
NFP121, Cnam/Paris
Cours 6-1
Les patrons
Composite, Interpréteur, Visiteur

jean-michel Douin, douin au cnam point fr
version : 12 Novembre 2019

Notes de cours

Sommaire pour les Patrons

- **Classification habituelle**

- **Créateurs**

- Abstract Factory, Builder, Factory Method Prototype Singleton

- **Structurels**

- Adapter Bridge Composite Decorator Facade Flyweight Proxy

- **Comportementaux**

- Chain of Responsibility. Command Interpreter Iterator

- Mediator Memento Observer State

- Strategy Template Method Visitor

Les patrons déjà vus en quelques lignes ...

- **Adapter**
 - Adapte l'interface d'une classe conforme aux souhaits du client
- **Proxy**
 - Fournit un mandataire au client afin de contrôler/vérifier ses accès
- **Observer**
 - Notification d'un changement d'état d'une instance aux observateurs inscrits
- **Template Method**
 - Laisse aux sous-classes une bonne part des responsabilités
- **Iterator**
 - Parcours d'une structure sans se soucier de la structure interne choisie
- **Strategy**
 - ...

Sommaire

- **Structures de données récursives**
 - Le patron Composite
- **Interprétation d'un composite**
 - Le patron Interpréteur
 - Un exemple prédéfini :
 - API Graphique en java(AWT), paquetage java.awt
- **Interprétation d'un composite**
 - Le patron Visiteur
- **Un exemple de composite : le langage *While***
 - Une mise en pratique
- **Annexes**
 - Parcours :
 - Itérateur et/ou visiteur

Principale bibliographie

- **GoF95**

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- Design Patterns, Elements of Reusable Object-oriented software Addison Wesley 1995

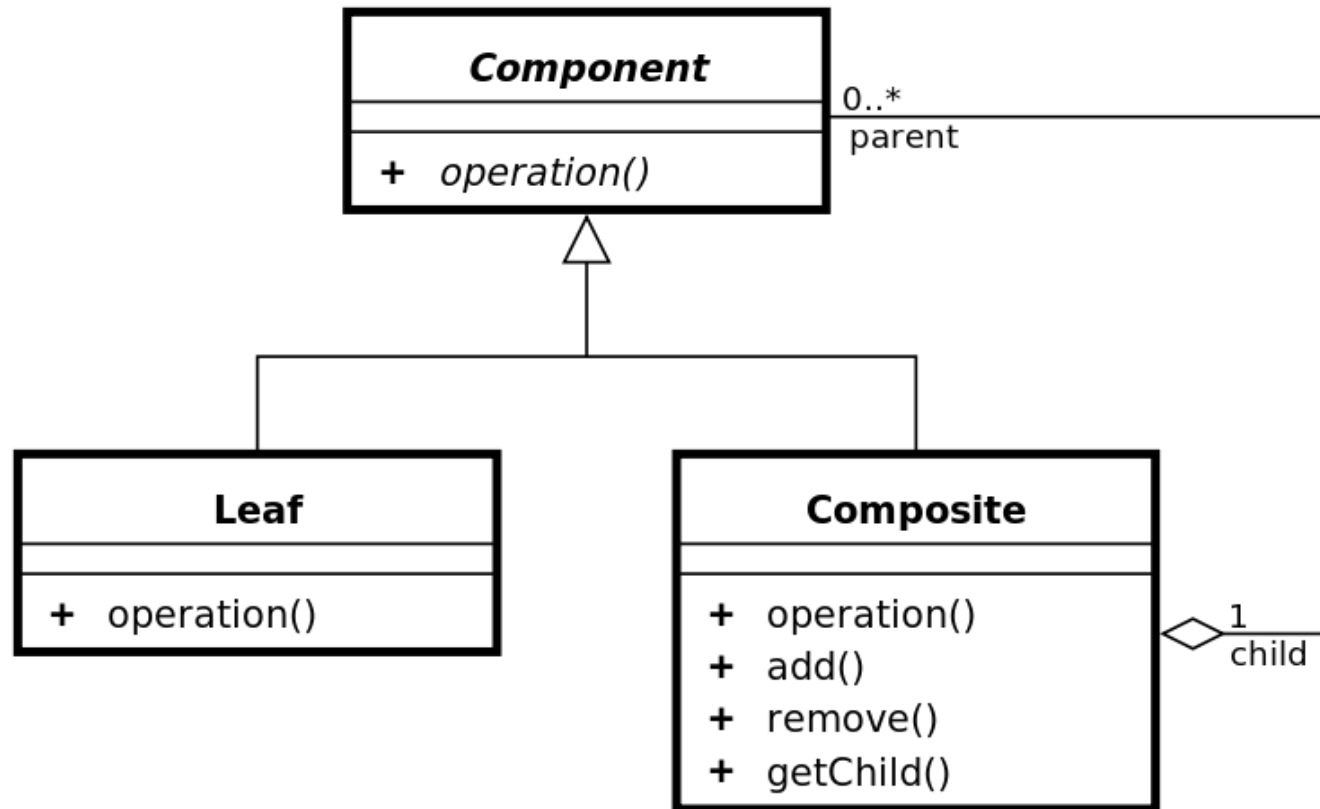
+

- <http://www.eli.sdsu.edu/courses/spring98/cs635/notes/composite/composite.html>
- <http://www.patterndepot.com/put/8/JavaPatterns.htm>

+

- <http://www.javaworld.com/javaworld/jw-12-2001/jw-1214-designpatterns.html>

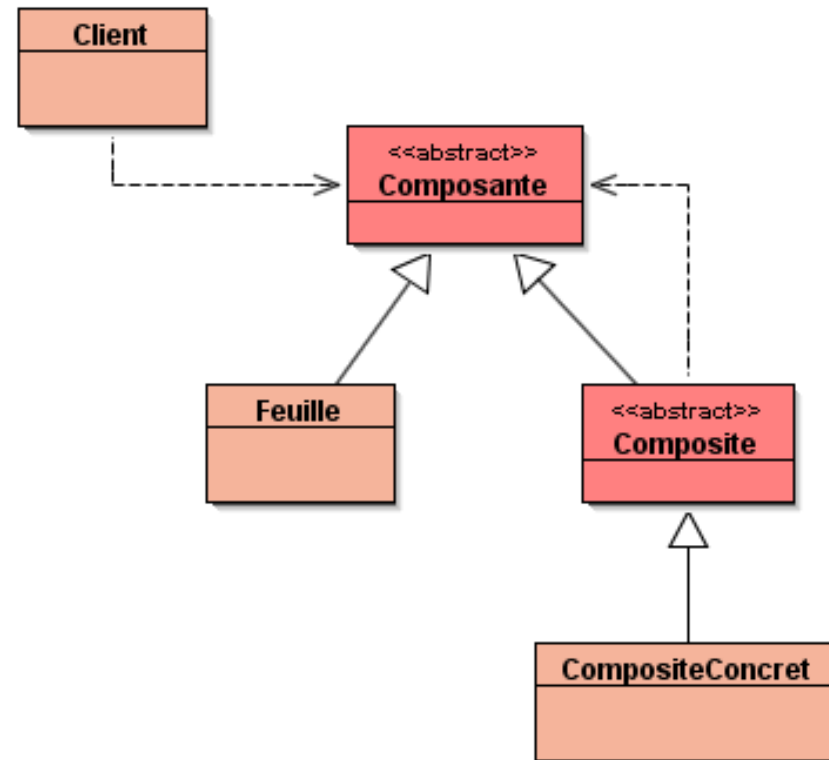
Le patron Composite



- L'original

Structures récursives : le pattern Composite

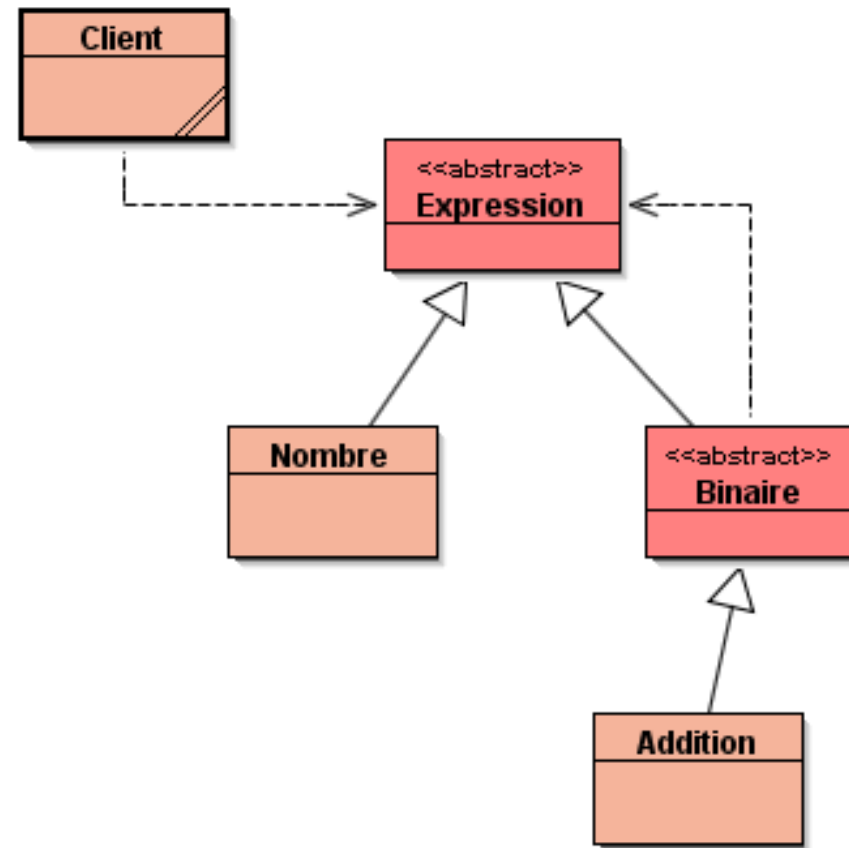
- Le client ne « voit » qu'une instance de *Composante*



Composante ::= Composite | Feuille
Composite ::= CompositeConcret
CompositeConcret ::= { Composante }
Feuille ::= 'symbole terminal'

Tout est Composante

Un exemple : Expression 3 , 3+2, 3+2+5,...



Expression ::= Binaire | Nombre

Binaire ::= Addition

Addition ::= Expression '+' Expression

Nombre ::= 'une valeur de type int'

Tout est Expression

Composite et Expression en Java

La Racine du composite : *Tout est Expression*

3 est un Nombre, est une Expression

3 + 2 est une addition, est Binaire, est une Expression

```
public abstract class Expression{ }
```

Au plus simple

Binaire est une Expression

```
public abstract class Expression{
```

```
public abstract class Binaire extends Expression{
```

```
    // binaire a deux opérandes ...
```

```
    protected Expression op1;
```

```
    protected Expression op2;
```

```
    public Binaire(Expression op1, Expression op2) {
```

```
        this.op1 = op1;
```

```
        this.op2 = op2;
```

```
    }
```

```
}
```

Addition est une opération Binaire

// Symbole non terminal

```
public class Addition extends Binaire{  
  
    public Addition(Expression op1, Expression op2) {  
        super(op1, op2);  
    }  
}
```

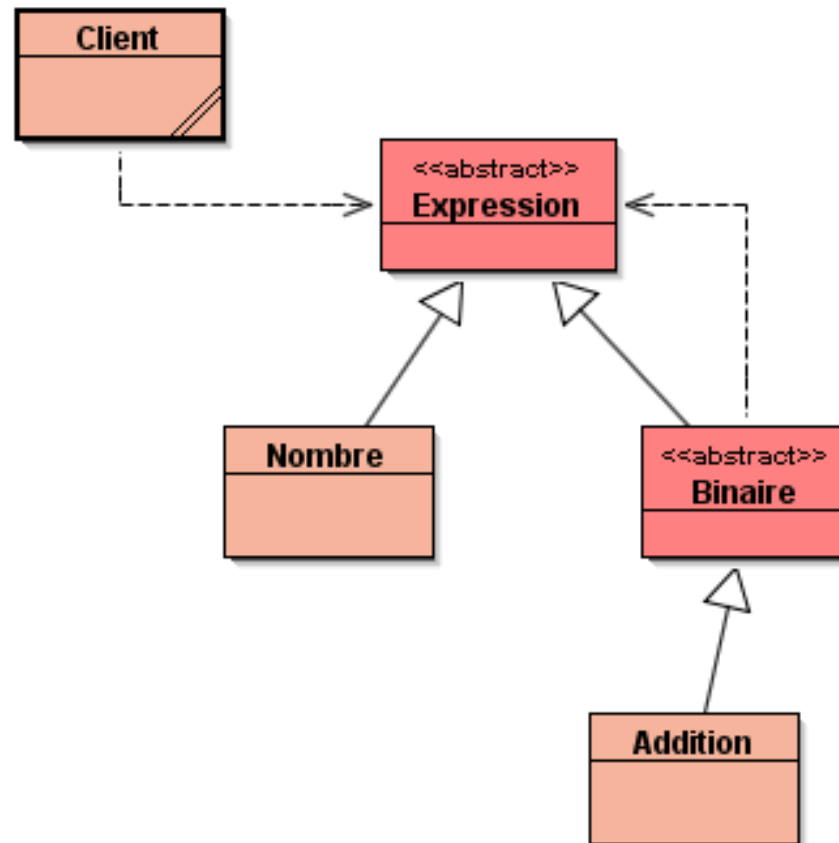
Soustraction, Division ...

