Les patrons Composite, Interpréteur, Visiteur & Décorateur

jean-michel Douin, douin au cnam point fr
version : 18 Septembre 2008

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 Responsability. Command Interpreter Iterator
    Mediator Memento Observer State
    Strategy Template Method Visitor

Les patrons 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 visitée

Sommaire

- **Structures de données récursives**
  - **Le pattern Composite**
  - **Le pattern Interpréteur**
  - **API Graphique en Java(AWT), paquetage java.awt**
  - **Parcours : Itérateur et/ou visiteur**
- **Comportement dynamique d ’un objet**
  - **Le pattern Décorateur**
  - **Entrées/Sorties, paquetage java.io, java.net.Socket**

Principale bibliographie

- **GoF95**
  - **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**
  - **Design Patterns, Elements of Reusable Object-oriented software Addison Wesley 1995**

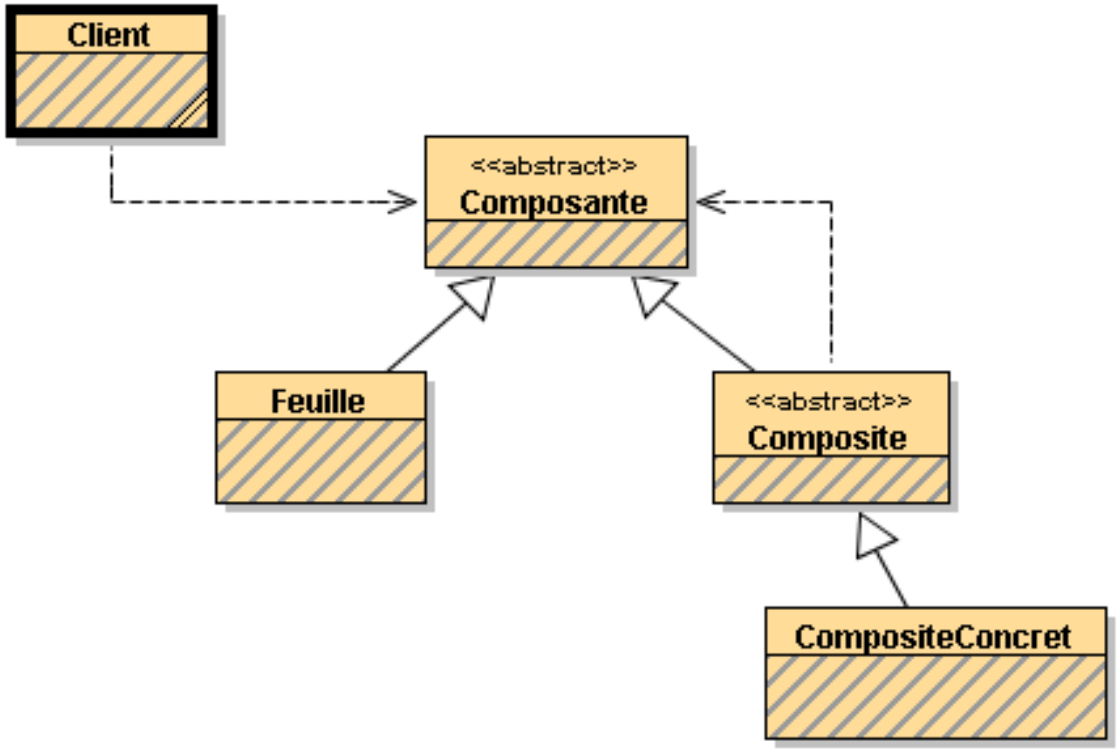
+

  - **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**
  - **www.oreilly.com/catalog/hfdesignpat/chapter/ch03.pdf**

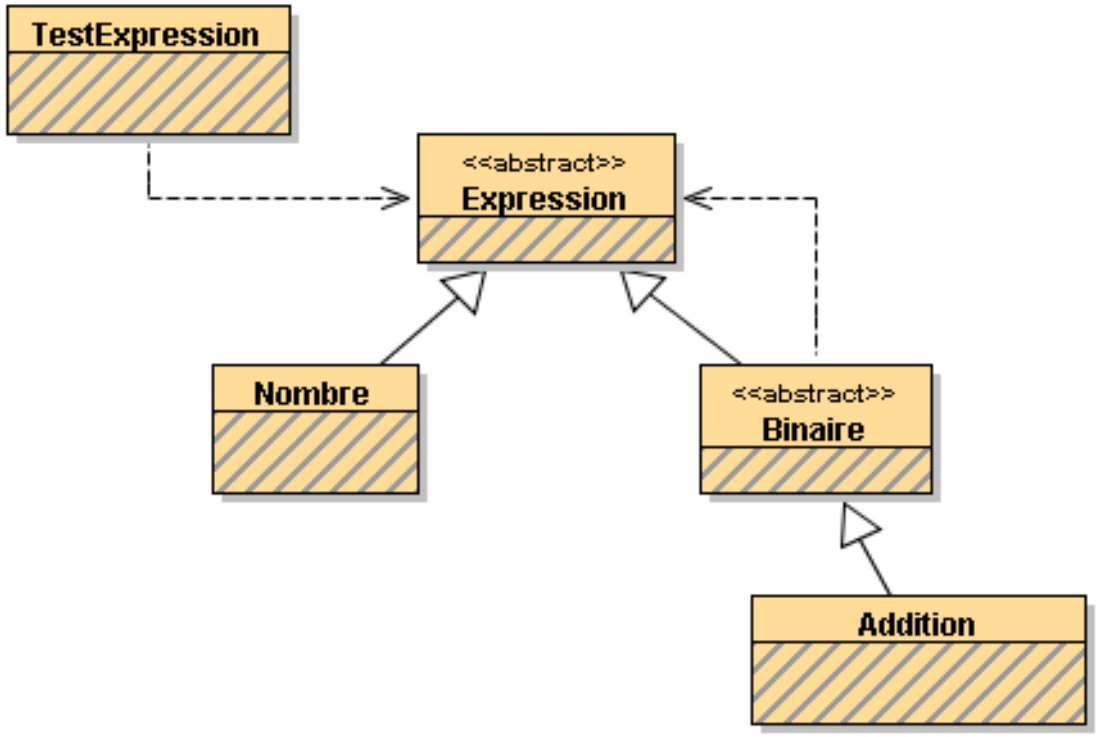
Structures récursives : le pattern Composite



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

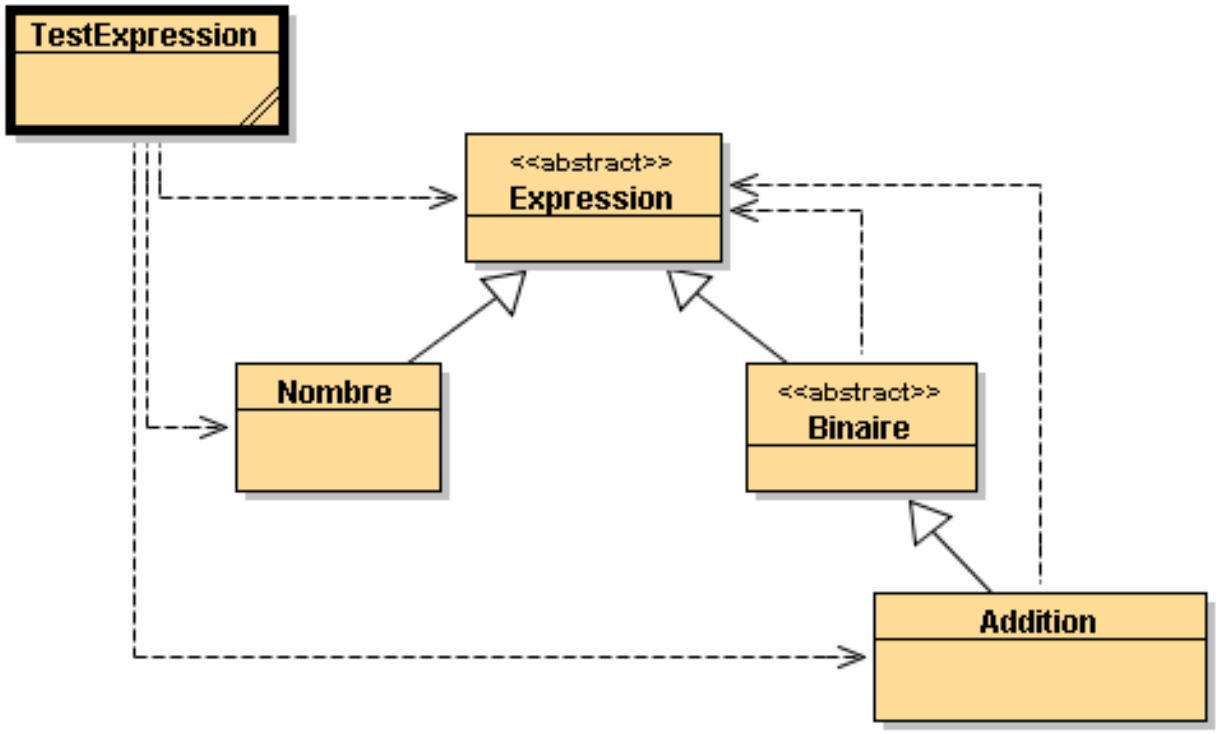
Composite et Expression en Java

```
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;
  }
}
```

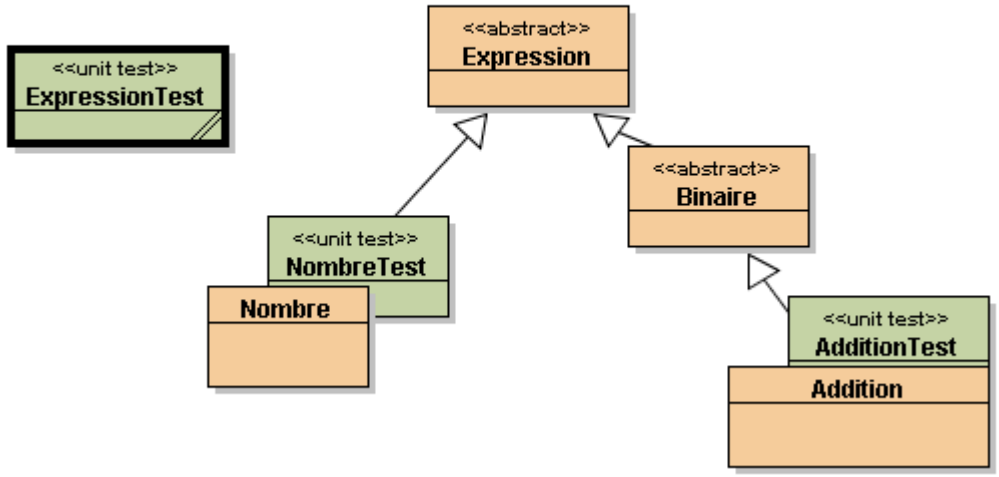
Composite et Expression en Java

```
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



Démonstration avec BlueJ



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);
  }
}
```

