Composite, Interpréteur et Visiteur

Cnam Paris
jean-michel Douin, douin@cnam.fr
Version du 10 Octobre 2003

Notes de cours java : les patterns Composite, Interpréteur et Visiteur

Les notes et les Travaux Pratiques sont disponibles en
http://jfod.cnam.fr/tp_cdi/{douin/}

Objectifs

- **Structures de données récursives**
  Le pattern Composite
- **Analyse, interprétation de ces structures**
  Le pattern Interpréteur (ou little language chez M.Grand)
- **Analyse, multiples interprétations et parcours**
  Le pattern 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

- **CDL : WhileL**

  sémantique opérationnelle d 'un sous-ensemble de java impératif

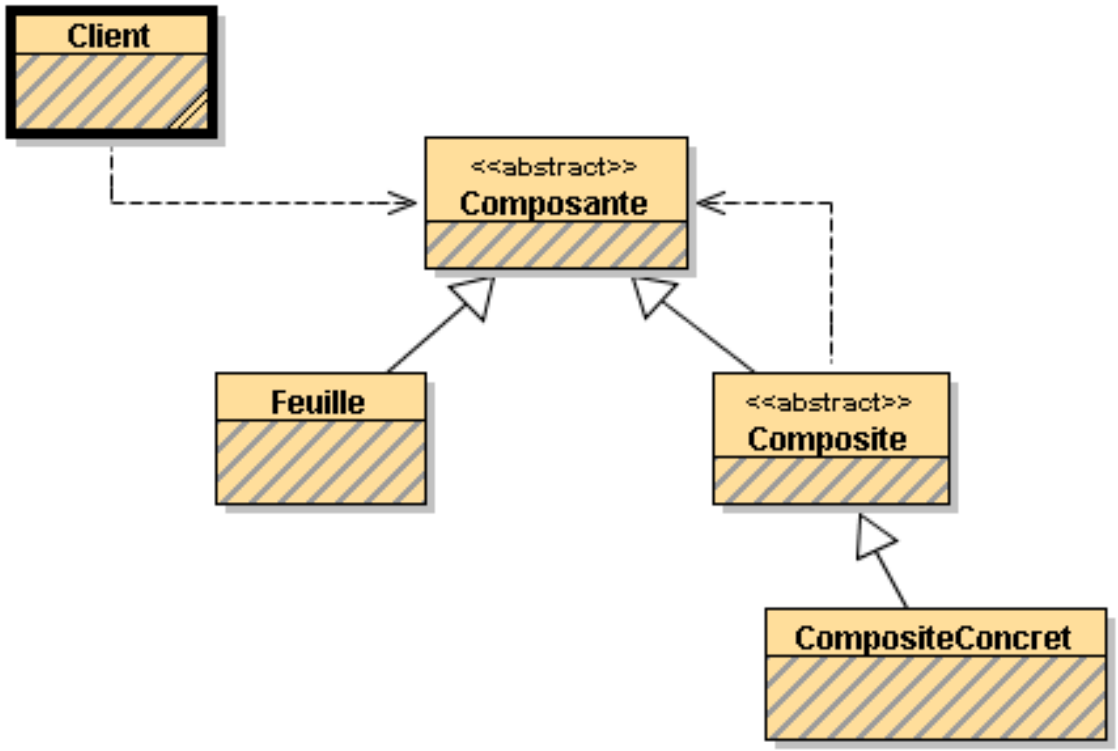
  en règles d'inférence

  http://deptinfo.cnam.fr/Enseignement/CycleProbatoire/CDI/CDL/CDL2000/COURS.CDL/Intro/Progr.htm voir chapitre 10

Sommaire

- **Structures récursives**
  - Les expressions arithmétiques
  - Les classes de l'API AWT utilise le pattern Composite
- **Interprétation d'une expression arithmétique**
  - calcul de la valeur
  - calcul de la valeur à l'aide d'une pile
- **Interprétation multiple : les visiteurs**
  - Un visiteur de calcul, un visiteur de transformation en String
- **WhileL : petit langage impératif, introduction à ..**
  - Les expressions booléennes
  - Les Instructions

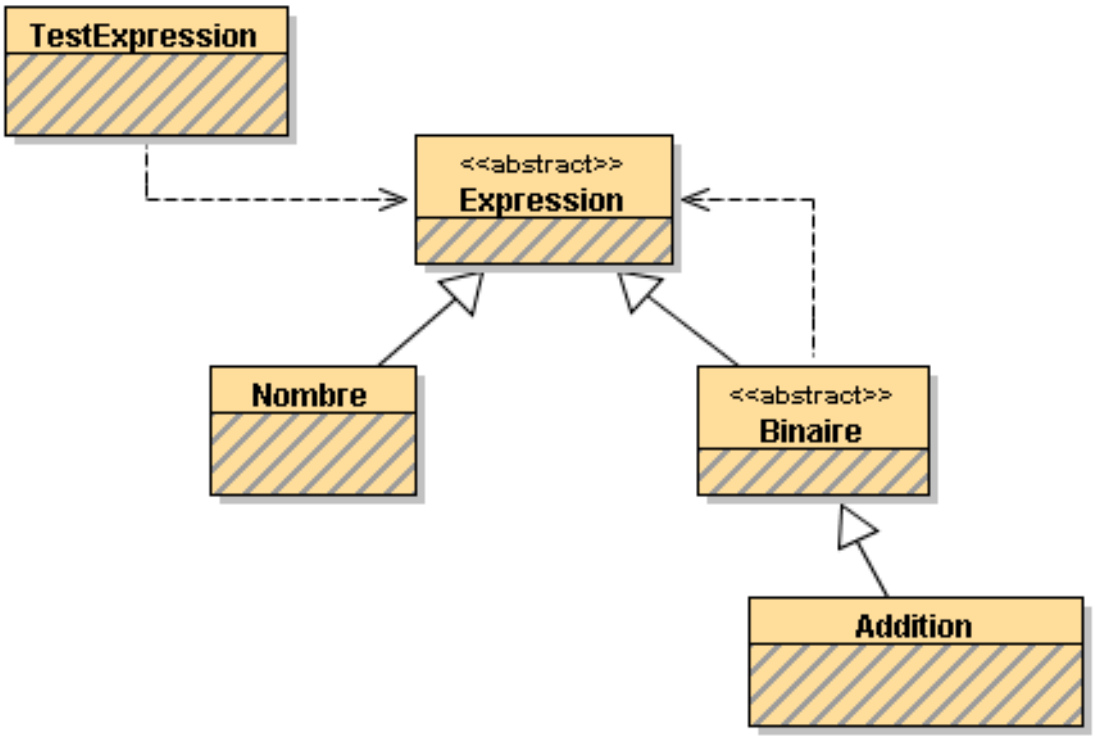
Structures récursives : le pattern Composite



