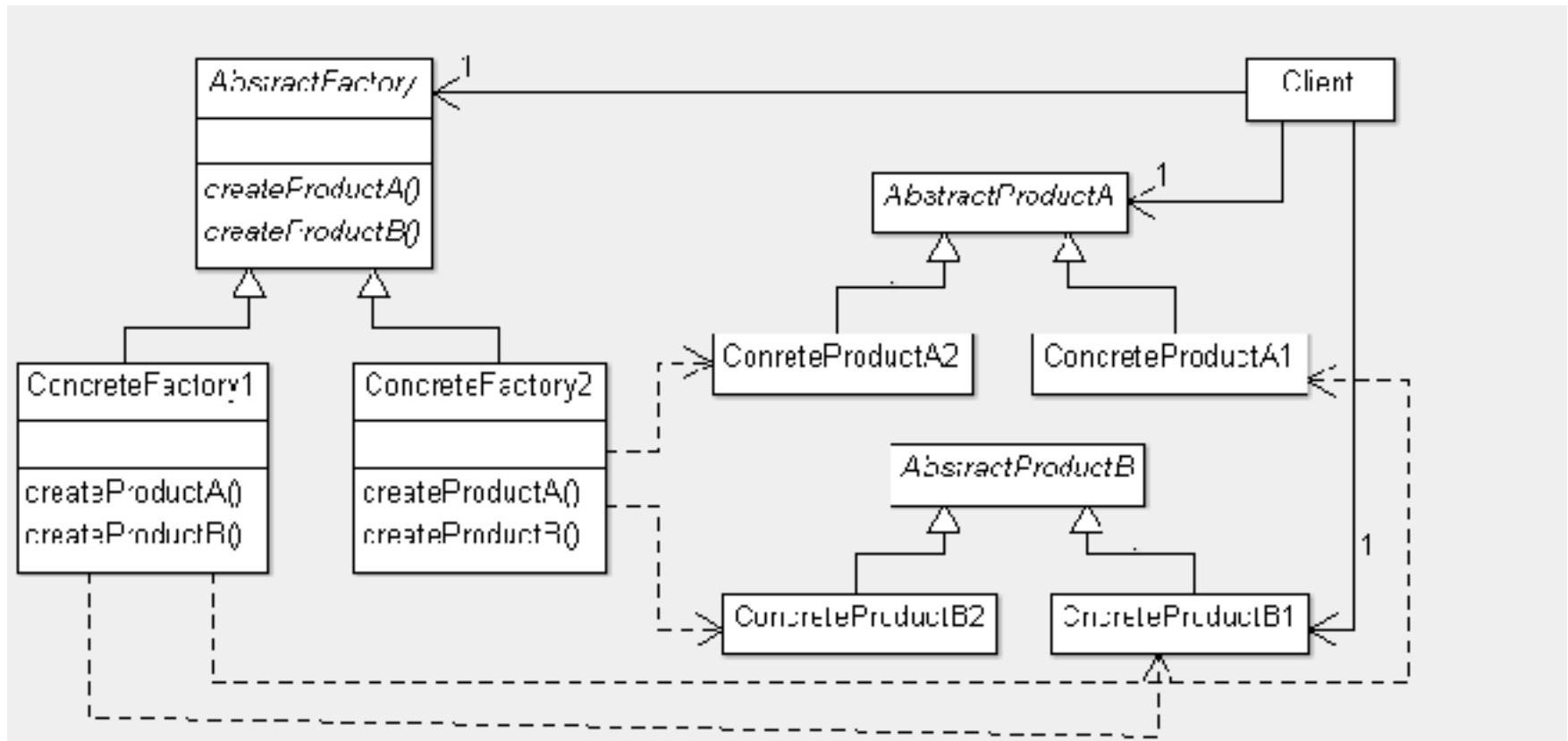
Principiul “inchis-deschis”

O posibila solutie:
fabrica de obiecte

Fabrica de obiecte (Abstract Factory)

- intentie
 - de a furniza o interfata pentru crearea unei familii de obiecte intercorelate sau dependente fara a specifica clasa lor concreta
- aplicabilitate
 - un sistem ar trebui sa fie independent de modul in care sunt create produsele, compuse sau reprezentate
 - un sistem ar urma sa fie configurat cu familii multiple de produse
 - o familie de obiecte intercorelate este proiectata astfel ca obiectele sa fie utilizate impreuna
 - vrei sa furnizezi o biblioteca de produse si vrei sa fie accesibila numai interfata, nu si implementarea