L 'AWT utilise 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

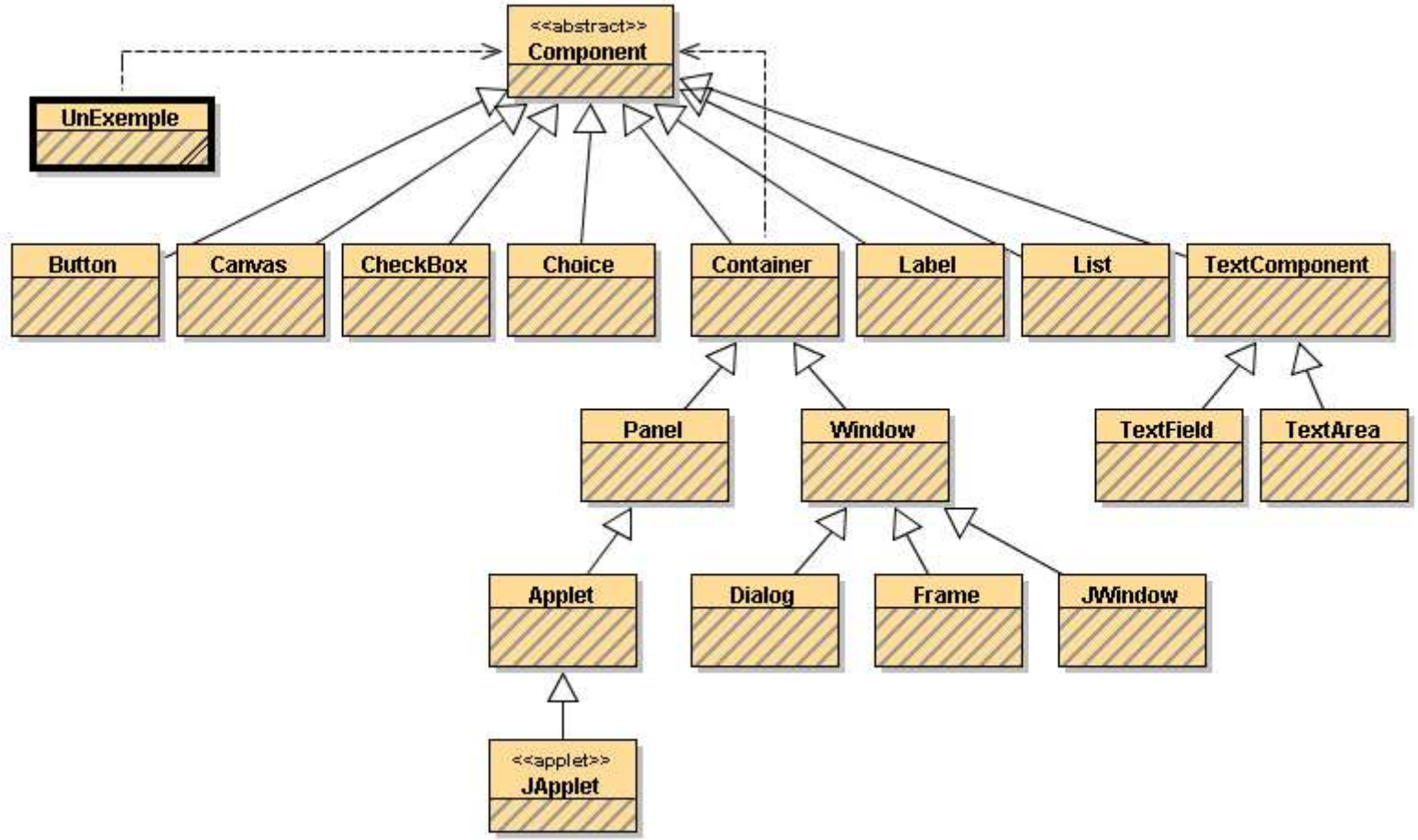
- Direct Known Subclasses:
- **BasicSplitPaneDivider, CellRendererPane, DefaultTreeCellEditor.EditorContainer, JComponent, Panel, ScrollPane, Window**

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, 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)

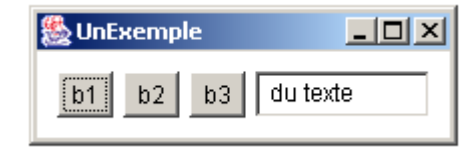
- Expression e = new Addition(new Nombre(1),new Nombre(3));

Une Expression de type composite (API AWT)

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

UnExemple

```
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);
  }}
```



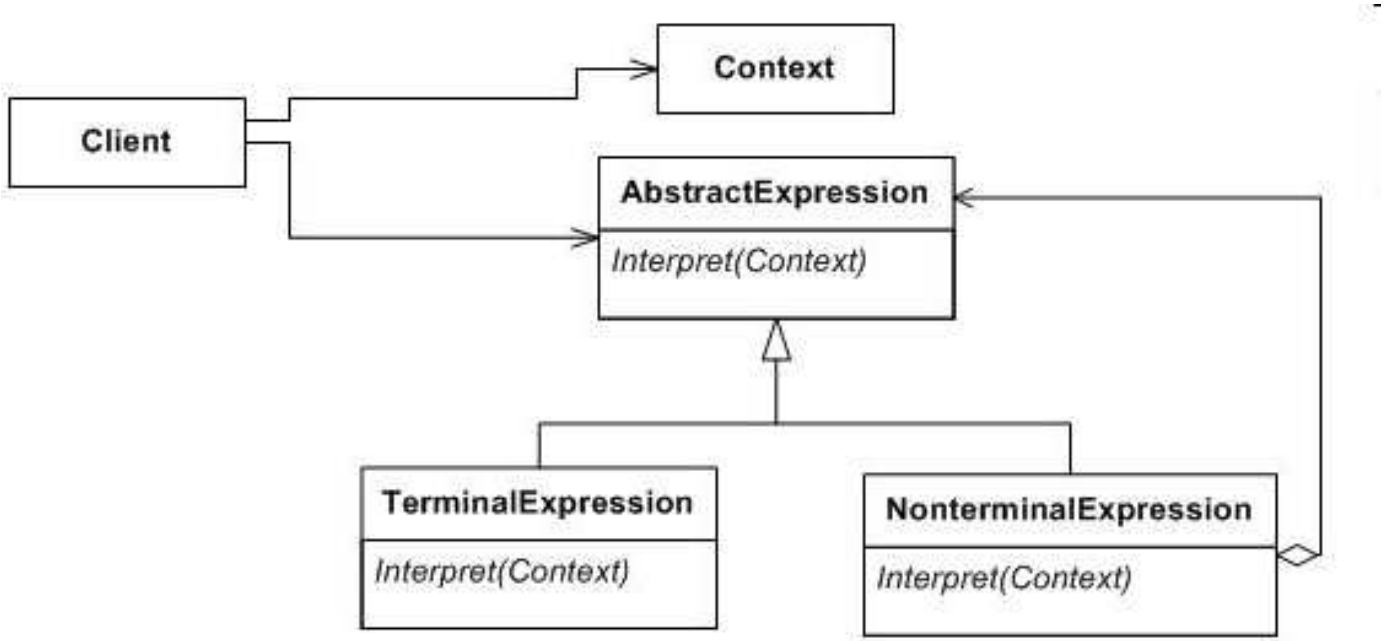
AppletteFahrenheit

```
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);
   …
 }
```

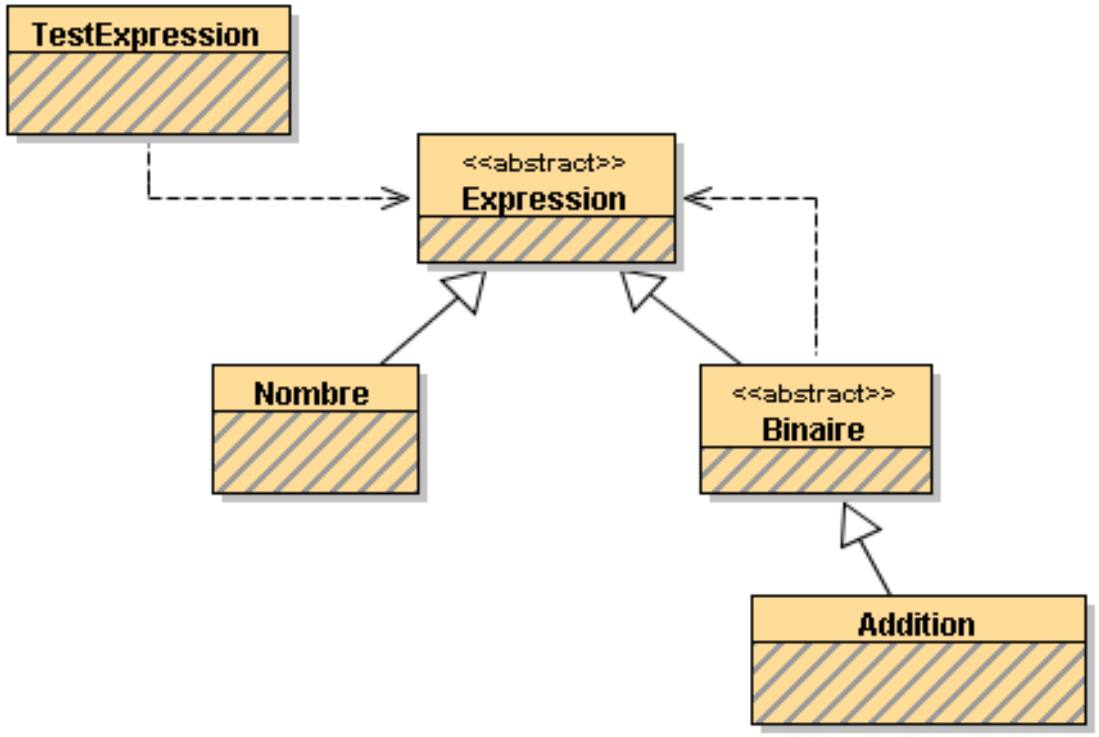
Le Pattern Expression : un premier bilan

- **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**

Pattern Interpréteur : l'original



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);
}}
```

Démonstration/Discussion

- **Simple**

- **Mais**

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);
}
```

– **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**

→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 !, 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

→ Existe-t-il un patron afin de prévenir toute évolution future ?

du logiciel …

Le pattern Visiteur vient à notre secours

- **il faudrait pouvoir effectuer de multiples interprétations de la même structure sans aucune modification du Composite**

