Composite, Interpréteur & décorateur

Cnam Paris
jean-michel Douin
Version du 9 Octobre 2006

Notes de cours java : les patterns Composite, interpréteur et Décorateur

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

Sommaire

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

Principale bibliographie

- **GoF95**

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

+

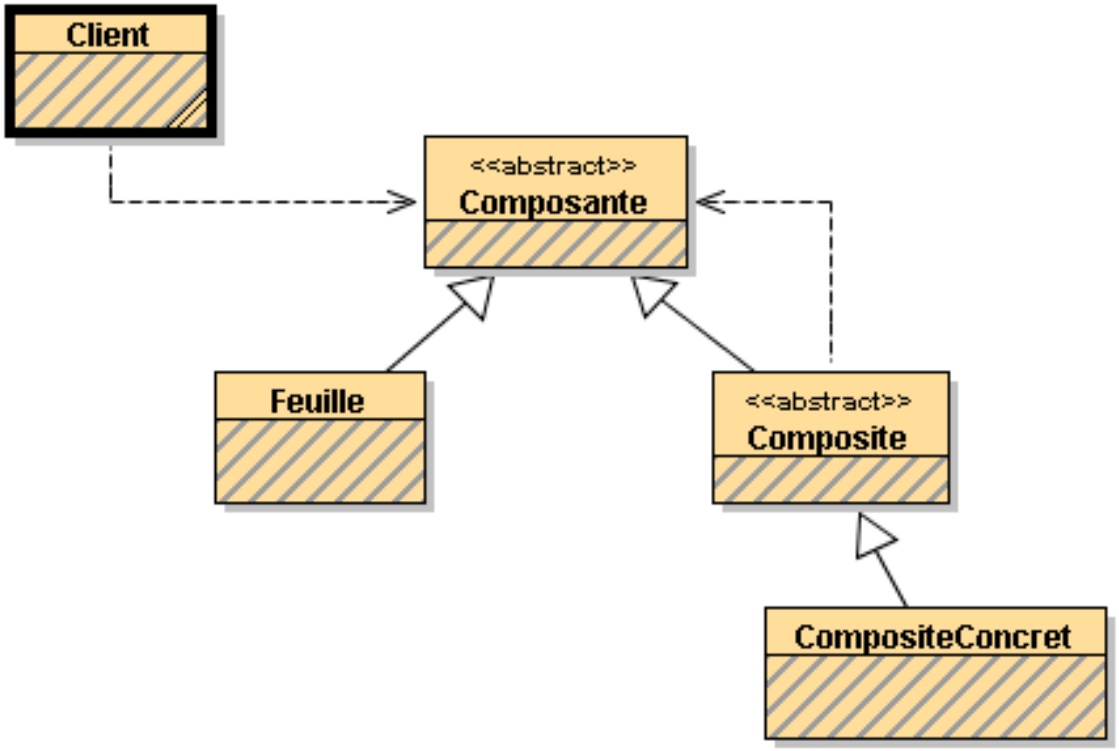
http://www.eli.sdsu.edu/courses/spring98/cs635/notes/composite/composite.html
http://www.patterndepot.com/put/8/JavaPatterns.htm

+

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

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

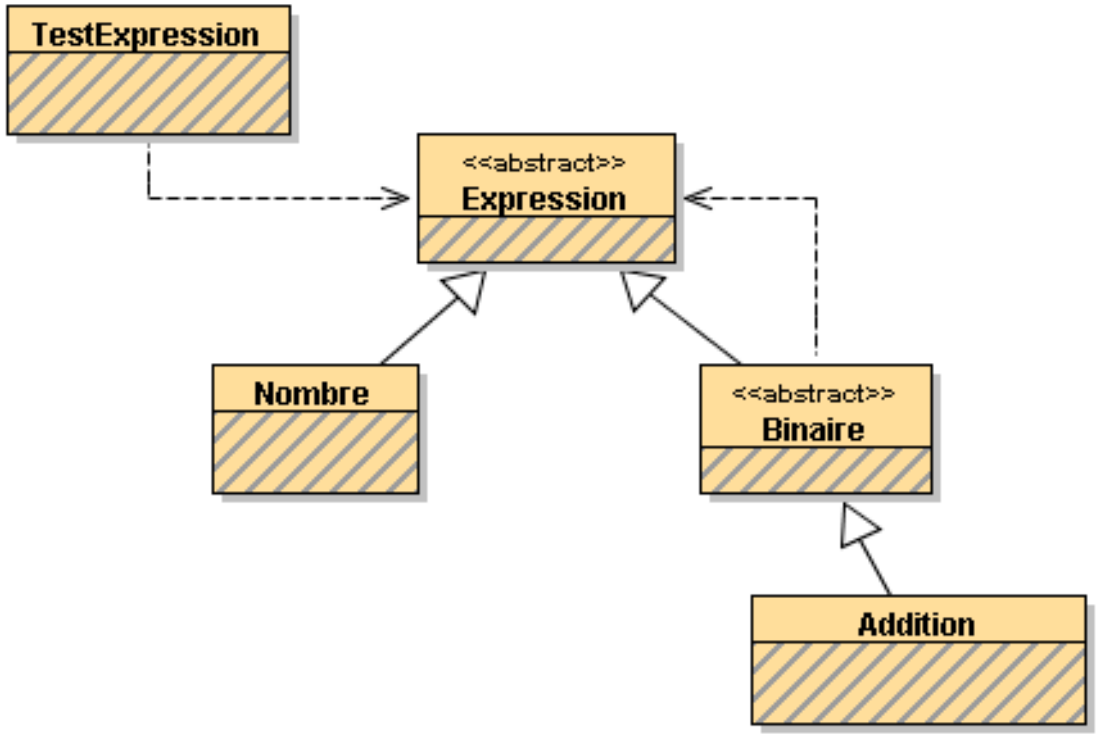
Structures récursives : le pattern Composite



