

***P00***

Sablonul  
*Object Factory*

# Cuprins

---

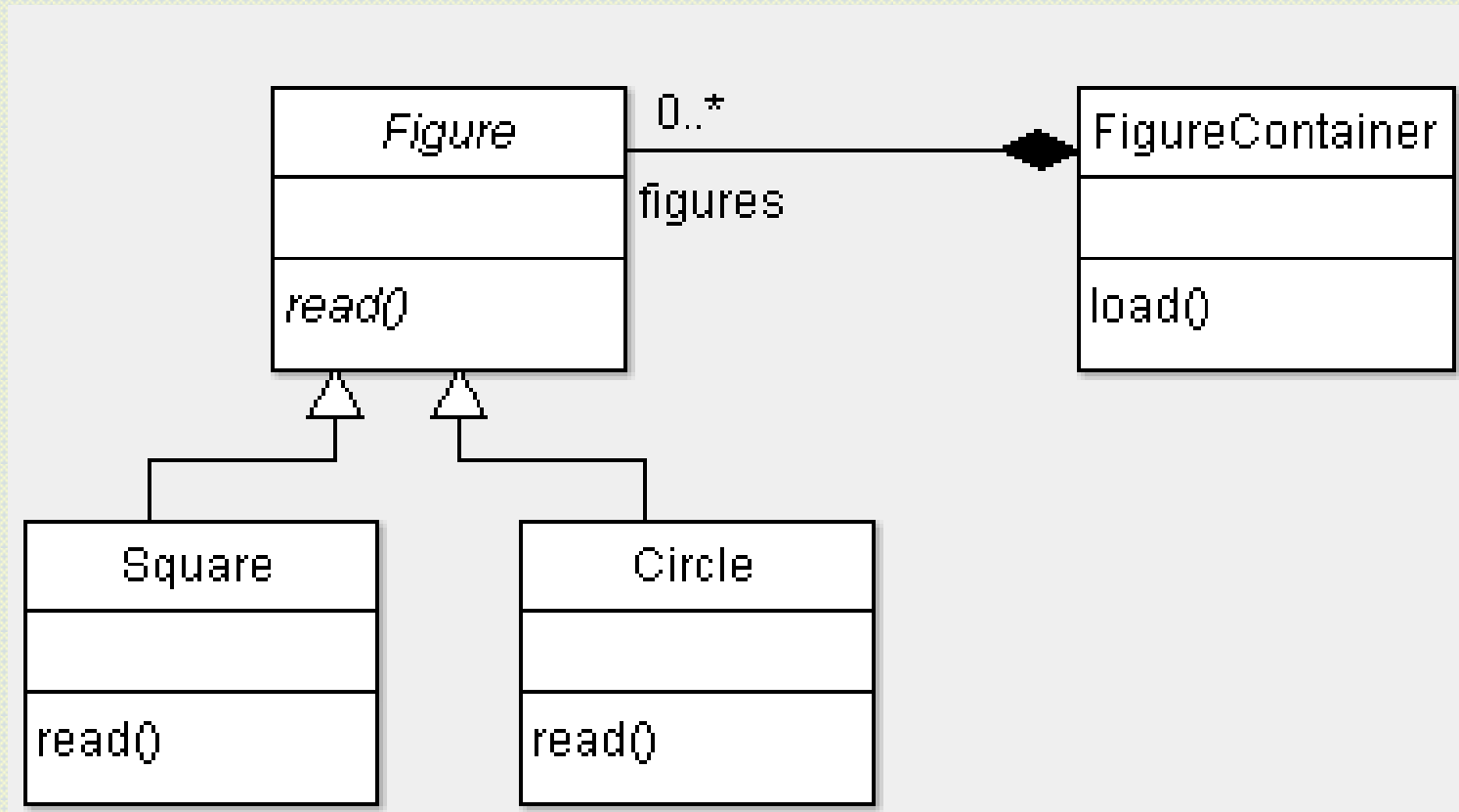
- principiul inchis-deschis
- fabrica de obiecte (Abstract Object Factory)  
(prezentare bazata pe GoF)
- studii de caz:
  - *expression factory*