- **→ Patron Visiteur**

Un visiteur par interprétation

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…**

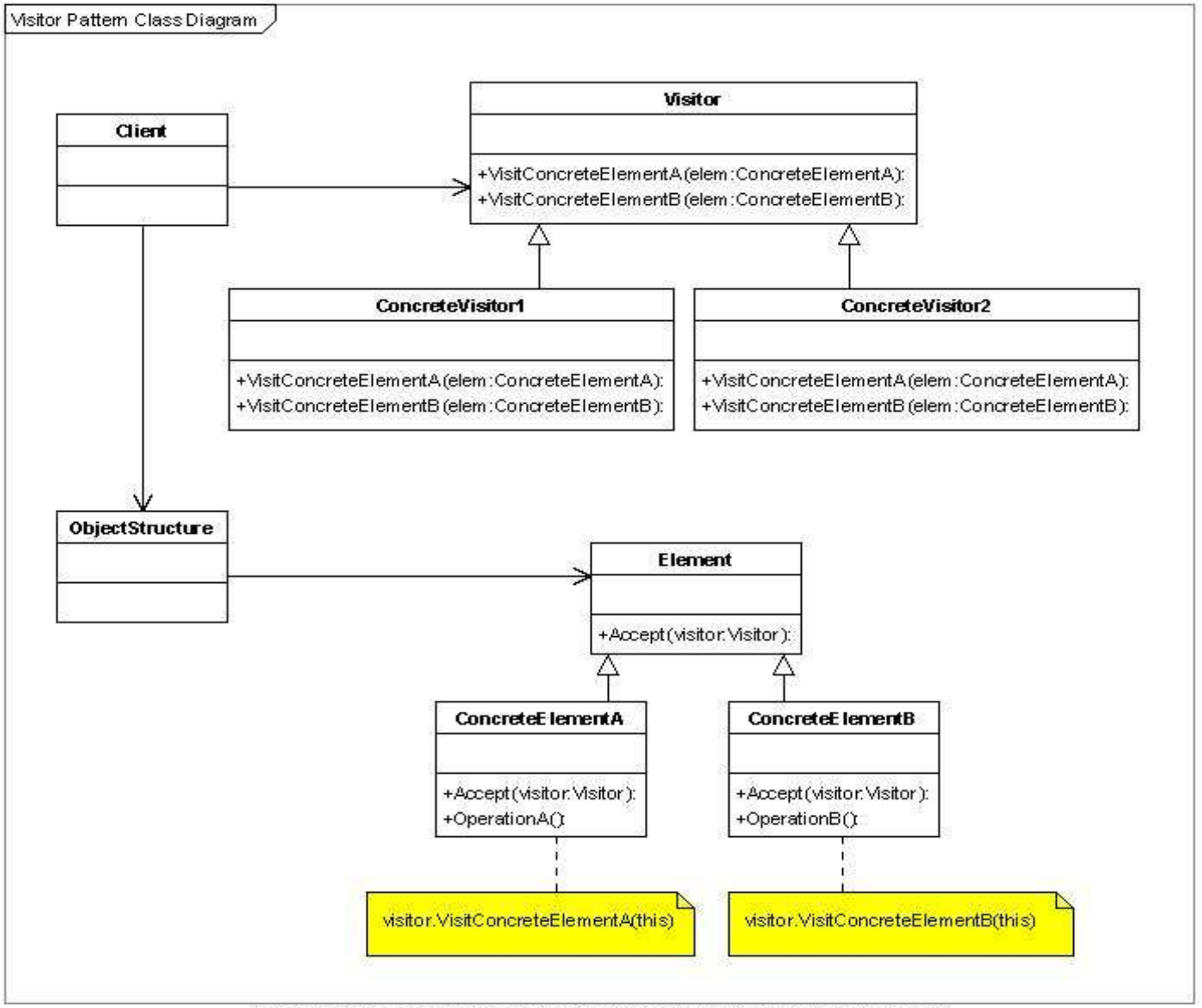
- **→ Ajout de cette méthode, ce sera la seule modification**

Du composite

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

- → emploi de la généricité, pour le type retourné

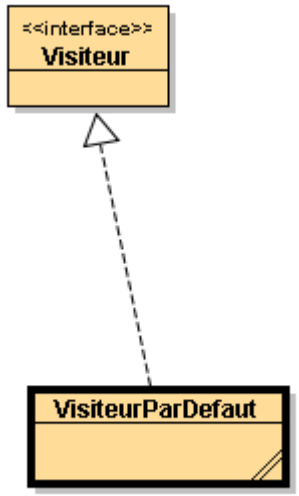
Le diagramme UML à l'origine



- **Lisible ?**

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 n;}
  public T visiteAddition(Addition a) {return a;}
}
```



*feuille concrète du composite

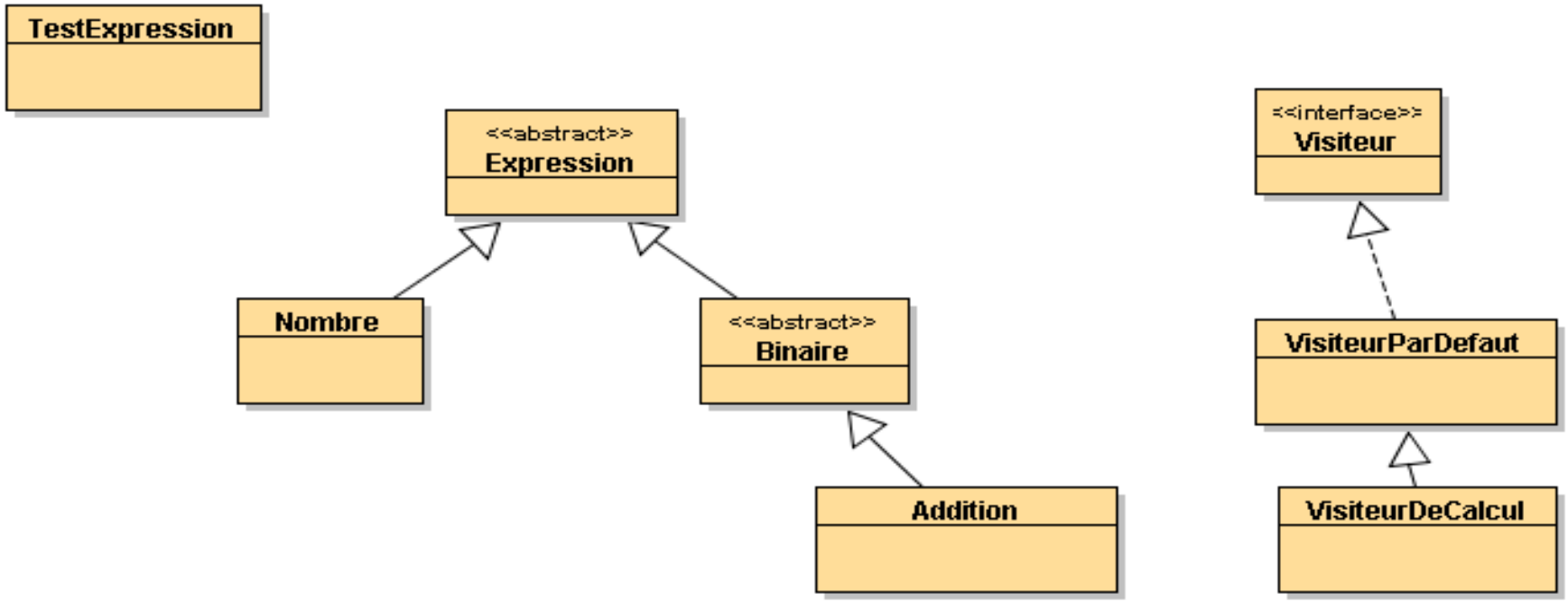
La classe Expression et Binaire
une fois pour toutes !

```
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

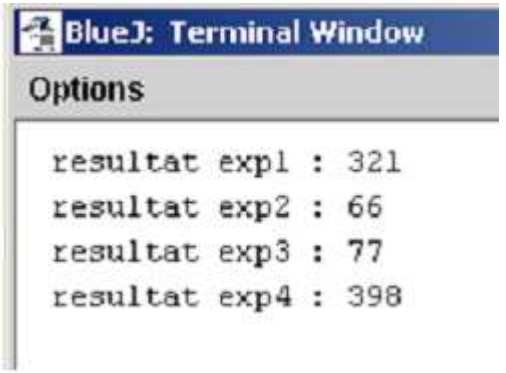


Le VisiteurDeCalcul

```
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;
  }
}
```

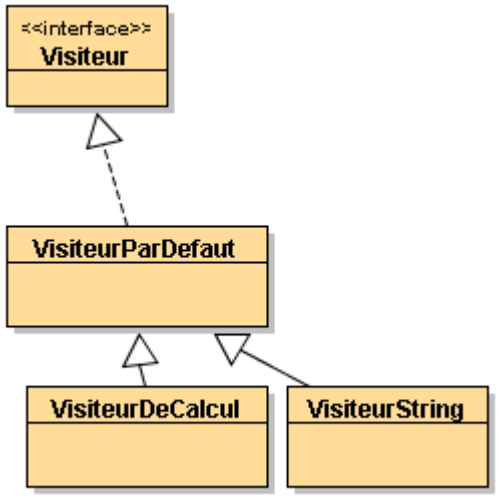
La classe TestExpression

```
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));
}}
```



