Conservatoire national des arts et métiers

NFP121 : Programmation avancée
TP n° 6 - WhileL et les Visiteurs

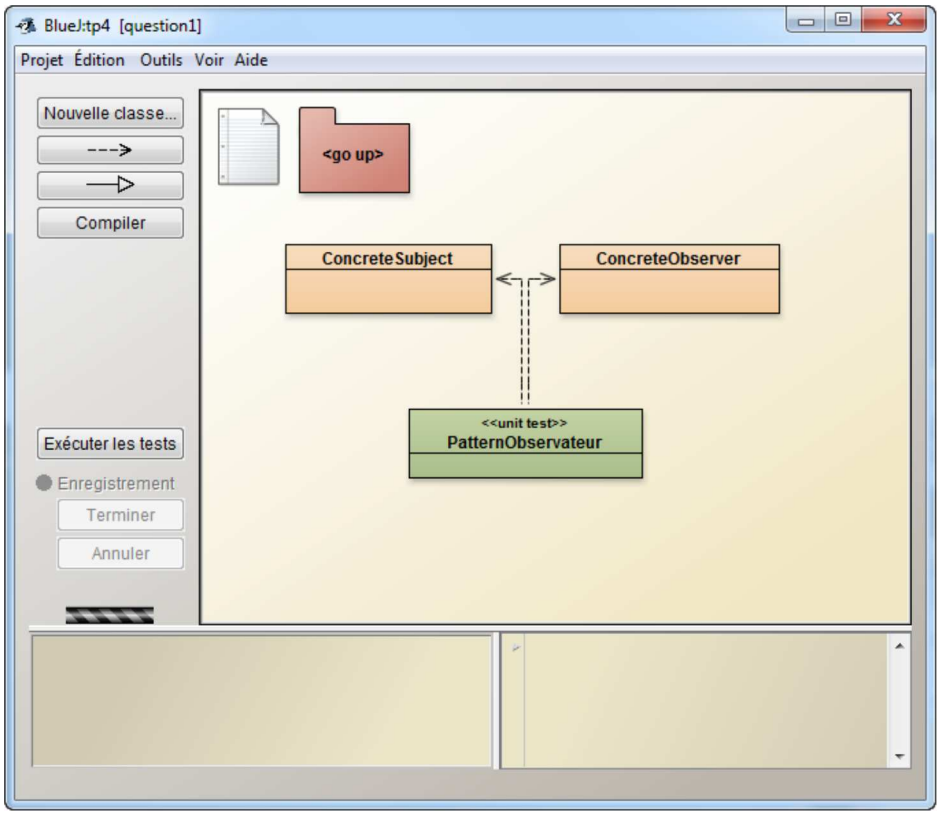
Thèmes du TP :

- Le pattern Composite
- Le pattern Visiteur
- le langage WhileL de Hennessy *(page 47, chapitre 4.3 an Imperative language)*

Le pattern composite

Aucun programme n'est demandé dans ce préliminaire, mais une lecture attentive facilitera l'exécution du reste du TP