Nombre est une Expression

// Symbole terminal

```
public class Nombre extends Expression{  
    private int valeur;  
    public Nombre(int valeur) {  
        this.valeur = valeur;  
    }  
}
```

Démonstration avec BlueJ



- Discussion, démonstration ...

Quelques instances d'Expression en Java

```
public class ExpressionTest extends junit.framework.TestCase {

    public static void test1(){

        Expression exp1 = new Nombre(321);

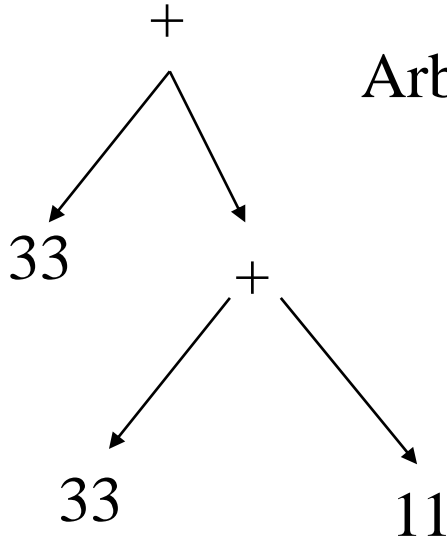
        Expression exp2 = new Addition(
            new Nombre(33),
            new Nombre(33)
        );

        Expression exp3 = new Addition(
            new Nombre(33),
            new Addition(
                new Nombre(33),
                new Nombre(11)
            )
        );

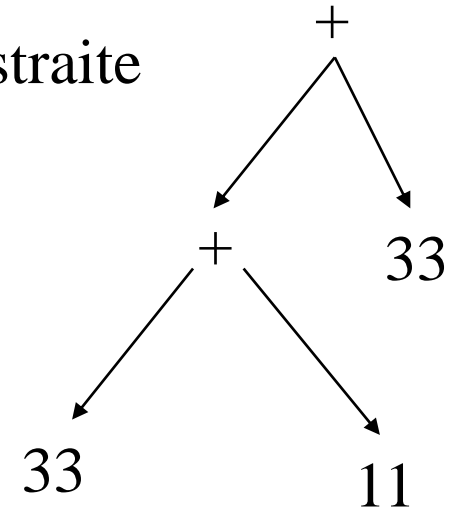
        Expression exp4 = new Addition(exp1,exp3);
    }
}
```

Discussion

Arbres de syntaxe abstraite

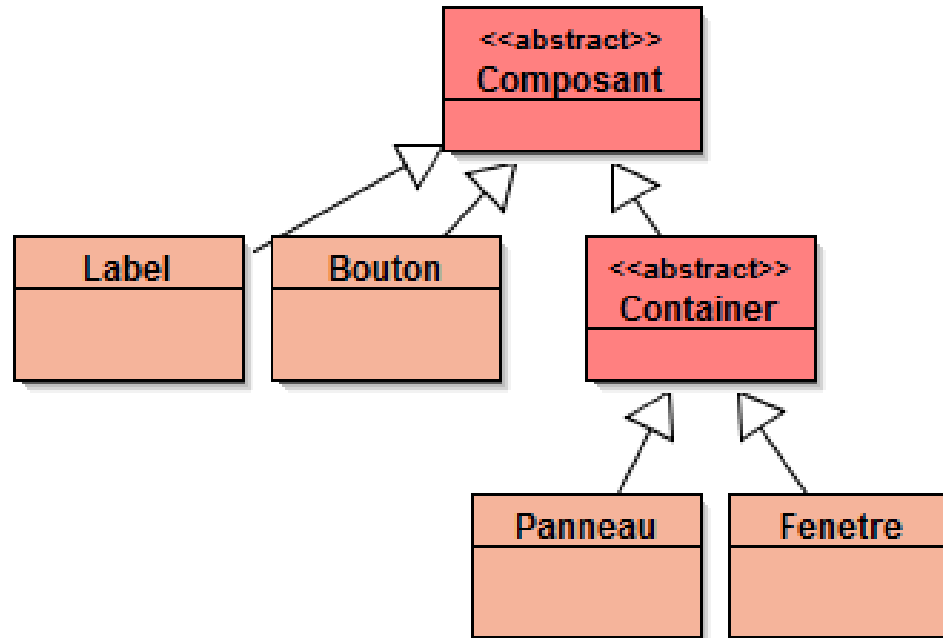


```
Expression exp3 =  
  new Addition(  
    new Nombre(33),  
    new Addition(  
      new Nombre(33),  
      new Nombre(11)  
    )  
  );
```



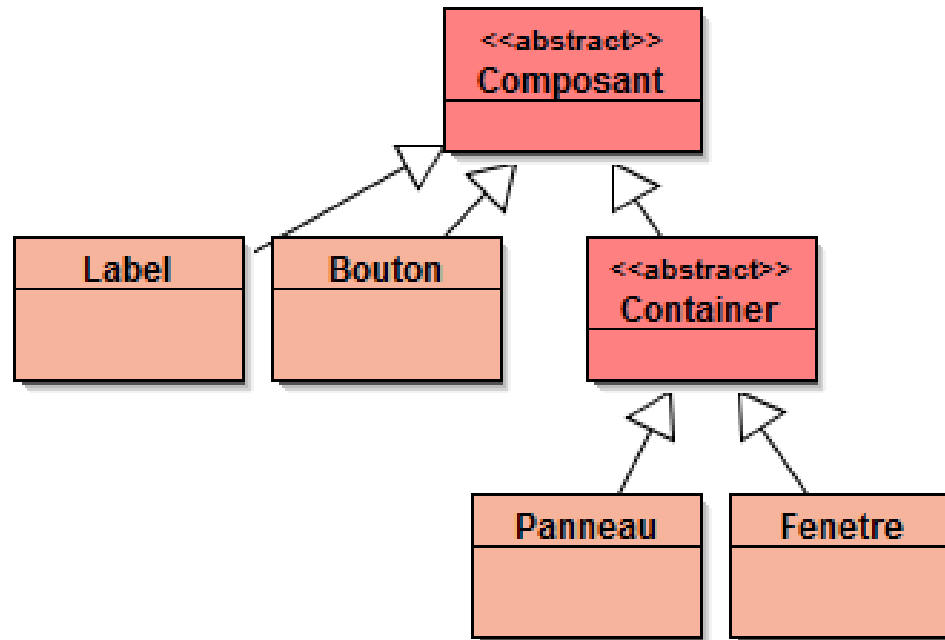
```
Expression exp3 =  
  new Addition(  
    new Addition(  
      new Nombre(33),  
      new Nombre(11)  
    ),  
    new Nombre(33)  
  );
```

Un autre exemple : une API graphique



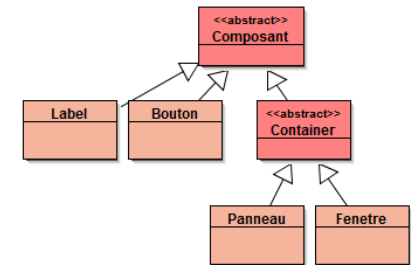
- $\text{Composant} ::= \text{Container} \mid \text{Feuille}$
- $\text{Container} ::= \text{Panneau} \mid \text{Fenetre}$
- $\text{Panneau} ::= \{\text{Composant}\}^*$
- $\text{Fenetre} ::= \{\text{Composant}\}^*$
- $\text{Feuille} ::= \text{Label} \mid \text{Bouton}$

Second exemple : une API graphique



```
public abstract class Composant{
    public abstract void dessiner();
}
```

API Graphique : une Feuille

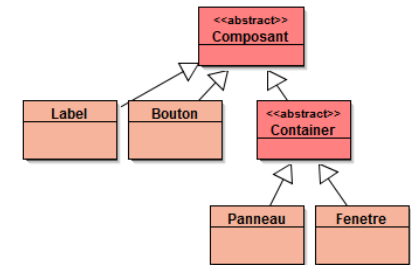


```
public class Label extends Composant{

    public void dessiner() {
        // dessin effectif d'un label
    }

}
```

API Graphique, Container

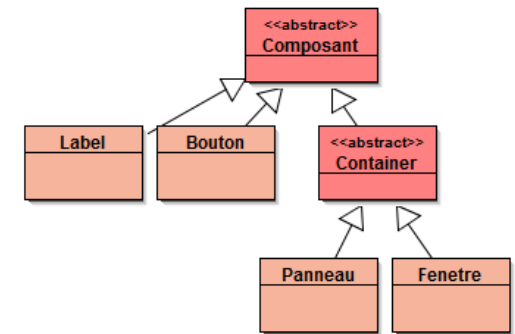


```
public abstract class Container extends Composant{
    protected List<Composant> liste;
```

```
    public void dessiner(){
        for(Composant c : liste){
            c.dessiner();
        }
    }
```

```
    public Container () {liste = new ArrayList<Composant>();}
    public void ajouter(Composant c) { liste.add(c); }
}
```

API Graphique, Panneau



```
public class Panneau extends Container{

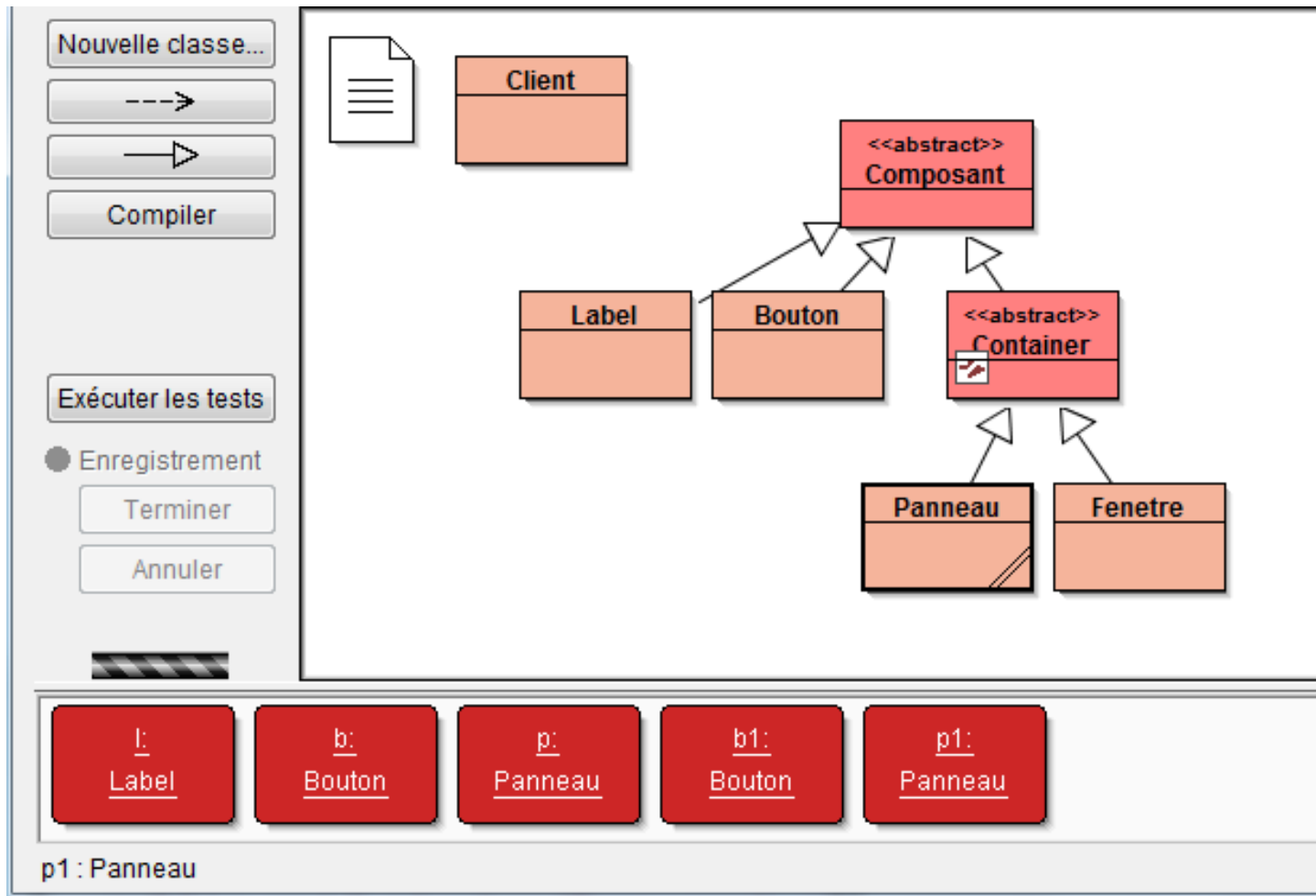
    public void dessiner(){
        // dessin effectif d'un Panneau

        super.dessiner();
        // dessin de ce qu'il contient

    }

}
```

Démonstration avec Bluej



L'AWT utilise-t-il le Pattern Composite ?