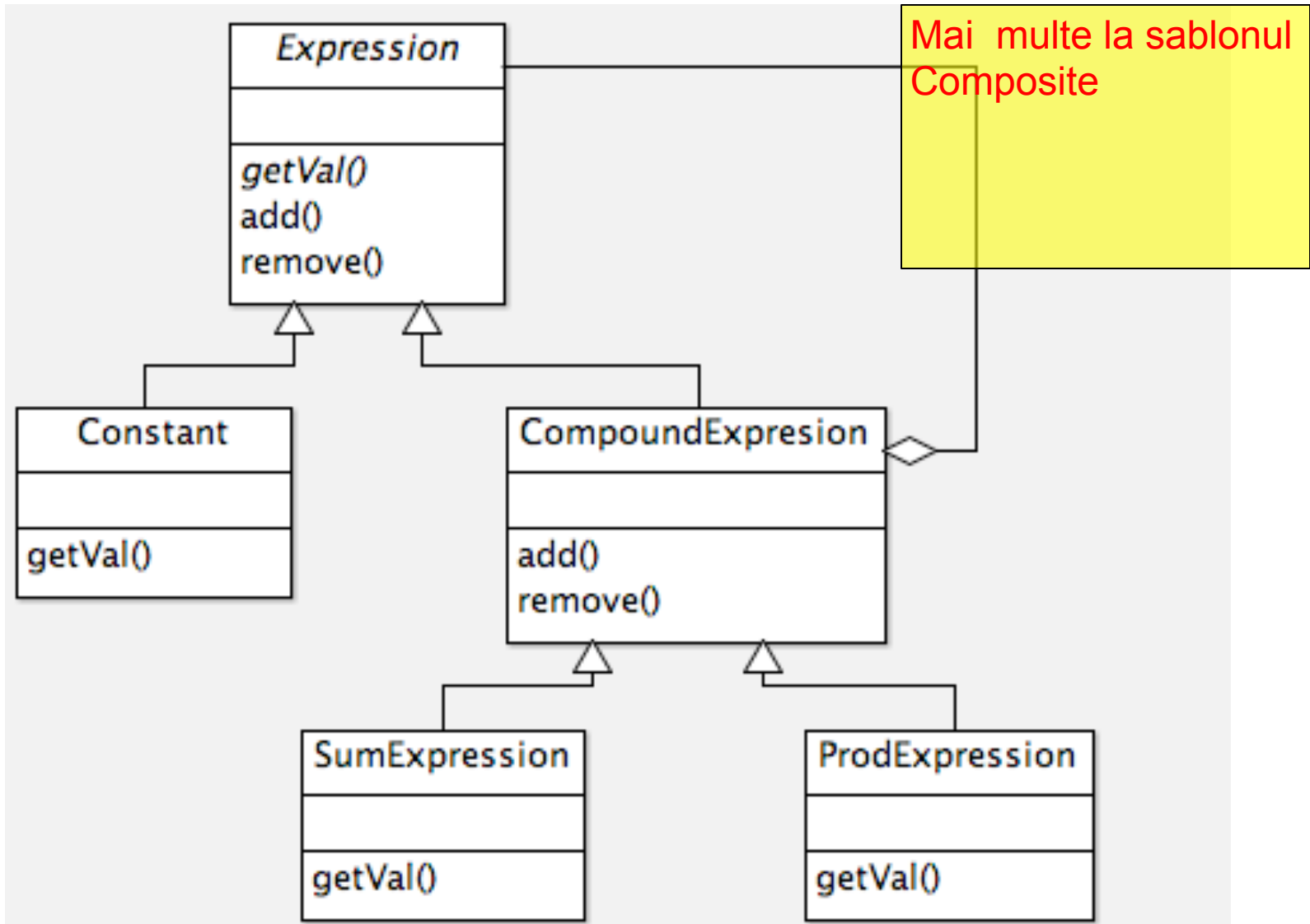
Fabrica de obiecte:: structura



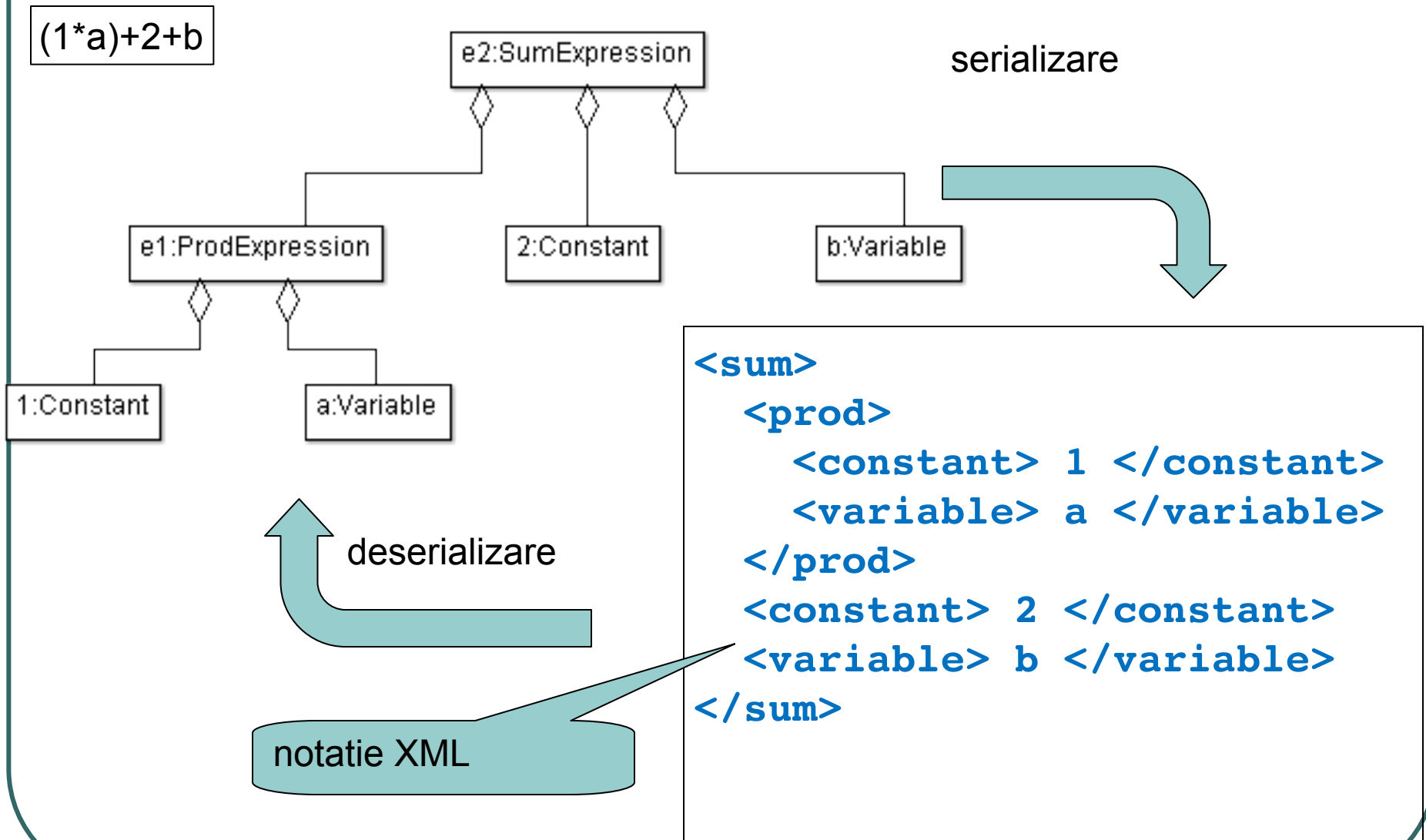
Fabrica de obiecte

- colaborari
 - normal se creeaza o singura instanta
- consecinte
 - izoleaza clasele concrete
 - simplifica schimbul familiei de produse
 - promoveaza consistenta printre produse
 - suporta noi familii de produse usor
 - **respecta principiul deschis/inchis**
- implementare
 - se face pe baza studiului de caz “expression factory”

Expresii : : structura



Expressions: Problema (diagr. de obiecte)



Expressions: Problema

- serializare
 - vizitator
- deserializare

```
switch (tag)
{
    case <sum>:
        ...
    case <prod>:
        ...
    case <constant>:
        ...
    case <variable>:
        ...
}
```

