

POO

Sabloane de proiectare
creationale

Cuprins

- sabloane de proiectare (software design patterns)
- sabloane creationale
 - Singleton
 - Builder
 - Object Factory

Sabloane de proiectare (Design Patterns)

- intai aplicate in proiectare urbanistica:
C. Alexander. A Pattern Language. 1977
- prima contributie in software: 1987, Kent Beck (creatorul lui Extreme Programming) & Ward Cunningham (a scris primul wiki)
- contributia majora: Design Patterns:
Gamma et al. Elements of Reusable Object-Oriented Software was published, 1994
 - cunoscuta ca **GoF** (Gang of Four)
 - in functie de scop, clasifica patternurile in
 - creationale
 - structurale
 - comportamentale
 - pot fi aplicate la nivel de clasa sau obiect

Ce este un sablon de proiectare?

- 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"*
- Elementele esentiale ale unui pattern (GoF):
 - nume
 - problema (si context)
 - solutie
 - consecinte
- GoF include 23 de sabloane

Formatul (template) 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)

Clase cu o singura instanta (Singleton)

- 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

Clase cu o singura instanta (Singleton)

- structura

Singleton
-uniqueInstance -data
+getValue() +setValue() +instance()

- participant: Singleton
- colaborari: clientii clasei

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

```
class Singleton  
{
```

manerul cu care se are acces
la instanta

```
public:
```

```
    static Singleton& instance()  
    {return uniqueInstance;}
```

```
    int getValue() { return i; }
```

```
    void setValue(int x) { i = x; }
```

```
private:
```

```
    static Singleton uniqueInstance;
```

```
    int i;
```

```
    Singleton(int x) : i(x) { }
```

constructor

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

operator atribuire

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

constructor de copiere

```
};
```

Clase cu o singura instanta

```
Singleton Singleton::uniqueInstance(47) ;
```

```
int main()
```

```
{
```

```
    Singleton& s1 = Singleton::instance() ;
```

```
    cout << s1.getValue() << endl;
```

```
    Singleton& s2 = Singleton::instance() ;
```

```
    s2.setValue(9) ;
```

```
    cout << s1.getValue() << endl;
```

```
    return 0;
```

```
}
```

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 s4 = s2;
```

```
s4.setValue(23);
```

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

```
cout << s2.getValue() << endl; // 9
```

- daca se comenteaza operatorul de atribuire, atunci se poate executa urmatorul cod:

```
s4 = s2;
```

```
s4.setValue(43);
```

```
cout << s4.getValue() << endl; // 43
```

```
cout << s2.getValue() << endl; // 9
```

Demo

Instanta unica dinamica 1/2

```
class Singleton {  
public:  
    static Singleton* instance() {  
        if (uniqueInstance == 0) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
    int getValue() { return i; }  
    void setValue(int x) { i = x; }
```

Instanta unica dinamica 2/2

protected:

```
int i;
```

```
Singleton(int x = 0) : i(x) { }
```

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

```
    // nu mai e necesar (de ce?)
```

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

```
    // nu mai e necesar (de ce?)
```

private:

```
// pointer la instanta unica
```

```
static Singleton* uniqueInstance;
```

```
};
```


Demo

```
Singleton* s1 = Singleton::instance();  
s1->setValue(47);  
cout << s1->getValue() << endl; // 47  
Singleton* s2 = Singleton::instance();  
s2->setValue(9);  
cout << s1->getValue() << 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* s4 = s2;
```

```
s4.setValue(23);
```

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

```
cout << s2.getValue() << endl; // 23
```

```
s4 = s2;
```

```
s4.setValue(43);
```

```
cout << s4.getValue() << endl; // 43
```

```
cout << s2.getValue() << endl; // 43
```

- de ce?

Diferenta dintre pointer si referinta 2/2

```
s2->setValue(9);
```

- dar urmatoarele 4 instructiuni nu se compileaza daca se decommenteaza constructorul de copiere

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

```
s5.setValue(23);
```

```
cout << s5.getValue() << endl; // 23
```

```
cout << s2->getValue() << endl; // 9
```

- si urmatoarele 4 instructiuni nu se compileaza daca se decommenteaza si operatorul de atribuire

```
s5 = *s2;
```

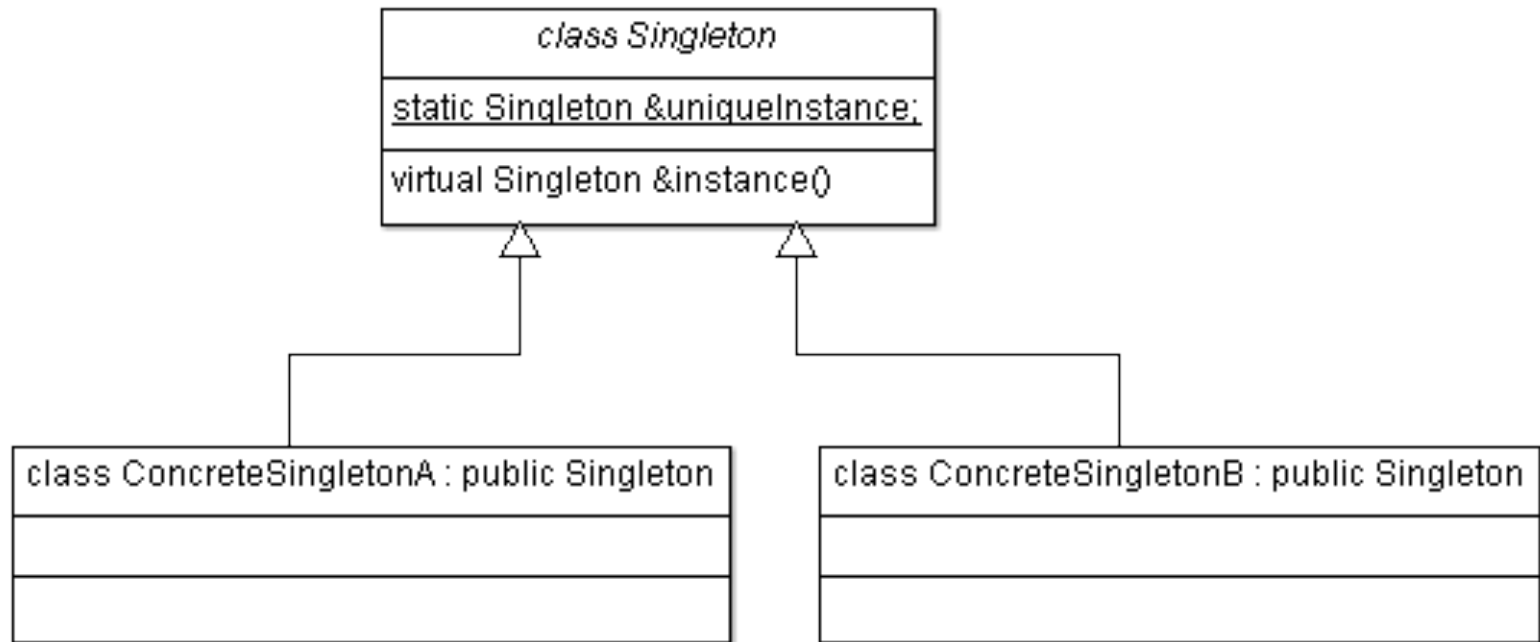
```
s5.setValue(43);
```