- **Component, Container, Label, JLabel**

Objets graphiques en Java

- **Comment se repérer dans une API de 180 classes (java.awt et javax.swing) ?**

La documentation : une liste (indigeste) de classes.
exemple

- **java.awt.Component**
 - Direct Known Subclasses:
 - **Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent**

+--java.awt.Component

|

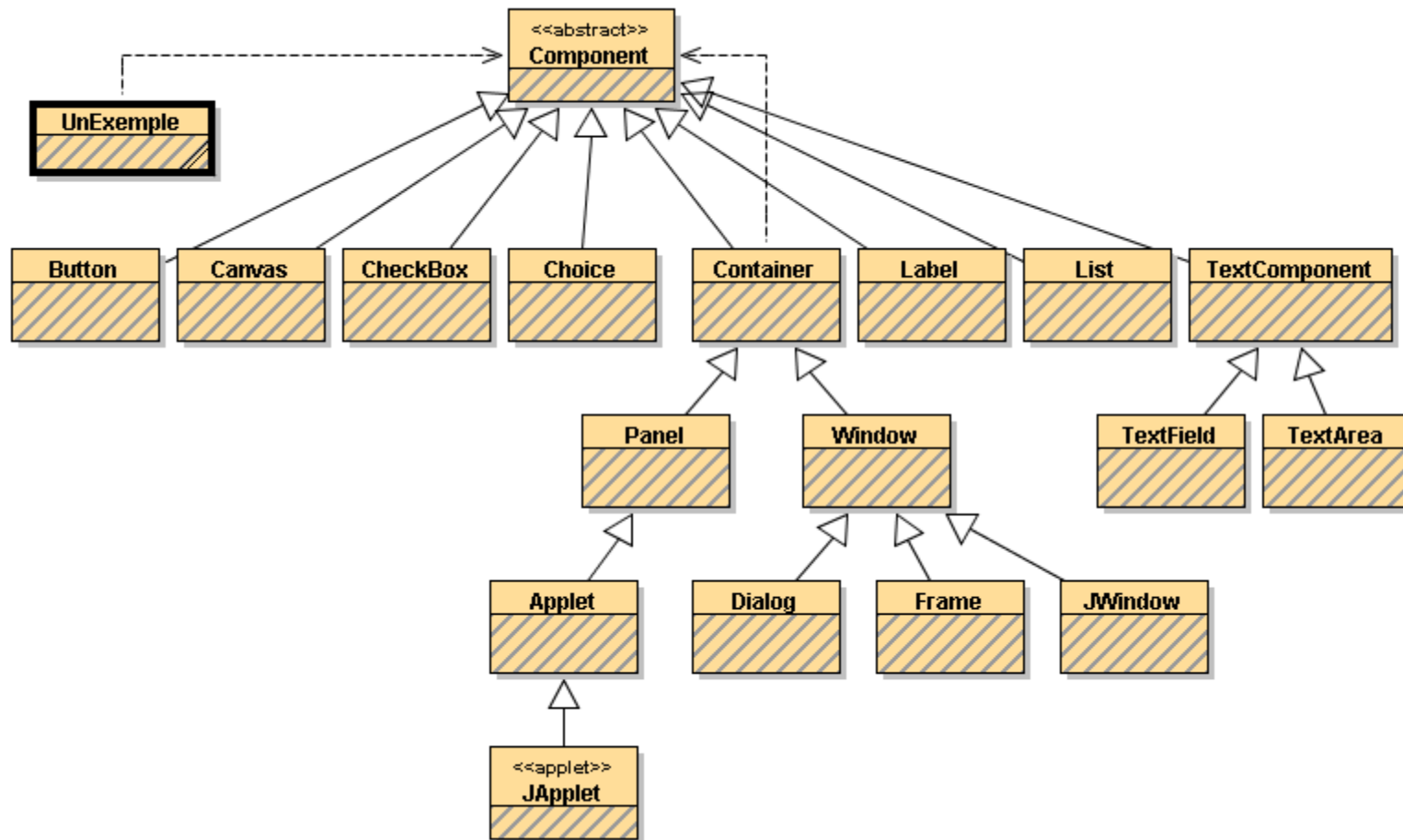
+--java.awt.Container

- Direct Known Subclasses:
 - **BasicSplitPaneDivider, CellRendererPane, DefaultTreeCellEditor.EditorContainer, JComponent, Panel, ScrollPane, Window**

Pattern Composite et API Java

- **API Abstract Window Toolkit**
 - Une interface graphique est constituée d'objets
 - De composants
 - Bouton, menu, texte, ...
 - De composites (composés des composants ou de composites)
 - Fenêtre, applet, ...
- **Le Pattern Composite est utilisé**
 - Une interface graphique est une expression respectant le Composite (la grammaire)
- **En Java au sein des paquetages
java.awt et de javax.swing.**

l'AWT utilise un Composite



- `class Container extends Component ...{`
 - `Component add (Component comp);`



Une Expression de type composite (Expression)

- Expression `e = new Addition(new Nombre(1),new Nombre(3));`

Une Expression de type composite (API AWT)

- Container `p = new Panel();`
- `p.add(new Button("b1 ")) ;`
- `p.add(new Button("b2 ")) ;`

Démonstration ici UnExemple

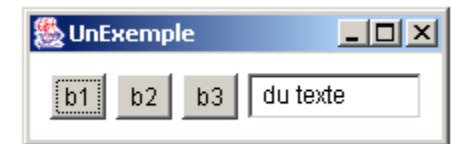
```
import java.awt.*;
public class UnExemple{

    public static void main(String[] args) {
        Frame f = new Frame("UnExemple");

        java.awt.Container p = new Panel();
        p.add(new Button("b1"));
        p.add(new Button("b2"));
        p.add(new Button("b3"));

        java.awt.Container p2 = new Panel();
        p2.add(p);p2.add(new TextField(" du texte"));

        f.add(p2);
        f.pack();f.setVisible(true);
    }
}
```



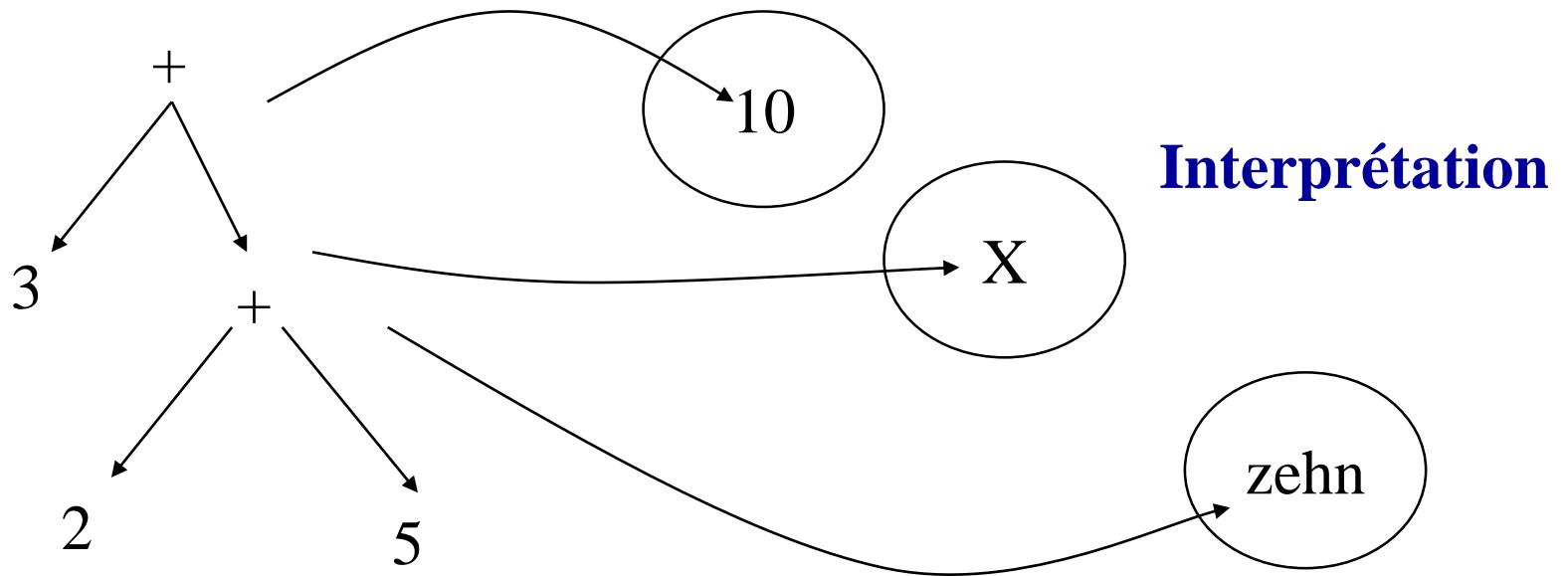
Le Pattern Expression : un premier bilan

- **Composite :**
 - Représentation de structures de données récursives
 - Techniques de classes abstraites
 - Liaison dynamique
 - Manipulation uniforme (tout est Composant)
 - C'est bien !, mais que peut-on en faire ?

Pourquoi faire ?

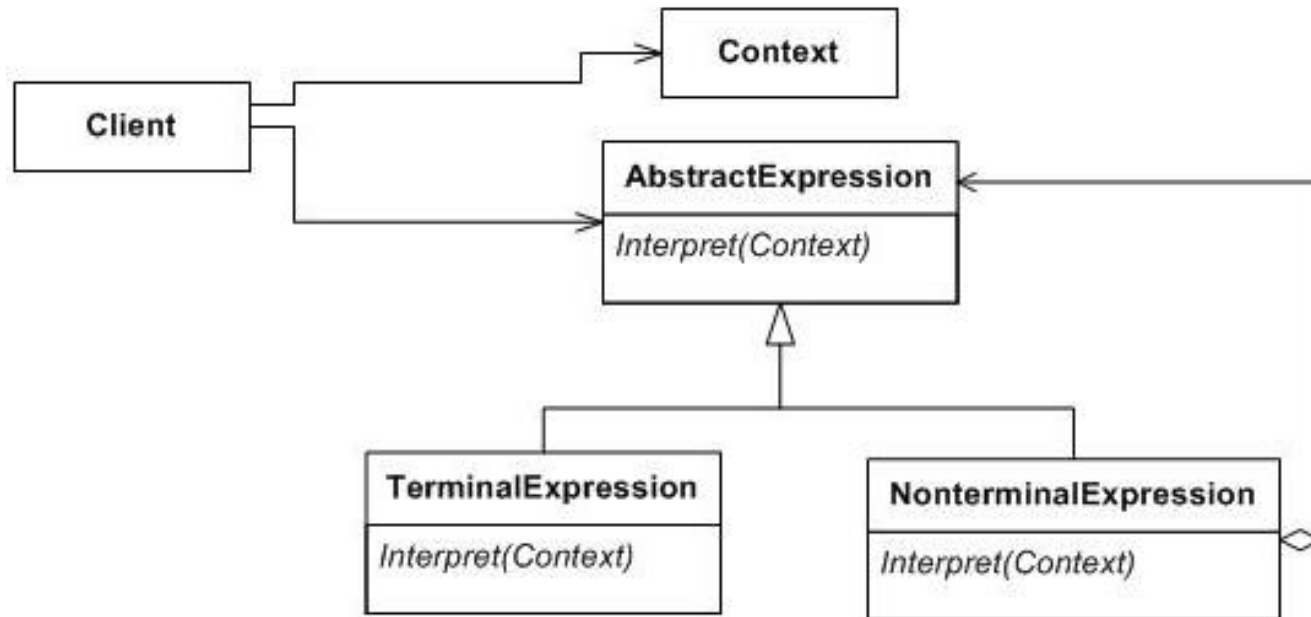
- **Un Composite**

- Une instance d'un composite est une représentation interne, parmi d'autres ...
 - Apparenté arbre de syntaxe abstraite
- Interprétation, transformation, évaluation, compilation,



- Une évaluation de cette structure : le Pattern Interpréteur
 - // 3+2
 - Expression **e** = new Addtition(new Nombre(3),new Nombre(2));
 - // appel de la méthode interpreter, installée dans la classe Expression
 - int resultat = **e**.interpreter();
 - assert(resultat == 5); // enfin