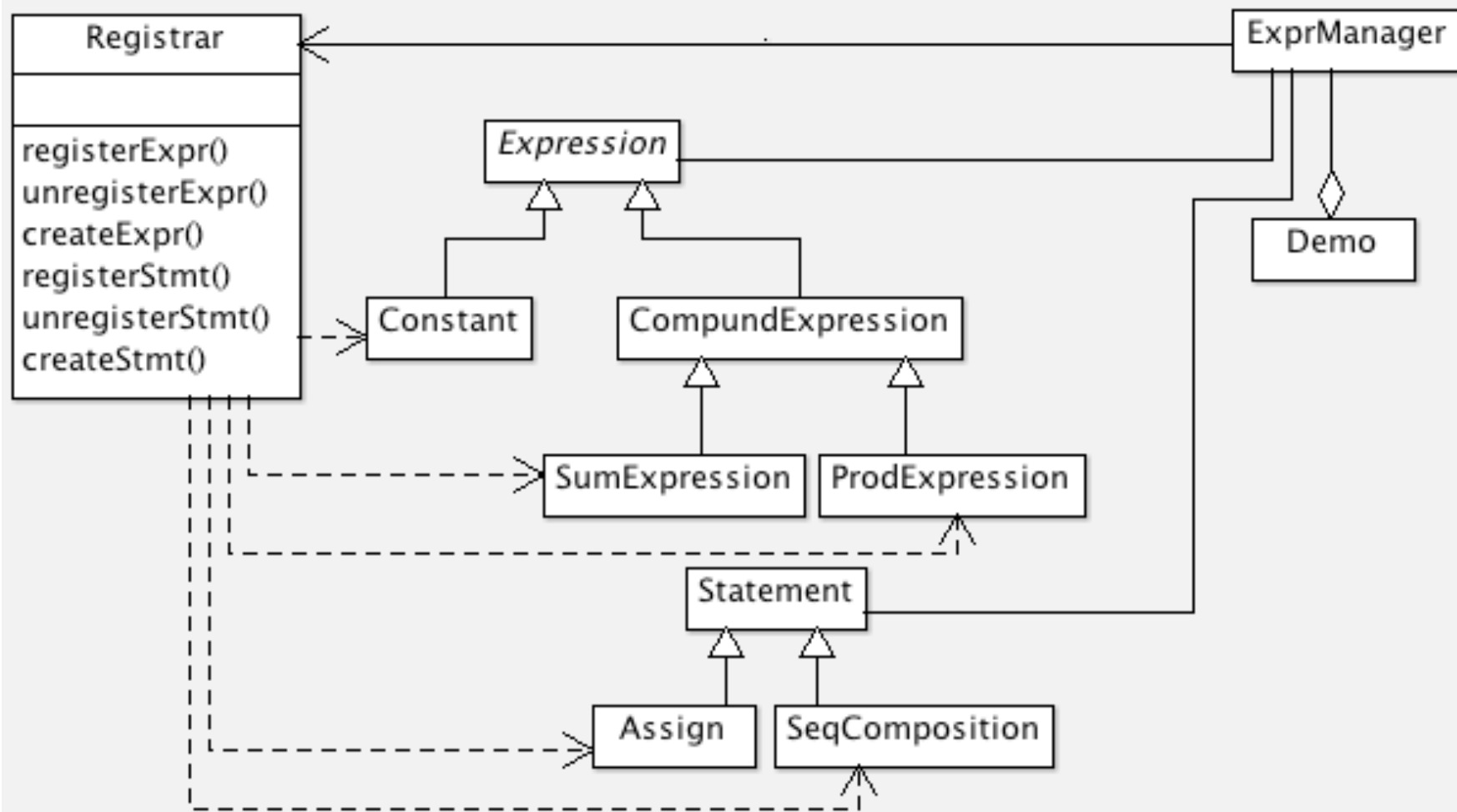
sablonul Visitor va fi facut la cursurile urmatoare

se incalca principiul inchis-deschis

Solutia: object factory

- corespondenta cu modelul standard
 - AbstractProductA = Expression
 - AbstractProductB = Statements (neimplementat inca, lasat ca exercitiu)
 - ConcreteFactory = Registrar (registru de expresii, instructiuni)
 - Client = ExprManager (responsabila cu deserializarea)

“Object factory” pentru expresii, instr.