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

Tout est Composante

Un exemple : Expression



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

Tout est Expression

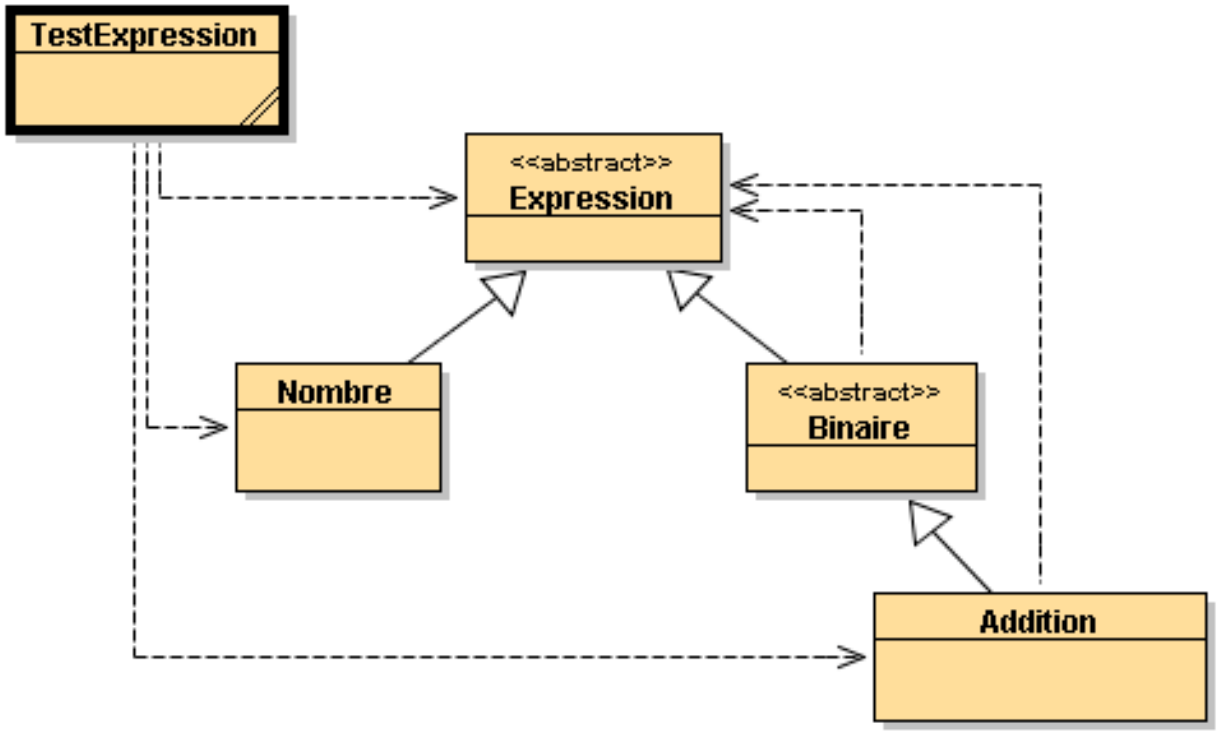
Composite et Expression en Java

```
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;
  }
}
```