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

Tout est Composante

Composite et Expression



Expression ::= Binaire | Nombre
Binaire ::= Addition
Addition ::= Expression ‘+’ Expression
Nombre ::= ‘une valeur de type int’

Tout est Expression

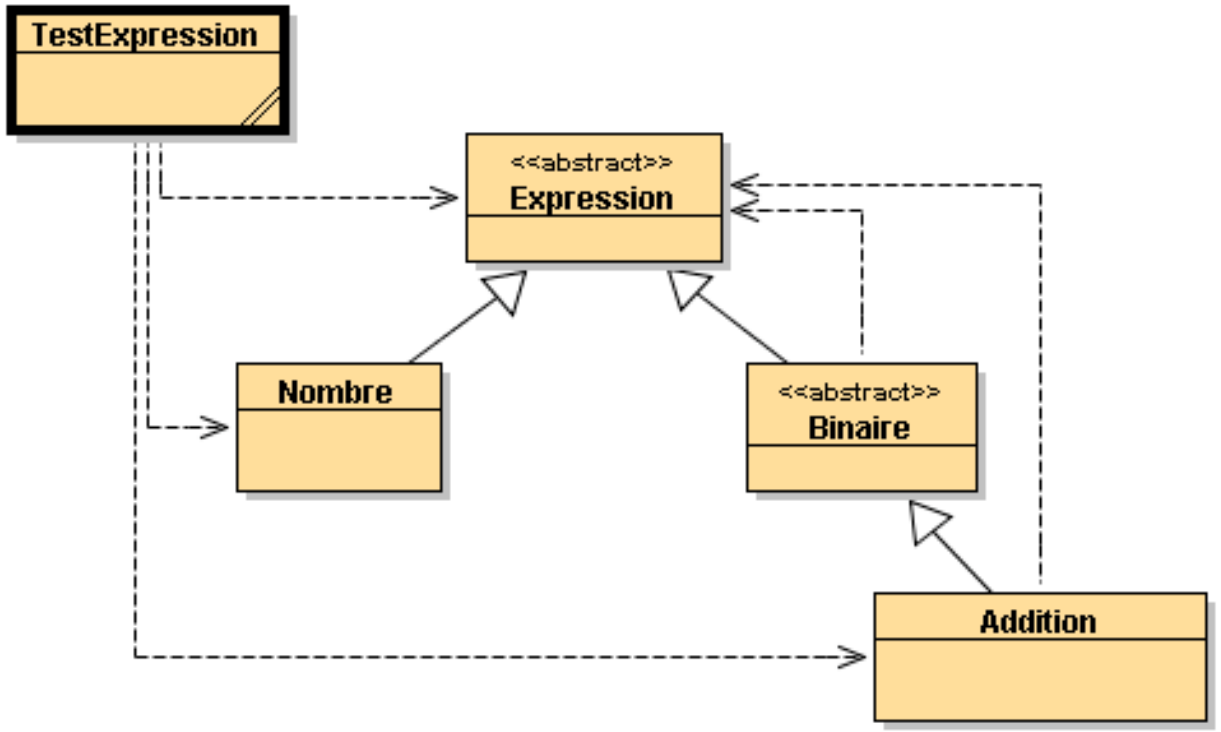
Composite et Expression en Java

```
public abstract class Expression{}


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

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

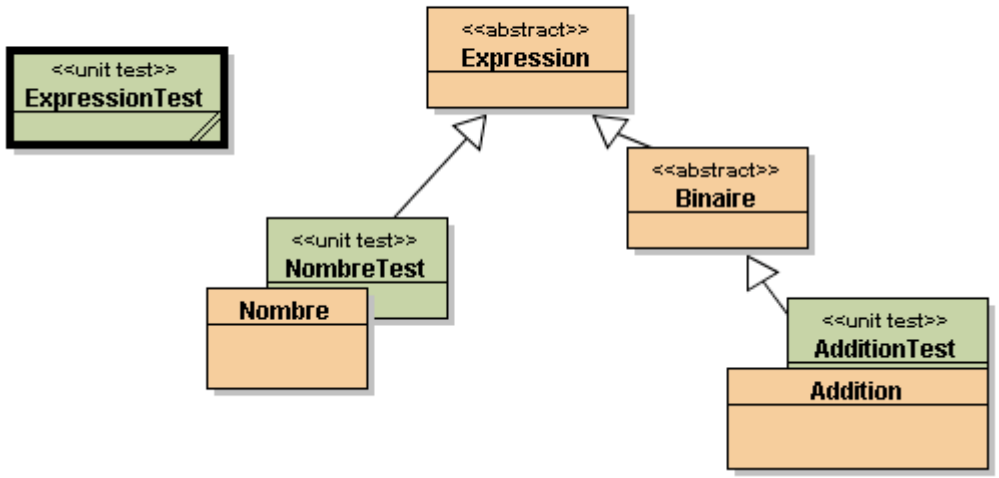
Composite et Expression en Java

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

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

Le diagramme au complet



Avec BlueJ 1.3.0



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);
  }
}
```

