POO

Sablonul

Composite

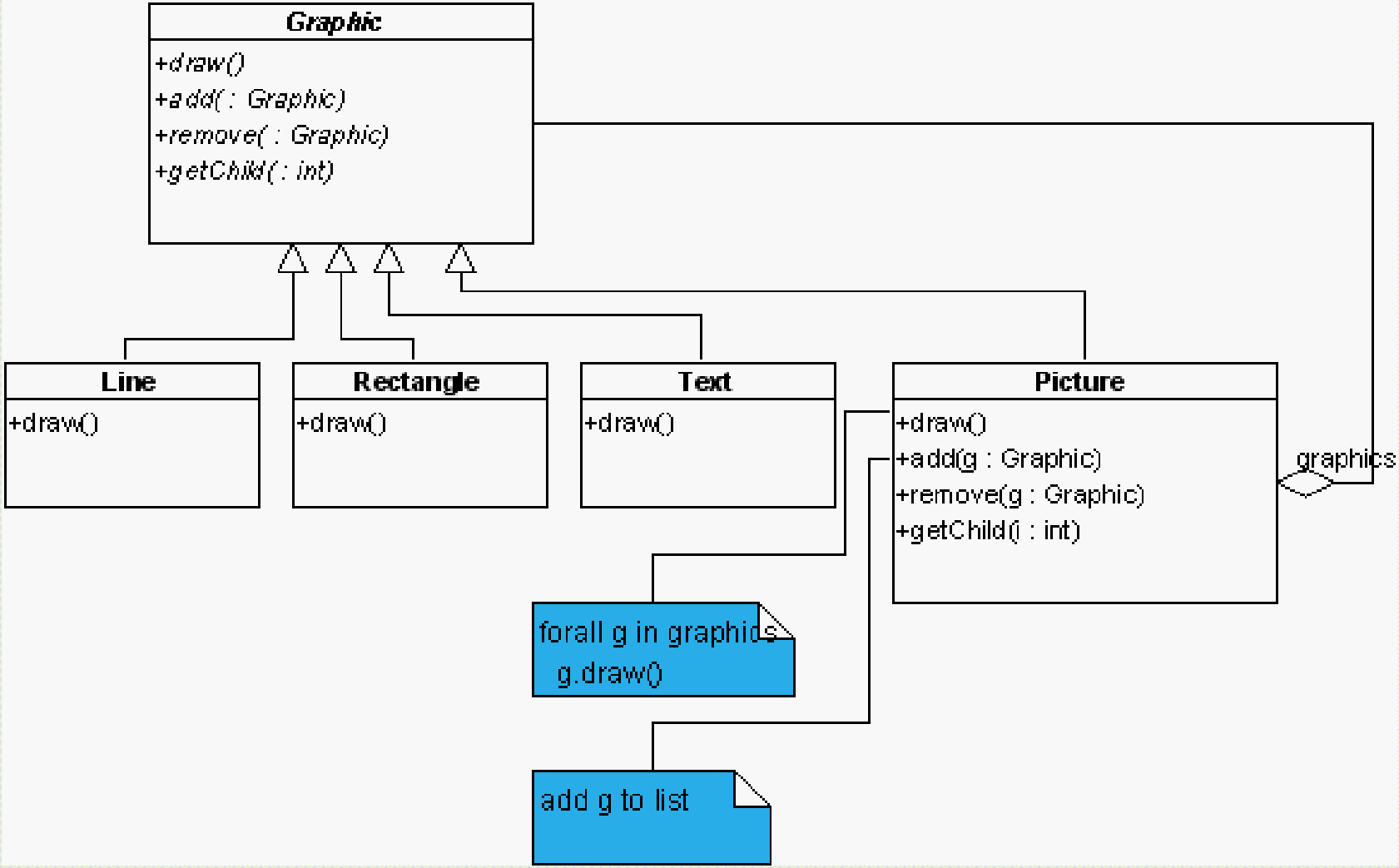
Cuprins

- Composite pattern
  (prezentare bazata pe GoF)
- studii de caz:
  - expresii

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



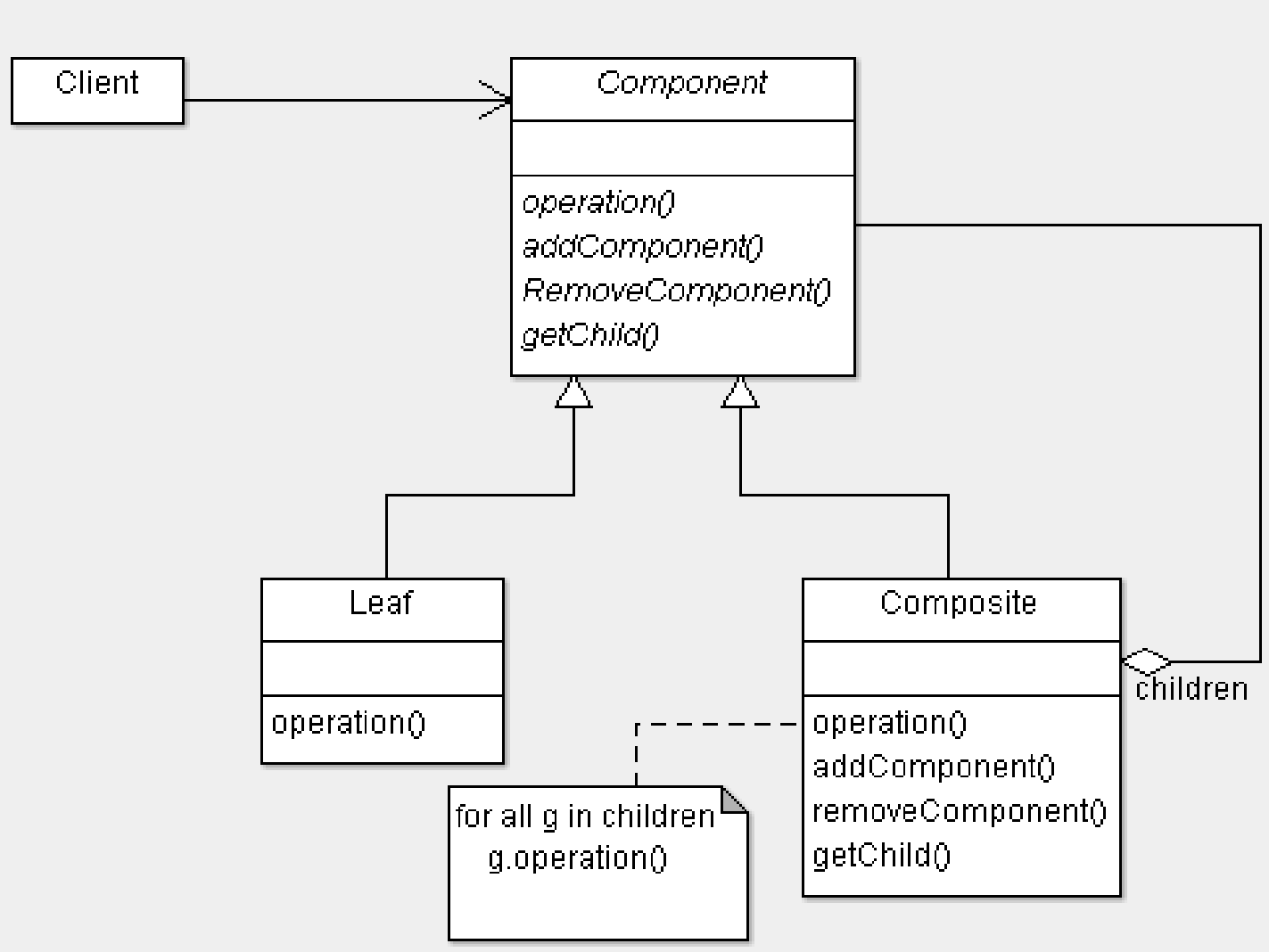
Composite:: caracterul recursiv al str.