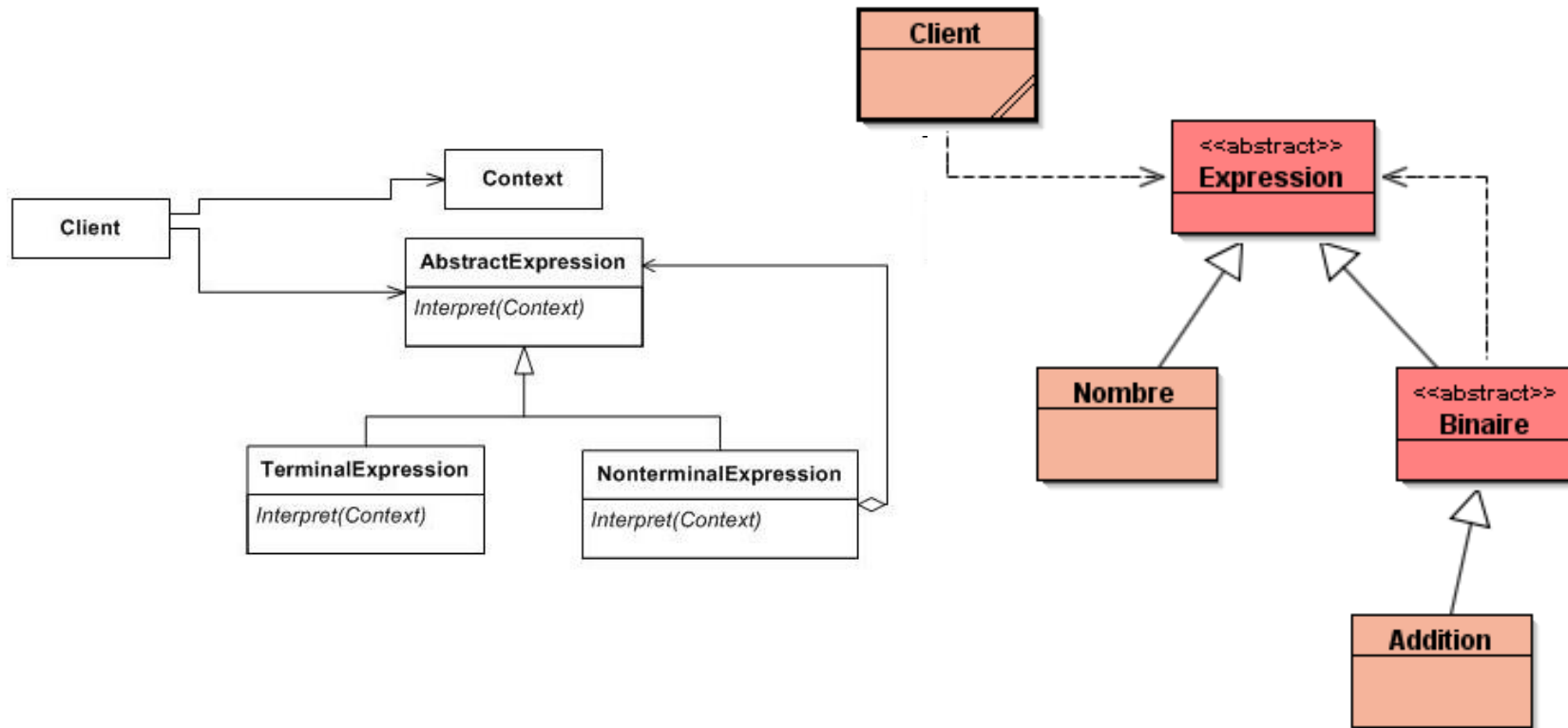
Pattern Interpréteur : l'original



- **De GoF95 ...**

- une interprétation, évaluation dans un certain contexte
- Le Composite est inclus

Le Pattern Interpréteur : Composite + évaluation



- Première version : chaque classe possède la méthode
 - **public int interpreter();**
 - abstraite pour les classes abstraites, concrètes pour les autres
 - ici pas de contexte (pour le moment)

Le pattern interpréteur

```
public abstract class Expression{
    abstract public int interpreter();
}

public abstract class Binaire extends Expression{
    protected Expression op1;
    protected Expression op2;

    public Binaire(Expression op1, Expression op2){
        this.op1 = op1;
        this.op2 = op2;
    }

    public Expression op1(){ return op1; }
    public Expression op2(){ return op2;}

    abstract public int interpreter();
}
```

Classe Nombre

```
public class Nombre extends Expression{  
    private int valeur;  
    public Nombre(int valeur){  
        this.valeur = valeur;  
    }  
  
    public int valeur(){ return valeur;}  
  
    public int interpreter();  
        return valeur;  
    }  
}
```

Le pattern interpréteur

```
public class Addition extends Binaire{  
  
    public Addition(Expression op1, Expression op2){  
        super(op1, op2);  
    }  
  
    public int interpreter(){  
        return op1.interpreter() + op2.interpreter();  
    }  
}
```

Interprétations simples

// quelques assertions

```
Expression exp1 = new Nombre(321);
```

```
int resultat = exp1.interpreter();
```

```
assert resultat == 321;
```

```
Expression exp2 = new Addition(  
    new Nombre(33),  
    new Nombre(33)  
);
```

```
resultat = exp2.interpreter();
```

```
assert resultat==66;
```

Quelques « interprétations » en Java

```
Expression exp3 = new Addition(  
    new Nombre(33),  
    new Addition(  
        new Nombre(33),  
        new Nombre(11)  
    )  
);
```

```
resultat = exp3.interpreter();  
assert resultat == 77;
```

```
Expression exp4 = new Addition(exp1, exp3);  
resultat = exp4.interpreter();  
assert resultat == 398;
```

```
}}
```

Démonstration/Discussion

- **Simple**
- **Mais**
 - **La demande d'interprétations reste vive...**

Interprétations suite

- **Une expression en String**
 - Préfixée, infixée, postfixée
- **Une expression en une suite d'instructions**
 - Pour une machine à pile,
 - Pour une machine à registres,
 - ..
- **Une expression évaluée à l'aide d'une pile**
 - suite à un traumatisme dû à un TP au Cnam sur des piles...
- **Une expression dans le système Bibi-binaire,**
 - HO, HA, HE, HI, BO, BA, BE, BI, KO, KA, KE, KI, DO, DA, DE, DI
 - 0 1 2 3 4 5 6 7 8 9 A B C D E F
 - Exemple 2000 -> 49C -> BOKADO
 - [https://fr.wikipedia.org/wiki/Système Bibi-binaire](https://fr.wikipedia.org/wiki/Système_Bibi-binaire) Bobby Lapointe

Démonstration ...

Evolution ... une galerie

- Une expression peut se calculer à l'aide d'une pile :

– Exemple : $3 + 2$ engendre
cette séquence `empiler(3)`
 `empiler(2)`
 `empiler(depiler() + depiler())`

Le résultat se trouve (ainsi) au sommet de la pile

→ **Nouvel entête de la méthode interpreter**

Evaluation sur une pile...

```
public abstract class Expression{  
    abstract public <T> void interpreter(PileI<T> p);  
}
```

Ou bien

```
public abstract class Expression{  
    abstract public <T> void interpreter(Stack<T> stk);  
}
```

Démonstration ...

– Mais ... encore !

Cette nouvelle méthode engendre une

modification de toutes les classes !!

Evolution ... bis

- **L'interprétation d'une expression utilise une mémoire**
 - **Le résultat de l'interprétation est en mémoire**

→ **Nouvel entête de la méthode interpreter**

```
public abstract class Expression{  
    abstract public <T> void interpreter(Memoire<T> p) ;  
}
```

→ mêmes critiques : modification de toutes les classes

.....

→ **Encore une modification de toutes les classes !**, la réutilisation prônée par l'usage des patrons est plutôt faible ...

Evolution... ter ... non merci !

- mêmes critiques : modification de toutes les classes
- Encore une modification de toutes les classes !, la réutilisation prônée par l'usage des patrons est de plus en plus faible ...
- Design pattern : pourquoi faire ?, utile/inutile,
 - *NFP121 ? Ai-je bien fait ?*
- Existe-t-il un patron afin de prévenir toute évolution future ?
du logiciel ...

Le pattern Visiteur vient à notre secours

- **Objectifs :**
 - Il faudrait pouvoir effectuer de multiples interprétations de la même structure **sans aucune modification du Composite**
- **→ Patron Visiteur**

Un **visiteur** par **interprétation** du composite

Dans la famille des itérateurs avez-vous le visiteur ? ...

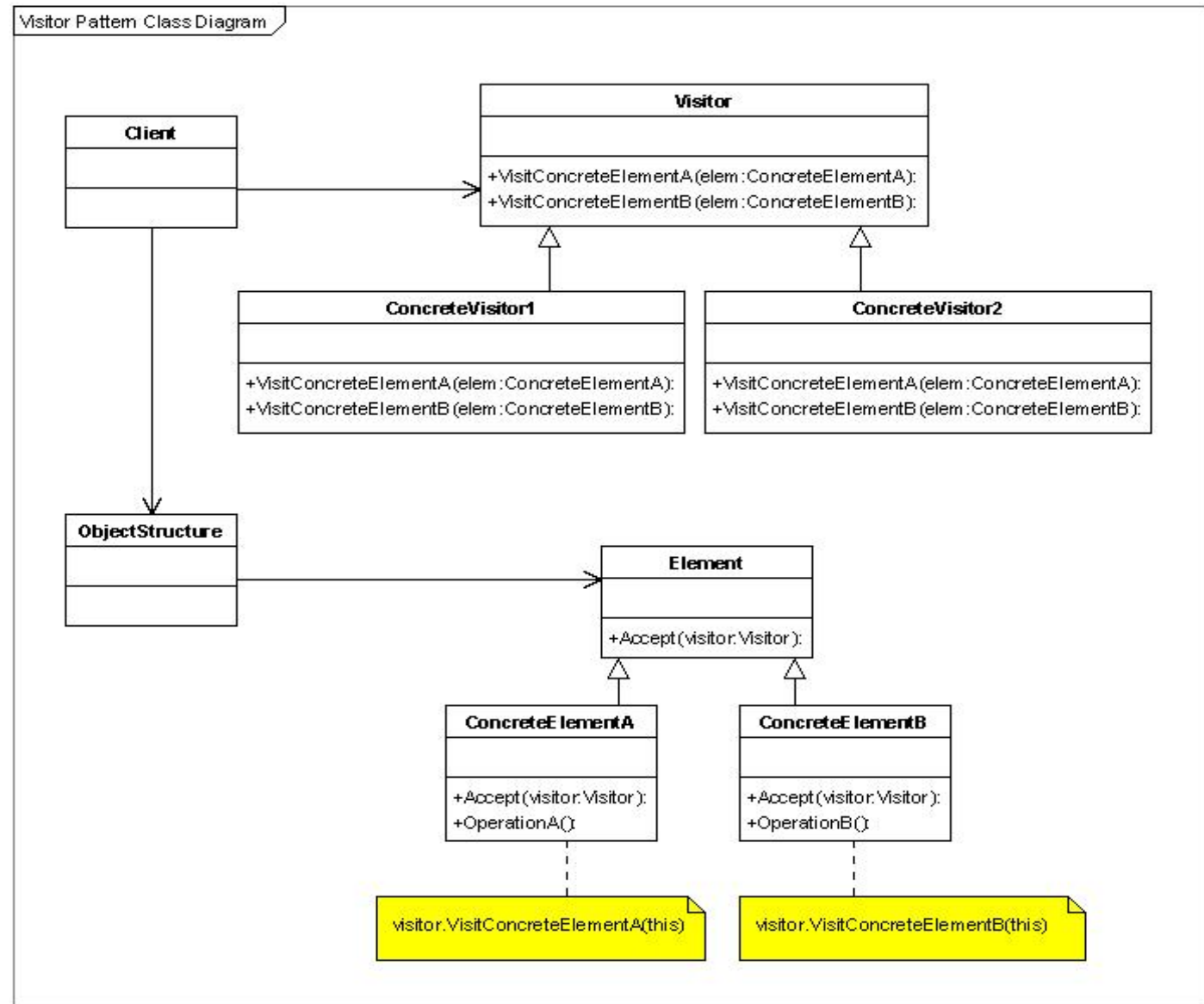
Patron visiteur

- **L'utilisateur de la classe Expression devra proposer ses Visiteurs**
 - VisiteurDeCalcul, VisiteurDeCalculAvecUnePile
 - Le problème change de main...
- **→ Ajout de cette méthode, dans toutes les classes**
 - ce sera la modification ultime

```
public abstract class Expression{  
    abstract public <T> T accepter(Visiteur<T> v) ;  
}
```

- **→ emploi de la généricité, pour le type retourné (<T> T)**

Le diagramme UML à l'origine



Created with Poseidon for UML Community Edition. Not for Commercial Use.