```
cout << s5.getValue() << endl; // 43
```

```
cout << s2->getValue() << endl; // 9
```

Demo

Clase singleton derivate



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

Repozitoriu de clase singleton 1/3

```
class Singleton {
public:
    static void register(const char* name,
                        Singleton*);

    static Singleton* instance();
protected:
    static Singleton* lookup(const char* name);
private:
    static Singleton* uniqueInstance;
    static List<NameSingletonPair>* registry;
};
```

Repozitoriu de clase singleton 2/3

- metoda instance cauta in registru adresa instantei unice pentru o clasa concreta din ierarhie

```
Singleton* Singleton::instance () {  
    if (uniqueInstance == 0) {  
        const char* singletonName = getenv("SINGLETON");  
        // furnizata la incarcarea aplicatiei  
        uniqueInstance = lookup(singletonName);  
        // lookup intoarce 0 daca nu gaseste  
    }  
    return uniqueInstance;  
}
```

Repozitoriu de clase singleton 3/3

- o clasa singleton concreta din ierarhie trebuie sa inregistreze in registru adresa instantei unice

```
ConcreteSingletonA::Singleton() {  
    // ...  
    Singleton::register("ConcreteSingletonA",  
                        this);  
}
```

- in fisierul cu implementarea trebuie sa avem
`static ConcreteSingletonA theSingletonA;`

P00

Builder

(prezentare bazata pe GoF)

Intentie

- Separa constructia unui obiect complex de reprezentarile sale astfel incat procesul de constructie poate crea diferite reprezentari
- clasificare: creational

Motivatie

Default color

Red

Default color

- RTF

```
{\rtf1\ansi\deff0
```

```
{\colortbl;\red0\green0\blue0;\red255\green0\blue0;}
```

```
Default color\line
```

```
\cf2
```

```
Red\line
```

```
\cf1
```

```
Default color
```

```
}
```

Motivatie

- LaTeX

```
\rm Default color\\
```

```
\color{red}Red\\
```

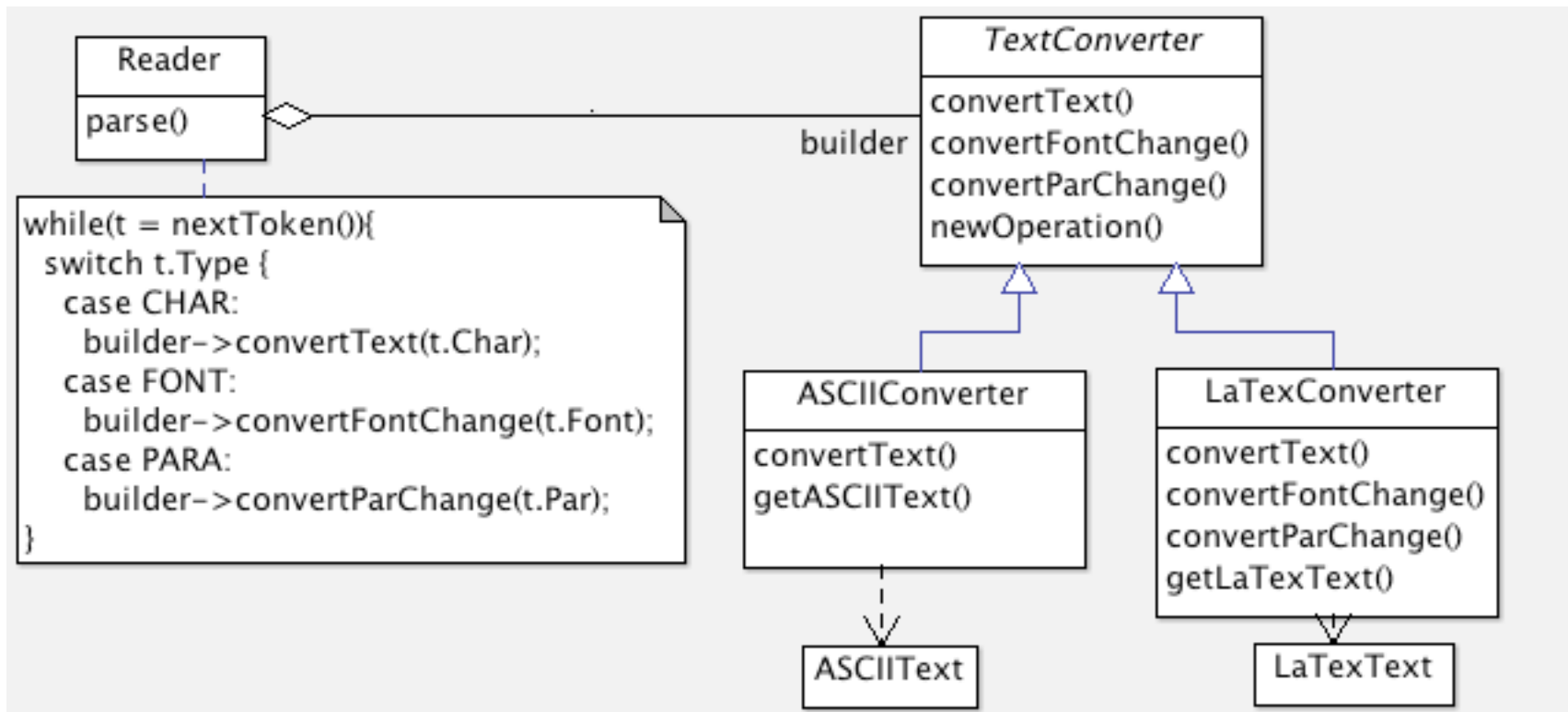
```
\color{black}Default color
```

- ASCII