- 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 sunt tratate uniform

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

- defineste o ierarhie de clase constand din obiecte primitive si compuse
- obiectele primitive pot fi compuse in obiecte mai complexe, care la randul lor pot fi compuse in alte obiecte mai complexe samd (recursie)
- ori de cate ori un client asteapta un obiect primitiv, el poate lua de asemnea si un obiect compus
- clientul e foarte simplu; el trateaza obiectele primitive si compuse in mod uniform
- clientului nu-i pasa daca are de-a face cu un obiect primitiv sau compus (evitarea utilizarii 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 (Flyweigth)

Composite::implementare

- *Maximizarea interfetei Component*
  - *Component* ar trebui sa implementeze cat mai multe operatii comune (pt *Leaf* si *Composite*)
  - aceste op 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 op. pt. *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 itermediul pointerului *this*

Implementarea metodei getComposite()

```
class Composite;
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 {
// ...
};
```

problema de tip “oul sau gaina”

implementarea implicita

implementarea pt. un compus

Exemplu utilizare a metodei getComposite()

```
Composite* aComposite = new Composite;
Leaf* aLeaf = new Leaf;
Component* aComponent;
Composite* test;

aComponent = aComposite;
if (test = aComponent->getComposite()) {
  test->Add(new Leaf);
}

aComponent = aLeaf;
if (test = aComponent->getComposite()) {
  test->Add(new Leaf);
}
```

crearea unui obiect compus si a unei frunze

adauga, pentru ca test va fi diferit de zero

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

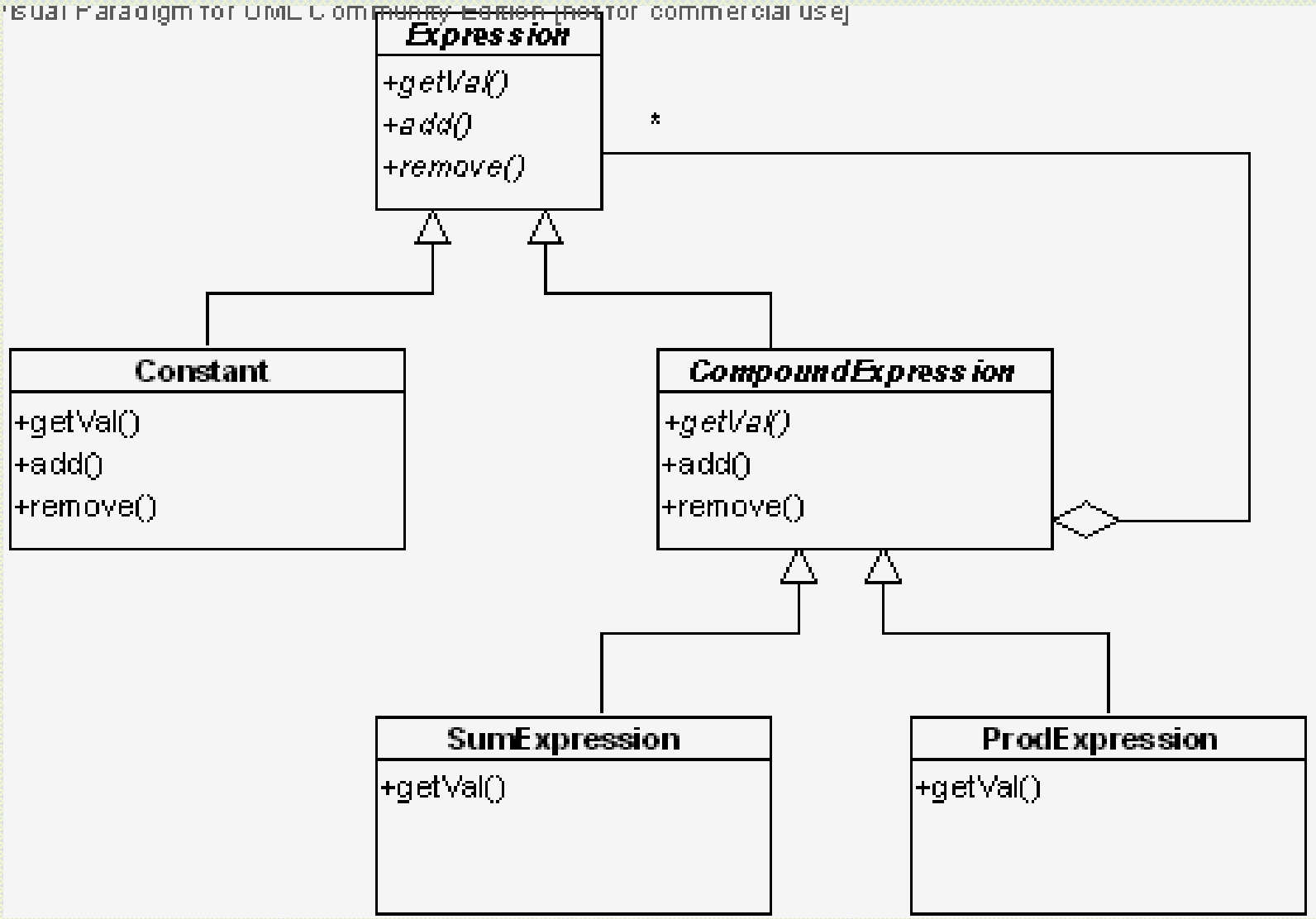
- *Ar trebui implementata o lista de fii in Component*?
  - ar fi tentant
  - dar …
  - ar fi irosire de spatiu
- *Ordinea copiilor*
  - sunt aplicatii in care conteaza
  - daca da, atunci accesul si managementul copiilor trebuie facut cu grija
- *Cine sterge componentele?*
  - fara GC, responsabilitatea 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 e1, e2, e3, … sunt expresii atunci suma lor
    e1 + e2 + e3 + … este expresie
  - daca e1, e2, e3, … sunt expresii atunci produsul lor
    e1 * e2 * e3 * … 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

