TP 3

Thèmes :

- *Pattern Décorateur*
- le chapitre 3
  extrait de Head First Design Patterns
- Décorer une classe "InputStream"
- Les pattern Composite et Interpréteur

- **Visualisez le sujet en ouvrant `index.html` du répertoire qui a été créé à l'ouverture de `tp3.jar` par BlueJ; vous aurez ainsi accès aux liens fournis pour vous aider.**
- **Soumettez chaque question à l'outil d'évaluation *jnews/junit3*, puis rendez le tp avant la date limite grâce à l'outil *jnews/depot*.**

Le décorateur de boisson

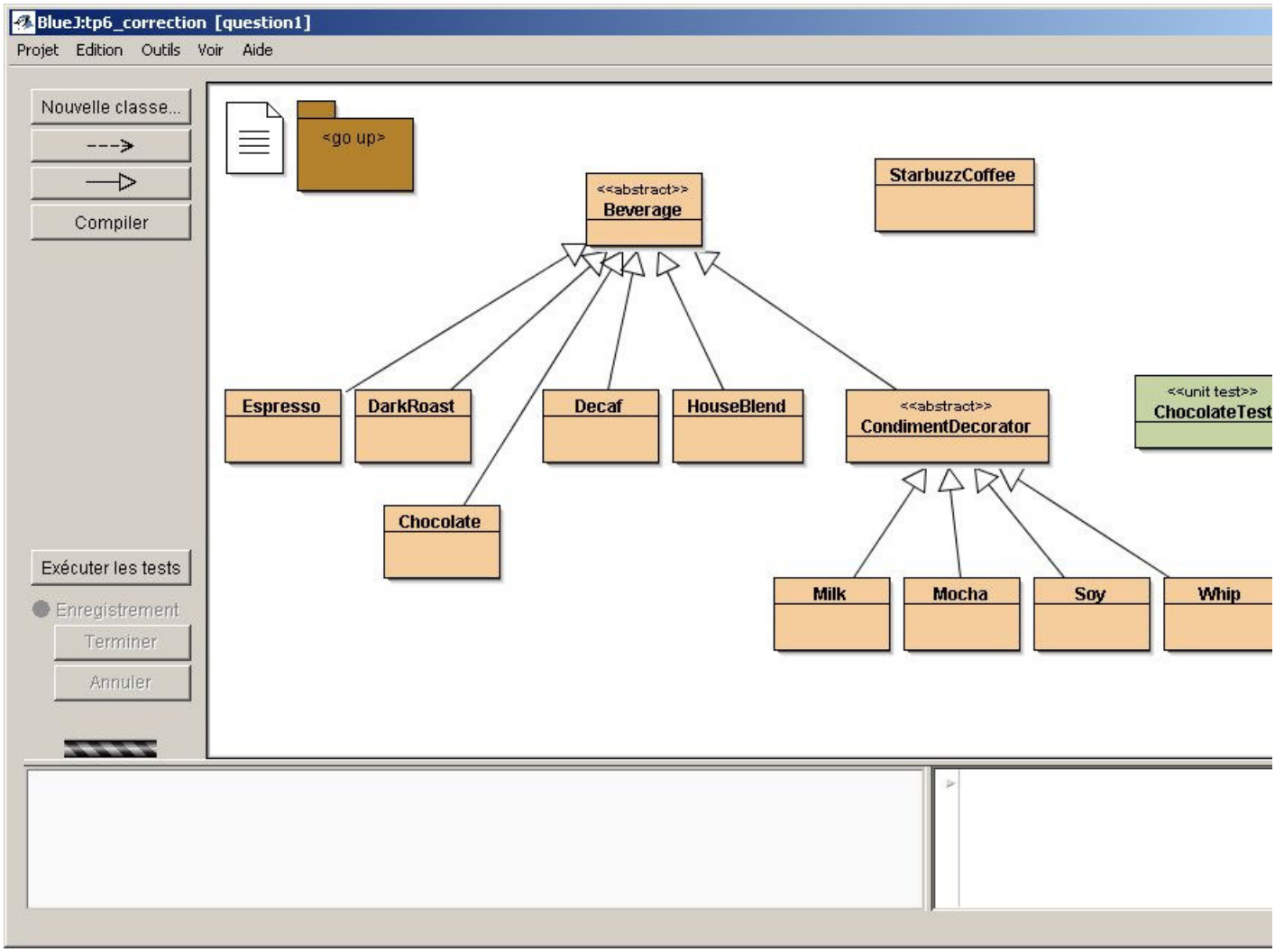
L'exemple présenté dans le chapitre 3 extrait de Head First Design Patterns, décrit une boisson et des compléments possibles. Le pattern décorateur est utilisé afin, de "décorer" la boisson choisie avec les souhaits d'un client d'une part, et de fournir au client le prix exact de la boisson qu'il a commandée d'autre part.