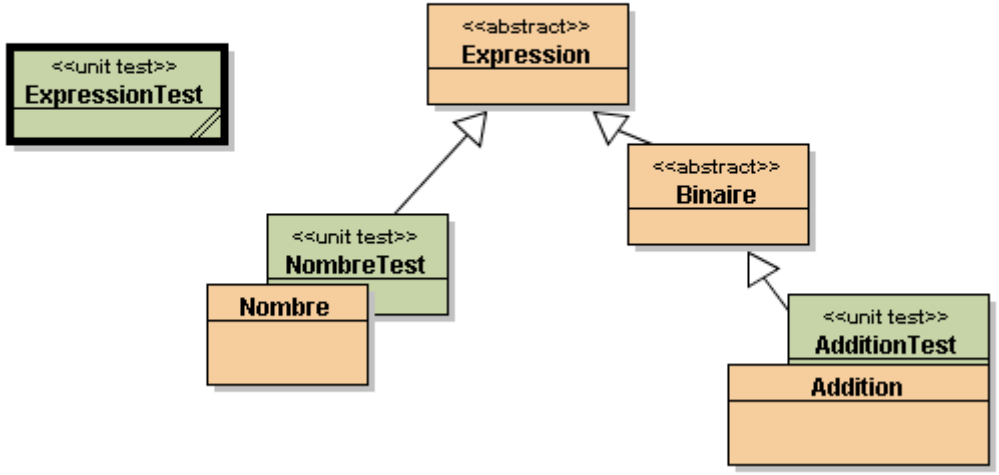
Composite et Expression en Java

```
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


<<unit test>>
ExpressionTest
<<abstract>>
Expression
<<abstract>>
Binaire
<<unit test>>
NombreTest
Nombre
<<unit test>>
AdditionTest
Addition

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 ?

- **Compent, Container, Label, Jlabel ….**

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 (composés des composants ou de composites)

  Fenêtre, applette, …

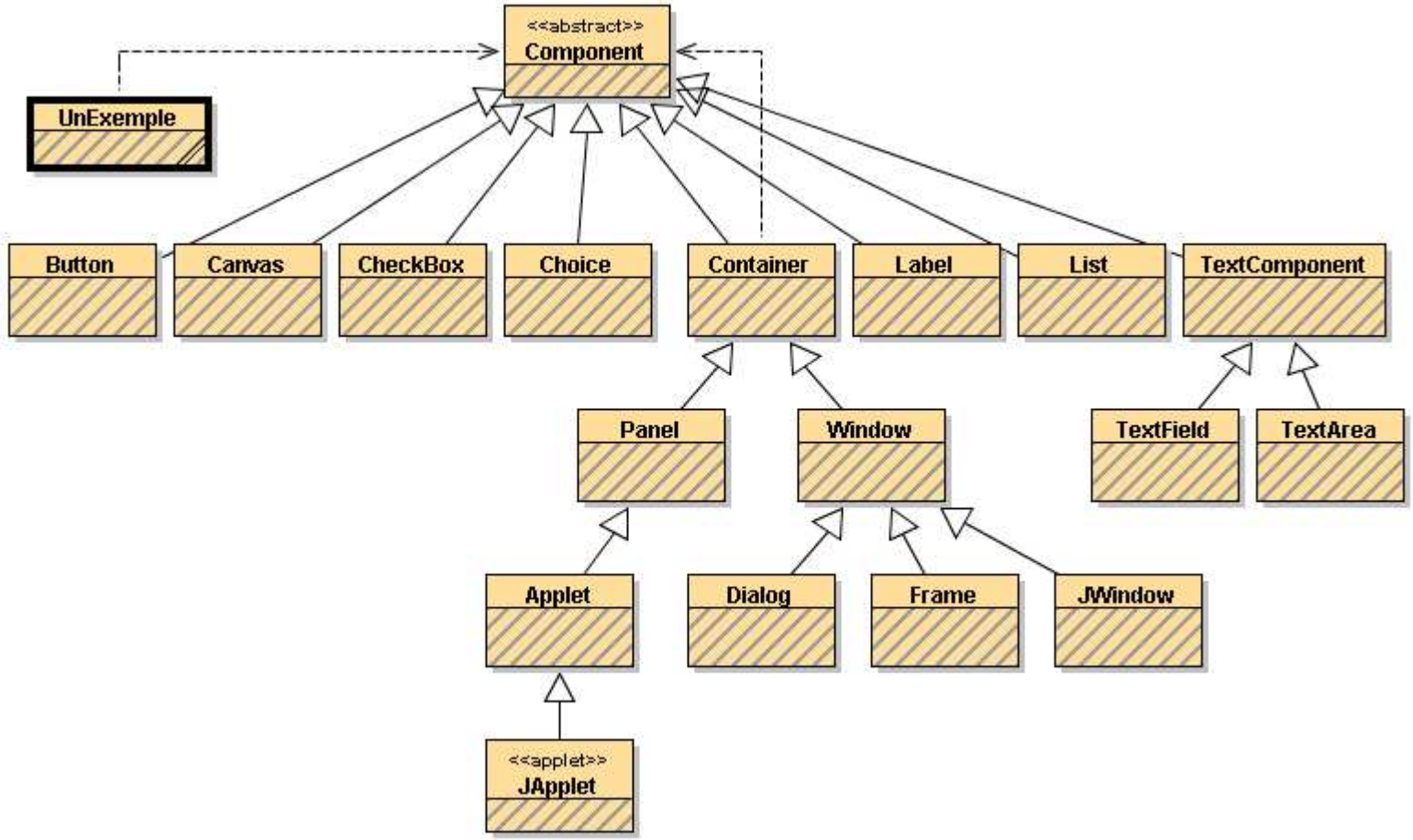
- **Le Pattern Composite est utilisé**

  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)

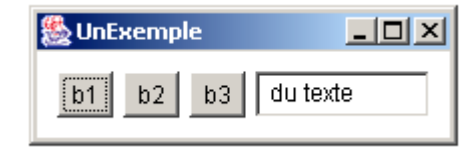
- Expression e = new Addition(new Nombre(1),new Nombre(3));

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

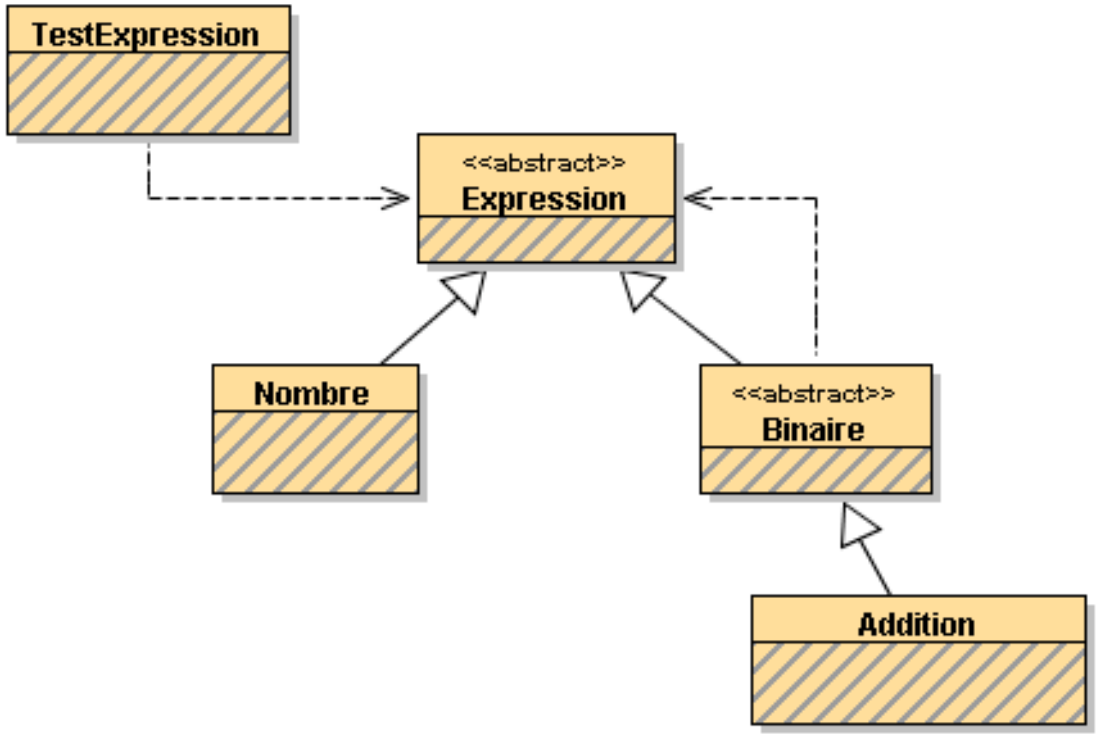
  Techniques de classes abstraites

  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
```