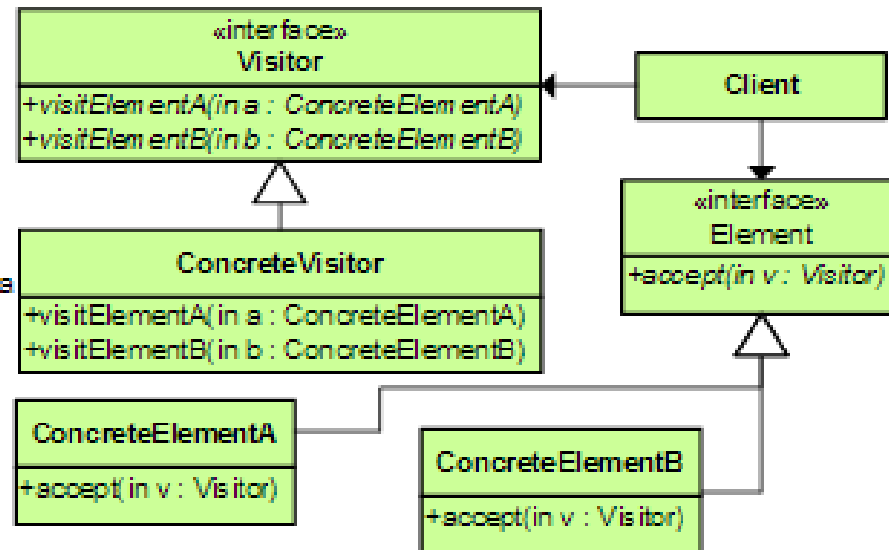
- **Lisible ?**
 - Peu ...
- **Un exemple ?**
 - volontiers

Le diagramme UML à l'origine

Visitor

Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.

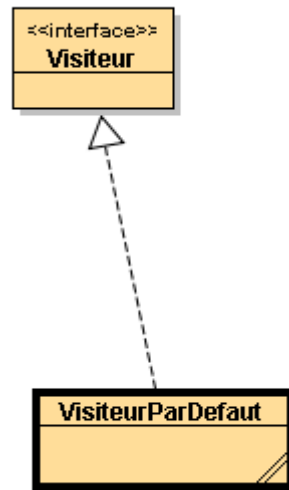


- **Lisible ?**
- http://jfod.cnam.fr/NFP121/supports/extras_designpatternscard.pdf
- **Un exemple au complet : les expressions**

Le pattern Visiteur une méthode par feuille*

```
public abstract interface Visiteur<T>{  
    public abstract T visiteNombre(Nombre n);  
    public abstract T visiteAddition(Addition a);  
}
```

```
public class VisiteurParDefaut<T> implements Visiteur<T>{  
    public T visiteNombre(Nombre n) {return null;}  
    public T visiteAddition(Addition a) {return null;}  
}
```



*feuille concrète
du composite

La classe Expression et Binaire

une fois pour toutes !

```
public abstract class Expression{
    abstract public <T> T accepter(Visiteur<T> v) ;
}

public abstract class Binaire extends Expression{
    protected Expression op1;
    protected Expression op2;

    public Binaire(Expression op1, Expression op2){
        this.op1 = op1;
        this.op2 = op2;
    }

    public Expression op1(){return op1;}
    public Expression op2(){return op2;}
    abstract public <T> T accepter(Visiteur<T> v) ;
}
```

La classe Nombre et Addition

une fois pour toutes

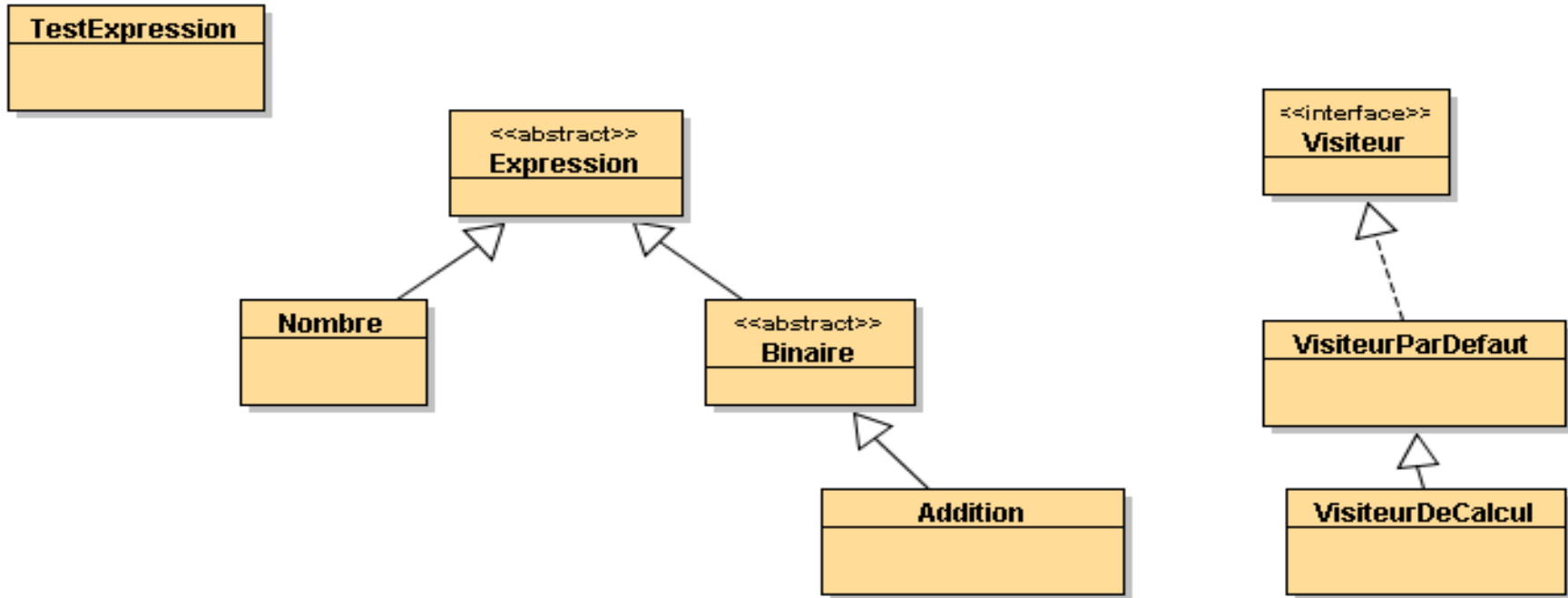
```
public class Nombre extends Expression{
    private int valeur;
    public Nombre(int valeur){
        this.valeur = valeur;
    }
    public int valeur(){ return valeur;}

    public <T> T accepter(Visiteur<T> v){
        return v.visiteNombre(this);
    }
}

public class Addition extends Binaire{
    public Addition(Expression op1, Expression op2){
        super(op1, op2);
    }

    public <T> T accepter(Visiteur<T> v){
        return v.visiteAddition(this);
    }
}
```

Le Composite a de la visite



Le Composite est figé

Toutes les visites
sont permises

Le VisiteurDeCalcul

```
public class VisiteurDeCalcul
    extends VisiteurParDefaut<Integer>{

    public Integer visiteNombre(Nombre n) {
        return n.valeur;
    }

    public Integer visiteAddition(Addition a) {
        Integer i1 = a.op1().accepter(this);
        Integer i2 = a.op2().accepter(this);
        return i1 + i2;
    }
}
```

La classe TestExpression

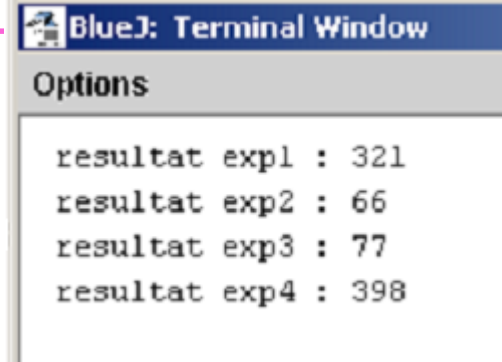
```
public class TestExpression{

    public static void main(String[] args){
        Visiteur vc = new VisiteurDeCalcul();
        Expression exp1 = new Nombre(321);
        System.out.println(" resultat exp1 : " + exp1.accepter(vc) );

        Expression exp2 = new Addition(
            new Nombre(33),
            new Nombre(33)
        );
        System.out.println(" resultat exp2 : " + exp2.accepter(vc) );

        Expression exp3 = new Addition(
            new Nombre(33),
            new Addition(
                new Nombre(33),
                new Nombre(11)
            )
        );
        System.out.println(" resultat exp3 : " + exp3.accepter(vc) );

        Expression exp4 = new Addition(exp1,exp3);
        System.out.println(" resultat exp4 : " + exp4.accepter(vc) );
    }
}
```



BlueJ: Terminal Window

Options

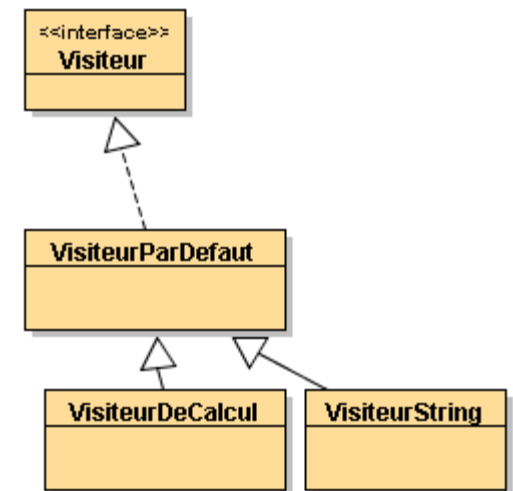
```
resultat exp1 : 321
resultat exp2 : 66
resultat exp3 : 77
resultat exp4 : 398
```

Le Composite a une autre visite

```
public class VisiteurString
    extends VisiteurParDefault<String>{

    public String visiteNombre(Nombre n) {
        return Integer.toString(n.valeur());
    }

    public String visiteAddition(Addition a) {
        String i1 = a.op1().accepter(this);
        String i2 = a.op2().accepter(this);
        return "(" + i1 + " + " + i2 + ")";
    }
}
```



La classe TestExpression re-visitée

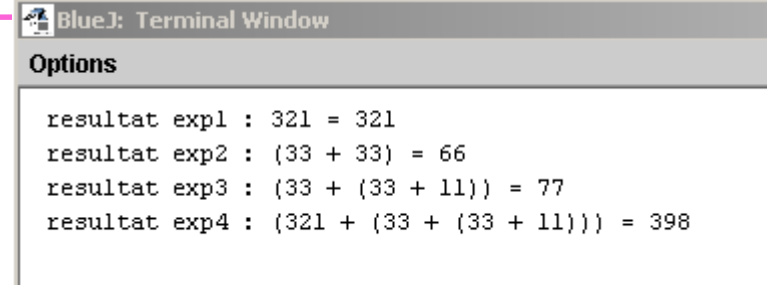
```
public class TestExpression{

    public static void main(String[] args){
        Visiteur<Integer> vc = new VisiteurDeCalcul();
        Visiteur<String> vs = new VisiteurString();
        Expression exp1 = new Nombre(321);
        System.out.println(" resultat exp1 : " + exp1.accepter(vs) + " = " +
                           exp1.accepter(vc));

        Expression exp2 = new Addition(new Nombre(33),new Nombre(33));
        System.out.println(" resultat exp2 : " + exp2.accepter(vs) + " = " +
                           exp2.accepter(vc));

        Expression exp3 = new Addition(
                                new Nombre(33),
                                new Addition(new Nombre(33),new Nombre(11))
                            );
        System.out.println(" resultat exp3 : " + exp3.accepter(vs) + " = " +
                           exp3.accepter(vc));

        Expression exp4 = new Addition(exp1,exp3);
        System.out.println(" resultat exp4 : " + exp4.accepter(vs) + " = " +
                           exp4.accepter(vc));
    } }
```



BlueJ: Terminal Window

Options

```
resultat exp1 : 321 = 321
resultat exp2 : (33 + 33) = 66
resultat exp3 : (33 + (33 + 11)) = 77
resultat exp4 : (321 + (33 + (33 + 11))) = 398
```


Le Composite pourrait avoir d'autres visites

```
public class AutreVisiteur extends VisiteurParDefaut<T>{

    public T visiteNombre(Nombre n) {
        // une implémentation
    }

    public T visiteAddition(Addition a) {
        // une implémentation
    }
}
```

Le pattern Visiteur

- **Contrat rempli :**

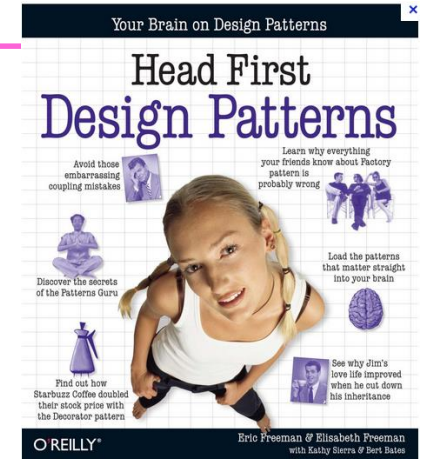
- Aucune modification du composite :-> couplage faible entre la structure et son analyse
- Tout type de visite est à la charge du client :
 - tout devient possible ...