## 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”

---

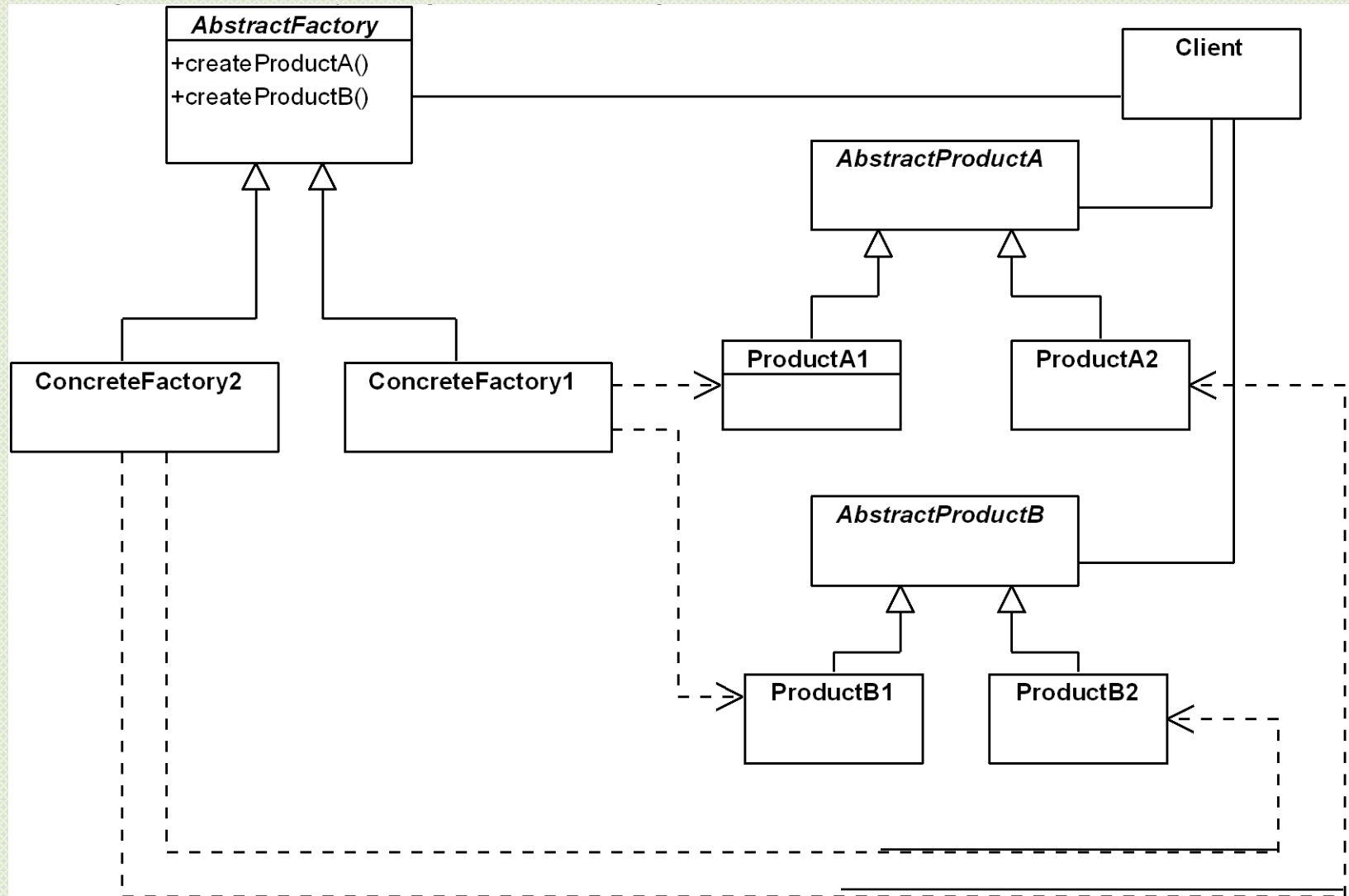
O posibila solutie:  
fabrica de obiecte

# 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 pentru astfel ca obiectele sa fie utilizate impreuna
  - vrei sa furniziei o biblioteca de produse ai vrei sa accesibila numai interfata, nu si implementarea