Exemple : un café corsé avec du lait, s'écrit:

```
Beverage darkRoastWithMilk = new Milk( new DarkRoast());
```

et l'obtention de son prix :

```
double price = darkRoastWithMilk.cost();
```



Les différentes boissons héritent de la classe abstraite **Beverage**,
La classe abstraite **CondimentDecorator**, instance du pattern Décorateur, représente les exigences possibles du client...

.1) Complétez cette architecture

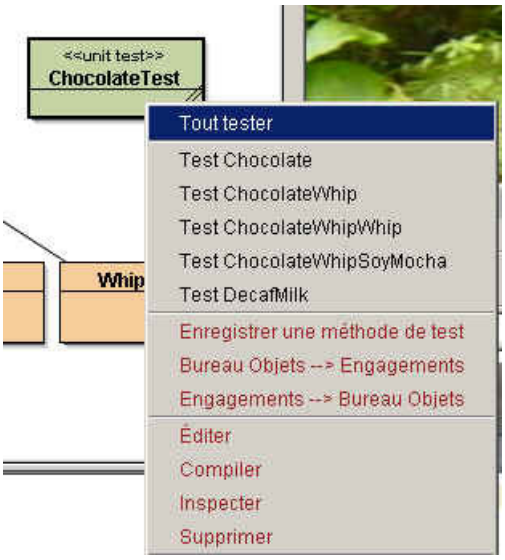
en ajoutant le chocolat (classe **Chocolate**) comme boisson; le coût de cette boisson est de **2.10**

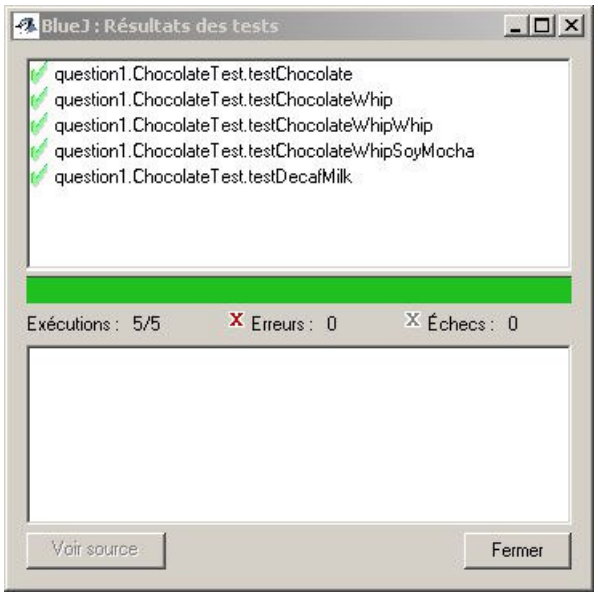
.2) Complétez la classe ChocolateTest

avec toutes les méthodes correspondant aux boissons suivantes :

- un chocolat seul
- un chocolat avec de la crème(Whip)
- un chocolat avec deux rations de crème
- un chocolat avec de la crème, du soja(soy) et du moka(mocha),... (les goûts ne se discutent pas ...)
- un café décaféiné avec du lait

Remarque: le 3ème paramètre d'`assertEquals` est la précision de comparaison entre 2 réels.





.3) Ajoutez le condiment sucre de betteraves

(**classe BeetSugar**), dont le coût est de 0.1, sa description : **"Beet Sugar"**.