```
Default color\nRed\nDefault color
```

- ...

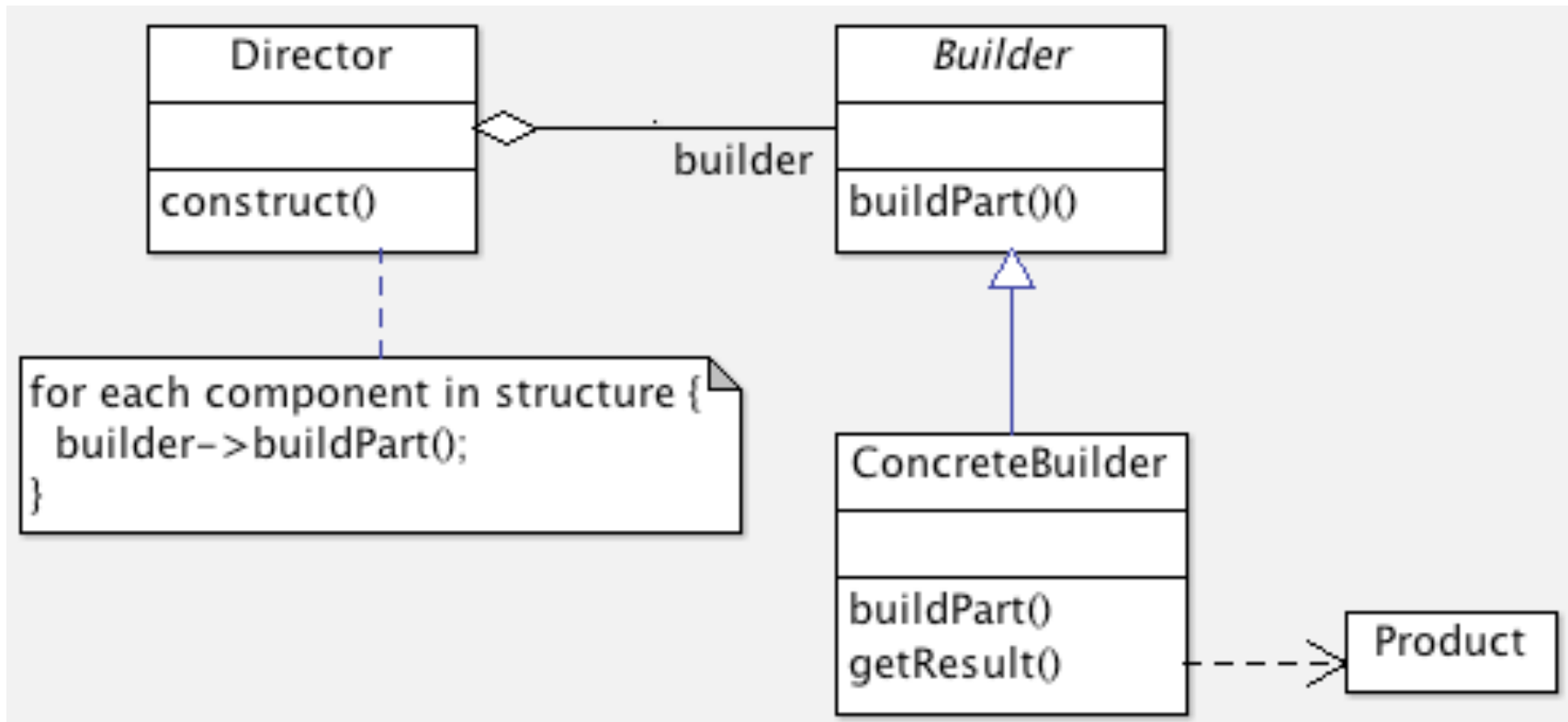
Motivatie



Aplicabilitate

- algoritmul de creare a unui obiect complex trebuie sa fie independent de componente si asamblarea lor
- constructia trebuie sa permita mai multe reprezentari

Structura



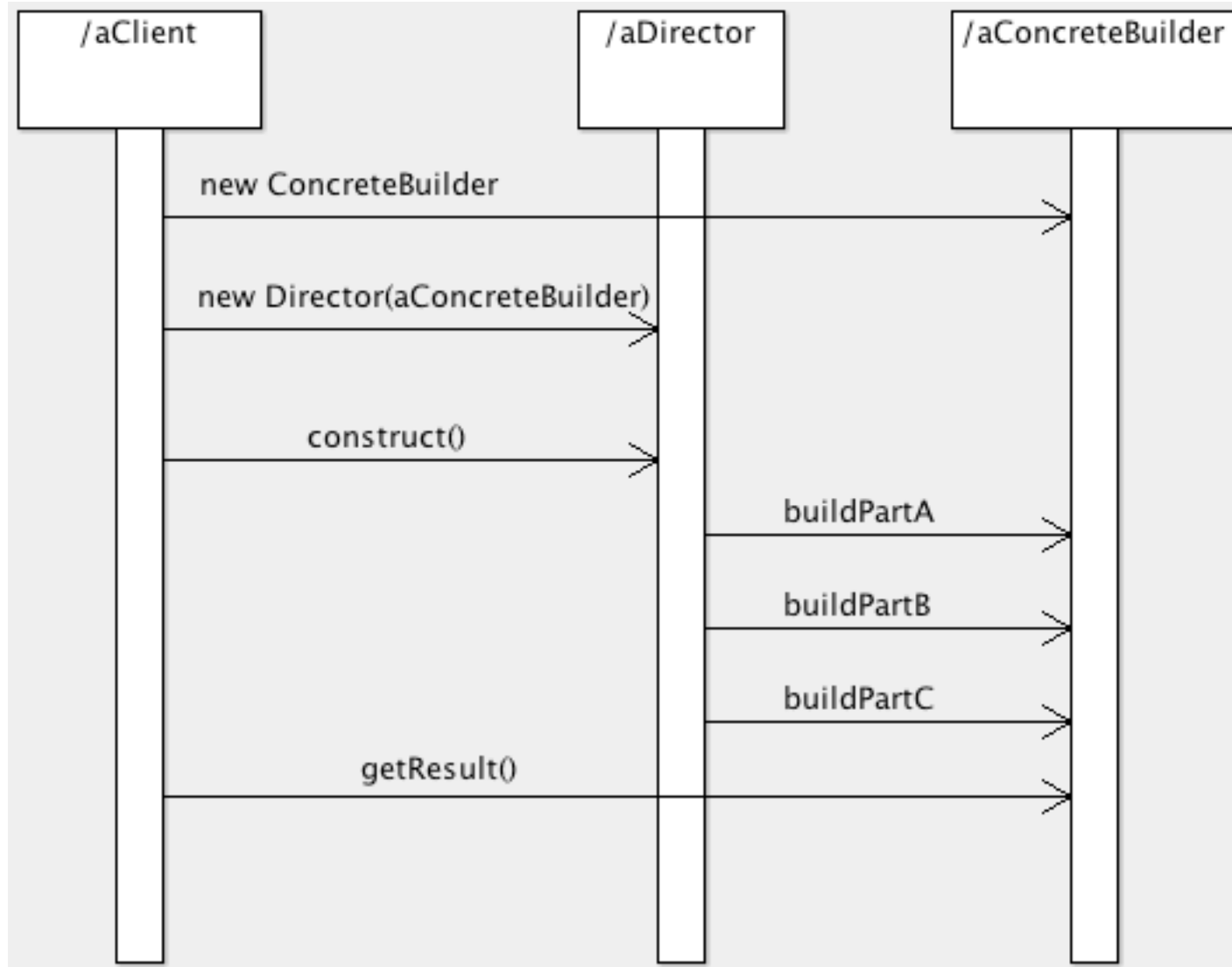
Participanti

- Builder
 - definește o interfață abstractă pentru crearea componentelor necesare pentru a construi un obiect Product
- ConcreteBuilder
 - implementează interfața Builder, construind și asamblând componentele
 - definește și păstrează o “urma” a reprezentării create
 - definește o interfață pentru regasirea produsului
- Director
 - construiește obiectul complex
- Product
 - reprezintă obiectul complex care se construiește
 - include clasele ce definesc partile constituente.

Colaborari

- clientul creeaza un obiect Director si-l configureaza cu un obiect Builder adecvat
- obiectul Director notifica obiectul Builder ori de cate ori o componenta trebuie construita
- obiectul Builder proceseaza cererile primite de la obiectul Director si adauga componentele