# Fabrica de obiecte:: structura



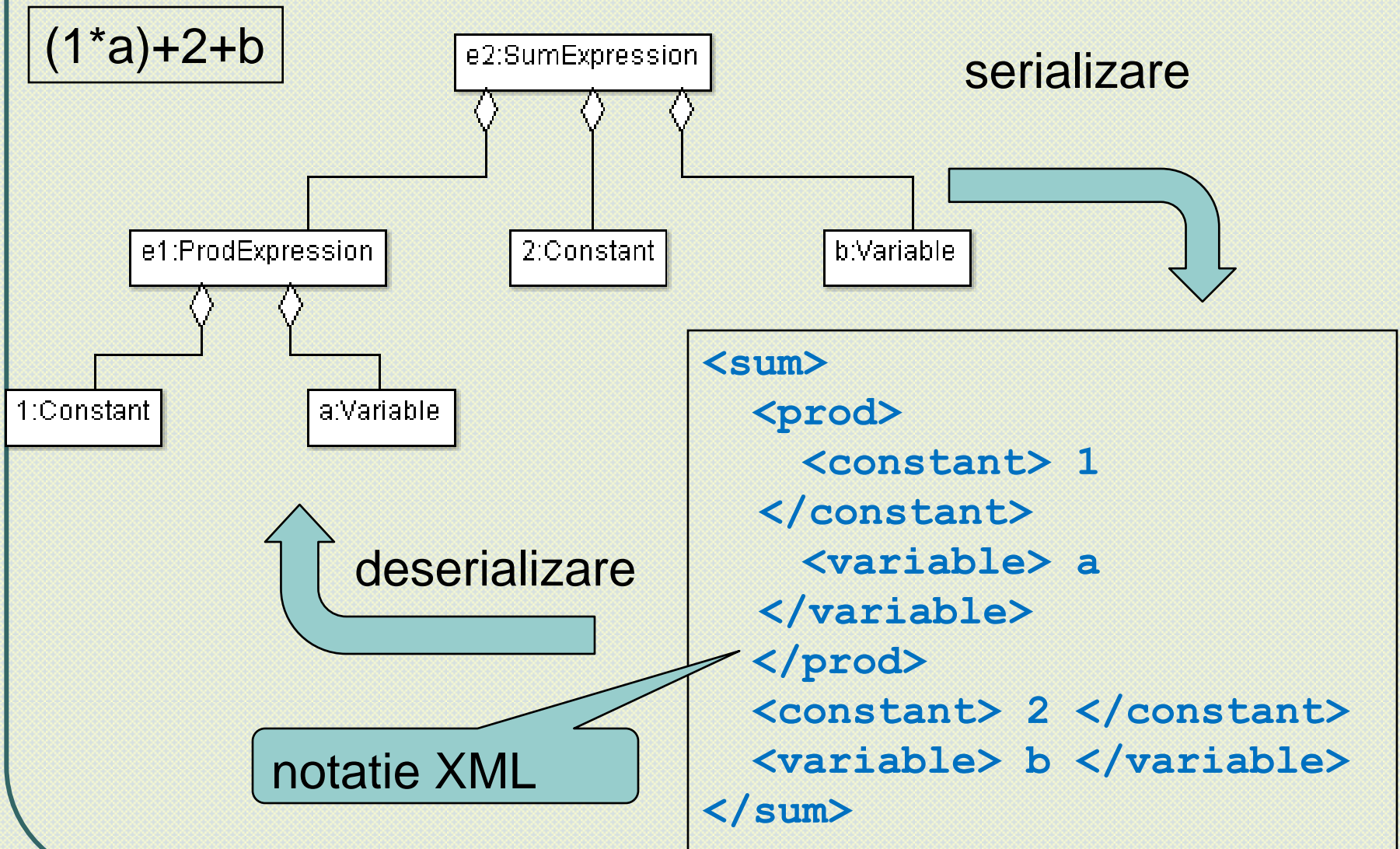


# Fabrica de obiecte

---

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

# Expressions: Problema



# Expressions: Problema

---

- serializare
  - vizitator
- deserializare

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

se incalca principiul  
inchis-deschis

# Solutia: object factory

---

- Corespondenta cu modelul standard
  - AbstractProductA = Expression
  - AbstractProductB = Statements (neimplementat inca)
  - ConcretFactory = Registrar (registru de expresii) + ExprManager (responsabila cu deserializarea)

# 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)

## Funcții delegat (callback)

---

- o funcție **delegat (callback)** este o funcție care nu este invocată explicit de programator; responsabilitatea apelării este delegată altei funcții care primește ca parametru adresa funcției delegat
- Fabrica de obiecte utilizează funcții delegat pentru crearea de obiecte: pentru fiecare tip este delegată funcția care creează obiecte de acel tip
- pentru “expression factory” declarăm un alias pentru tipul funcțiilor de creare a obiectelor Expression