.4) Ajoutez la méthode **public String toString();** -- dans quelle(s) classe(s) ? --

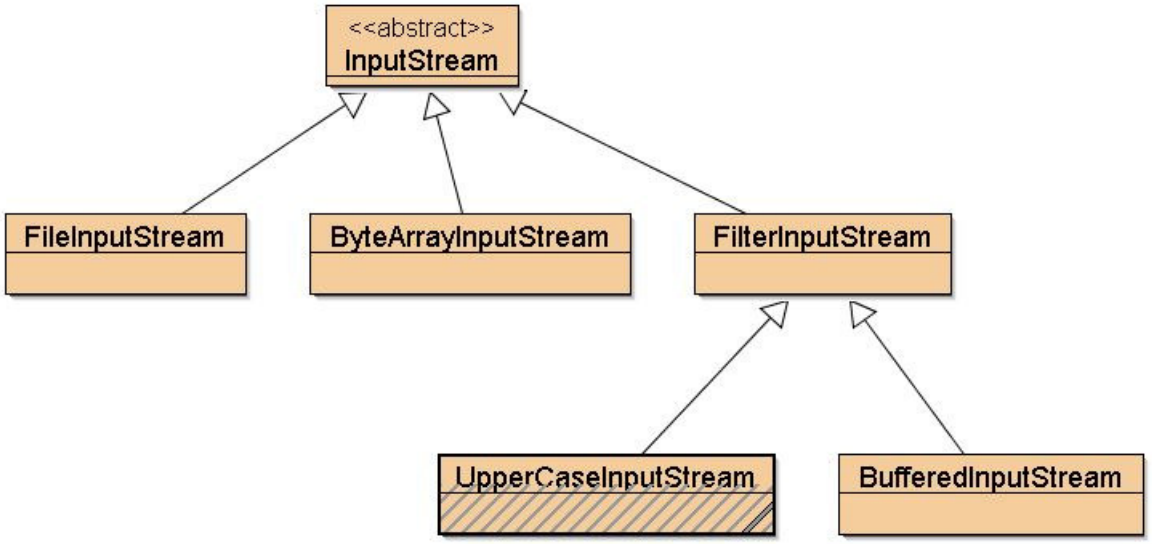
qui se contente de retourner **la description et le coût** du produit choisi par le client,
selon ce format *description $cost*

un exemple et la trace attendue

```
Beverage b = new BeetSugar( new Whip( new Mocha( new Soy( new HouseBlend
() ) ) ) );
System.out.println(b);
```

trace obtenue: **House Blend Coffee, Soy, Mocha, Whip, Beet Sugar $1.44**

Le décorateur de flux d'entrées/sorties



Ci-dessus le décorateur (incomplet) des entrées-sorties en Java, package java.io + la classe UpperCaseInputStream à développer

Proposez la classe UpperCaseInputStream (en grisé ci dessus),décorateur de FilterInputStream, qui transforme en Majuscule tous les caractères du fichier transmis en paramètre dans le constructeur.

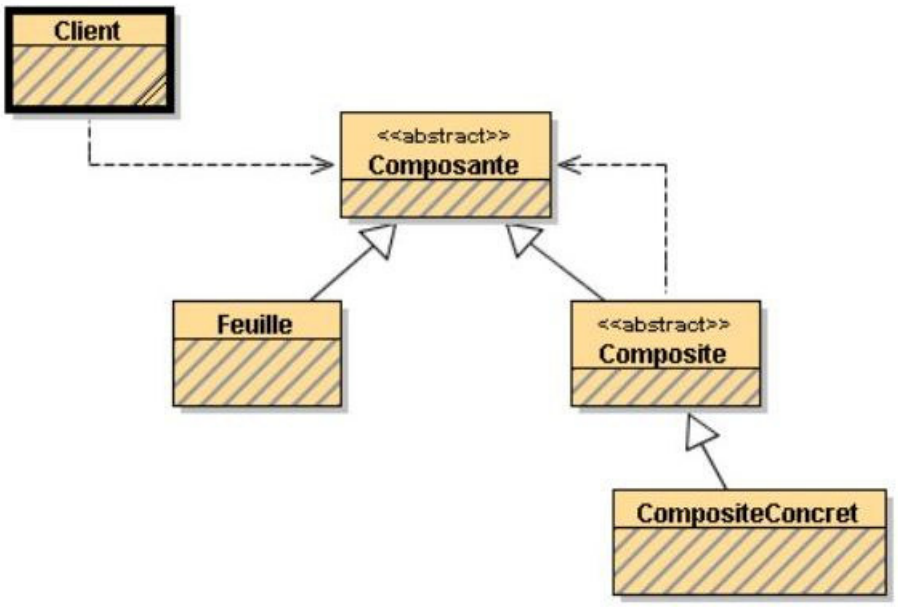
ci-dessous une instance et une utilisation possibles :

```
InputStream is = new UpperCaseInputStream(
                    new BufferedInputStream(
                      new FileInputStream( "README.TXT" ) ) );
int c = is.read();
while( c != -1){
  System.out.print((char)c);
  c = is.read();
}
is.close();
```