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);
}
```

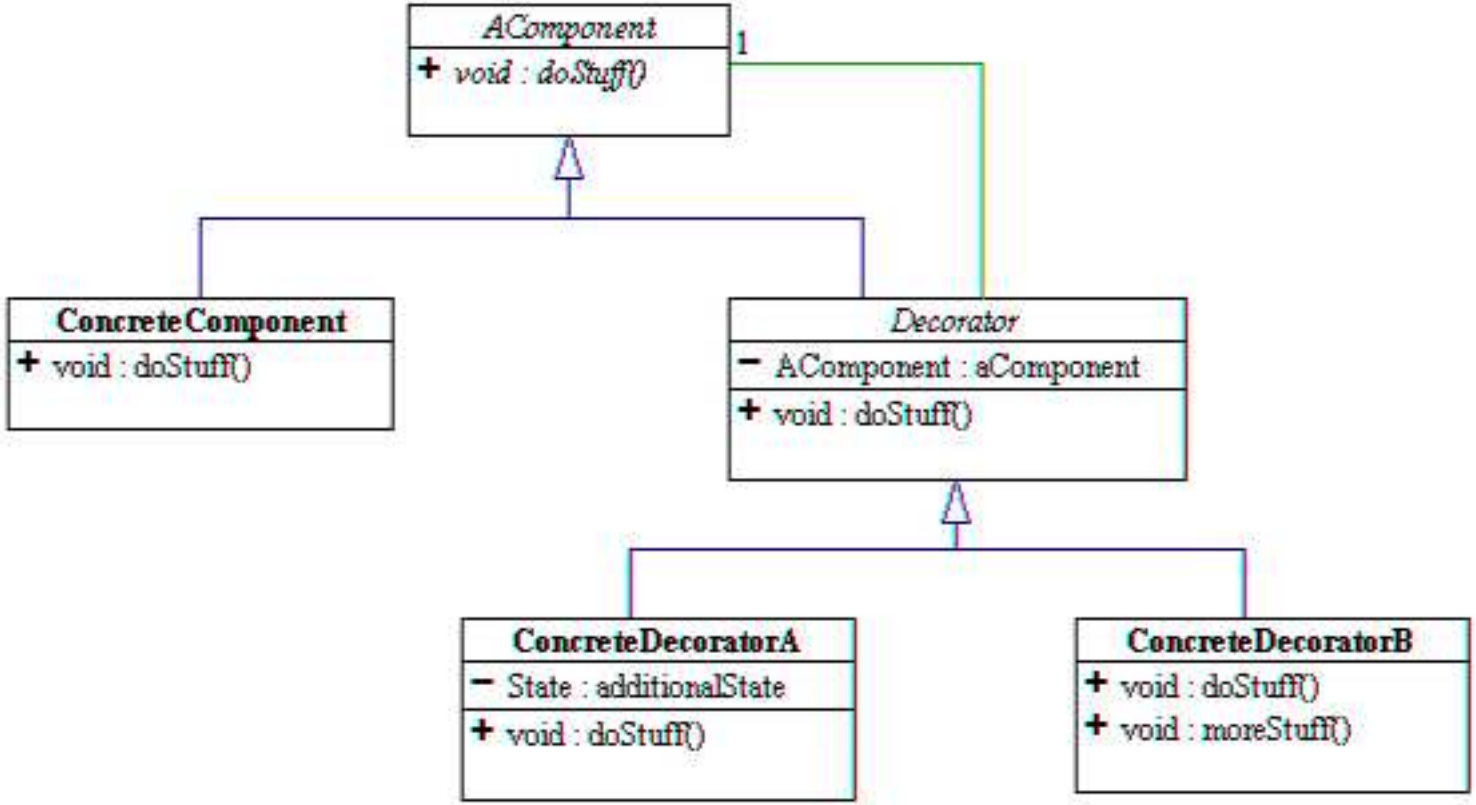
Dans la famille des itérateurs avez-vous le visiteur …

Discussion

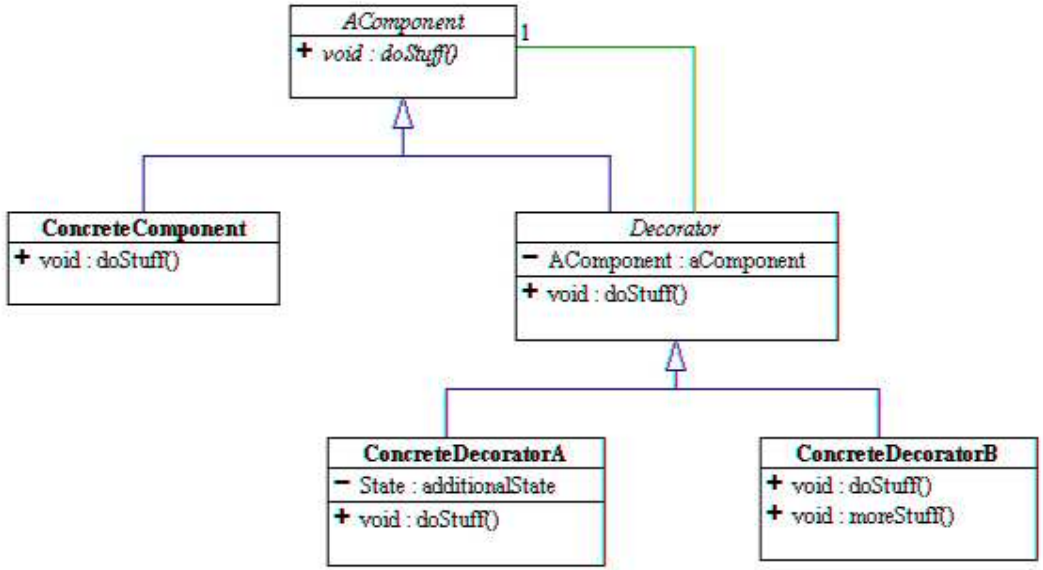
- **Composite**
- **Interpréteur**

le Pattern Décorateur

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



Le Pattern : mis 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();
  }
}
```