Le Composite a une autre visite

```
public class VisiteurString
             extends VisiteurParDefaut<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**

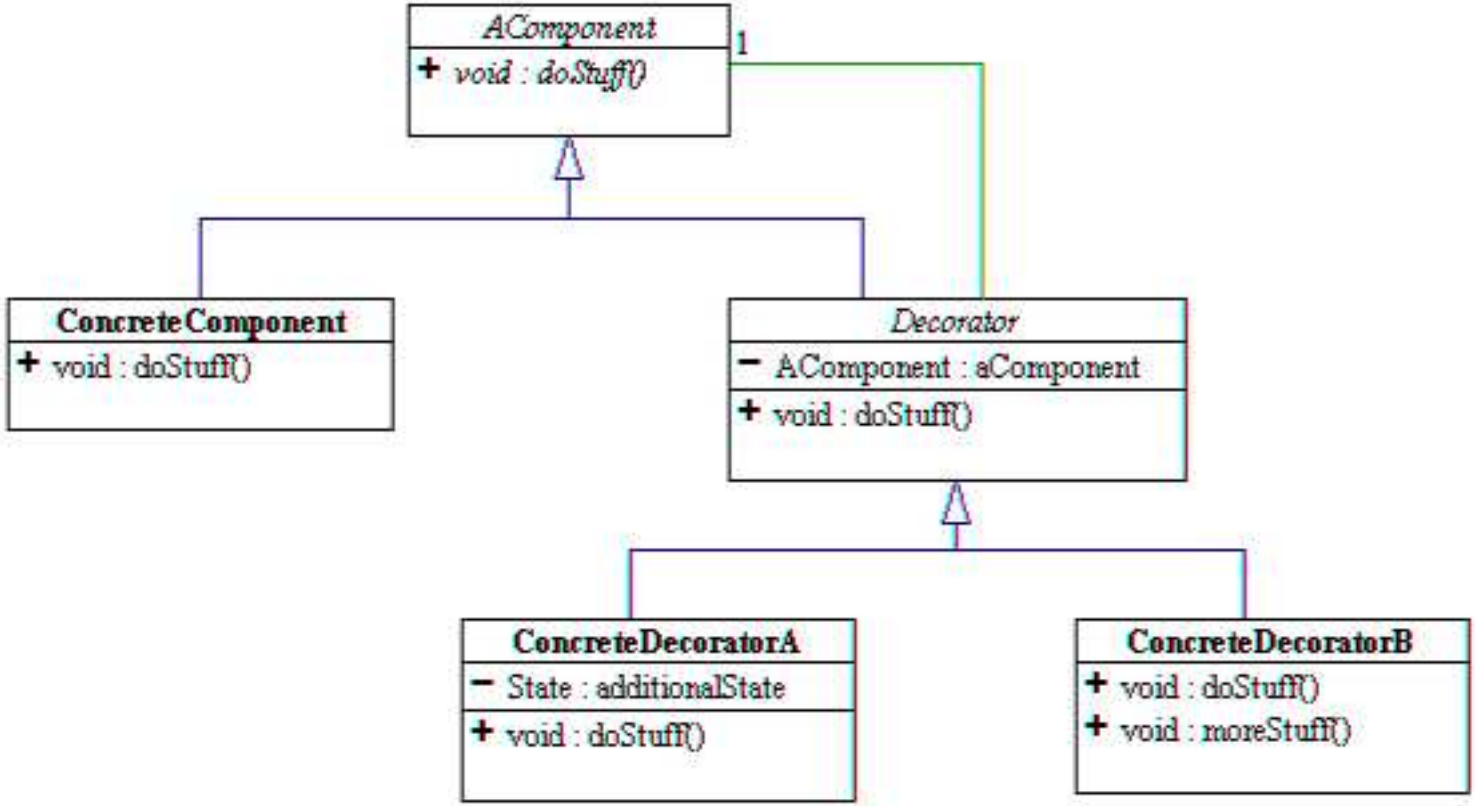
- **Alors à suivre…**

Discussion

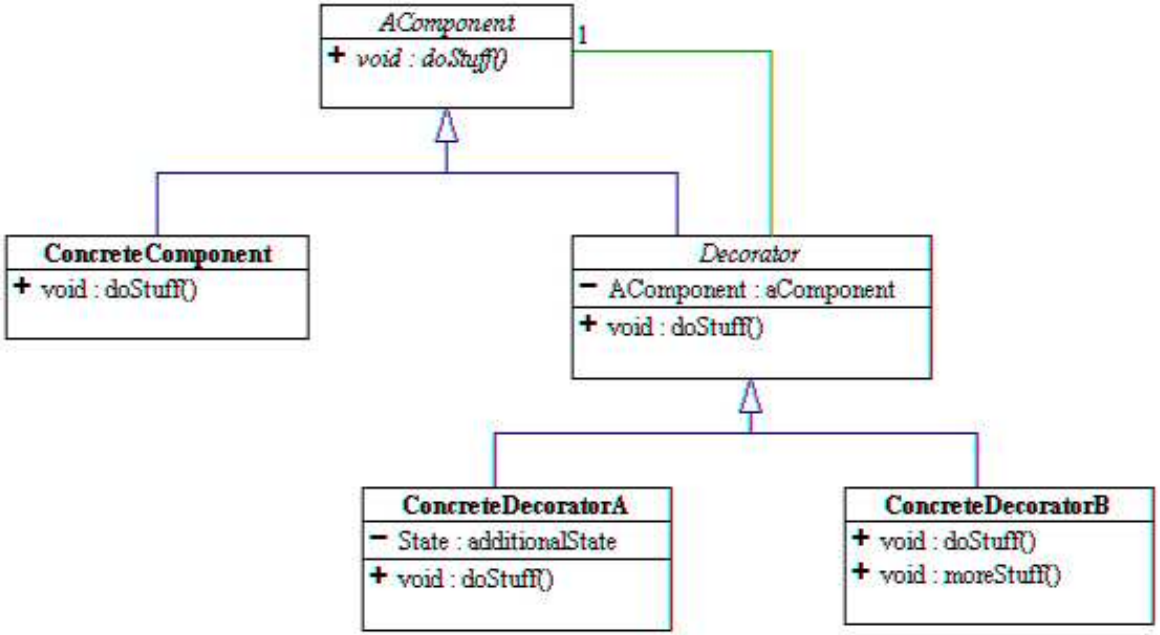
- **Composite**
- **Interpréteur**
- **Visiteur**
- **Itérateur, est-ce un oubli ?**
  - **Voir en annexe**

le Pattern Décorateur

- **Ajout dynamique de responsabilités à un objet**
- **Une alternative à l ’héritage ?**



Le Pattern : mise en œuvre



- *AComponent* interface ou classe abstraite
- **ConcreteComponent** implémente* *AComponent*

- *Decorator* implémente *AComponent* et contient une instance de *Acomponent*
- **ConcreteDecoratorA, ConcreteDecoratorB** héritent de *Decorator*

- *** implémente ou hérite de**

Une mise en œuvre(1)

```
public interface AComponent{
  public abstract String doStuff();
}
public class ConcreteComponent implements AComponent{
  public String doStuff(){
    //instructions concrètes;
    return "concrete..."
  }
}

public abstract class Decorator implements AComponent{
  private AComponent aComponent;
  public Decorator(AComponent aComponent){
      this.aComponent = aComponent;
  }
  public String doStuff(){
    return aComponent.doStuff();
  }
}
```

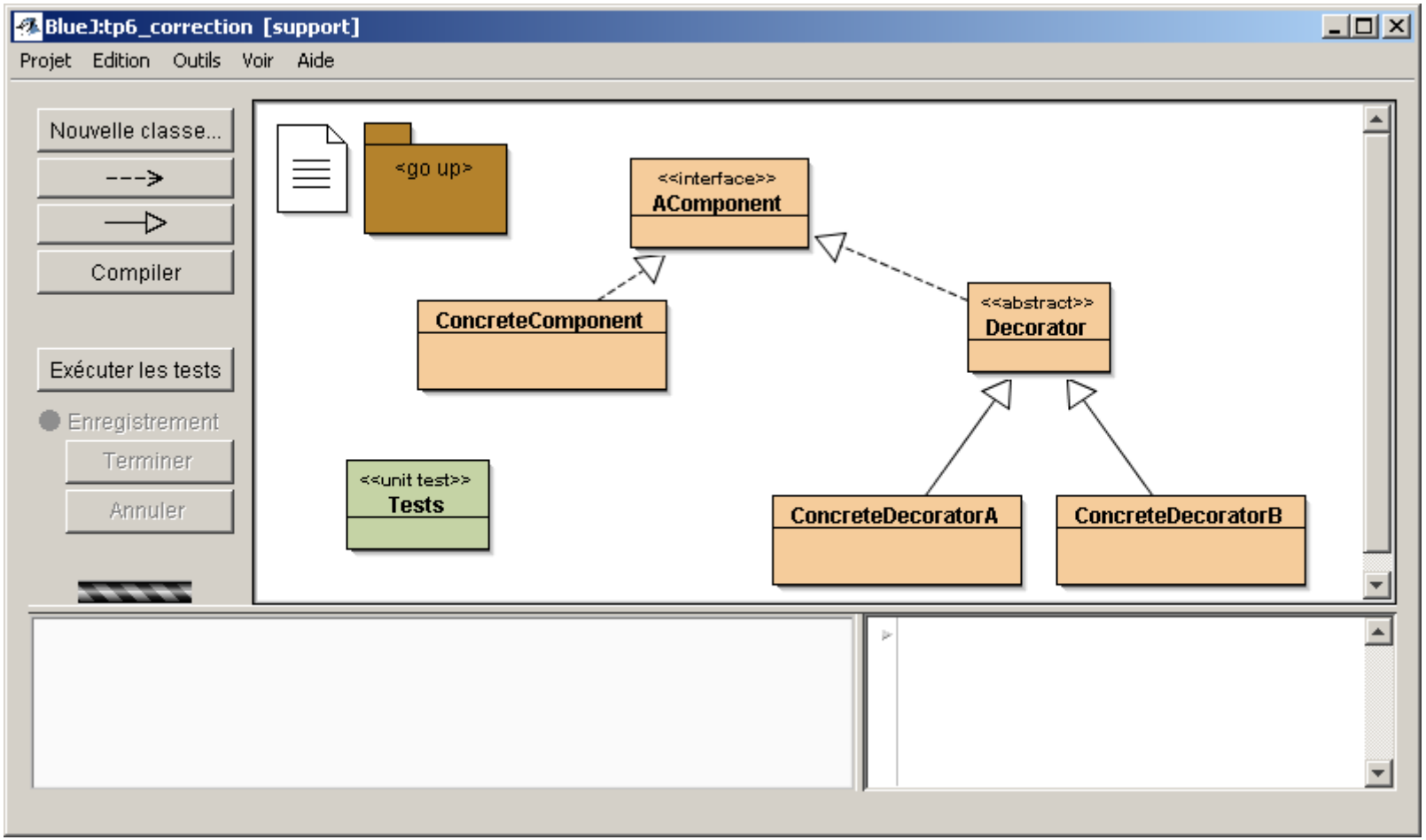
Mise en œuvre(2)

```
public class ConcreteDecoratorA extends Decorator{
  public ConcreteDecoratorA(AComponent aComponent){
    super(aComponent);
  }
  public String doStuff(){
    //instructions decoratorA;
    return "decoratorA... "  + super.doStuff();
  }
}
```

Déclarations

```
Acomponent concret = new ConcreteComponent();

Acomponent décoré = new ConcreteDecoratorA( concret);
décoré.doStuff();