Mais

- Convient aux structures qui n'évoluent pas ou peu
 - Une nouvelle feuille du pattern Composite engendre une nouvelle redéfinition de tous les visiteurs
- Alors à suivre...
 - Une proposition sera faite lors du cours sur l'introspection...

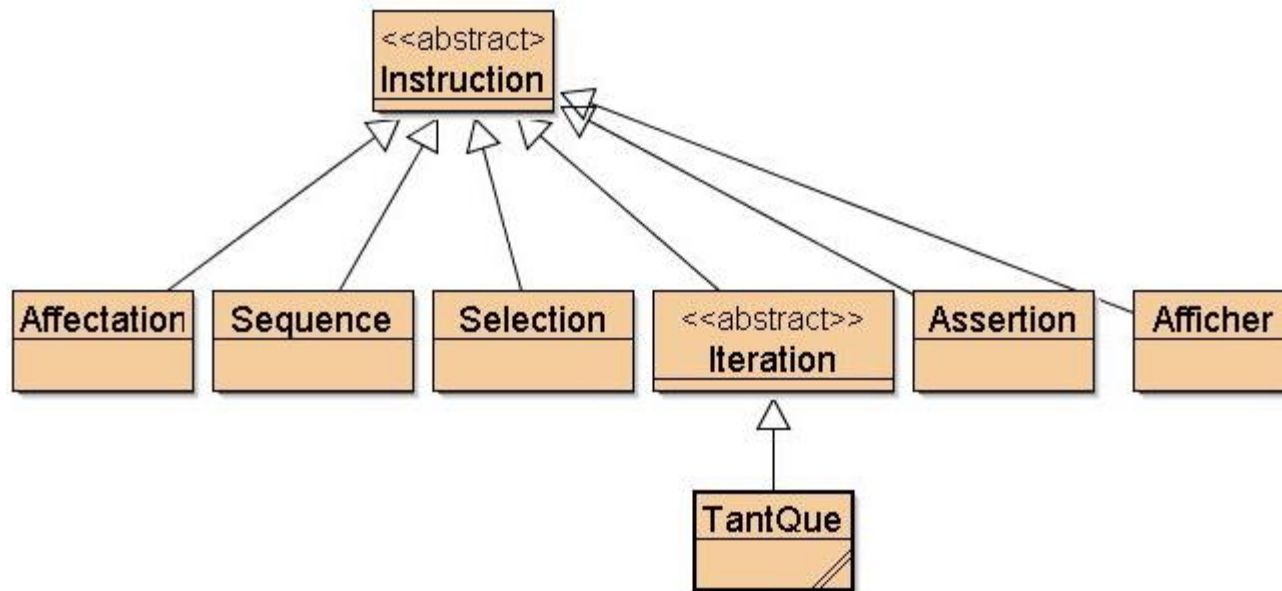
Discussion

- Composite
- Interpréteur
- Visiteur
 - <https://today.java.net/pub/a/today/2008/04/10/source-code-analysis-using-java-6-compiler-apis.html>
- Itérateur, est-ce un oubli ?
 - Voir en annexe
 - (prévoir un aspirine ... code extrait de tête la première, migraine en vue)
 - <http://www.sws.bfh.ch/~amrhein/ADP/HeadFirstDesignPatterns.pdf>



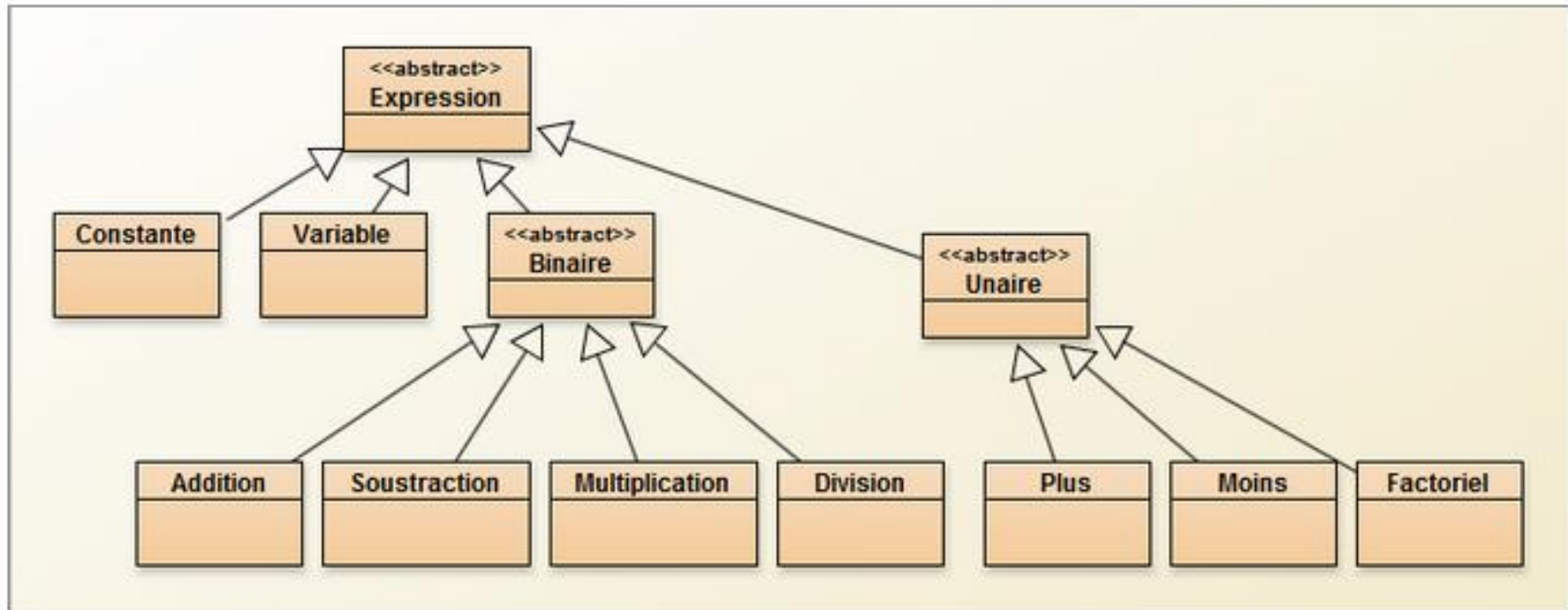
While Un petit langage, le tp

- Composites pour un tout petit langage impératif
 - Instructions
 - Affectation, Séquence, Selection, Itération
 - Diverses : Assertion, Afficher



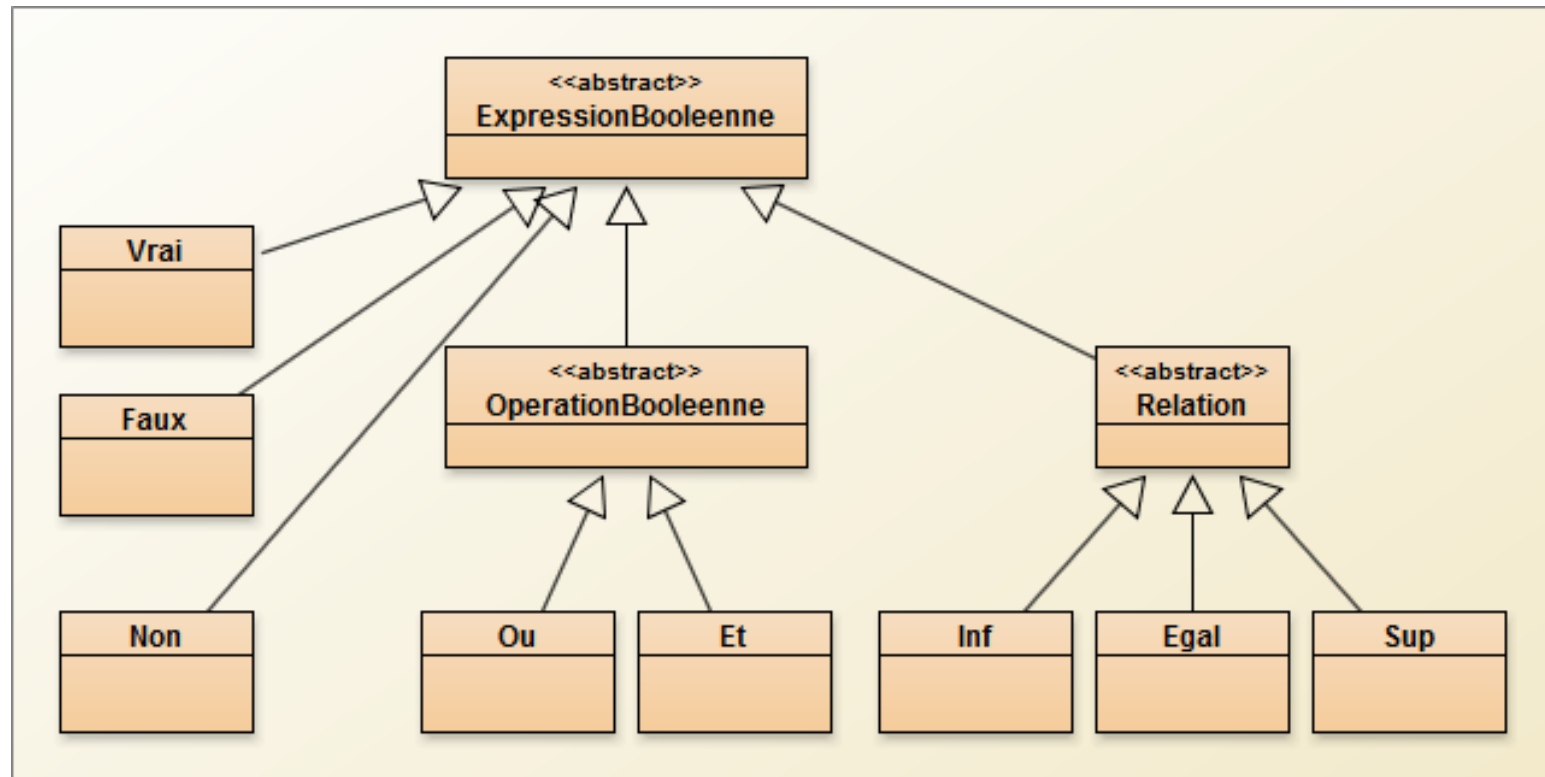
Expressions

- Composite Expressions arithmétiques (cf. ce support)

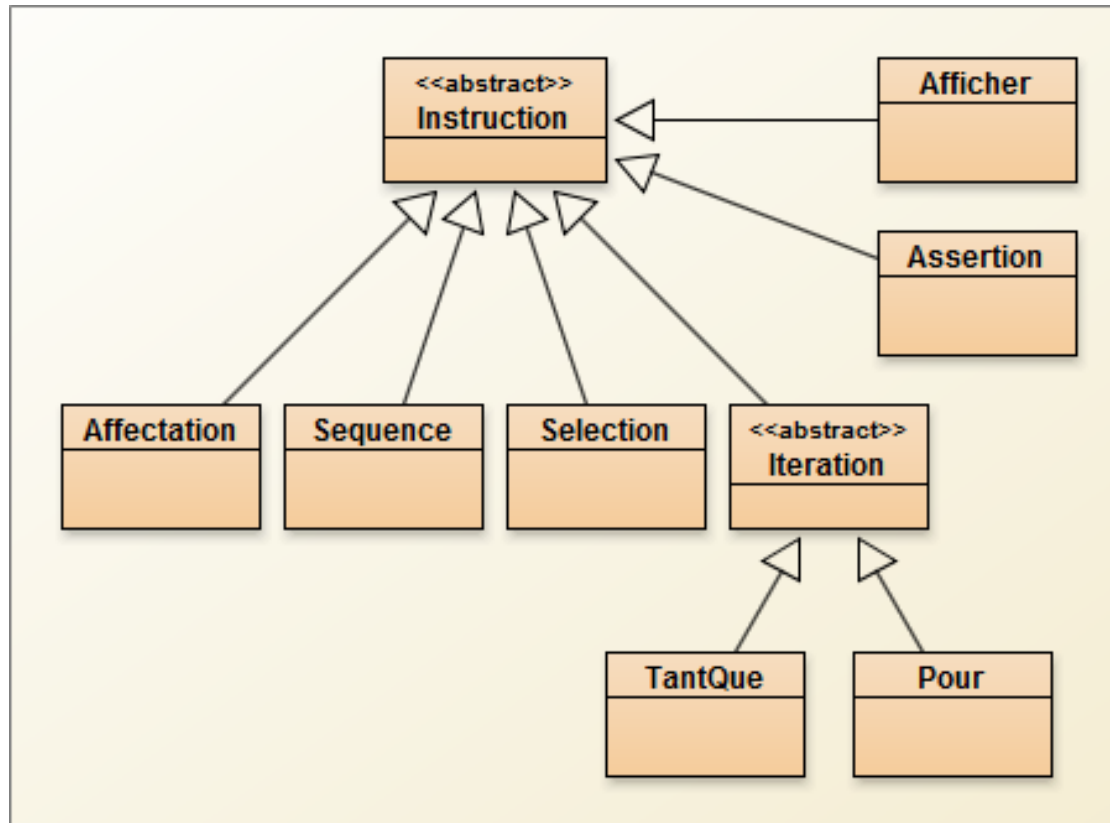


Expressions

- **Composite Expressions booléennes**



Instructions



- (Affectation =) (Sequence ;) (Selection si-alors si-alors-sinon)
- (Iteration tantQue Pour)
- (Afficher) (Assertion)