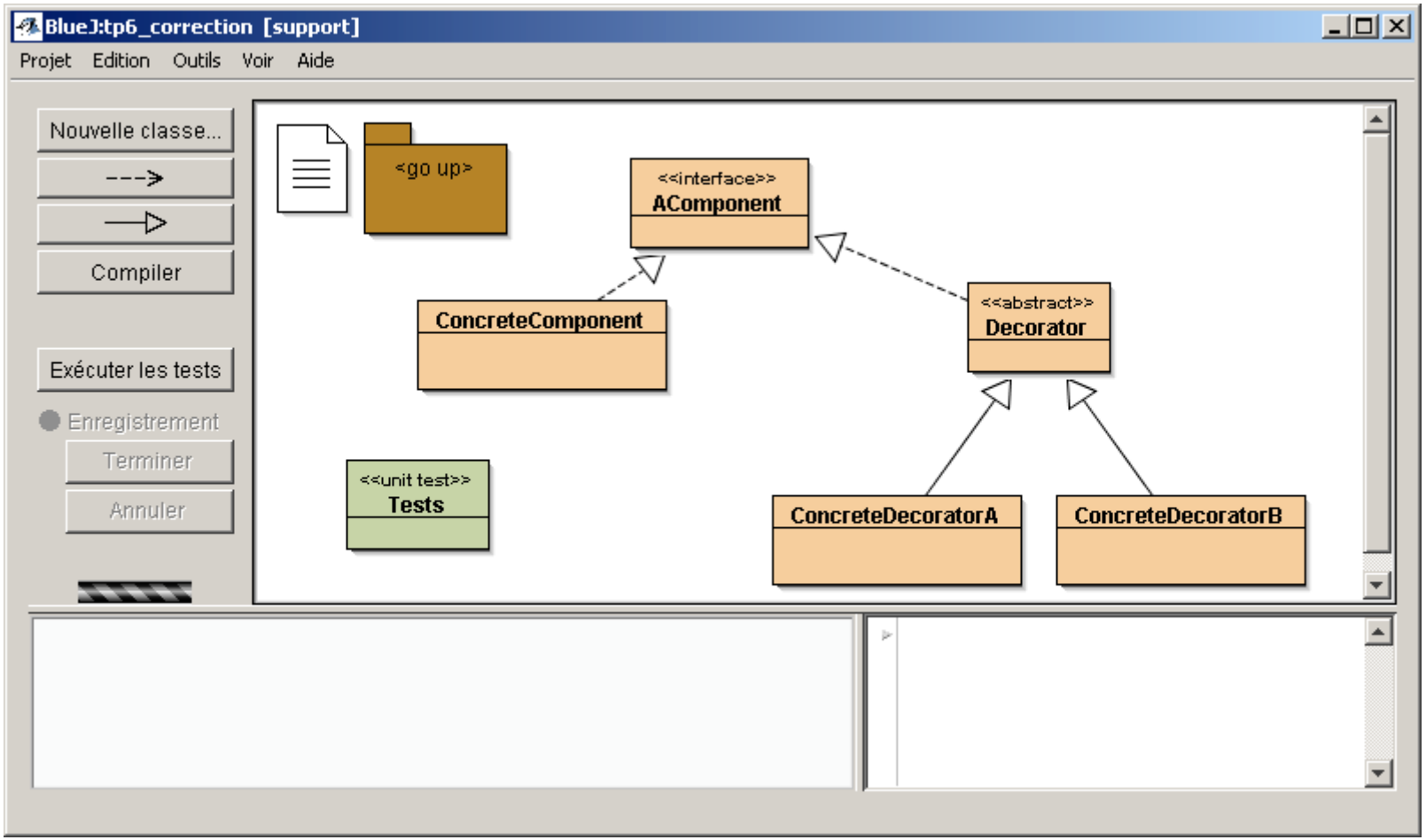
Mise en œuvre(2)