Acomponent décoré2 = new ConcreteDecoratorA( décoré);
```

Mise en oeuvre, bluej



- **Démonstration ...**

Quelques assertions

Tests

Classe Édition Outils Options

Compiler | Défaire | Couper | Copier | Coller | Chercher... | Chercher suivant | Fermer | Implémentation

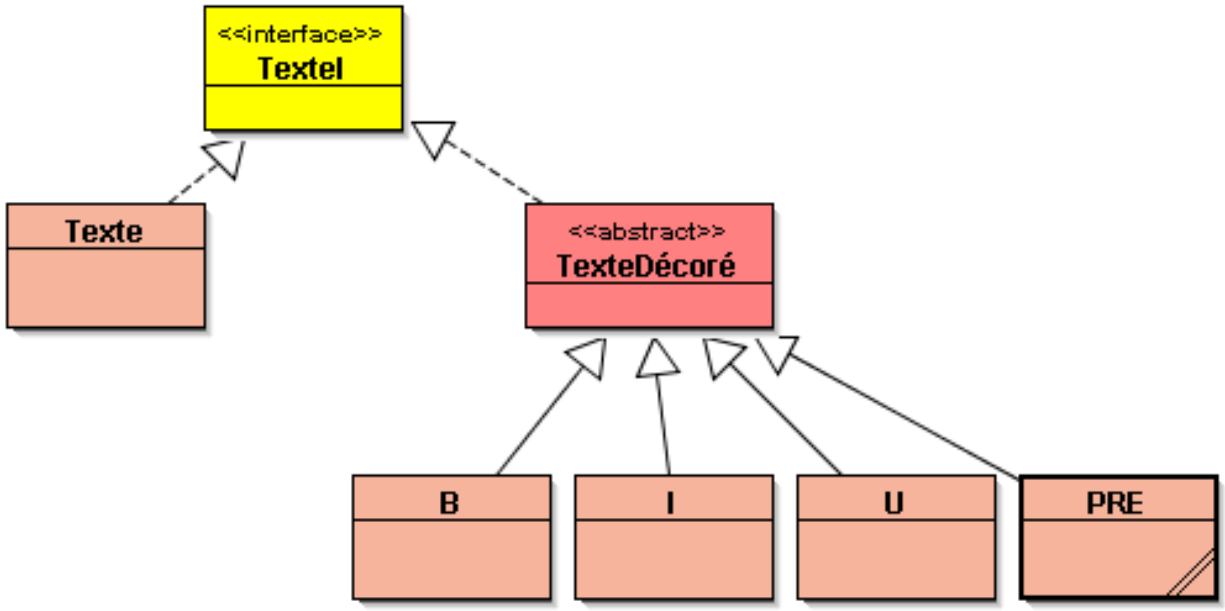
```
5
6   public void testl(){
7     AComponent concret = new ConcreteComponent();
8     assertEquals("concrete...", concret.doStuff());
9
10    AComponent decorA = new ConcreteDecoratorA(concret);
11    assertEquals("decoratorA...concrete...", decorA.doStuff());
12
13    AComponent decorBA = new ConcreteDecoratorB(decorA);
14    assertEquals("decoratorB...decoratorA...concrete...", decorBA.doStuff());
15
16    AComponent decorBB = new ConcreteDecoratorB(new ConcreteDecoratorB(concret));
17    assertEquals("decoratorB...decoratorB...concrete...", decorBB.doStuff());
18
19  }
20}
21
22
```

enregistré

- **(decoratorB (decoratorA (concrete)))**

Instance imbriquées

- **Instances gigognes + liaison dynamique = Décorateur**
- **Un autre exemple : un texte décoré par des balises HTML**
  - **<b>exemple<b>**



Le TexteI, Texte et TexteDécoré

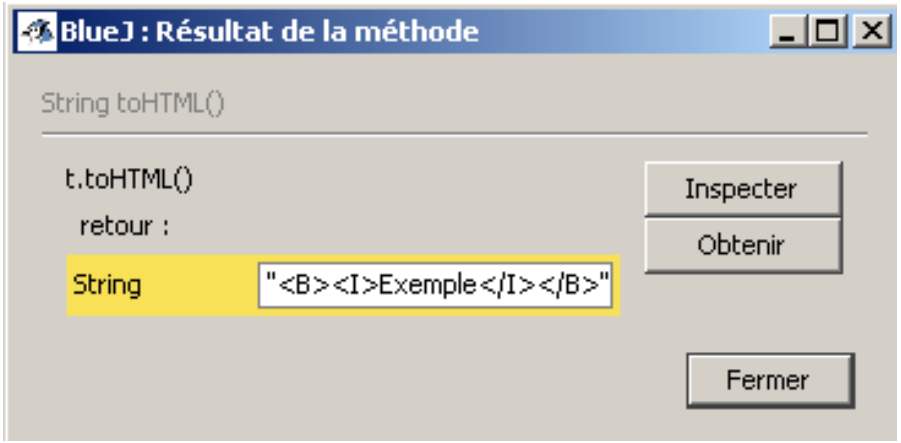
```
public interface TexteI{
  public String toHTML();
}

public class Texte implements TexteI{
  private String texte;
  public Texte(String texte){this.texte = texte;}
  public String toHTML(){return this.texte;}
}

public abstract class TexteDécoré implements TexteI{
  private TexteI unTexte;

  public TexteDécoré(TexteI unTexte){
    this.unTexte = unTexte;
  }
  public String toHTML(){
    return unTexte.toHTML();
  }
}
```

B, I …

```
public class B extends TexteDécoré{

  public B(TexteI unTexte){
    super(unTexte);
  }

  public String toHTML(){
    return "<B>" + super.toHTML() + "</B>";
  }
}

public class I extends TexteDécoré{

  public I(TexteI unTexte){
    super(unTexte);
  }

  public String toHTML(){
    return "<I>" + super.toHTML() + "</I>";
  }
}
```

Exemple

- **Textel t = new B(new I(new Texte("Exemple")));**
- **String s = t.toHTML();**



- **Démonstration**

Encore un autre exemple

- **inspiré de www.oreilly.com/catalog/hfdesignpat/chapter/ch03.pdf**

- **Confection d ’une Pizza à la carte**
  - **3 types de pâte**
  - **12 ingrédients différents, (dont on peut doubler ou plus la quantité)**
  - **si en moyenne 5 ingrédients, soit 792* combinaisons !**

  - **? Confection comme décoration ?**

  - **Une description de la pizza commandée et son prix**

* n parmi k, n! / k!(n-k)!

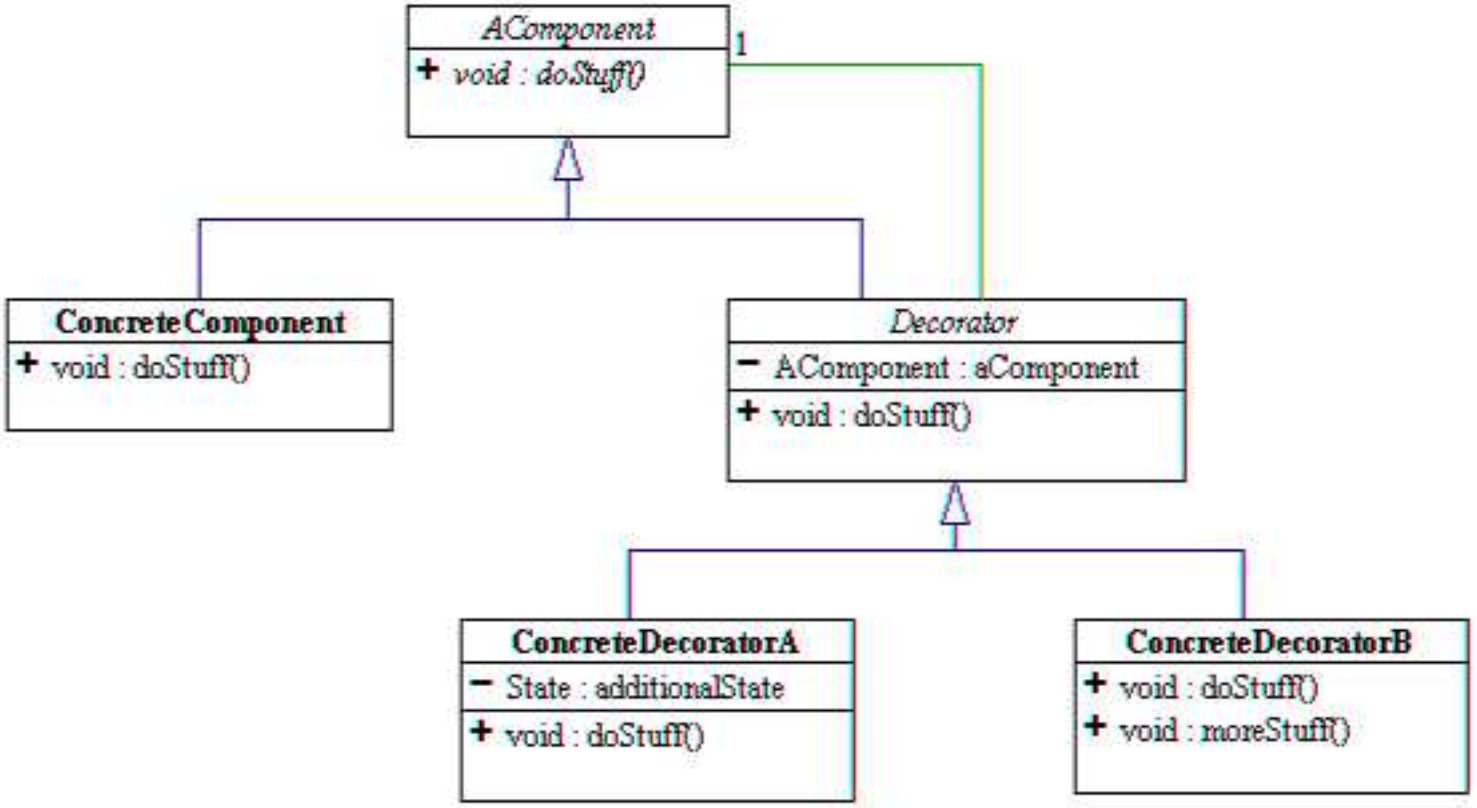
3 types de pâte

- **Pâte solo, (très fine…)**
- **Pâte Classic**
- **Pâte GenerousCrust©**

12 ingrédients différents

- **Mozarella, parmesan, Ham, Tomato, Mushrooms, diced onion, etc...**

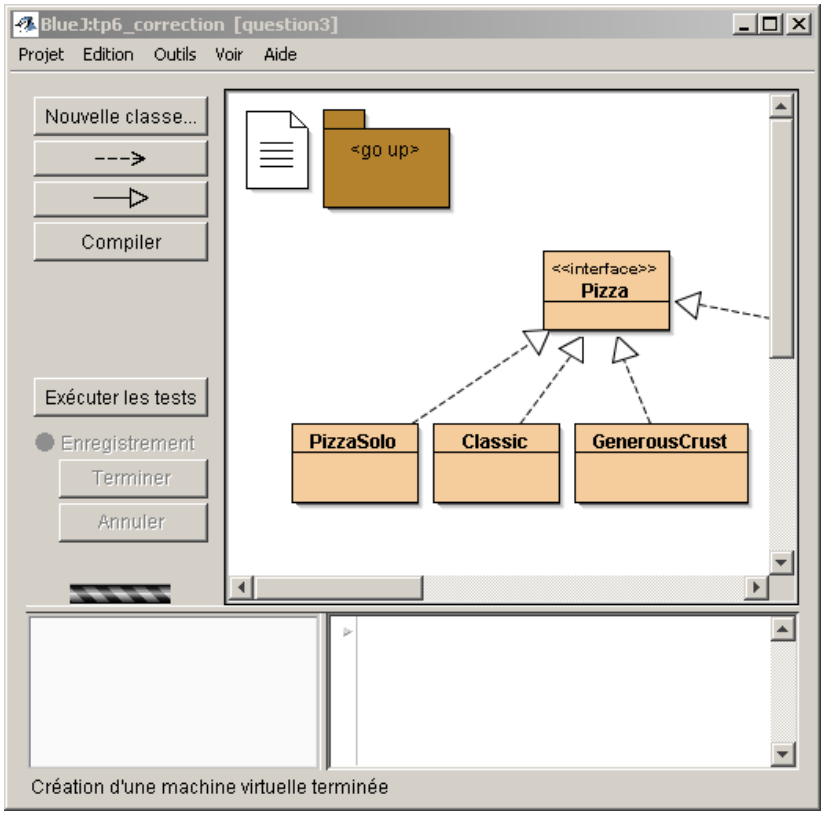
Le décorateur de pizza



- AComponent --> une interface Pizza
- ConcreteComponent --> les différentes pâtes

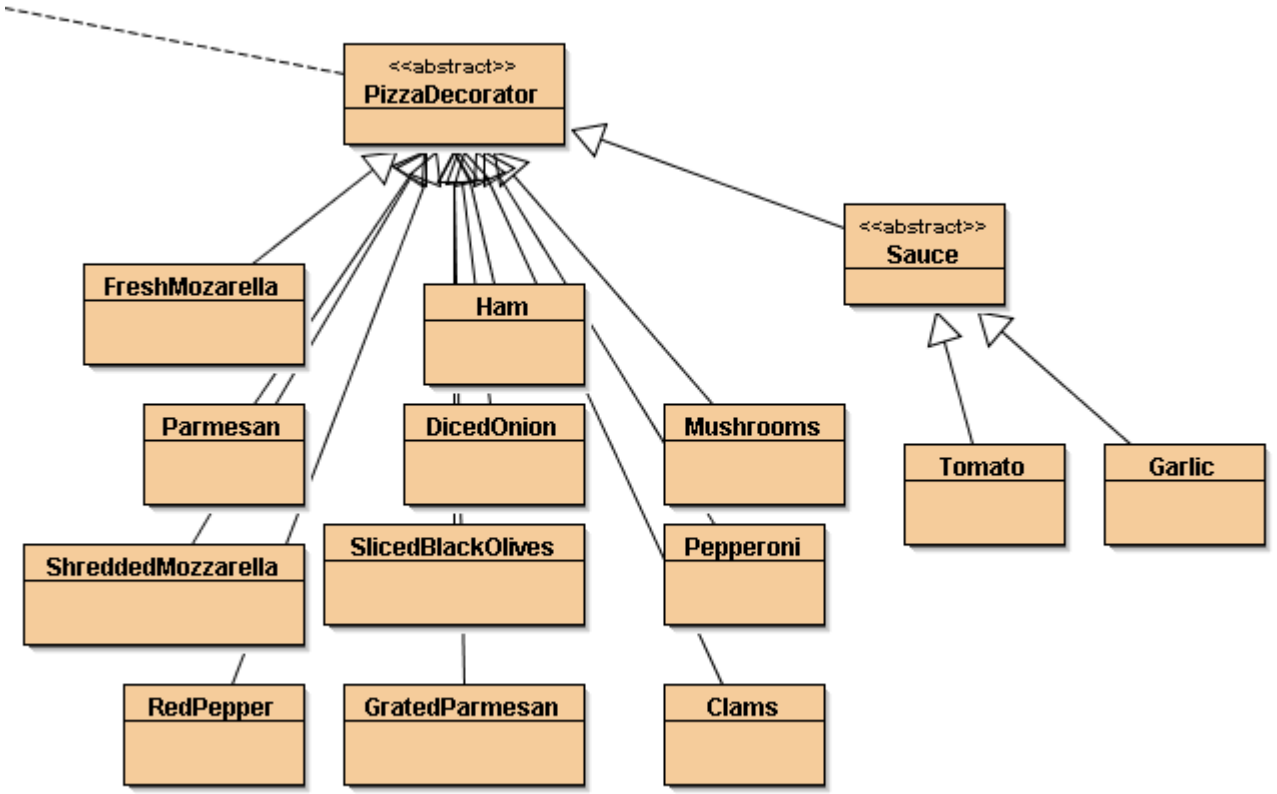
- Decorator l'ingrédient, la décoration
- ConcreteDecorator Parmesan, Mozarella, ...

3 types de pâte