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

# Registrar 1/3

```
class Registrar
```

```
{
```

```
    bool registerExpr(string tag,
```

```
                        CreateExprFn createExprFn )
```

```
{
```

```
    return catalog.insert( map<string,
```

```
                          CreateExprFn>:: value_type (tag,
```

```
                          createExprFn) ) .second;
```

```
}
```

metoda responsabila cu inregistrarea  
unui nou tip de obiecte Pizza

inserarea in tablou

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 tip Pizza

```
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:

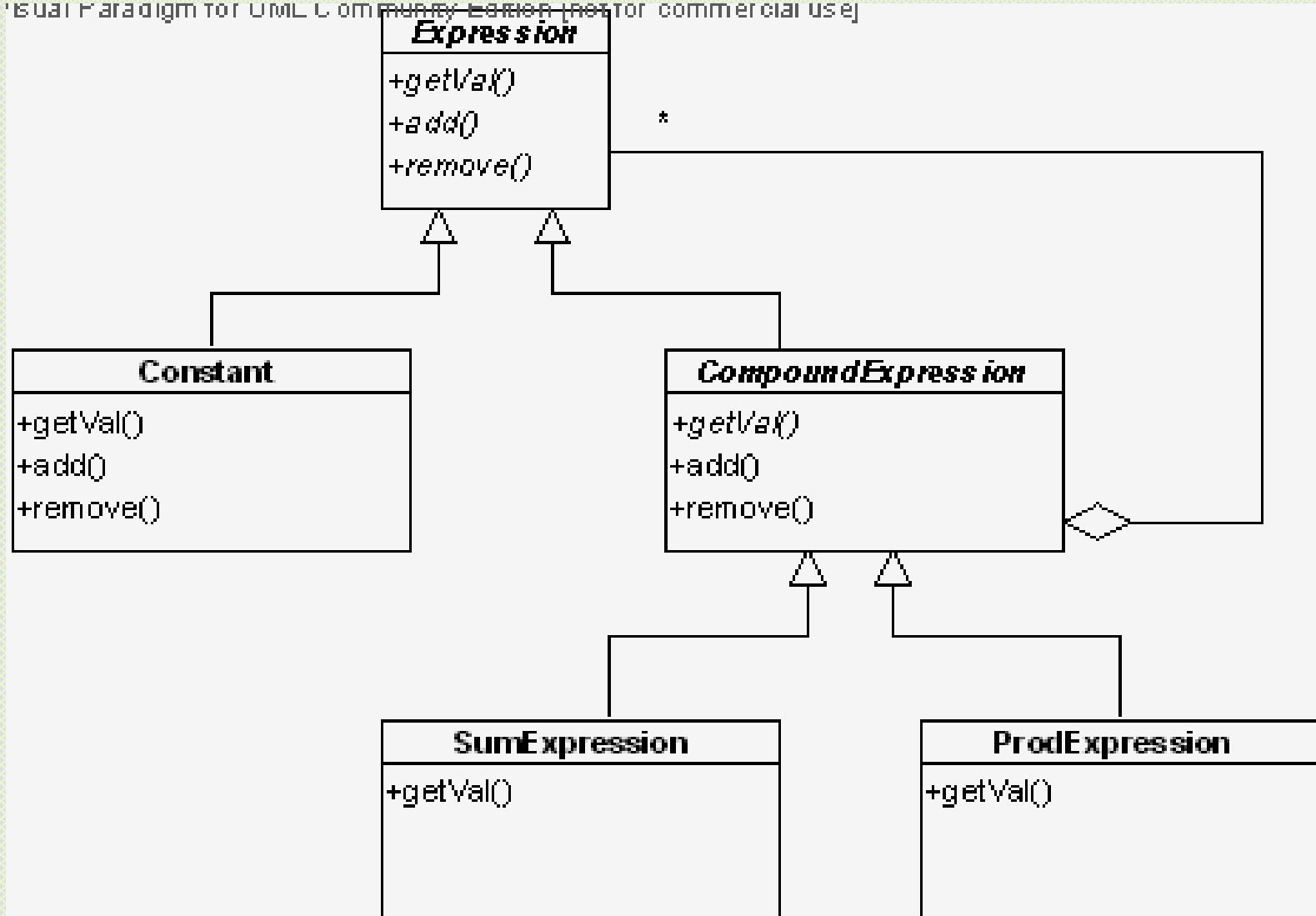
```
map<string, CreateExprFn> catalog;  
  
};
```



catalogul este un tablou asociativ

# Produsele din familia Expression

Visual Paradigm for UML Community Edition (not for commercial use)



## Deserializarea (fabrica de obiecte din descrieri XML)

---

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

# Functia recursiva de creare obiecte 1/3