- clientul “preia” produsul de la obiectul Builder

Diagrama de colaborare



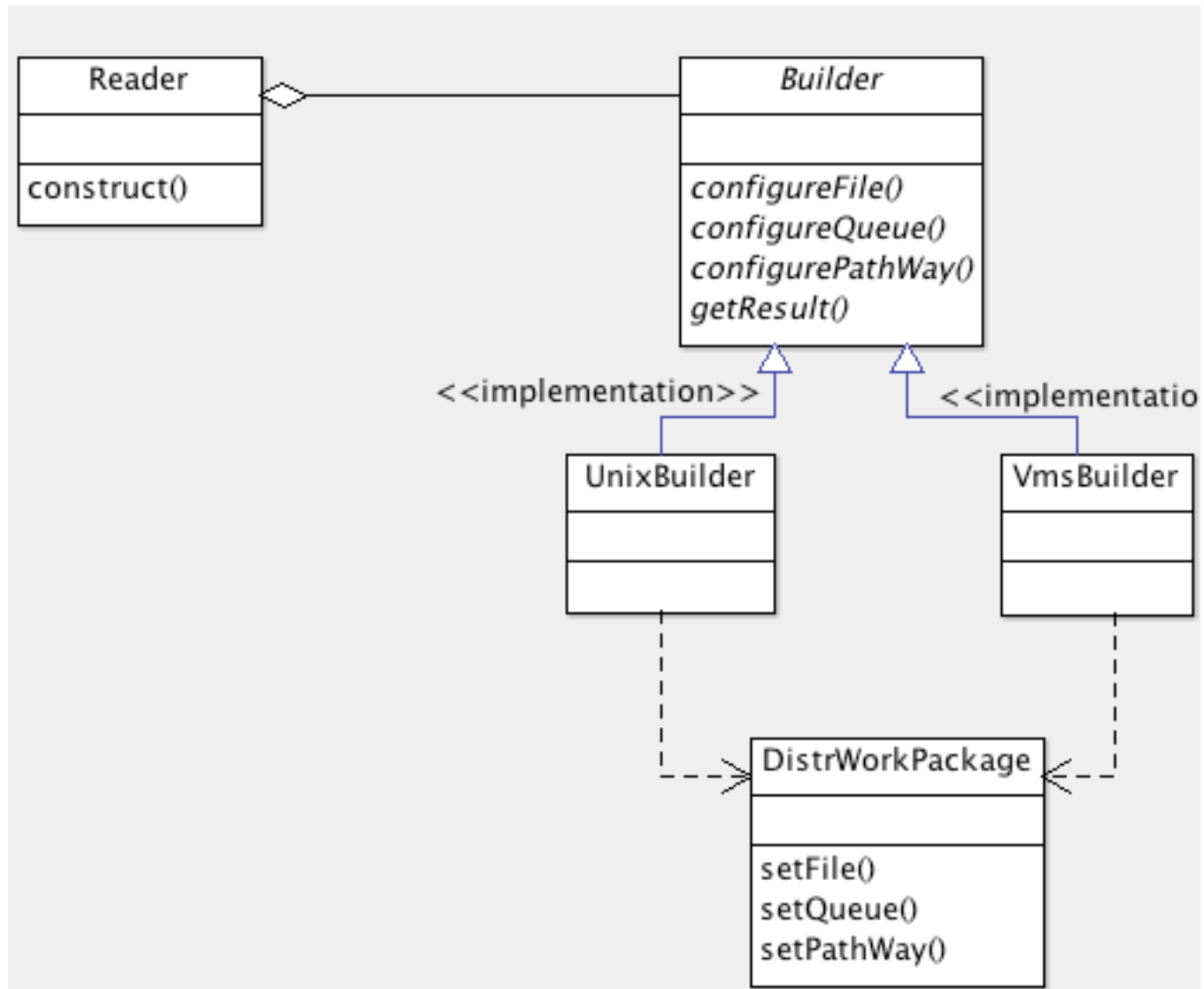
Consecinte

- Lasa libertatea de a varia reprezentarea interna a produsului
 - obiectul Builder furnizeaza obiectului Director o interfata pentru constructia produsului
 - interfata ii permite obiectului Builder sa ascunde reprezentarea interna
- Izoleaza codul destinat construirii produsului de reprezentare
 - clientii nu cunosc nimic despre reprezentare
- Permite sa ai controlul asupra procesului de constructie
 - produsul este construit componenta cu componenta, prin intermediul obiectului Builder

Implementare

- vom discuta exemplul “distributed work packages” din cartea “Design Patterns Explained Simply”
(https://sourcemaking.com/design_patterns/builder/cpp/1)
- problema consta in a defini o interfata abstracta pentru pachetele de lucru distribuite ce sa fie persistenta si independenta de platforma
- aceasta presupune implementari specifice platformelor pentru fisier (file), cozi de prioritate (queue), cai de concurenta (concurrent pathways)

Distributed Work Package Diagram



Produsul (DistrWorkPackage)