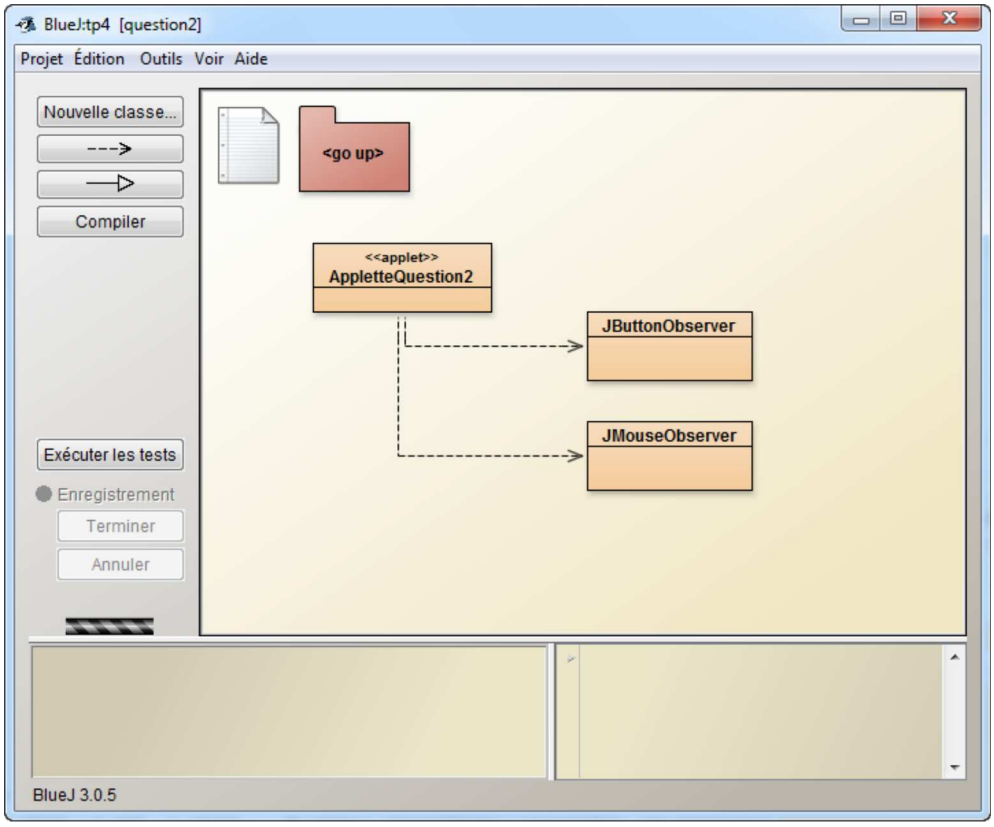
Regrouper dans une hiérarchie des objets (simples, complexes y compris récursifs).



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

Informellement	Formilisé en :
Une Composante est un Composite ou une Feuille	Composante ::= Composite | Feuille
Composite est composé de 0 ou plusieurs CompositeConcret	Composite ::= {CompositeConcret}
Feuille est une 'symbole terminal' un composite primitif	Feuille ::= 'symbole terminal'
de plus un Composite *peut être "récursif i.e. défini en terme de* Composante	...

On applique ce pattern pour représenter la structure d'une Expression Arithmétique sur les nombres entiers :

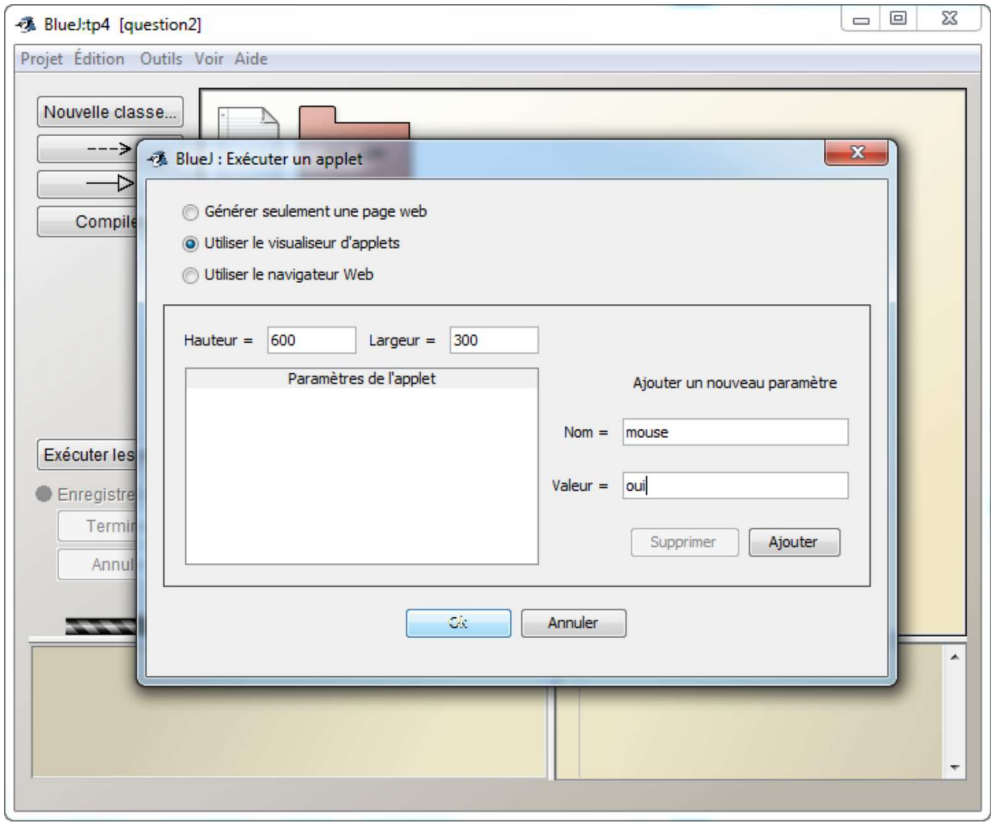


Avec la grammaire :