Les patterns Composite et Interpréteur

ou comment créer une hiérarchie d'objets (simples, complexes, y compris récursifs).

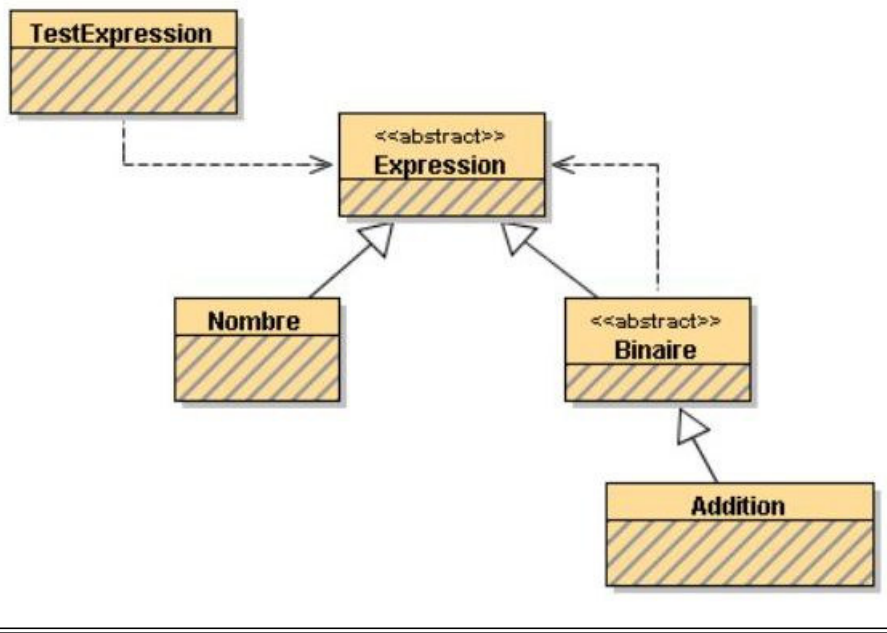
Le pattern Composite



On décrit souvent une telle structure de données par une une grammaire :

informellement	formalisé en :
une Composante est un Composite ou une Feuille Composite est composé de 0 ou plusieurs CompositeConcret Feuille est un 'symbole terminal' c-à-d un composite primitif *de plus, un Composite peut être "récursif"* *c-à-d défini en terme de Composante*	Composante ::= Composite | Feuille Composite ::= {CompositeConcret} Feuille ::= 'symbole terminal'

On applique maintenant, le pattern Composite pour représenter la structure d'une Expression Arithmétique sur les nombres entiers :



avec la grammaire :