```
class DistrWorkPackage
{
    public:
        DistrWorkPackage(char *type);
        void setFile(char *f, char *v);
        void setQueue(char *q, char *v);
        void setPathway(char *p, char *v);
        const char *getState();
    private:
        char _desc[200], _temp[80];
};
```

Interfata Builder

```
class Builder
{
    public:
        virtual void configureFile(char*) = 0;
        virtual void configureQueue(char*) = 0;
        virtual void configurePathway(char*) = 0;
        DistrWorkPackage *getResult()
        {
            return _result;
        }
    protected:
        DistrWorkPackage *_result;
};
```

Un *Builder* concret: UnixBuilder

```
class UnixBuilder : public Builder {
public:
    UnixBuilder() {
        _result = new DistrWorkPackage("Unix");
    }
    void configureFile(char *name) {
        _result->setFile("flatFile", name);
    }
    void configureQueue(char *queue) {
        _result->setQueue("FIFO", queue);
    }
    void configurePathway(char *type) {
        _result->setPathway("thread", type);
    }
};
```

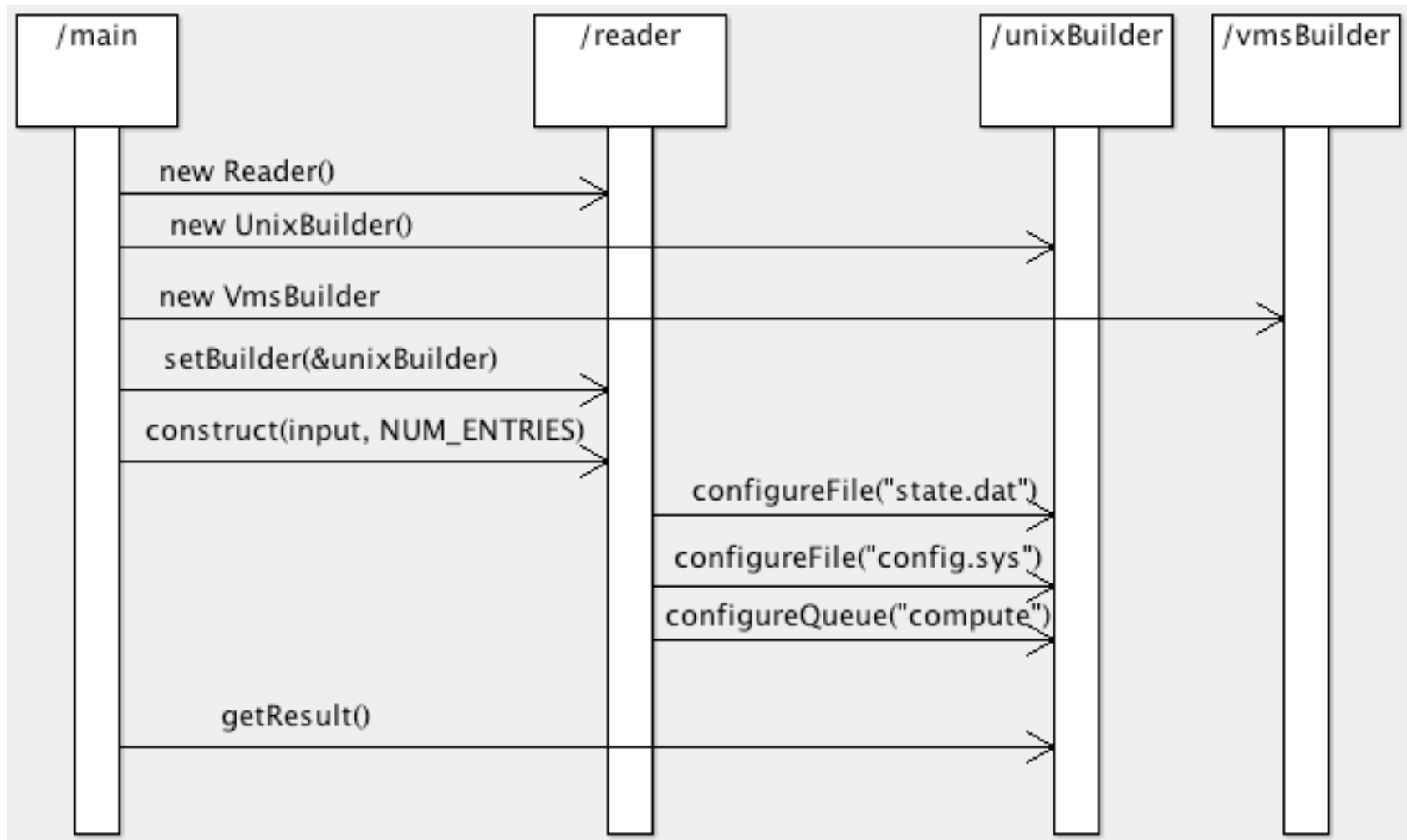

Clasa Director (Reader)

```
class Reader
{
    public:
        void setBuilder(Builder *b)
        {
            __builder = b;
        }
        void construct(PersistenceAttribute[], int);
    private:
        Builder *__builder;
};
```

metoda construct()

```
void Reader::construct(PersistenceAttribute list[], int num)
{
    for (int i = 0; i < num; i++)
        if (list[i].type == File)
            _builder->configureFile(list[i].value);
        else if (list[i].type == Queue)
            _builder->configureQueue(list[i].value);
        else if (list[i].type == Pathway)
            _builder->configurePathway(list[i].value);
}
```

Demo – diagrama de colaborare



Demo - cod

```
int main()
{
    UnixBuilder unixBuilder;
    VmsBuilder vmsBuilder;
    Reader reader;

    reader.setBuilder(&unixBuilder);
    reader.construct(input, NUM_ENTRIES);
    cout << unixBuilder.getResult()->getState() << endl;