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 et les opérations unaires Plus, Moins et Factorielle et la possibilité de désigner un nombre par une Variable, on obtient la structure de Données :

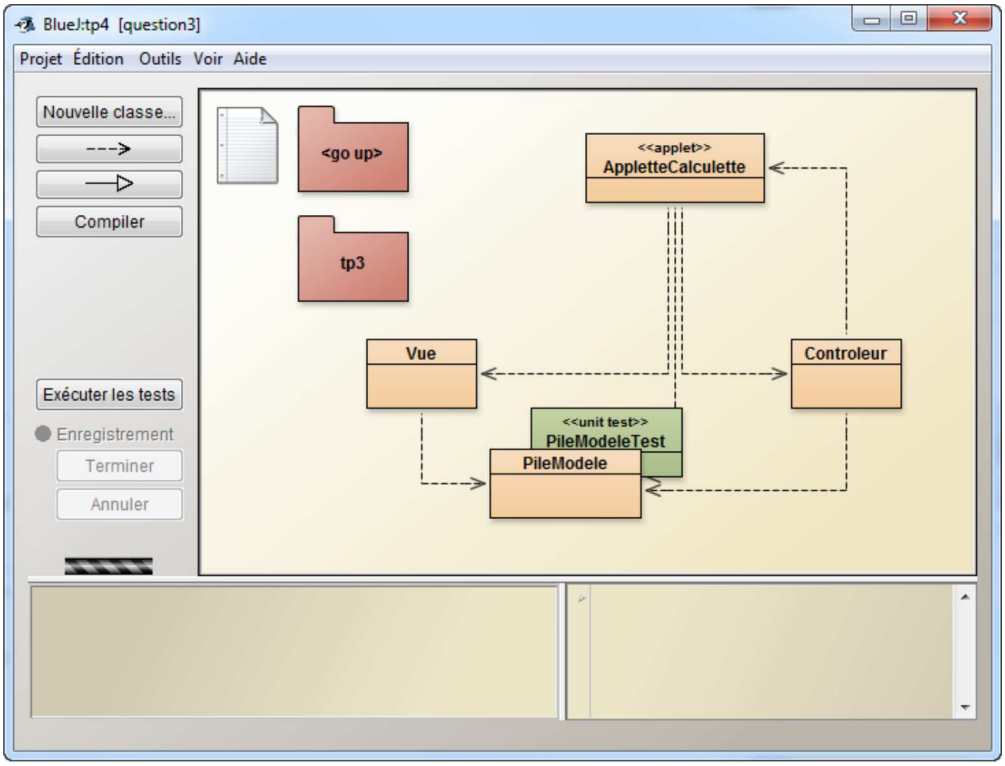
Composite des Expressions Arithmétiques entières



Le pattern interpreter/interpréteur

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 (dans un monde connu) 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** dans laquelles nous trouverons les valeurs associées aux Variables.

Contexte



Donc l'entête de la méthode d'interprétation dans la classe abstraite Expression est la suivante :

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

Ceci 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 montrent quelques utilisations de l'interpréte.

Remarques :

- l'évaluation n'est pas la seule interprétation possible des expressions.
- l'affichage des expressions (infixé, postfixé, préfixé) peut être vu comme une interprétation.
- La simplification (évaluation des sous expressions purement numériques) en est une aussi.
- etc...
- Donc, pour implémenter une nouvelle interprétation il faut "ouvrir" pratiquement toutes les classes de la structure de données avec tous les dangers que cela comporte.
- Alors dans la question1 nous allons utiliser le pattern visiteur pour éviter cette opération.