Objets graphiques en Java

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

La documentation : une énumération (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 (contenant des composants)

  Fenêtre, applette, …

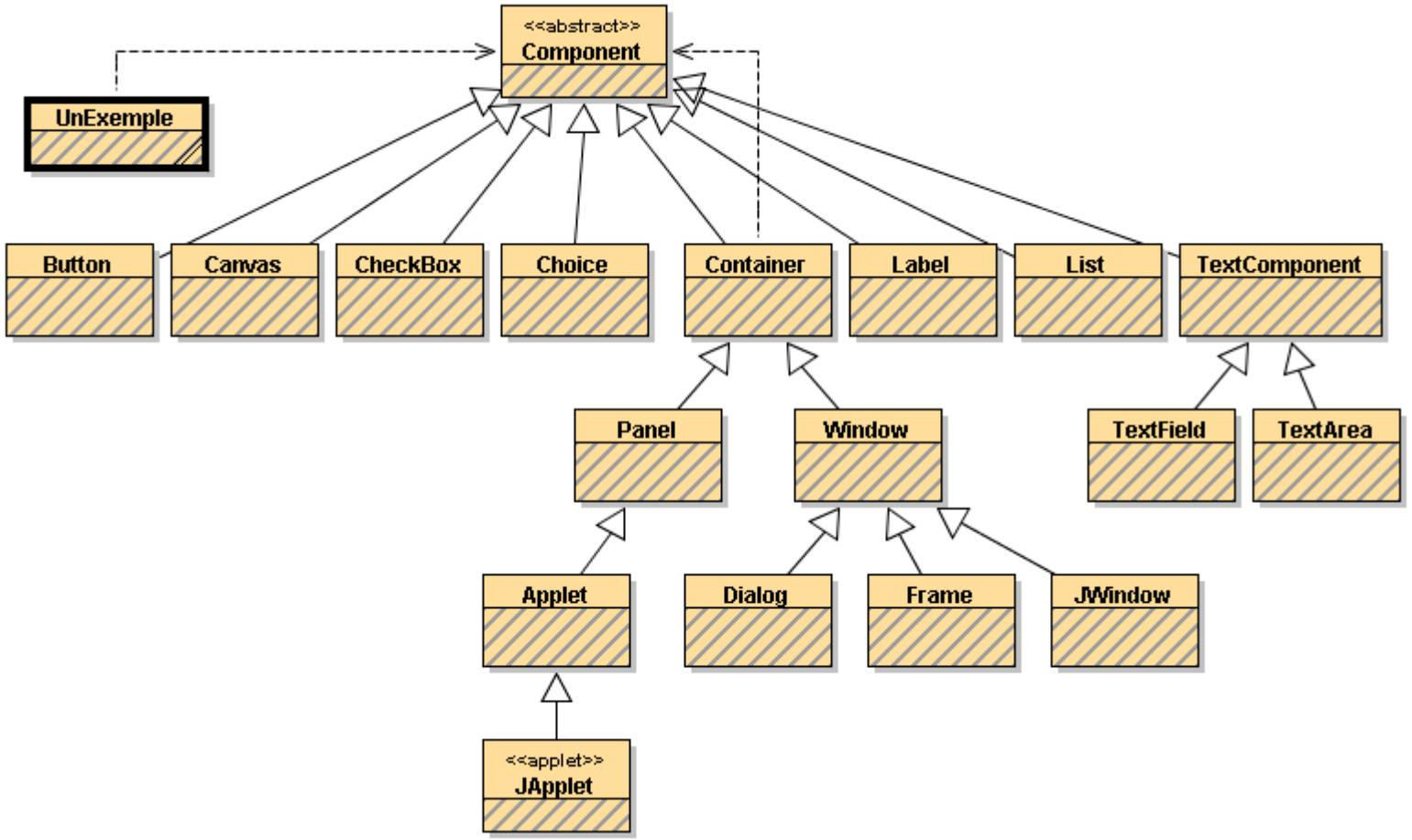
- **Le Pattern Composite est utilisé**

  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);`

Discussion

Une Expression de type composite (Expression)

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

Une Expression de type composite (API AWT)

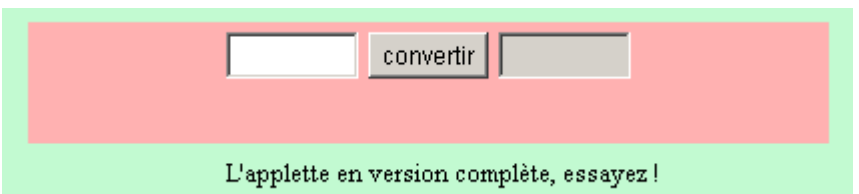
- Container p = new Panel();
- p.add(new Button("b1"));
- p.add(new Button("b2"));

UnExemple : la classe

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

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

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

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

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

AppletteFahrenheit : souvenirs…

```
public class AppletteFahrenheit extends Applet {
  private TextField entree = new TextField(6);
  private Button bouton = new Button("convertir");
  private TextField sortie = new TextField(6);

  public void init() {
     add(entree); // ajout de feuilles au composite
     add(boutonDeConversion);                  // Applet
     add(sortie);
   …
 }
```