Registru de clase (Registrar)

- este o clasa care sa gestioneze tipurile de expresii
 - inregistreaza un nou tip de expresie (apelata ori de cate ori se defineste o noua clasa derivata)
 - eliminarea unui tip de expresie inregistrat (stergerea unei clase derivate)
 - crearea de obiecte expresie
 - la nivel de implementare utilizam perechi
(tag, createExprFn)
 - ... si functii delegat (vezi slide-ul urmator)
- se poate utiliza sablonul Singleton pentru a avea o singura fabrica (registru)

Functii delegat (callback)

- o functie **delegat (callback)** este o functie care nu este invocata explicit de programator; responsabilitatea apelarii este delegata altei functii care primeste ca parametru adresa functiei delegat
- Fabrica de obiecte utilizeaza functii delegat pentru crearea de obiecte: pentru fiecare tip este delegata functia carea creeaza obiecte de acel tip
- pentru “expression factory” declaram un alias pentru tipul functiilor de creare a obiectelor Expression

```
typedef Expression* ( *CreateExprFn ) ();
```

Registrar 1/3

```
class Registrar
```

```
{
```

```
    bool registerExpr(string tag,  
                      CreateExprFn createExprFn )
```

```
{
```

```
    return catalog.insert(  
        std::pair<string, CreateExprFn>(   
            tag, createExprFn)  
        ) .second;
```

```
}
```

metoda responsabila cu inregistrarea
unui nou tip de obiecte Expression

inserarea in catalog (un
map)

a doua componenta a valorii
intoarse de insert (inserare cu
succes sau fara succes)

Registrar 2/3

```
void unregisterExpr(string tag)
{
    catalog.erase(tag);
}
```

metoda responsabila cu eliminarea unui obiect tip
Expression

```
Expression* createExpr(string tag)
{
    map<string, CreateExprFn>::iterator i;
    i = catalog.find(tag);
    if ( i == catalog.end() )
        throw string("Unknown expression tag");
    return (i->second)();
}
```

metoda responsabila cu crearea de obiecte
Expression

de fapt deleaga aceasta responsabilitate metodei care corespunde tipului dat ca parametru

Registrar 3/3

protected:

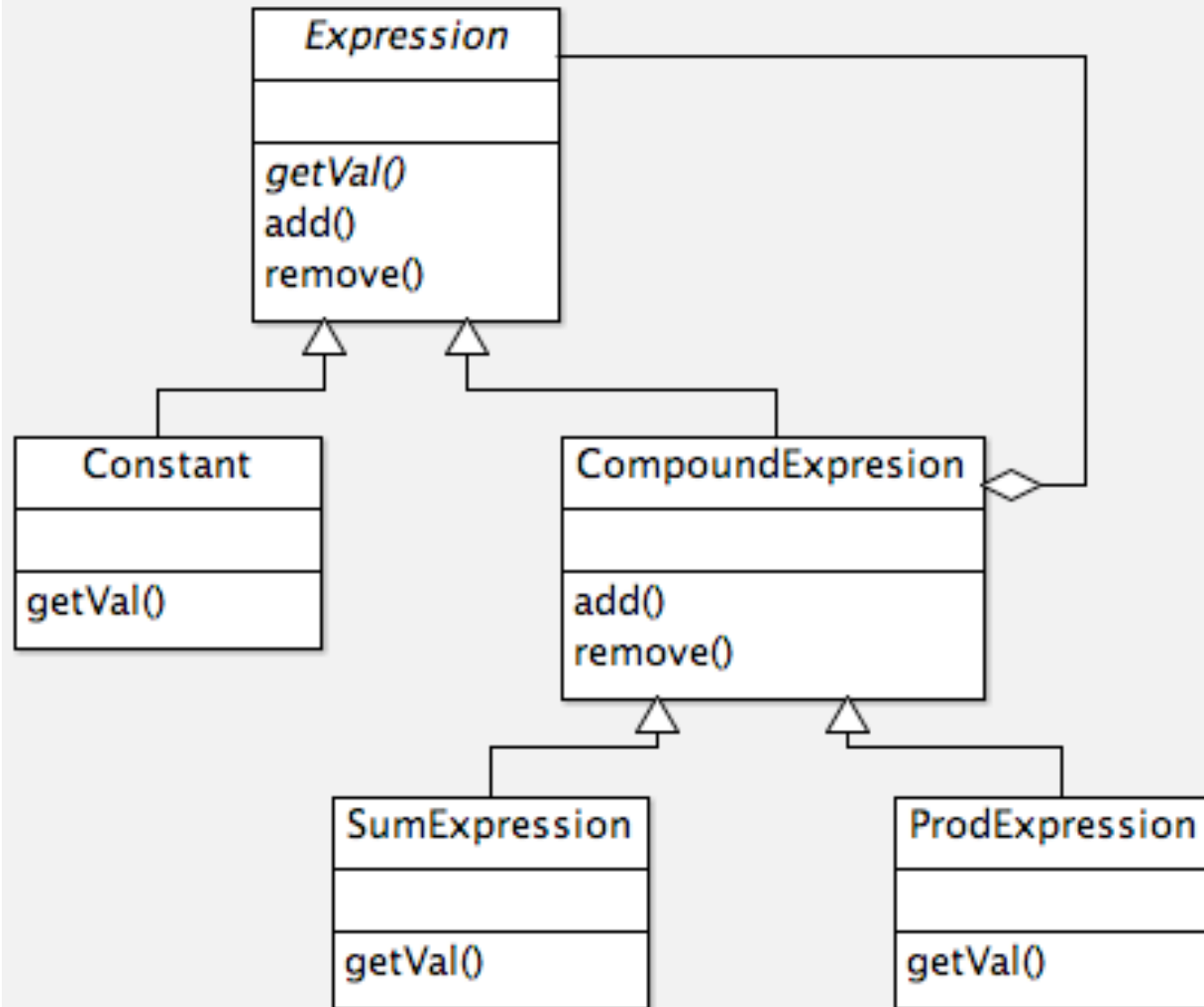
```
map<string, CreateExprFn> catalog;
```

```
};
```