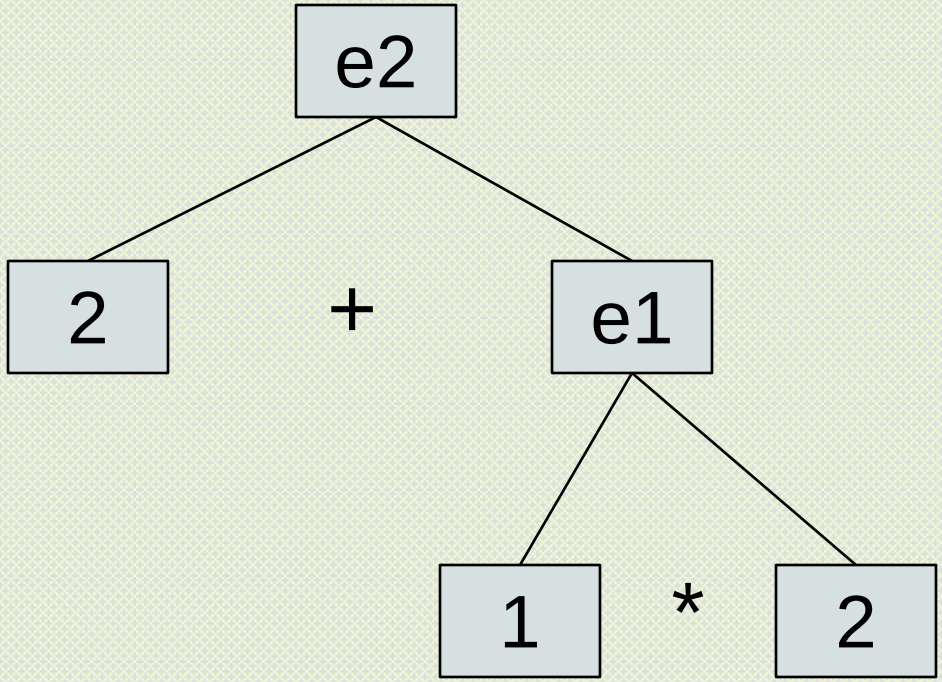
```
class SumExpression : public CompoundExpression
{
public:
   SumExpression() {}
   int getVal()
   {
       list<Expression*>::iterator i;
       int valTemp = 0;
       for ( i = members.begin();
                       i != members.end(); ++i)
            valTemp += (*i)->getVal();
       return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

declarare iterator pentru parcurgere componente

valoarea unei expresii suma este suma valorilor componentelor

Expresie de tip produs

Demo 1/2



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