Le Pattern Expression : un bilan

- **Composite :**

  Représentation de structures de données récursives

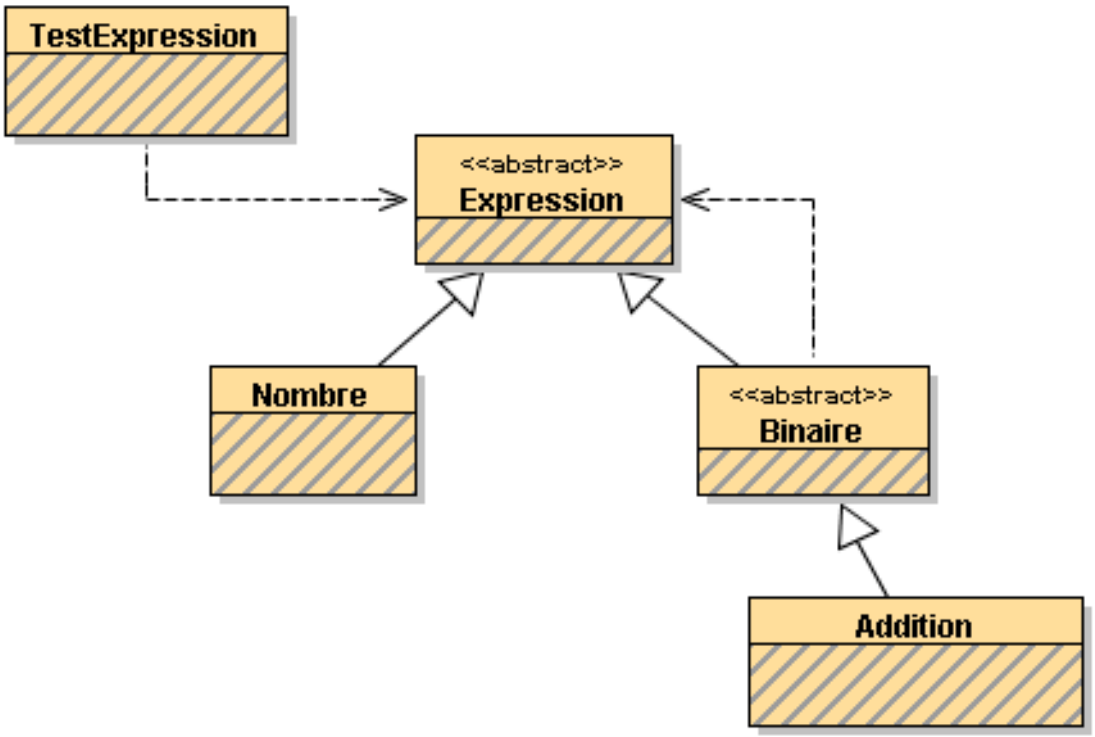
  Techniques de classes abstraites

  Manipulation uniforme (tout est Composant)

- **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
int resultat = e.interpreter();
assert( resultat == 5);      // enfin
```

Le Pattern Interpréteur : Composite + évaluation



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

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;
  }
  abstract public int interpreter();
}
```

Le pattern interpréteur

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

Quelques interprétations en Java

```
// une extrait de ExpressionTest (Junit)
  Expression exp1 = new Nombre(321);
   int resultat = exp1.interpreter();
   assertEquals(resultat,321);

  Expression exp2 = new Addition(
                      new Nombre(33),
                      new Nombre(33)
                    );
   resultat = exp2.interpreter();
   assertEquals(resultat,66);

  Expression exp3 = new Addition(
                      new Nombre(33),
                      new Addition(
                        new Nombre(33),
                        new Nombre(11)
                      )
                    );
   resultat = exp3.interpreter(); assertEquals(resultat,77);

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

Evolution …

- **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

Evolution …

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

Et …

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

→Modification de l'entête de la méthode interpreter

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

→mêmes critiques : modification de toutes les classes ……..

→Encore une modification de toutes les classes, réutilisation plutôt faible …

Le pattern Visiteur au secours du pattern Interpréteur

- **Multiples interprétations de la même structure sans aucune modification du Composite**

- **L'utilisateur de la classe Expression devra proposer ses Visiteurs**
  VisiteurDeCalcul, VisiteurDeCalculAvecUnePile ….

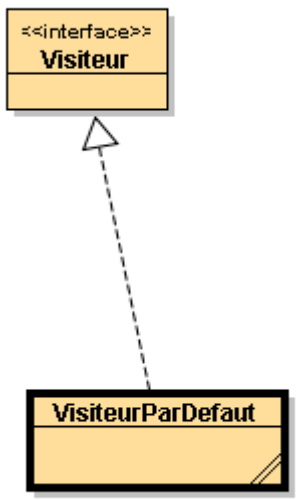
- **→ Ajout de cette méthode, ce sera la seule modification**

```
public abstract class Expression{
  abstract public Object accepter(Visiteur v);
}
```

Dans la famille des itérateurs avez-vous le visiteur …

Le pattern Visiteur une méthode par feuille

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

public class VisiteurParDefaut implements Visiteur{
  public Object visiteNombre(Nombre n) {return n;}
  public Object visiteAddition(Addition a) {return a;}
}
```