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

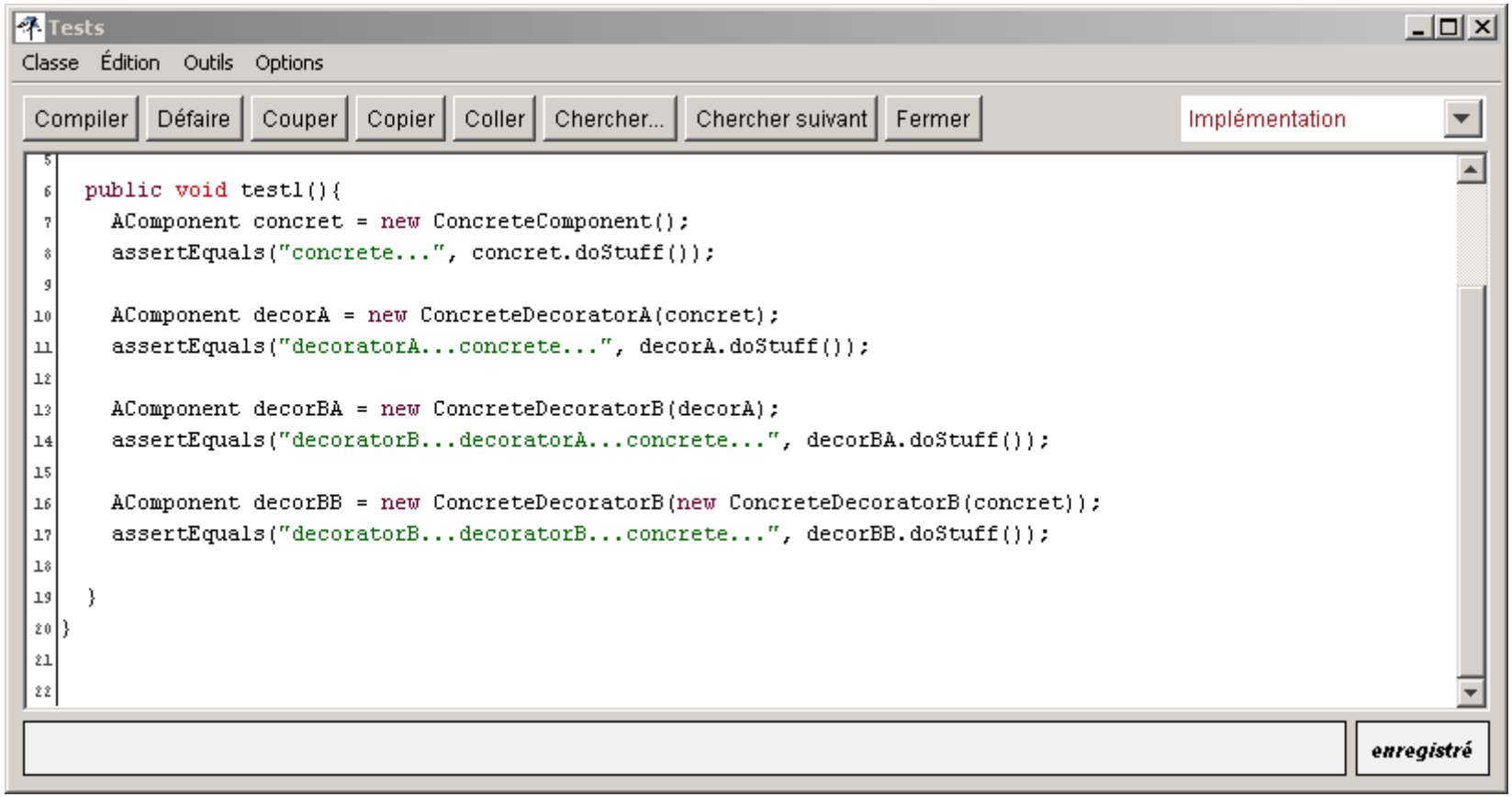
Déclarations

```
Acomponent concret = new ConcreteComponent();

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

Mise en oeuvre, bluej



Quelques assertions



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

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)

  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)!

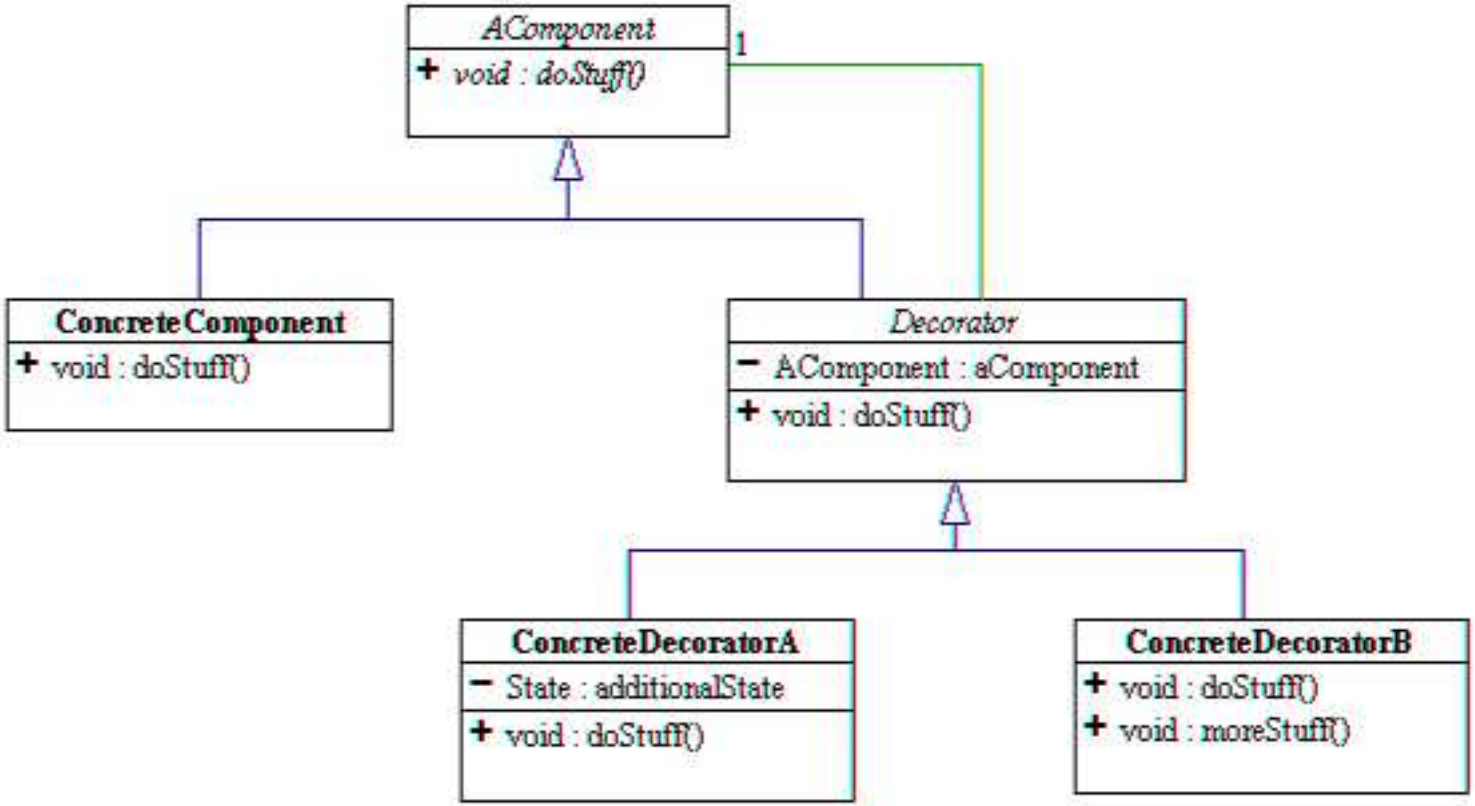
3 types de pâte

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

12 ingrédients différents

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

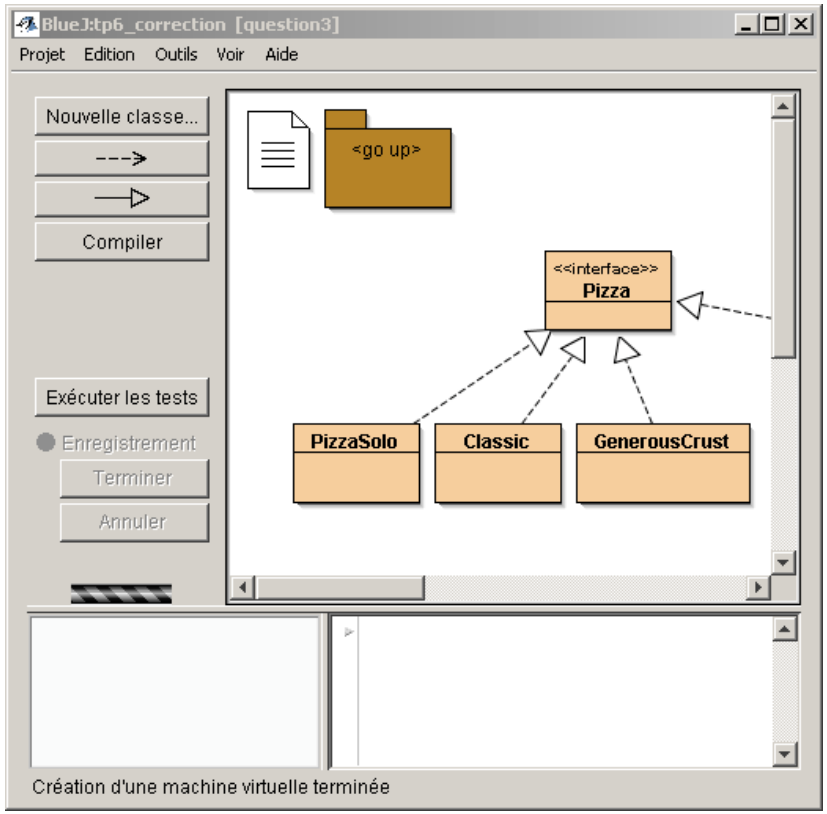
Le décorateur, rappel



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

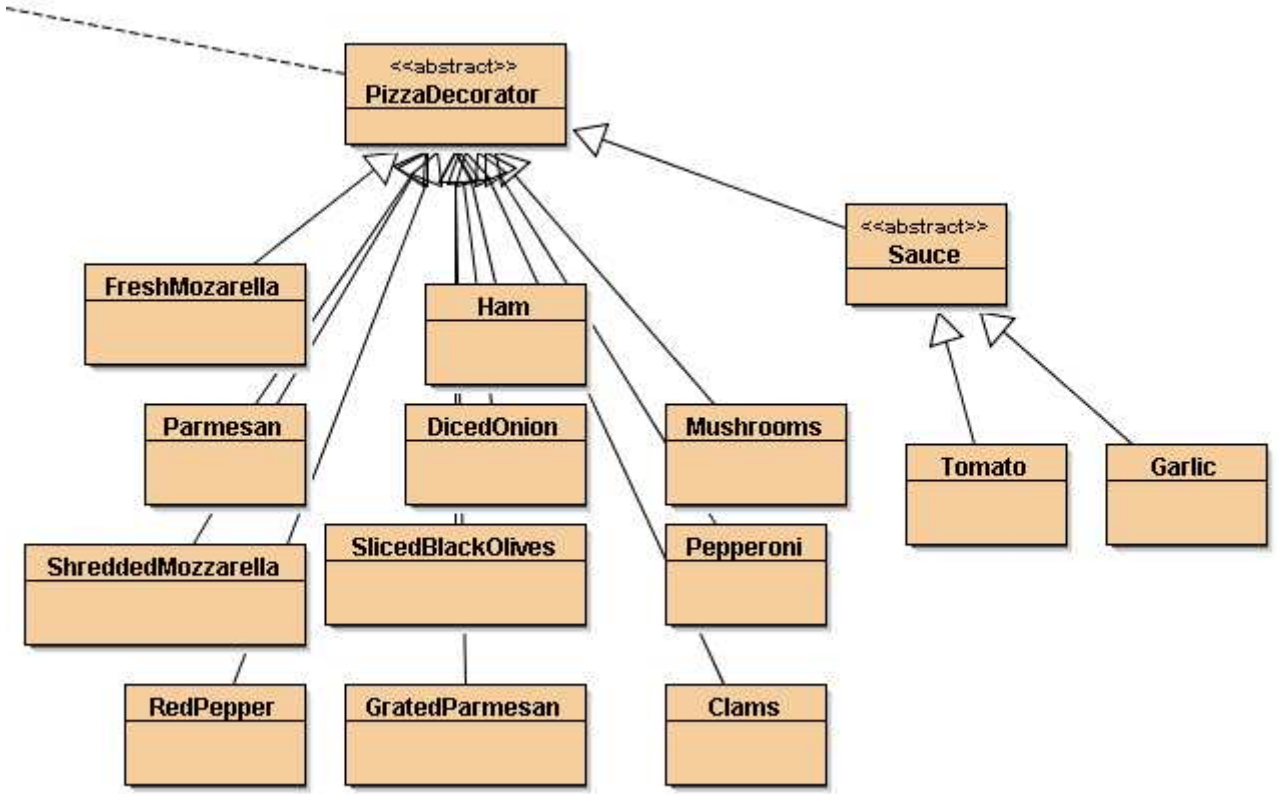
- Decorator l'ingrédient, la décoration
- ConcreteDecorator Parmesan, Mozarella, ...

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();
}

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;}
}
```