catalogul este un tablou asociativ

Produsele din familia Expression



expr-manager.h – functii de creare obiecte

```
Expression* createConstant() {  
    return new Constant();  
}
```

```
Expression* createVariable() {  
    return new Variable();  
}
```

```
Expression* createProd() {  
    return new ProdExpression();  
}
```

...

Clasa ExprManager - constructorul

```
class ExprManager {
public:
    ExprManager() {
        reg = new Registrar();
        reg->registerExpr("<constant>",
                        createConstant);
        reg->registerExpr("<variable>",
                        createVariable);
        reg->registerExpr("<prod>", createProd);
        reg->registerExpr("<sum>", createSum);
    }
}
```

Deserializarea (fabrica de obiecte din descrieri XML)

```
public:
    Expression* loadf(ifstream& f)
    {
        if (f.eof())
            throw "Unknown file.";
        string tag;
        f >> tag;
        return loadfRec(f, tag);
    }
protected:
    Registrar *reg;
```

Funcția recursivă de creare obiecte 1/3

protected:

```
Expression* loadfRec(ifstream& f,  
                    string tag)
```

```
{
```

```
    Expression* expr1 = reg->createExpr(tag);
```

```
    string endTag = tag.insert(1, "/");
```

```
    Expression* expr2;
```

memoreaza expresiile
componente, daca expr1
este compusa

calculeaza tagul de
sfarsit

creaza obiectul
pentru nodul curent

... cazul obiectelor compuse

```
if (expr1->getCompoundExpression()) {  
    if (f.eof())  
        throw "File illformatted."  
    string nextTag;  
    f >> nextTag;  
    while (endTag != nextTag && !f.eof()) {  
        expr2 = loadfRec(f, nextTag);  
        expr1->add(expr2);  
        f >> nextTag;  
    }  
}
```

citeste urmatorul tag

daca nu s-a ajuns la tagul de sfarsit, inseamna ca avem o noua componenta pe care o cream recursiv si o adaugam la expresia compusa

... cazul obiectelor elementare

```
else {  
    if (f.eof())  
        throw "File illformatted.";  
    expr1->loadInfo(f);  
    if (f.eof())  
        throw "File illformatted.";  
    f >> tag;  
}  
return expr1;  
}  
};
```

incarca informatia
din nodul frunza

consuma tagul de
sfarsit

Concluzii

- sabloanele de proiectare constituie o modalitate de a învăța cum să vă construiască programele.
- un sablon de proiectare este un sfat care vine de la oameni care au distilat soluțiile lor cele mai comune în sfaturi de cunoștințe simple, digerabile și cu nume sugestiv
- POO și sabloanele de proiectare sunt subiecte distincte
- POO te învață cum să programezi, este o metodologie de programare sau un concept de programare
- sabloanele de proiectare te învață cum să te gândești despre programe, îți sugerează metode de construire a unor clase / obiecte pentru a rezolva un anumit scenariu într-un program, metode dovedite a avea succes
- multe alte sabloane utile se găsesc în GoF
-