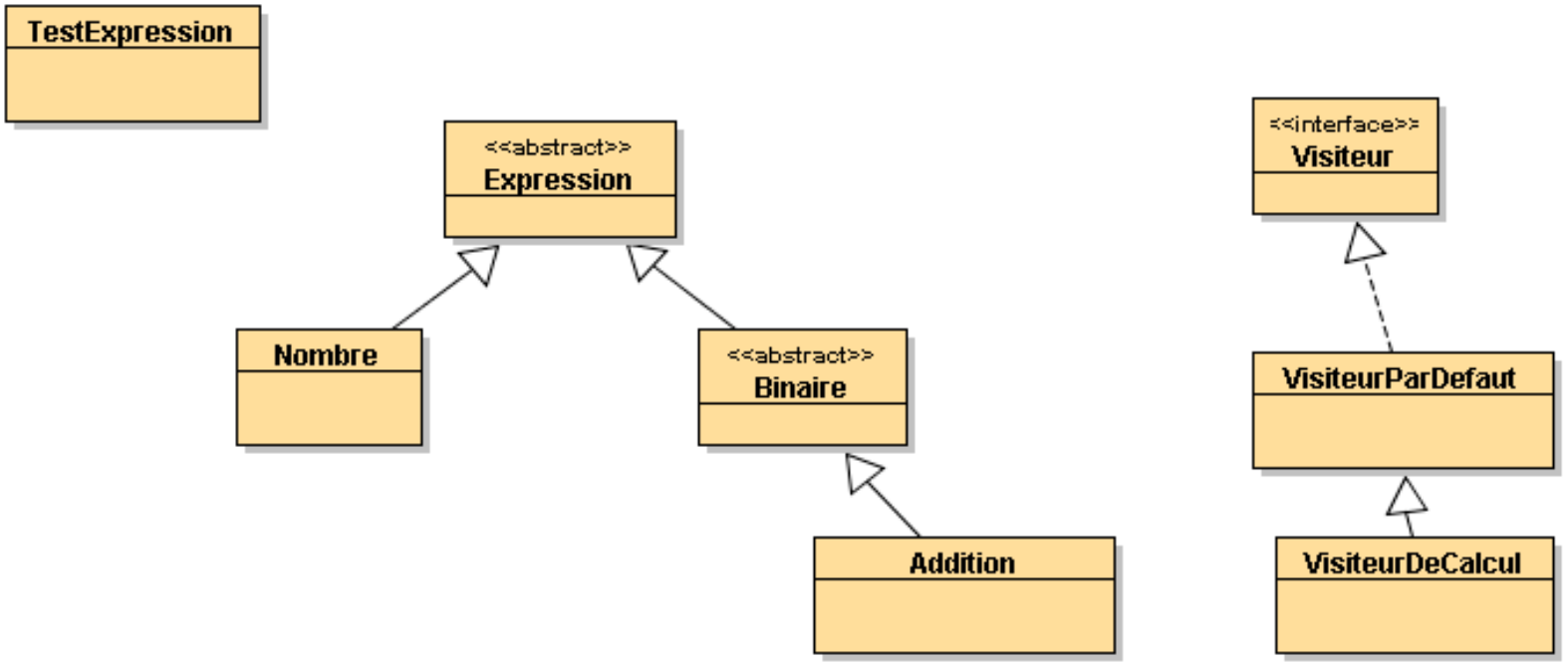
La classe Expression,Binaire une fois pour toutes

```
public abstract class Expression{
  abstract public Object accepter(Visiteur 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 Object accepter(Visiteur 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 Object accepter(Visiteur v){
    return v.visiteNombre(this);
  }
}
public class Addition extends Binaire{
  public Addition(Expression op1,Expression op2){
      super(op1,op2);
  }

   public Object accepter(Visiteur v){
     return v.visiteAddition(this);
   }
}
```

Le Composite a de la visite

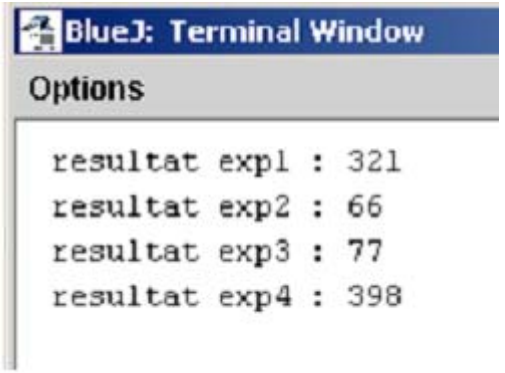


Le VisiteurDeCalcul

```
public class VisiteurDeCalcul extends VisiteurParDefaut{


  public Object visiteNombre(Nombre n) {
    return n;
  }
  public Object visiteAddition(Addition a) {
    Object i1 = a.op1().accepter(this);
    Object i2 = a.op2().accepter(this);
    return new Nombre( ((Nombre)i1).valeur() +
                       ((Nombre)i2).valeur());
  }
}
```

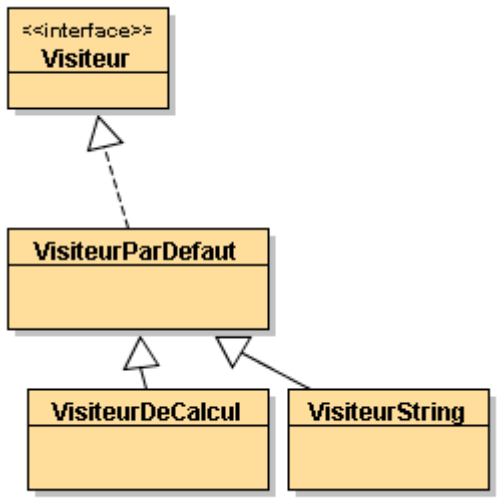
La classe TestExpression

```
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));
}}
```



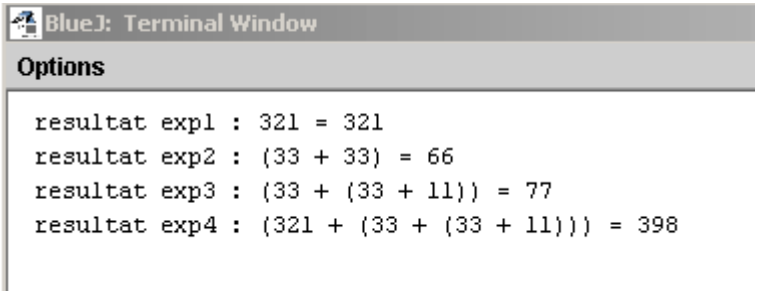
Le Composite a un autre visiteur

```
public class VisiteurString extends VisiteurParDefaut{

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

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



La classe TestExpression re-visitée