Expression ::= Binaire | Nombre | Variable

Binaire ::= Addition | Multiplication | Soustraction | Division

Addition ::= Expression '+' Expression

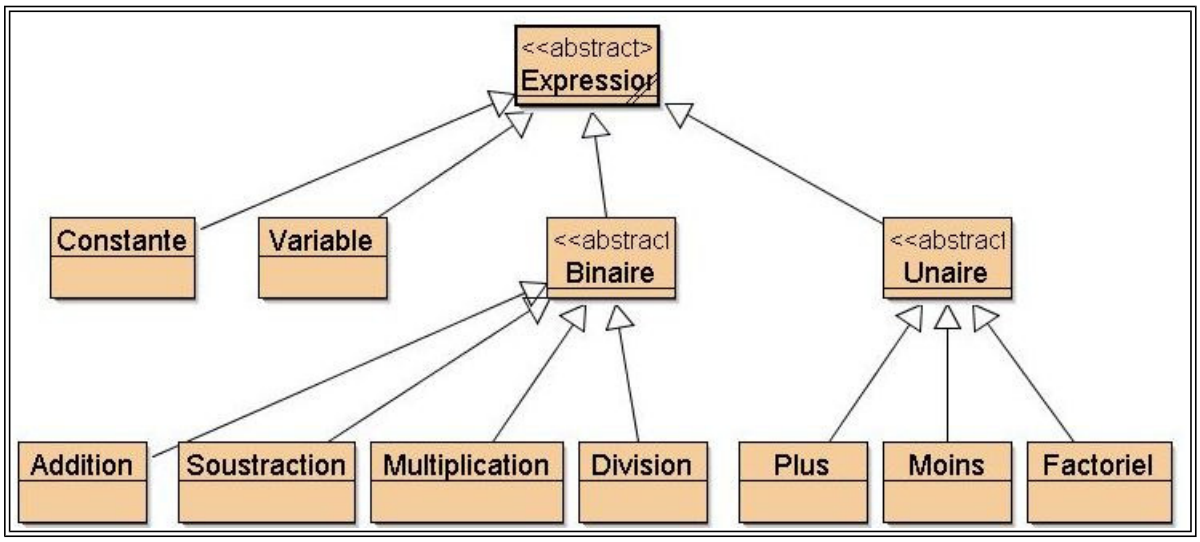
Multiplication ::= Expression '' Expression*

...

Nombre ::= 'une valeur de type int'

En ajoutant la Multiplication, la Division, la Soustraction, les opérations unaires Plus, Moins et Factoriel, ainsi que la possibilité de désigner un nombre par une Variable ou une Constante, on obtient la structure de Données :

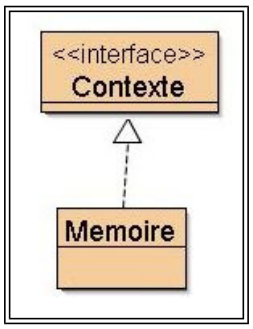
Composite des Expressions Arithmétiques entières



Le pattern Interpéteur

On reprend le pattern composite avec l'idée d'effectuer un traitement uniforme sur chacune des feuilles de la structure. Un traitement typique est une interprétation de la structure de données : par exemple ici une évaluation des expressions. Pour cela on ajoute un contexte à la structure de données : ici une mémoire où on trouvera les valeurs associées aux Variables.

contexte



Donc la spécification dans la classe abstraite Expression :

```
public int interprete(Contexte c);
```

impose l'implémentation par chaque feuille de la structure de données.

Les choix d'implantation de la classe Mémoire sont fixés, cf. le code java correspondant.

Enfin, une classe de tests unitaires montre quelques utilisations de l'interprète.

question3 **.1) Complétez toutes les sous-classes de la classe Binaire ainsi que la classe Factoriel**

Vérifiez avec la classe de tests TestInterpreteur

question3 **.2) Écrivez toutes les implémentations de la méthode toString**

afin d'obtenir une représentation infixée d'une "Expression" **conforme aux tests unitaires**, cf. la classe TestsToString

ci-dessous, un extrait de la classe de test :