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;}
}
```

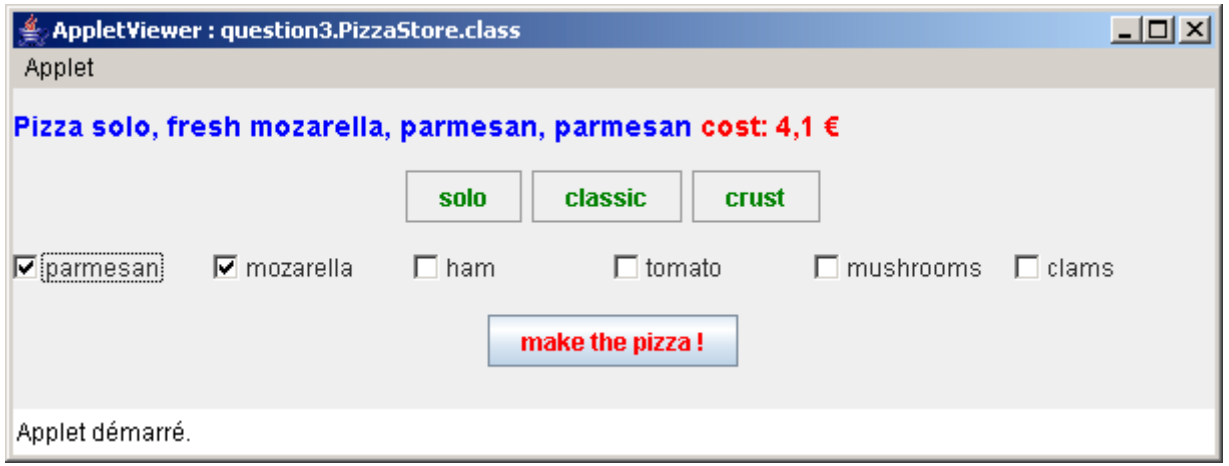
Pizza Solo + Mozarella