```
public interface Pizza{
  abstract public String getDescription();
  abstract public double cost();
}
```

Les ingrédients



- !

PizzaDecorator

```
public abstract class PizzaDecorator implements Pizza{
  protected Pizza pizza;
  public PizzaDecorator(Pizza pizza){
     this.pizza = pizza;
  }
  public abstract String getDescription();
  public abstract double cost();
}
```

Ham & Parmesan

```
public class Ham extends PizzaDecorator{
  public Ham(Pizza p){super(p);}
  public String getDescription(){
    return pizza.getDescription() + ", ham";
  }
  public double cost(){return pizza.cost() + 1.50;}
}


public class Parmesan extends PizzaDecorator{
  public Ham(Pizza p){super(p);}
  public String getDescription(){
    return pizza.getDescription() + ", parmesan";
  }
  public double cost(){return pizza.cost() + 0.75;}
}
```

Pizza Solo + Mozarella + quel coût ?

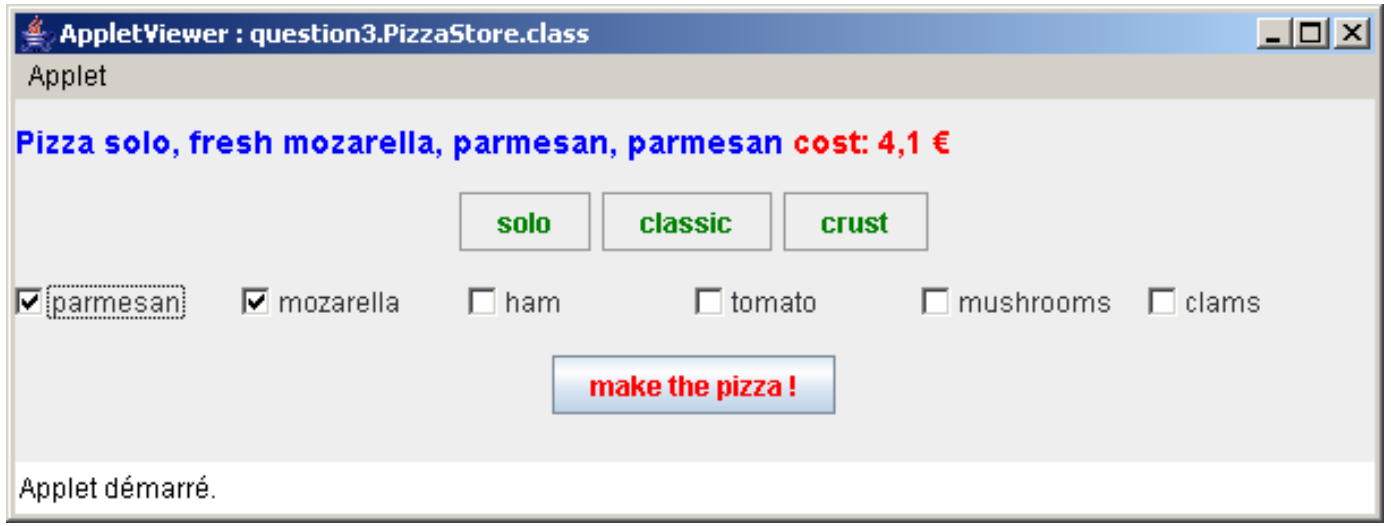
```
double coût = new Parmesan(
                  new FreshMozarella(
                    new PizzaSolo())).cost();
assert coût == 5.8;
```

Une pizza aux 2 fromages

```
Pizza p=new Mozarella(new Mozarella(new Parmesan(new PizzaSolo())));
```

Magique ou liaison dynamique ???

L 'IHM du pizzaiolo



- **Pizza p; // donnée d 'instance de l 'IHM**
- **choix de la pâte, ici solo**