    reader.setBuilder(&vmsBuilder);
    reader.construct(input, NUM_ENTRIES);
    cout << vmsBuilder.getResult()->getState() << endl;
    return 0;
}
```

P00

Sablonul
Object Factory

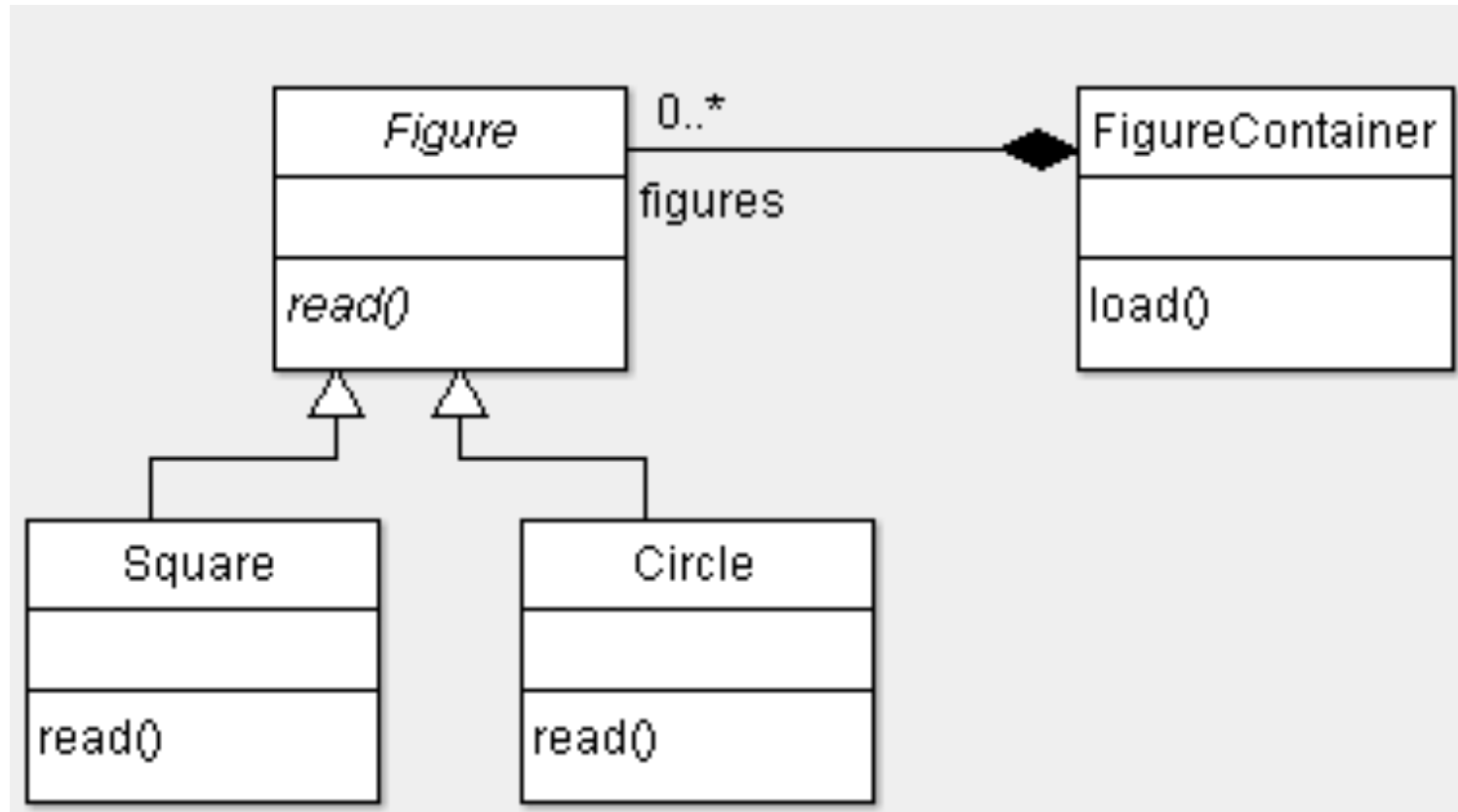
Cuprins

- 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

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

Circle::read()

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

Circle::read()

adaugarea unui nou tip de figura presupune modificarea acestui cod

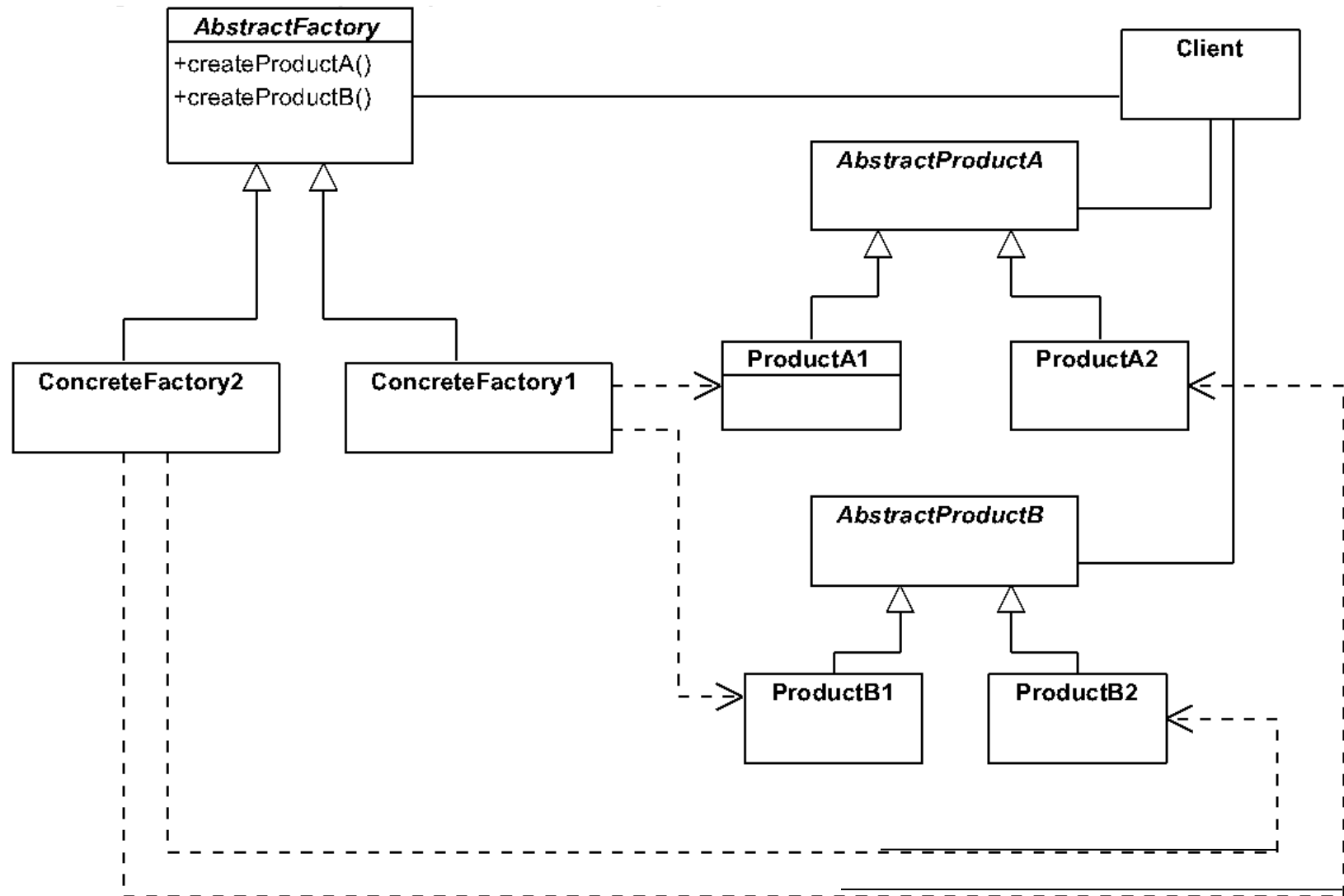
Principiul “inchis-deschis”

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 furnizezi o biblioteca de produse si vrei sa fie accesibila numai interfata, nu si implementarea