```
public class TestExpression{

  public static void main(String[] args){
    Visiteur vc = new VisiteurDeCalcul();
    Visiteur 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));
  } }
```



Le Composite pourrait avoir d'autres visiteurs

```
public class <AutresVisiteurs> extends VisiteurParDefaut{

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

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

Le pattern Visiteur

- **Contrat rempli :**

  Aucune modification du composite :-> couplage faible entre la structure et son analyse

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…**

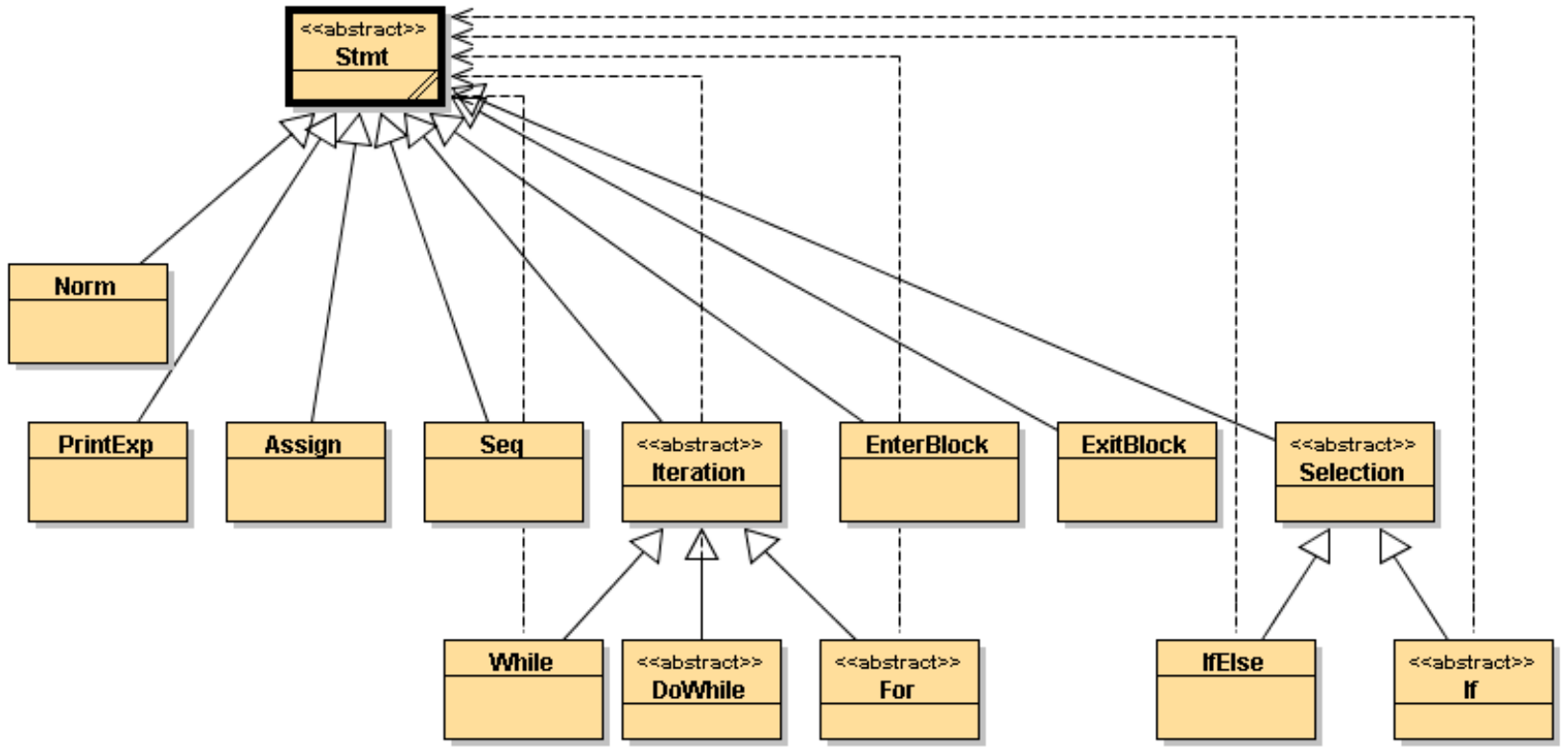
WhileL

- **Règles d'inférence de CDL B3**
- **WhileL en Java, sans visiteur**
- **WhileL en Java avec la syntaxe Jbook,**