```
boutonSolo.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent ae){
      pizza = new PizzaSolo();
      validerLesDécorations();
    }
  });
```

L 'IHM : les ingrédients ici Ham*2

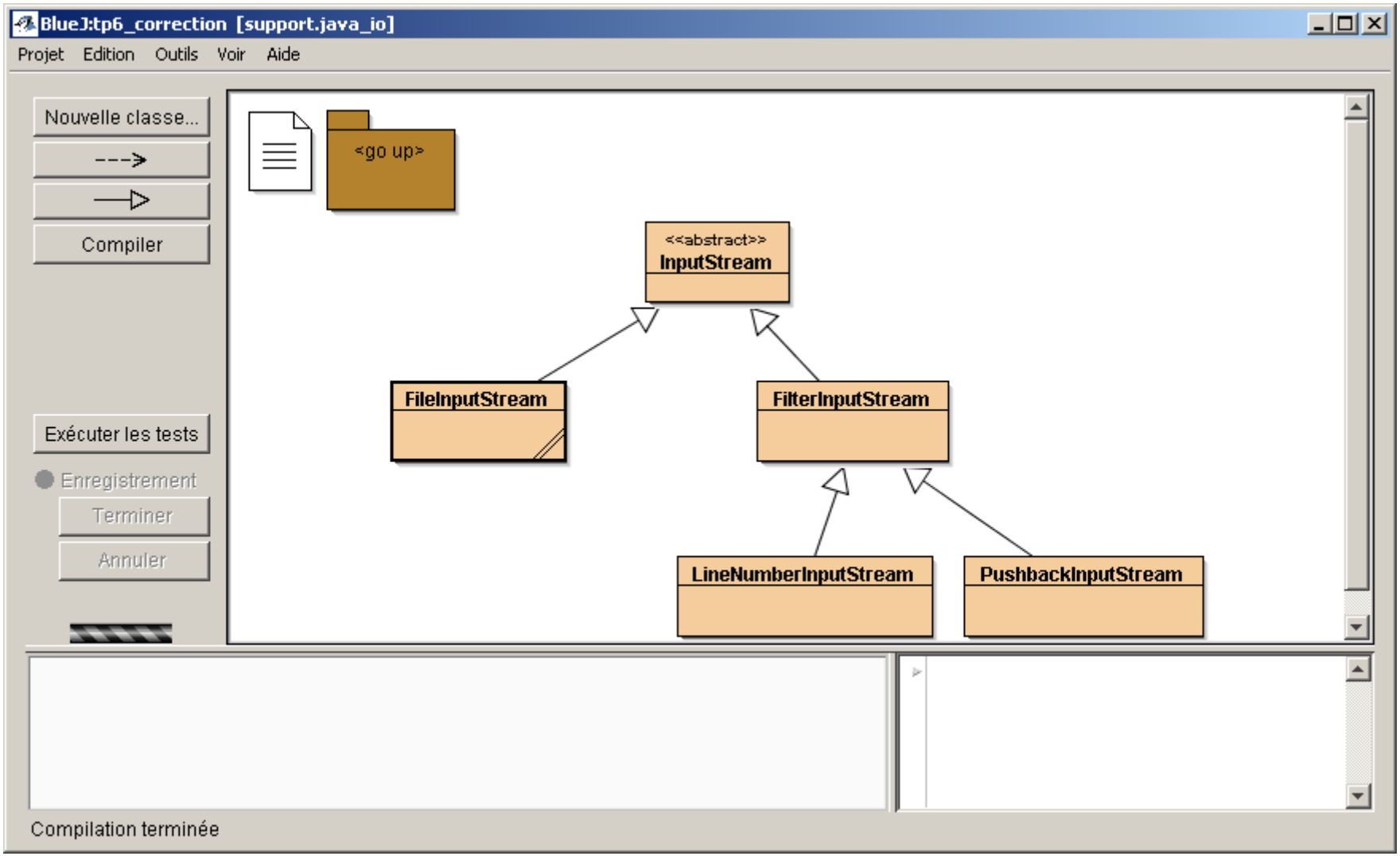

```
ham.addItemListener(new ItemListener(){
   public void itemStateChanged(ItemEvent ie){
     if(ie.getStateChange()==ItemEvent.SELECTED)
        pizza = new Ham(pizza);
     afficherLaPizzaEtSonCoût();
   }
});
```

http://jfod.cnam.fr/progAvancee/tp8/question1.PizzaStore.html

Démonstration, Discussion

- **Est-ce une alternative à l'héritage ?**
  - **Héritage statique,**
- **Instance au comportement dynamique**

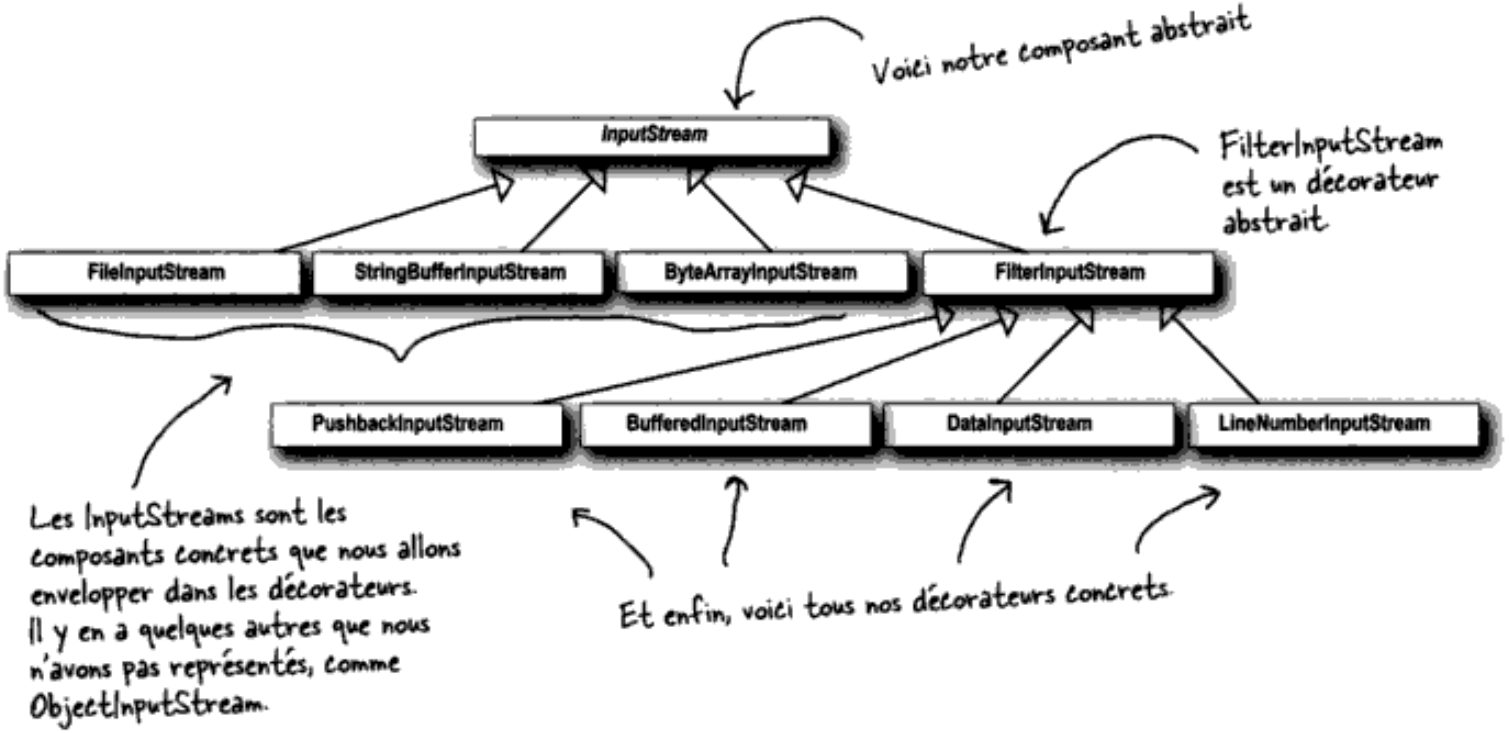
java.io



- **Le Décorateur est bien là**

Extrait de java : tête la première

Décoration des classes de java.io



Utilisation

```
InputStream is = new LineNumberInputStream(
                        new FileInputStream(
                            new File("LectureDeFichierTest.java")));
```

Reader, récidive

```
LineNumberReader r =
        new LineNumberReader(
            new FileReader(new File("README.TXT")));
```

- System.out.println(r.readLine());
- System.out.println(r.readLine());

java.net.Socket

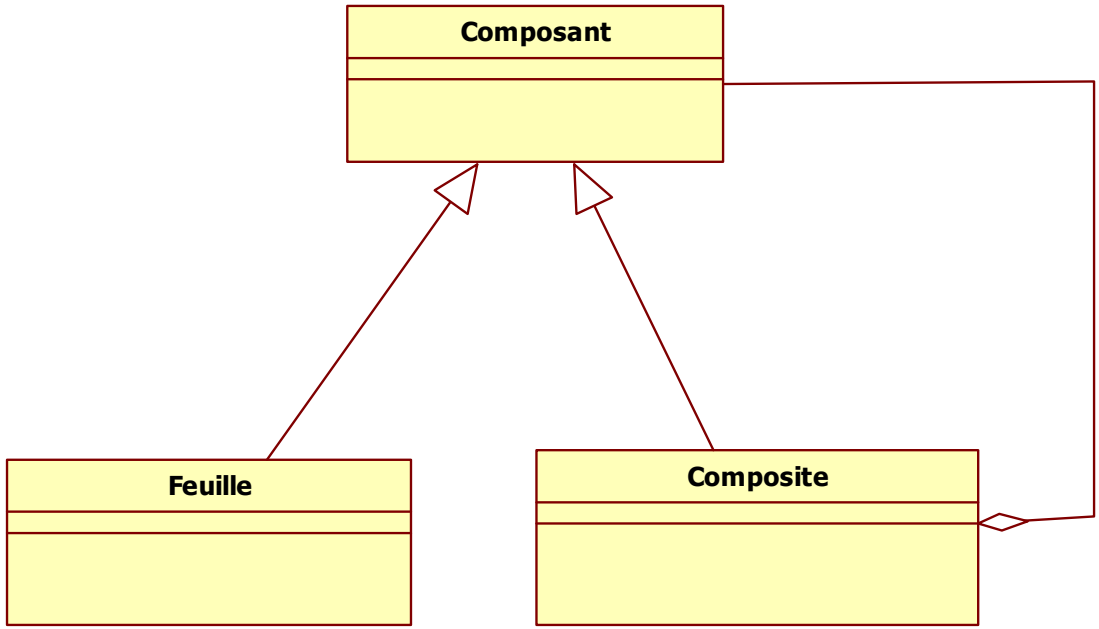
```
Socket socket = new Socket("vivaldi.cnam.fr", 5000);


 BufferedReader in =
   new BufferedReader(
      new InputStreamReader(socket.getInputStream()));


    System.out.println(in.readLine());
```

Conclusion

Annexes

- **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**

Composite et Iterator



- **Structure récursive « 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 commposite
    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

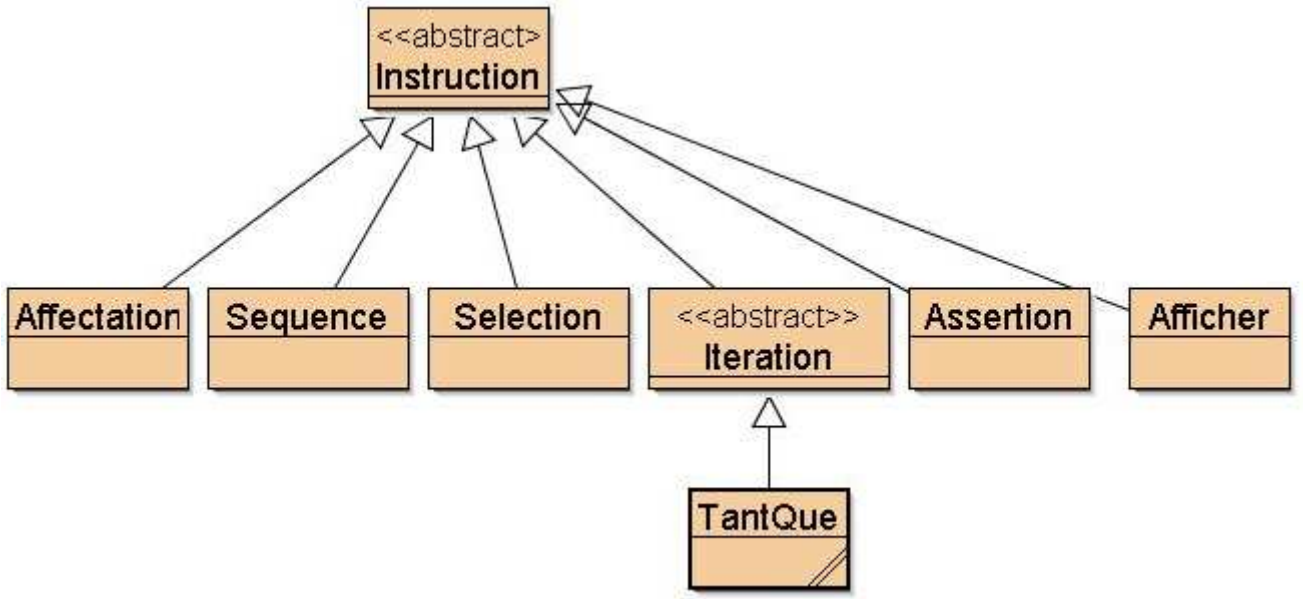
```
public void testIterator (){
    try{
      Composite g = new Composite();
      g.ajouter(new Composant());
      g.ajouter(new Composant());
      g.ajouter(new Composant());
      Composite g1 = new Composite();
      g1.ajouter(new Composant());
      g1.ajouter(new Composant());
      g.ajouter(g1);