Fabrica de obiecte:: structura

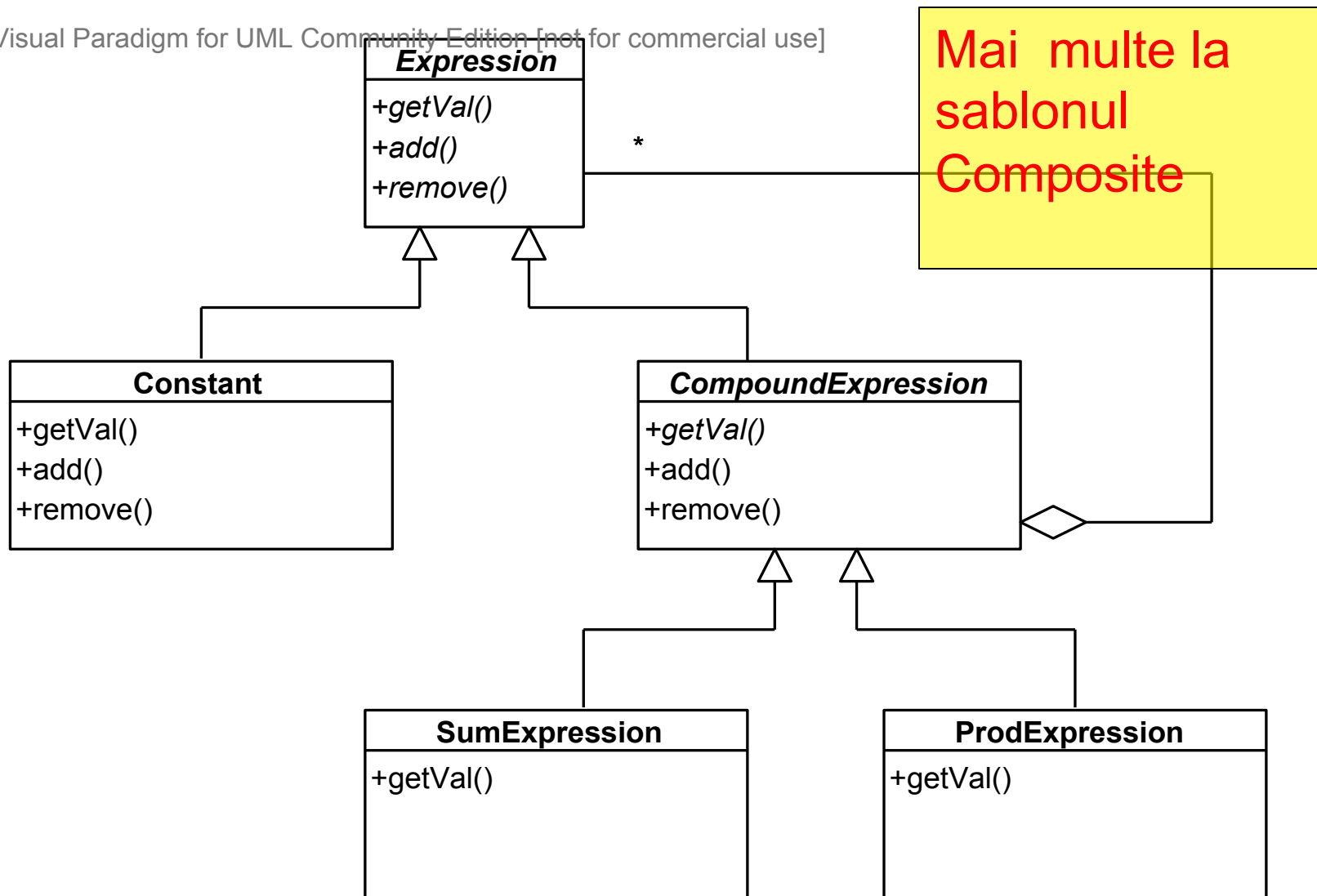


Fabrica de obiecte

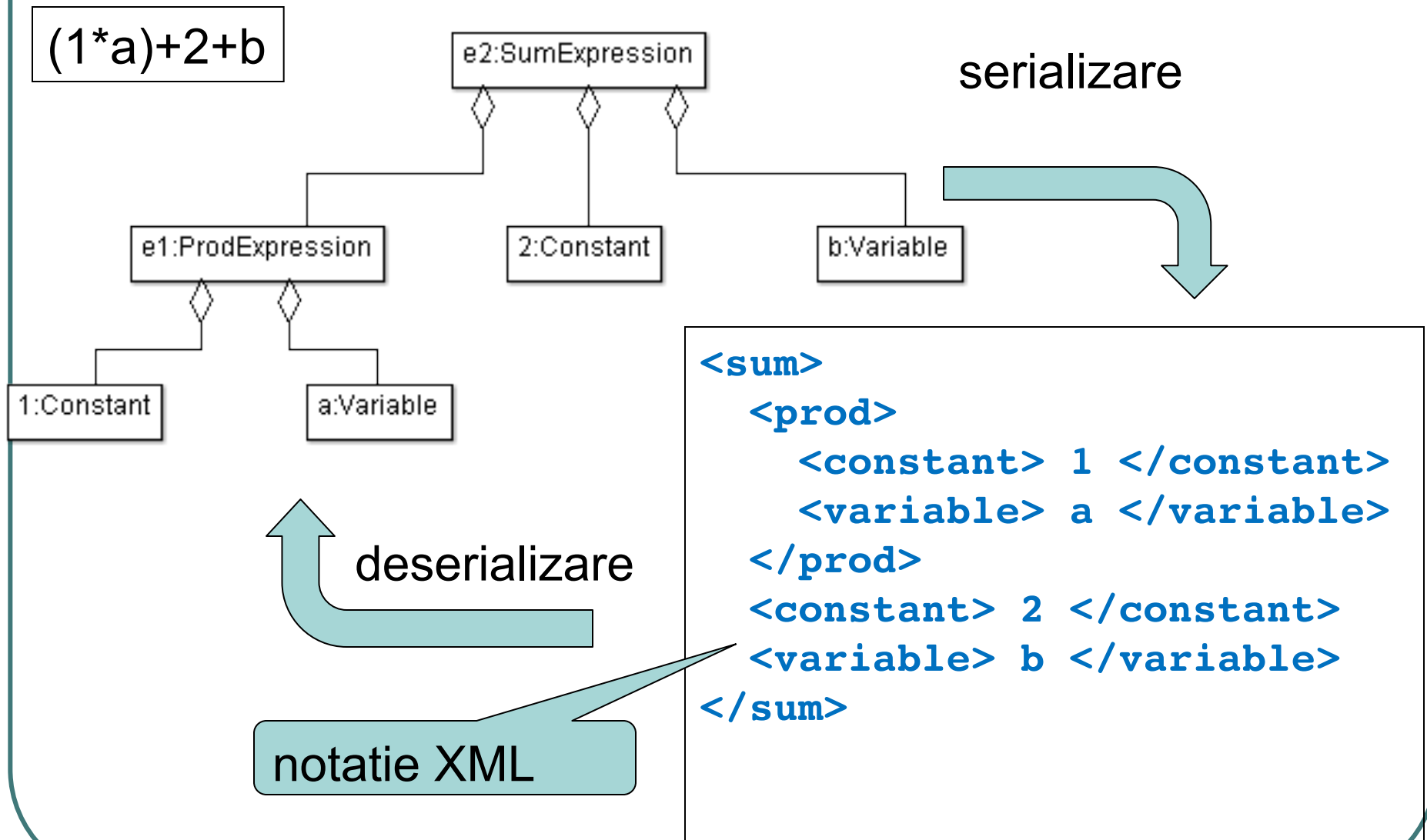
- colaborari
 - normal se creeaza o singura instanta
- Consecinte
 - izoleaza clasele concrete
 - simplifica schimbul familiei de produse
 - promoveaza consistenta printre produse
 - suporta noi timpul noi familii de produse usor
 - **respecta principiul deschis/inchis**
- implementare
 - se face pe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

Expresii : : structura

Visual Paradigm for UML Community Edition [not for commercial use]



Expressions: Problema (diagr. de obiecte)



Expressions: Problema

- serializare
 - vizitator
- deserializare