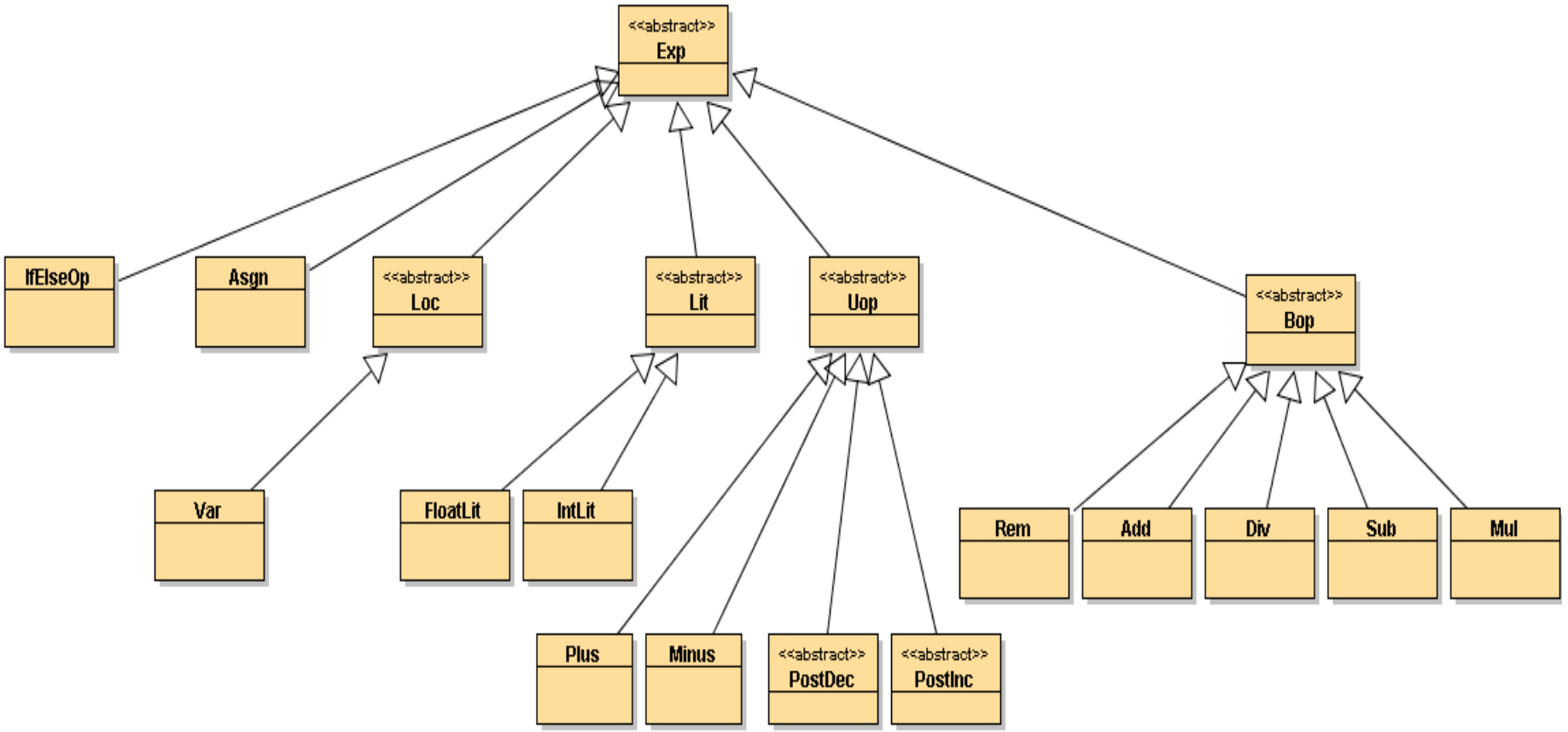
  Composite pour les instructions Stmt
  Composite pour les expressions arithmétiques Expr
  Composite pour les expressions booléennes Bexp