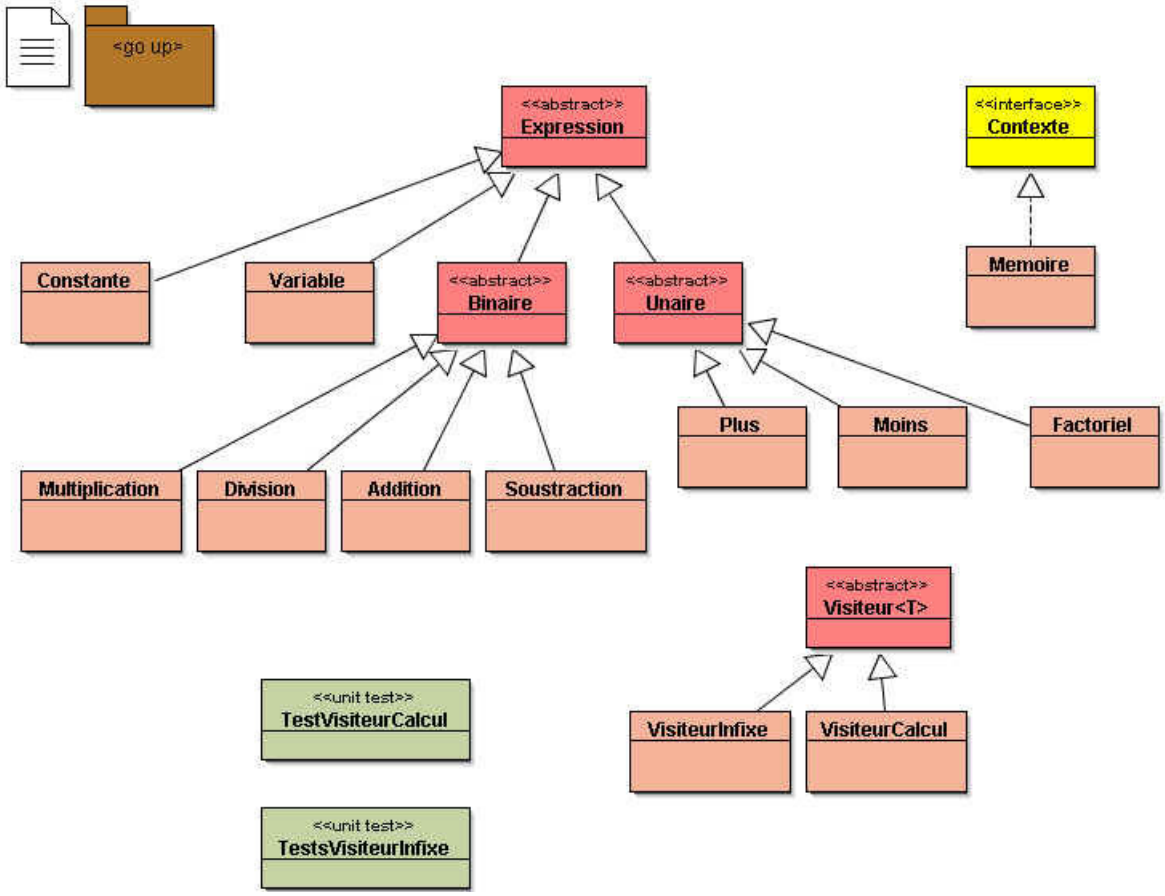
```
Expression expr = new Addition(new Constante(3), new Constante(2));
assertEquals(expr.toString(),"(3 + 2)");
expr = new Addition(expr, new Constante(2));
assertEquals(expr.toString(),"((3 + 2) + 2)");
expr = new Addition(expr, new Factoriel(x1));
assertEquals(expr.toString(),"(((3 + 2) + 2) + x1!)");
expr = new Soustraction(expr, new Factoriel(x1));
assertEquals(expr.toString(),"((((3 + 2) + 2) + x1!) - x1!)");
```

---

question4

Le pattern Visiteur

On reprend le pattern composite de la question précédente dans lequel les méthodes ***interprete*** et ***toString*** ont été supprimées. Proposer à la place de ces deux méthodes les visiteurs appropriés : VisiteurCalcul et VisiteurInfixe



ci-dessous, un extrait de la classe de test TestsVisiteurInfixe:

```
Visiteur<String> vi = new VisiteurInfixe(m);
Expression expr = new Addition(new Constante(3), new Constante(2));
assertEquals(expr.accepter(vi),"(3 + 2)");
expr = new Addition(expr, new Constante(2));
assertEquals(expr.accepter(vi),"((3 + 2) + 2)");
expr = new Addition(expr, new Factoriel(x1));
assertEquals(m.toString(),"{x=3, x1=5}");
assertEquals(expr.accepter(vi),"(((3 + 2) + 2) + x1!)");
expr = new Soustraction(expr, new Factoriel(x1));
assertEquals(expr.accepter(vi),"((((3 + 2) + 2) + x1!) - x1!)");
```