Un exemple

en entrée une mémoire dans laquelle $x==5$ et $f==1$

```
new TantQue(  
  new Sup(x,new Constante(1)),  
  new Sequence(  
    new Affectation(f,new Multiplication(f,x)),  
    new Affectation(x,new Soustraction(x,new Constante(1))))  
);
```

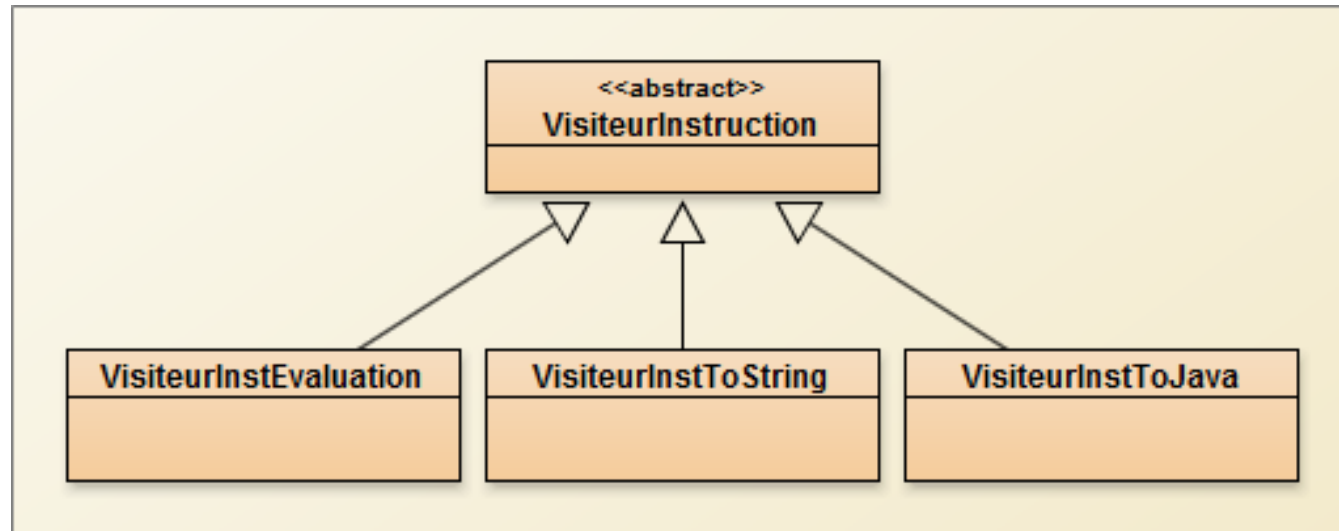
en sortie une mémoire dans laquelle f vaut 120

Un extrait du source Java

```
while(x > 1){  
    f = (f * x);  
    x = (x - 1);  
}
```


Visiteurs

- **Visite d'une instance d'un composite**
 - Pour une conversion en String
 - Pour une évaluation
 - Pour une conversion en source Java
 - Pour une génération de code
 - Pour ...



Exemple : évaluation de Factoriel !!!

```
public void testFactoriel(){
    Contexte m = new Memoire();
    Variable x = new Variable(m, "x", 5);
    Variable fact = new Variable(m, "fact", 1);

    VisiteurExpression ve = new VisiteurEvaluation(m);
    VisiteurExpressionBooleenne vb = new VisiteurBoolEvaluation(ve);
    VisiteurInstruction vi = new VisiteurInstEvaluation(ve, vb);

    Instruction i =
        new TantQue(
            new Sup(x, new Constante(1)),
            new Sequence(
                new Affectation(fact, new Multiplication(fact, x)),
                new Affectation(x, new Soustraction(x, new Constante(1))))
        );

    i.accepter(vi);
    assertTrue(" valeur erronée", m.lire("fact")==fact(5)); // ← vérification
}
```

Vérification en Java de factoriel...

```
private static int fact(int n) {  
    if(n==0) return 1;  
    else return n*fact(n-1);  
}
```

```
private static int fact2(int x) {  
    int fact = 1;  
    while(x > 1) {  
        fact = fact * x;  
        x = x - 1;  
    }  
    return fact;  
}
```

Exemple suite, un autre visiteur qui génère du source Java

```
public void test_CompilationDeFactoriel() {
    Contexte m = new Memoire();
    Variable x = new Variable(m, "x", 5);
    Variable fact = new Variable(m, "fact", 1);

    Instruction inst = // idem transparent précédent
        new TantQue(
            new Sup(x, new Constante(1)),
            new Sequence(
                new Affectation(fact, new Multiplication(fact, x)),
                new Affectation(x, new Soustraction(x, new Constante(1)))
            )
        );

    VisiteurExpression<String> ves = new VisiteurInfixe(m);
    VisiteurExpressionBooleenne<String> vbs = new VisiteurBoolToJava(ves);
    VisiteurInstruction<String> vs = new VisiteurInstToJava(ves, vbs, 4);

    // vérification par une compilation du source généré
}
```

Résultat obtenu : Le source généré

```
package question3;

public class Fact{

    public static void main(String[] args) throws Exception{
        int fact=1;
        int x=5;

        while(x > 1){
            fact = (fact * x) ;
            x = (x - 1);
        }
    }

}
```

Un visiteur qui engendre du bytecode

- **Bytecode** :code intermédiaire de la machine java
- **Mnémoniques** issus de l'assembleur jasmin

- <http://jasmin.sourceforge.net/>



Exemple suite, un autre visiteur qui génère du bytecode Java

```
public void test_CompilationDeFactoriel(){
    Contexte m = new Memoire();
    Variable x = new Variable(m, "x", 5);
    Variable fact = new Variable(m, "fact", 1);

    Instruction inst = // idem transparent précédent
        new TantQue(
            new Sup(x, new Constante(1)),
            new Sequence(
                new Affectation(fact, new Multiplication(fact, x)),
                new Affectation(x, new Soustraction(x, new Constante(1)))
            )
        );

    Code code = new Code("TestsFactoriel", m);
    VisiteurExprJasmin vej = new VisiteurExprJasmin(m, code);
    VisiteurBoolJasmin vbj = new VisiteurBoolJasmin(vej);
    VisiteurInstJasmin vij = new VisiteurInstJasmin(vej, vbj);

    // vérification par l'exécution de l'assembleur Jasmin
}
```

- **Retour sur la compilation**
 - **D'une expression arithmétique**
 - **D'une expression booléenne**
 - **D'une instruction**

Pour une expression arithmétique

- Générer du *bytecode* pour une expression arithmétique :
- **GenererCode(N) = *iconst_N* si $N \in [0..5]$**
GenererCode(N) = *iconst_m1* si $N = -1$
GenererCode(N) = *bipush N* si $N \in [-128..-2]$ et $[6..+127]$
GenererCode(N) = *sipush N* si $N \in [-32768..-129]$ et $[+128..+32767]$
GenererCode(N) = *ldc N* en dehors des valeurs ci-dessus
- **GenererCode(V) = *iload_N* avec $N = IDX$ si $IDX \in [0..3]$**
GenererCode(V) = *iload N* avec $N = IDX$ si $IDX \in [4..65535]$
- **GenererCode($E1 + E2$) = GenererCode($E1$), GenererCode($E2$), *iadd***
GenererCode($E1 - E2$) = GenererCode($E1$), GenererCode($E2$), *isub*
GenererCode($E1 * E2$) = GenererCode($E1$), GenererCode($E2$), *imul*
GenererCode($E1 / E2$) = GenererCode($E1$), GenererCode($E2$), *idiv*

Exemples 3+2...

expression arithmétique	L'instance du composite paquetage whileL	en pseudo-code pour machine à pile	en assembleur Jasmin, syntaxe ici
$3 + 2$	<code>new Addition(new Constante(3), new Constante(22))</code>	<code>empiler(3) empiler(22) additionner</code>	<code>iconst_3 bipush 22 iadd</code>
$(3 - x) / y$	<code>Memoire m=new Memoire(); Variable x = new Variable(m,"x",2); Variable x = new Variable(m,"y",3); new Division(new Soustraction(3,x), y);</code>	<code>empiler(3) empiler(x) soustraire empiler(y) diviser</code>	<code>iconst_3 iload_1 ; x en mémoire à l'adresse vars +1 isub iload_2 ; x en mémoire à l'adresse vars +2 idiv</code>

Pour une expression booléenne

```
GenererCode(Vrai) = iconst_1
GenererCode(Faux) = iconst_0
GenererCode(Non(Bexp)) = GenererCode(Bexp) ;
    ifne Etiq1
    iconst_1
    goto Fin
Etiq1: iconst_0
Fin:

GenererCode(BE1 et BE2) = GenererCode(BE1) ;
    ifeq Etiq1
    GenererCode(BE2) ;
    ifeq Etiq1
    iconst_1
    goto Fin
Etiq1: iconst_0
Fin:

GenererCode(E1 > E2) = GenererCode(E1) ;
    GenererCode(E2) ;
    if_cmple Etiq1
    iconst_1
    goto Fin
Etiq1: iconst_0
Fin:
```

Un exemple

expression booléennes	L'instance du composite	en pseudo-code pour machine à pile	en assembleur Jasmin
$i > 10$	Memoire m= ... new Sup(i, new Constante(10))	<i>empiler(i)</i> <i>empiler(10)</i> <i>si(dépiler() <= dépiler()) saut_en échec</i> <i>empiler(vrai);</i> <i>saut_en fin</i> <i>échec :</i> <i>empiler(faux);</i> <i>fin :</i>	<i>iload_1</i> <i>bipush 10</i> <i>if_icmple #_21</i> <i>iconst_1</i> <i>goto #_23</i> <i>#_21:</i> <i>iconst_0</i> <i>#_23:</i>

Pour les instructions

```
GenererCode(V := E1) = GenererCode(E); istore_N si IDX(V) appartient à [0..3]
GenererCode(V := E1) = GenererCode(E1); istore N si IDX(V) appartient à [4..32767]

GenererCode(I1 ; I2) = GenererCode(I1); GenererCode(I2)

GenererCode(si (BE) alors I1) = GenererCode(BE);
                                ifeq Fin
                                GenererCode(I1);
                                Fin:
GenererCode(si (BE) alors I1 sinon I2) = GenererCode(BE);
                                         ifeq Etiq1
                                         GenererCode(I1);
                                         goto Fin
                                         Etiq1: GenererCode(I2);
                                         Fin:

GenererCode(TantQue BE1 faire I) =
                                Debut: GenererCode(BE1);
                                ifeq Fin
                                GenererCode(I);
                                goto Debut
                                Fin:
```

Un exemple

```
while(i<10)
  i:=i+1;
assert i==10: "i != 10 ???"
```

```
Instruction i1 =
  new Sequence(
    new TantQue(
      new Inf(i,new Constante(10)),
      new Affectation(i, new Addition(i,new Constante(1)))
    ),
    new Assertion( new Egal(i, new Constante(10)), " i != 10 ???")
  );
```

```
début:
empiler(i)
empiler(10)
si(dépiler())>=dépiler())
  alors saut_en_fin
empiler(i)
empiler(1)
iadd
i := dépiler();
saut_en début
fin:
assert i==10:"i != 10";
```