```
public void testCheese(){
  assertEquals(5.8,
   new Parmesan(
     new FreshMozarella(
       new PizzaSolo())).cost());
}
```

un amateur de fromage pourrait commander celle-ci

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

L ’IHM



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

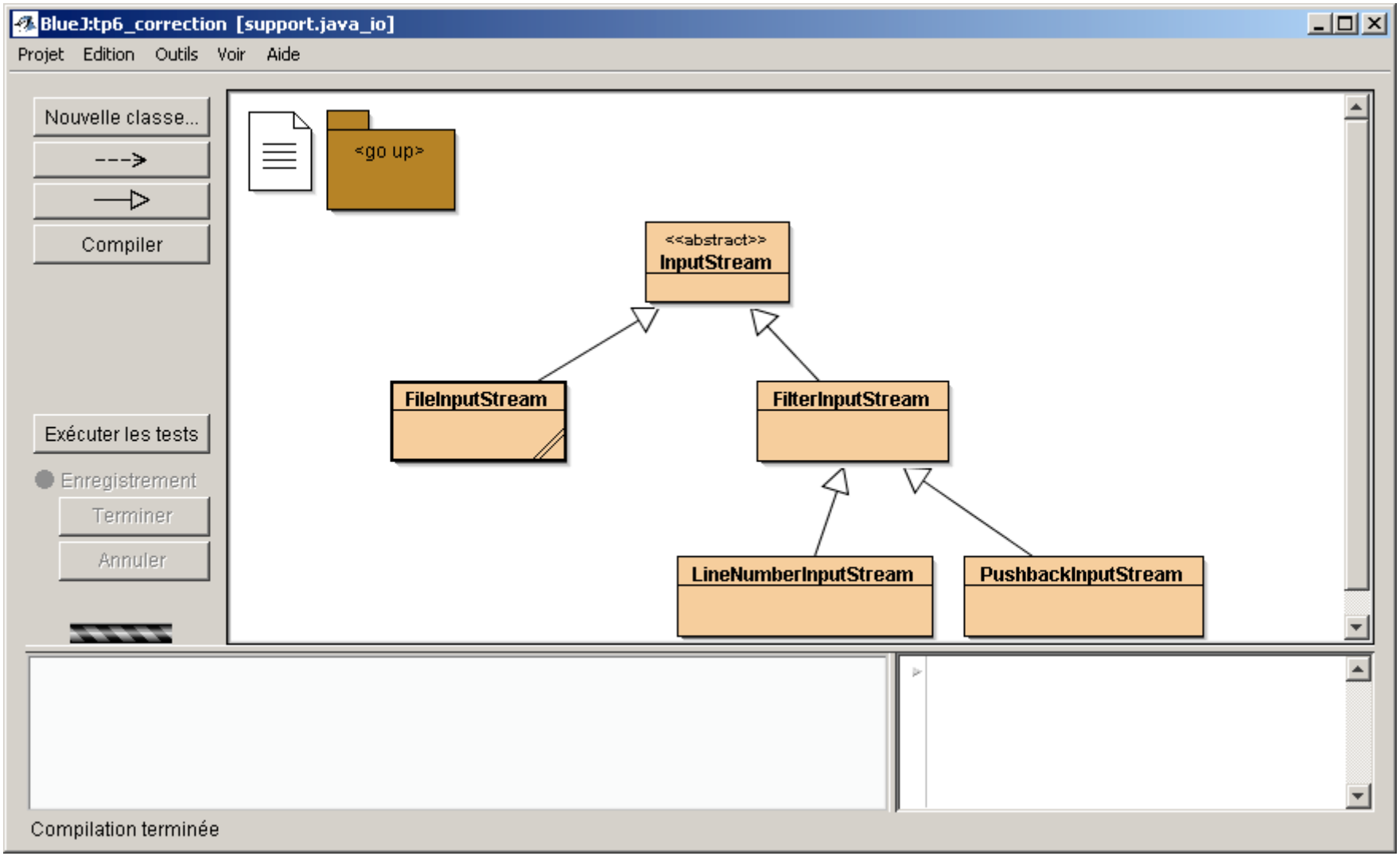
```
boutonSolo.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent ae){
      pizza = new PizzaSolo();
      validerLesDécorations();
    }
  });
```

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();
   }
});
```

Discussion

java.io



- **Le Décorateur est bien là**

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());

Conclusion