      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

- Itérator compliqué ?
- Préférer le visiteur

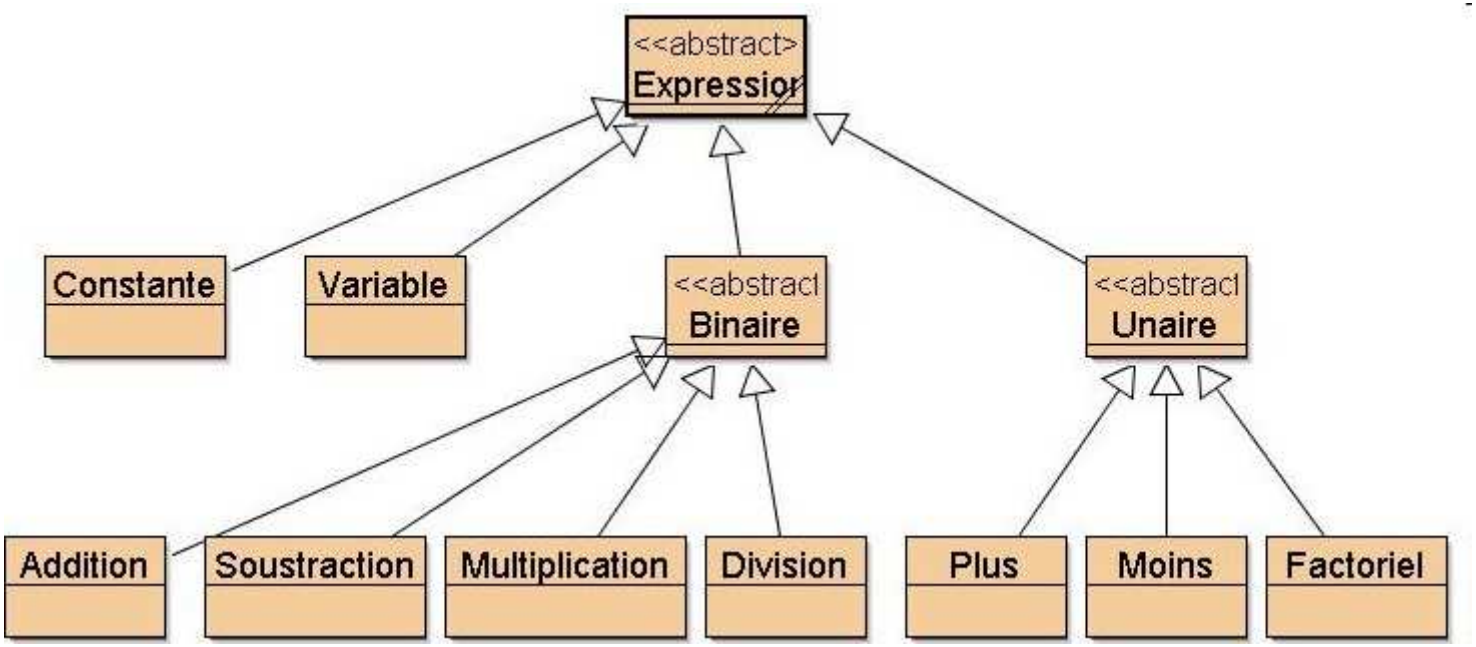
Un petit langage

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



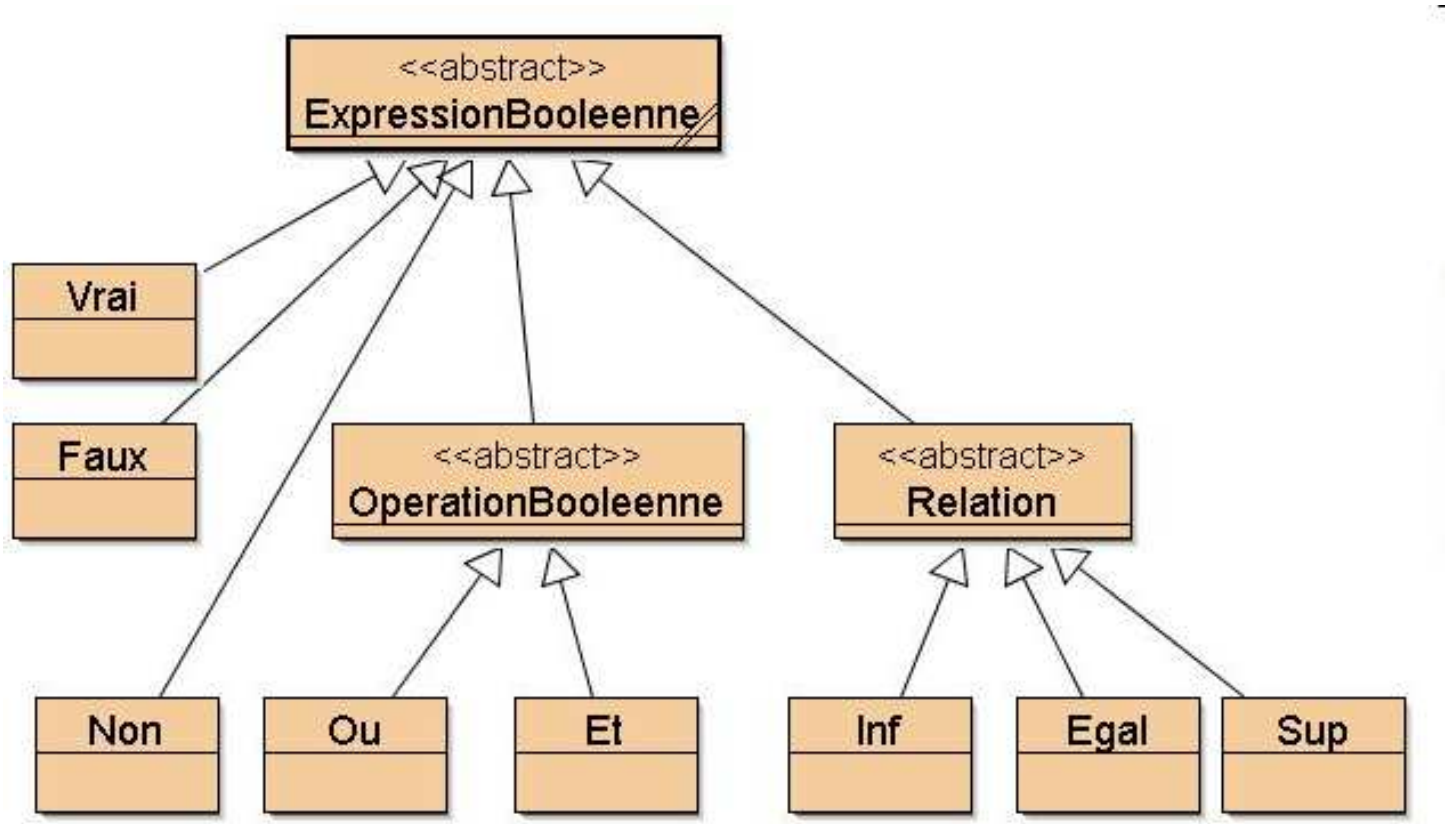
Expressions

- **Composite Expressions arithmétiques (cf. ce support)**



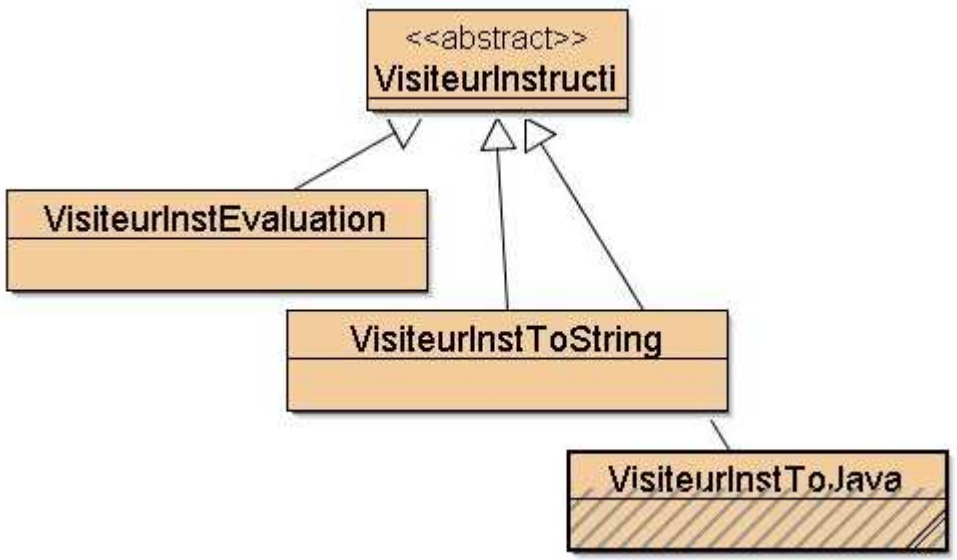
Expressions

- **Composite Expressions booléennes**



Visiteurs

- **Visite d'une instance d'un composite**
  - **Pour une conversion en String**
  - **Pour une évaluation**
  - **Pour une conversion en source Java**
  - **…**



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 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
}
```

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);
    }
  }

}
```

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
}
```

Le bytecode généré

compatible jasmin *jasmin.sourceforge.net/*

```
.class public TestsFactoriel
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit stack 3
.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:
        return
.end method
```

Un détail : la visite pour TantQue

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

Démonstration

Question …