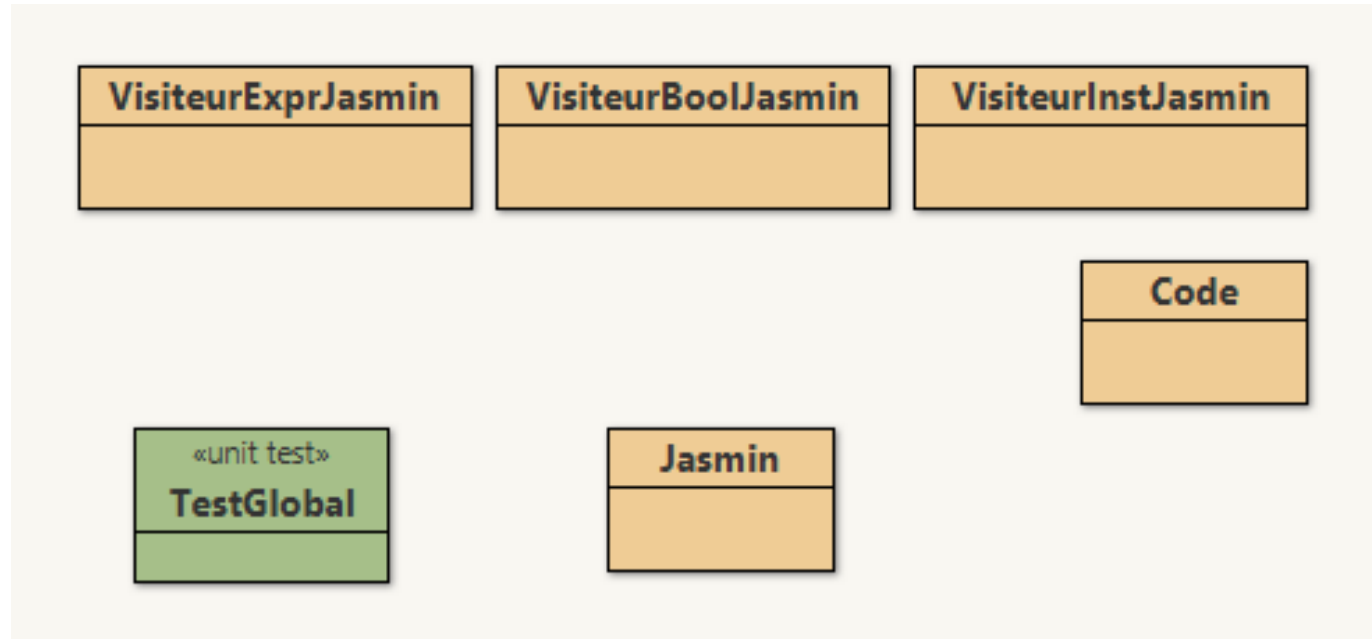
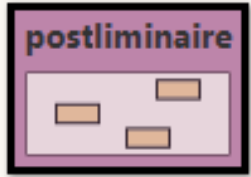
```
#_13:
iload_1
bipush 10
if_icmpge #_22
iconst_1
goto #_24
#_22:
iconst_0
#_24:
ifeq #_33
iload_1
iconst_1
iadd
istore_1
goto #_13
#_33:
iload_1
bipush 10
if_icmpne #_42
iconst_1
goto #_44
#_42:
iconst_0
#_44:
ifne #_55
new java/lang/AssertionError
dup
ldc " i != 10 ??? "
invokespecial java/lang
/AssertionError/<init>(Ljava
/lang/Object;)V
athrow
#_55:
```

Le bytecode généré pour factoriel ...

compatible.jasmin.jasmin.sourceforge.net/

```
.class public TestsFactoriel
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit stack 3                                #_25:
.limit locals 3                                ifeq #_43
    ldc 1                                       iload 1
    istore 1                                   iload 2
    ldc 5                                       imul
    istore 2                                   istore 1
#_14:                                           iload 2
    iload 2                                   iconst_1
    iconst_1                                   isub
    if_icmple #_23                             istore 2
    iconst_1                                   goto #_14
    goto #_25
#_23:                                           #_43:
    iconst_0                                   return
                                           .end method
```

Les visiteurs se chargent de tout



- Cf. le tp

Un détail : la visite pour TantQue

```
public Integer visite(TantQue tq){  
  
    int start = code.addLabel(code.currentPosition());  
    int hc = tq.cond().accepter(this.vbj);  
    code.add("ifeq");  
    int jumpIfAddr = code.add("labelxxxxx");  
    int h = tq.i1().accepter(this);  
    code.add("goto");  
    int jumpAddr = code.add("labelxxxxx");  
    code.setLabel(jumpAddr,start);  
    code.setLabel(jumpIfAddr,code.addLabel(code.currentPosition()));  
  
    return Math.max(hc,h);  
}
```

les visites pour Affectation et Sequence

```
public Integer visite(Affectation a){
    int h = a.exp().accepter(this.vej);
    int idx = code.varIndex(a.v().nom());
    if(idx >= 0 && idx <=3){
        code.add("istore_" + idx);
    }else{
        code.add("istore"); code.add(idx);
    }
    return h;
}
```

```
public Integer visite(Sequence seq){
    int h1 = seq.i1().accepter(this);
    int h2 = seq.i2().accepter(this);
    return Math.max(h1,h2);
}
```

Un détail : la visite pour TantQue

```
public Integer visite(Sup sup){
    int h1 = sup.op1().accepter(this.vbc);
    int h2 = sup.op2().accepter(this.vbc);
    code.add("if_icmple");
    int jumpIfAddr = code.currentPosition();
    code.add("labelxxxxx");
    code.add("iconst_1");
    code.add("goto");
    int jumpAddr = code.currentPosition();
    code.add("labelxxxxx");
    code.setLabel(jumpIfAddr,code.currentPosition());
    code.addLabel(code.currentPosition());
    code.add("iconst_0");
    code.setLabel(jumpAddr,code.currentPosition());
    code.addLabel(code.currentPosition());
    return Math.max(1, h1+h2);
}
```

Le code généré en jasmin !

```
.class public TestsFactoriel
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit stack 7
.limit locals 3
    ldc 1
    istore 1
    ldc 5
    istore 2
#_14:
    iload 2
    iconst_1
    if_icmple #_23
    iconst_1
    goto #_25
#_23:
    iconst_0
#_25:
    ifeq #_43
    iload 1
    iload 2
    imul
    istore 1
    iload 2
    iconst_1
    isub
    istore 2
    goto #_14
#_43:
    iload 1
    bipush 120
    if_icmpne #_53
    iconst_1
    goto #_55
#_53:
    iconst_0
#_55:
    ifne #_66
    new java/lang/AssertionError
    dup
    ldc " 5! != 120 ??? "
    invokespecial java/lang/AssertionError/<init>(Ljava/lang/Object;)V
    athrow
#_66:
    getstatic java/lang/System/out Ljava/io/PrintStream;
    iload 1
    invokevirtual java/io/PrintStream/println(I)V
    return
.end method
```

Démonstration

Présentation du TP

Aux multiples composites et visiteurs

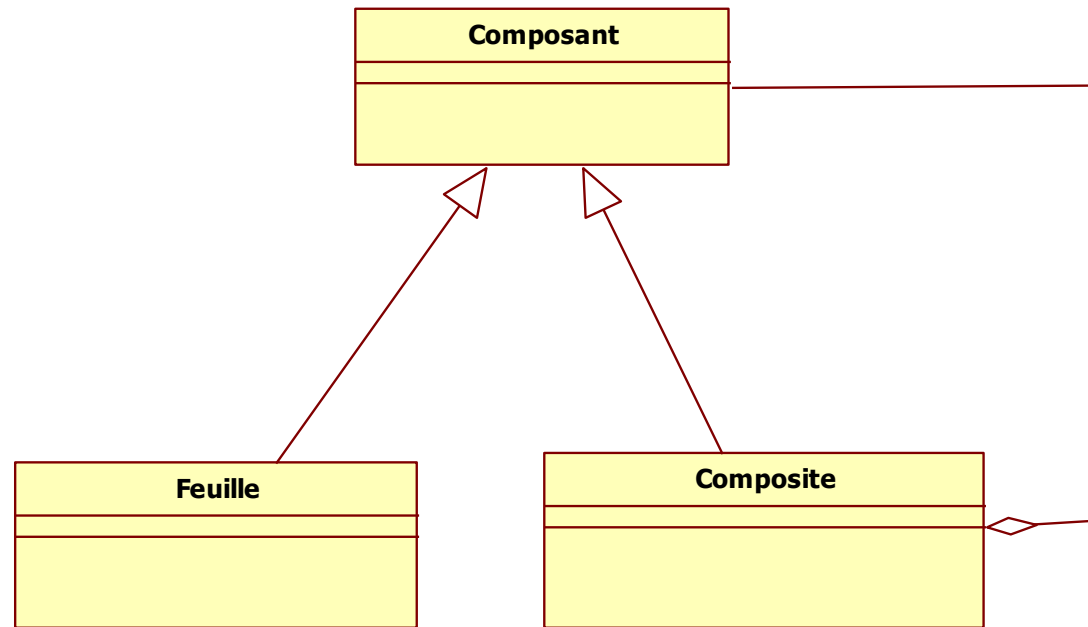
Le langage WhileL

<http://jfod.cnam.fr/progAvancee/Hennessy.pdf>

Chapitre 4.3 page 91

- **Patron Composite et parcours**
 - **Souvent récursif, « intra-classe »**
 - **Avec un itérateur**
 - **Même recette ...**
 - **Une pile mémorise l'itérateur de chaque classe « composite »**
 - **Avec un visiteur**
 - **Java API Compiler avec les visiteurs**

Composite et Iterator



- Structure réursive « habituelle »

- Classe Composite : un schéma

```
public class Composite
    extends Composant implements Iterable<Composant>{
    private List<Composite> liste;

    public Composite(...) {
        this.liste = ...
    }
    public void ajouter(Composant c) {
        liste.add(c);
    }

    public Iterator<Composant> iterator() {
        return new CompositeIterator(liste.iterator());
    }
}
```

CompositeIterator : comme sous-classe

```
private
class CompositeIterator
    implements    Iterator<Composant>{

    // une pile d'itérateurs,
    // un itérateur par composite
    private Stack<Iterator<Composant>> stk;

    public CompositeIterator (Iterator<Composant> iterator) {
        this.stk = new Stack<Iterator<Composant>>() ;
        this.stk.push(iterator) ;
    }
}
```

next

```
public Composant next() {
    if (hasNext()) {
        Iterator<Composant> iterator = stk.peek();
        Composant cpt = iterator.next();
        if (cpt instanceof Composite) {
            Composite gr = (Composite) cpt;
            stk.push(gr.liste.iterator());
        }
        return cpt;
    } else {
        throw new NoSuchElementException();
    }
}

public void remove() {
    throw new UnsupportedOperationException();
}
}
```

hasNext

```
public boolean hasNext() {  
    if (stk.empty()) {  
        return false;  
    } else {  
        Iterator<Composant> iterator = stk.peek();  
        if ( !iterator.hasNext() ) {  
            stk.pop();  
            return hasNext();  
        } else {  
            return true;  
        }  
    }  
}
```

Un test unitaire possible

```
public void testIterator () {
    try {
        Composite g = new Composite();
        g.ajouter(new Composant());
        g.ajouter(new Composant());
        g.ajouter(new Composant());
        Composite g1 = new Composite();
        g1.ajouter(new Composant());
        g1.ajouter(new Composant());
        g.ajouter(g1);

        for(Composite cpt : g) { System.out.println(cpt); }

        Iterator<Composite> it = g.iterator();
        assertTrue(it.next() instanceof Composant);
        assertTrue(it.next() instanceof Composant);
        assertTrue(it.next() instanceof Composant);
        assertTrue(it.next() instanceof Groupe);
        // etc.
    }
}
```

Discussions annexes

- Itérateur compliqué ?
- Le visiteur tu préféreras

API Compiler

The Java Compiler Tree API provides three implementations of `TreeVisitor`; namely, `SimpleTreeVisitor`, `TreePathScanner`, and `TreeScanner`. The demo application uses a `TreePathScanner` to extract information about the Java source file. The `TreePathScanner` is a `TreeVisitor` that visits all the child tree nodes and provides support for maintaining a path for the parent nodes. The `scan()` method of the `TreePathScanner` needs to be invoked to scan the tree. To visit nodes of a particular type, just override the corresponding `visitXYZ` method. Inside your visit method, call `super.visitXYZ` to visit descendant nodes. The code snippet of a typical visitor class is shown below:

```
[prettify]
public class CodeAnalyzerTreeVisitor extends TreePathScanner<Object, Trees> {
    @Override
    public Object visitClass(ClassTree classTree, Trees trees) {
        ---- some code ----
        return super.visitClass(classTree, trees);
    }
    @Override
    public Object visitMethod(MethodTree methodTree, Trees trees) {
        ---- some code ----
        return super.visitMethod(methodTree, trees);
    }
}
```

- <https://today.java.net/pub/a/today/2008/04/10/source-code-analysis-using-java-6-compiler-apis.html>