```
switch (tag)
```

```
{
```

```
    case <sum>:
```

```
        ...
```

```
    case <prod>:
```

```
        ...
```

```
    case <constant>:
```

```
        ...
```

```
    case <variable>:
```

```
        ...
```

```
}
```

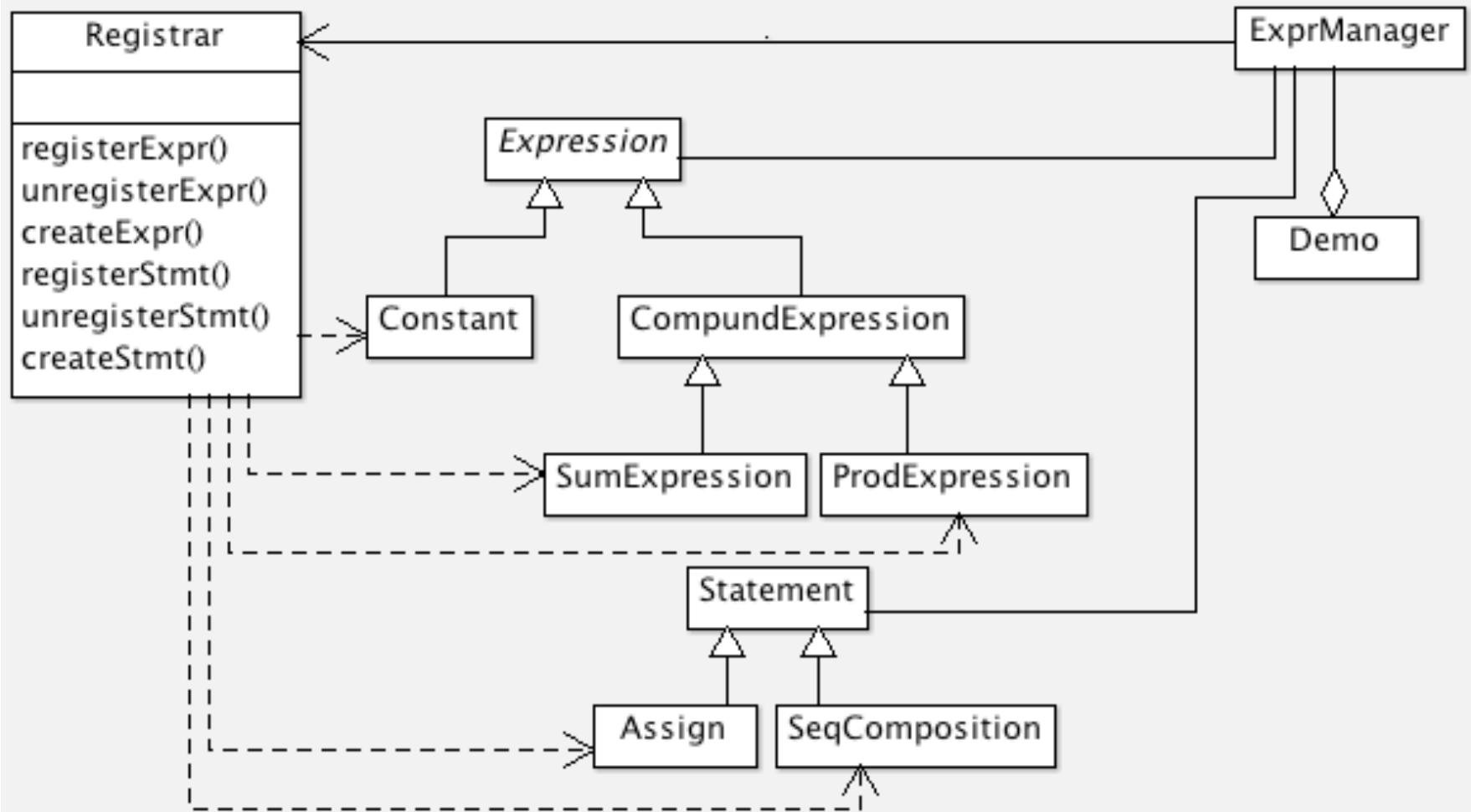
sablonul Visitor va fi
facut la cursurile
urmatoare

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)

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 Expression

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 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:

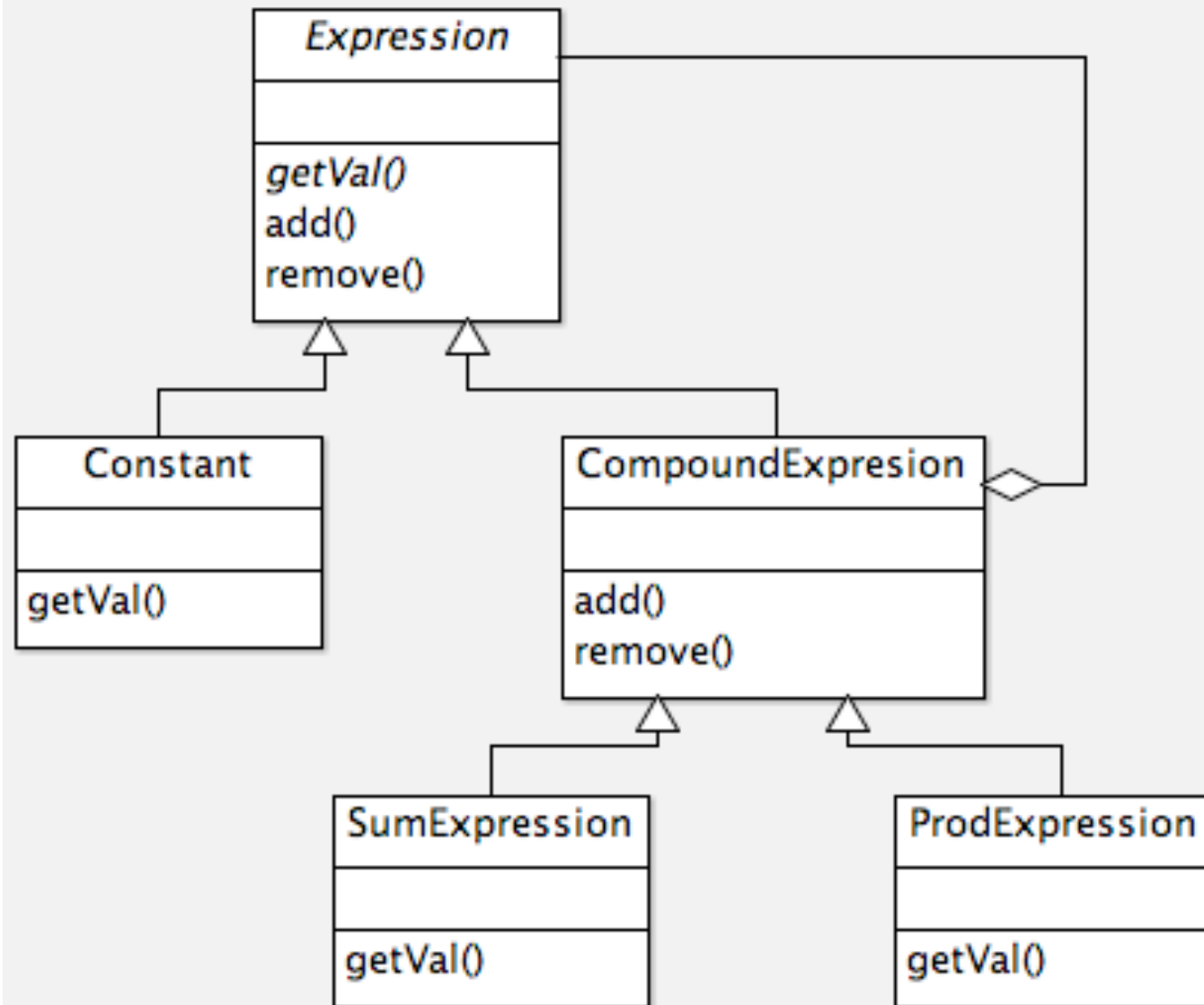
```
map<string, CreateExprFn> catalog;
```

```
};
```



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