  Visiteurs pour chaque Composite
  VisitorStmt, VisitorExpr, VisitorBexp

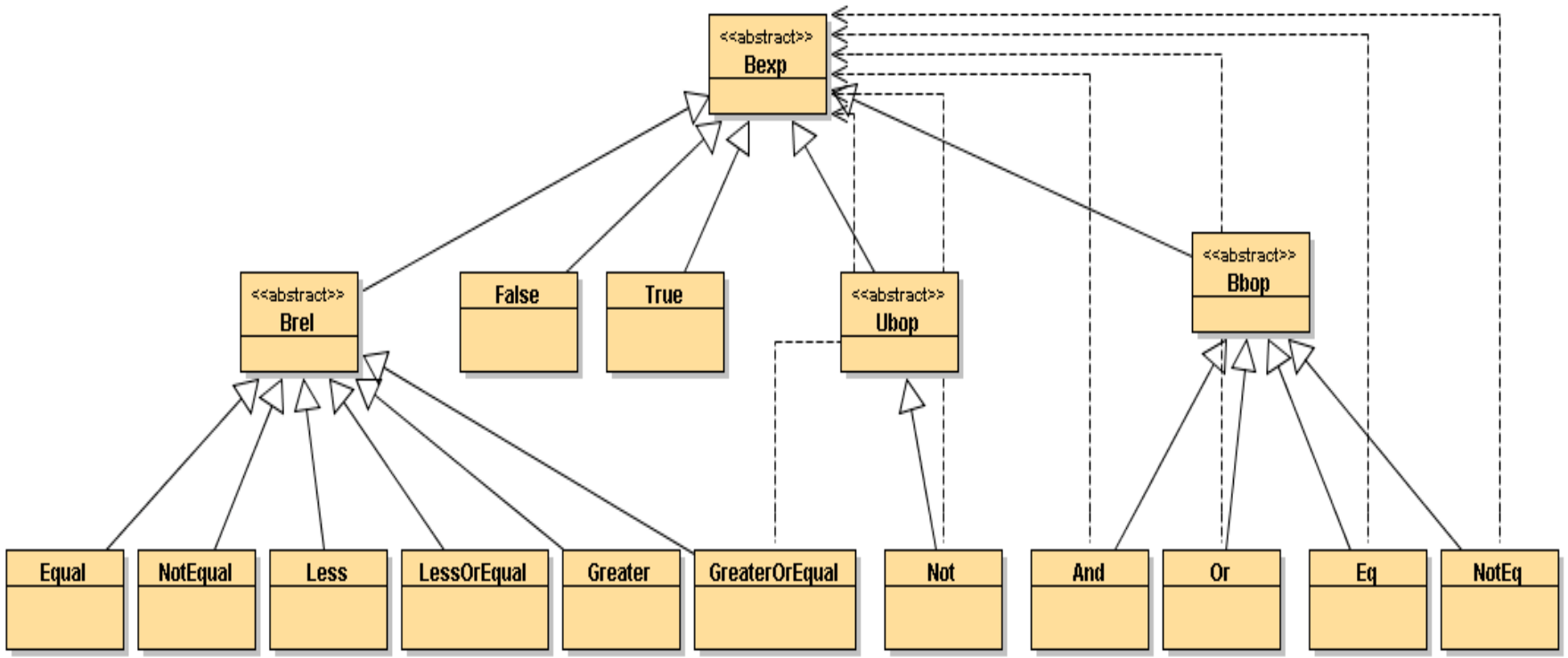
Les instructions Stmt



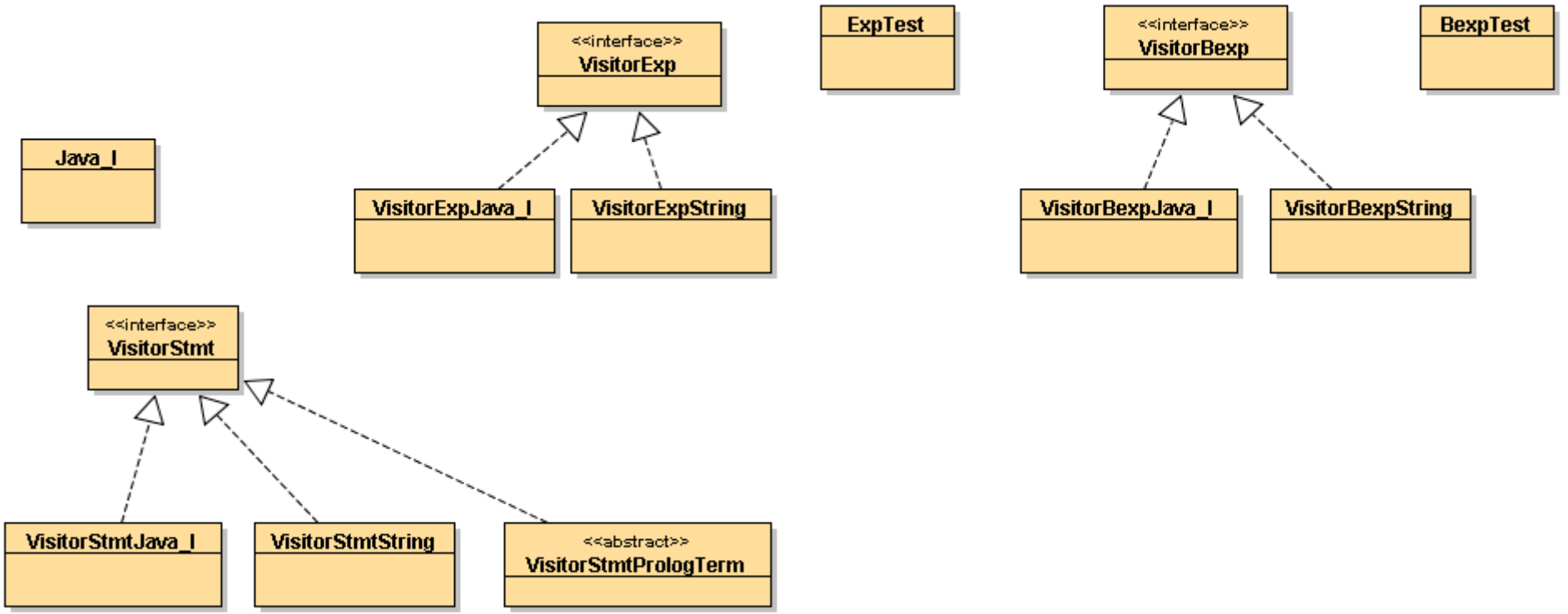
Les Expressions Expr



Les Expressions booléennes



Les visiteurs



Un Test : Java_I

```
s = new While( new Less(i,new IntLit(10)),
       new Seq(
         new Assign(i,new Add(i,new IntLit(1))),
           new Seq( new Assign(j,new Add(j,new IntLit(1))),
                    new Seq(
                         new Assign(k,new Add(k,new IntLit(1))),
                         new IfElse(new Greater(k,new IntLit(0)),
                         new Assign(l,new Add(l,new IntLit(100))),
                         Norm.getInstance())
                         )
                    )
             )
        );

   java_I.execJavaStmt(s);

   System.out.println(s.accept(vss) + " res : " + res +
                      " context: " + java_I.toString());
```

Un Test : Java_I

```
while(i<10){
  i := i + 1;
  j := j + 1;
  k := k + 1;
  if( k > 0){
    l = l + 100;
  } else{
  }
```

```
while(i<10){i=(i + 1);j=(j + 1);k=(k + 1);if(k>0){l=(l +
  100)}else{Norm}} res : Norm
context: [{l=800, k=8, j=8, i=10, res=12.900001}]
```

WhileL, Stmt, While

```
package jbook.java_I.stmt;
import jbook.java_I.*;
import jbook.java_I.bexp.Bexp;
public class While extends Iteration{
  protected Bexp bexp;
  protected Stmt s;

  public While(Bexp bexp, Stmt s){
    this.bexp = bexp;
    this.s = s;
  }

  public Bexp bexp(){ return this.bexp;}
  public Stmt s(){ return this.s;}

  public Object accept(VisitorStmt v){
    return v.visitWhile(this);
  }
}
```

WhileL, Stmt, VisitWhile

```
package jbook.java_I;

import jbook.Block;
import jbook.Context;
import jbook.java_I.*;
import jbook.java_I.bexp.*;
import jbook.java_I.stmt.*;

public class VisitorStmtJava_I implements VisitorStmt{
  private Context context;
  private VisitorExp  ve;
  private VisitorBexp vb;
// ……
  public Object visitWhile(While w){
    Object bexp = w.bexp().accept(vb);
    if( bexp instanceof False)
      return Norm.getInstance();
    else // bexp instanceof True
      return new Seq(w.s(),w).accept(this);
  }
}
```

Alors À suivre …

Conclusion

- **Voir WhileL au complet autre support**
  - **Règles d'inférence engendrent le code des visiteurs**
  - **Recherche sur le Web …**