```
static Expression* loadfRec(ifstream& f,  
                             Registrar& reg, 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, reg, 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

## Clientul (Demo) – inregistrarea

---

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

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

```
Registrar aReg;  
aReg.registerExpr("<constant>", createConstant);  
aReg.registerExpr("<variable>", createVariable);  
aReg.registerExpr("<prod>", createProd);  
aReg.registerExpr("<sum>", createSum);
```

# Clientul (Demo) – deserializarea

```
outfile.open("test-copy.xml", ios::out);  
ifstream infile("test.xml");  
  
try {  
    Expression* exprCopy =  
    ExprManager::loadf(infile, aReg);  
    exprCopy->accept(visitorStoref, outfile);  
}  
catch(string msg) {  
    cout << msg << endl;  
}  
infile.close();
```

fisierul care include descrierea XML

creeaza o copie pentru verificare



# Serializare 1/3

---

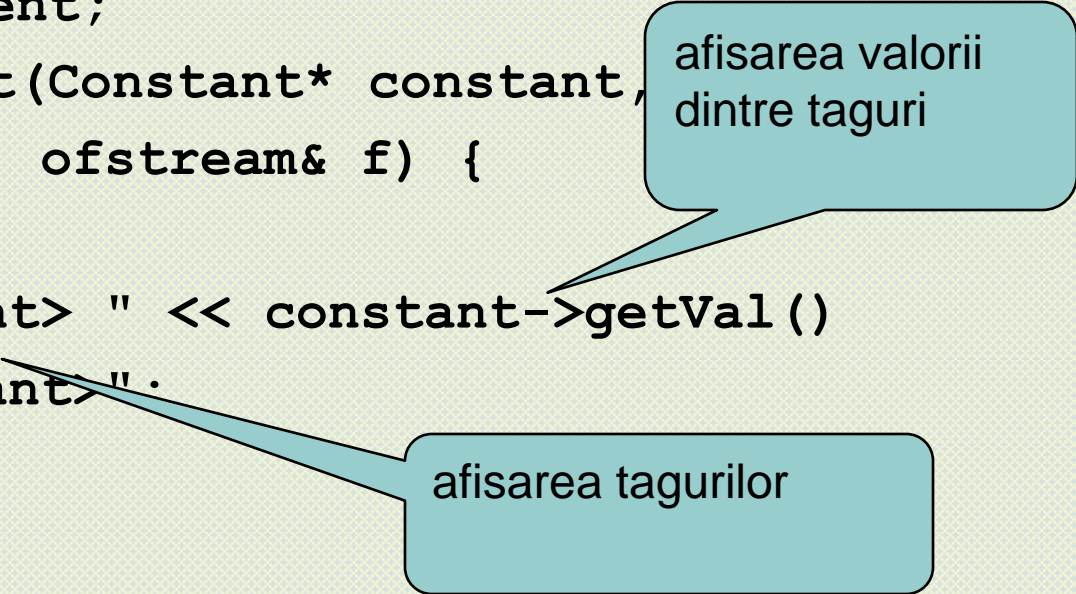
```
class Visitor {
public:
virtual void visitConstant(Constant*, ostream& f) = 0;
virtual void visitVariable(Variable*, ostream& f) = 0;
virtual void
    visitCompoundExpression(CompoundExpression*,
                             ostream& f) {}
virtual void visitProdExpression(ProdExpression*,
                                  ostream& f) {}
virtual void visitSumExpression(SumExpression*,
                                 ostream& f) {}

protected:
    Visitor(){};
};
```

## Serializare 2/3

```
class VisitorStoref : public Visitor {
public:
    static string indent;
    void visitConstant(Constant* constant,
                      ofstream& f) {
        f << indent;
        f << "<constant> " << constant->getVal()
          << " </constant>".
        f << endl;
    }

    // la fel pentru variabila
```



afisarea valorii  
dintre taguri

afisarea tagurilor

## Serializare 3/3

```
void visitProdExpression(ProdExpression* prod,
                          ofstream& f) {
    f << indent;
    f << "<prod>" << endl;
    indent += "  ";
    prod->CompoundExpression::accept(*this, f);
    indent = indent.erase(indent.size()-2,2);
    f << indent;
    f << "</prod>" << endl;
}
// la fel pentru suma
```

indentarea  
textului

serializarea  
componentelor

descresterea  
indentarii