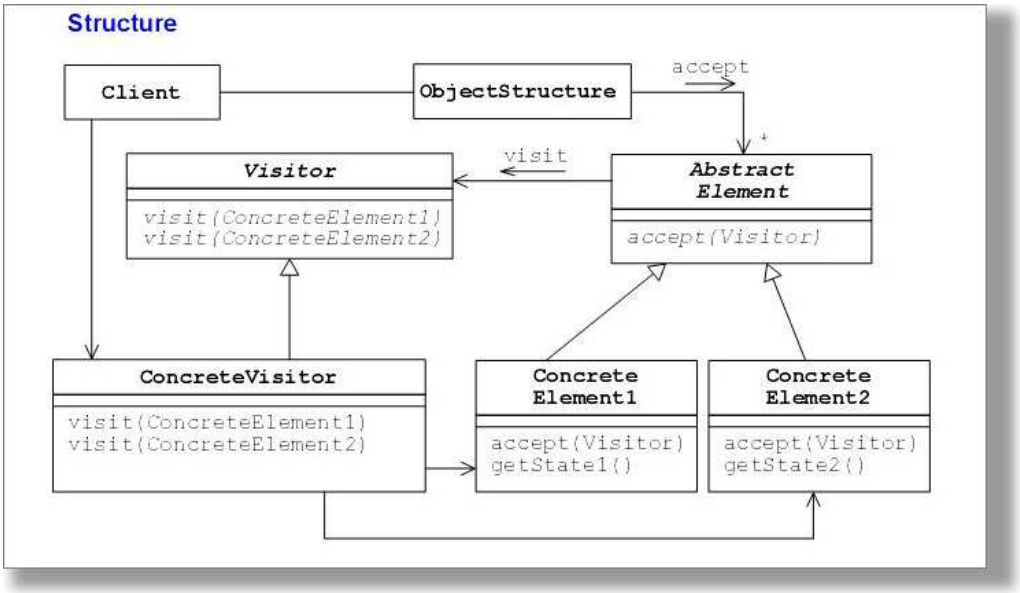
Le pattern visitor/visiteur

Idée : Pour éviter l'ouverture et la modification de toutes les classes de la structure pour l'implantation d'une nouvelle méthode, on décide d'implanter toutes les méthodes de manipulation de la structure dans des classes externes dites "visiteur". Alors dans chaque classe de la structure on ne trouve plus qu'une seule méthode qui accepte un visiteur.

Ainsi la classe Expression devrait s'écrire :

```
package question1;

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



Reprendre les Expressions du paquetage "preliminaire" et implanter par le pattern visiteur les différentes interprétations proposées : Evaluation (VisiteurEvaluation), Affichage infixe (VisiteurInfixe), Affichage Postfixe (VisiteurPostfixe).

En vous inspirant de la classe VisiteurInfixe qui est complète, complétez les classes VisiteurEvaluation et VisiteurPostfixe et proposez les tests des classes de tests appropriés (cf : classe TestsAFaire).

Remarque :

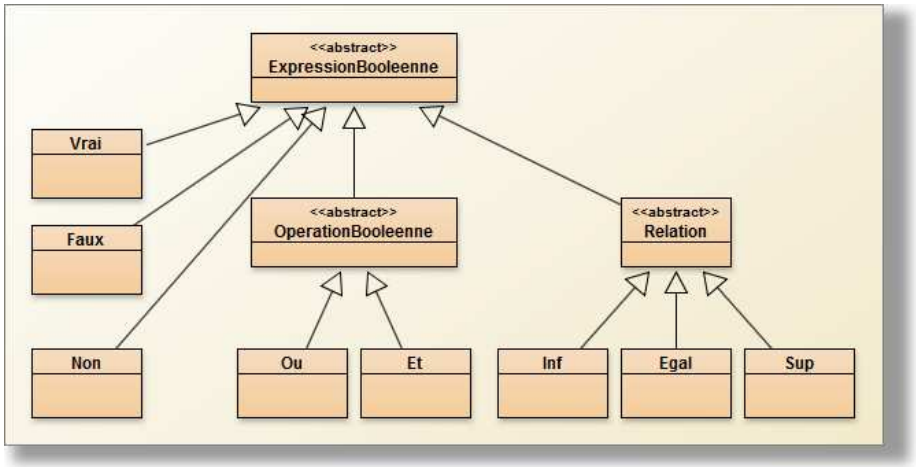
Toutes les classes feuilles contiennent la même méthode :

```
public <T> T accepter(VisiteurExpression<T> v){ return v.visite(this); }
```

Les Expressions Booléennes

Reproduire pour les Expressions Booléennes ce qui a été obtenu à la question 1 pour les Expressions Arithmétiques.

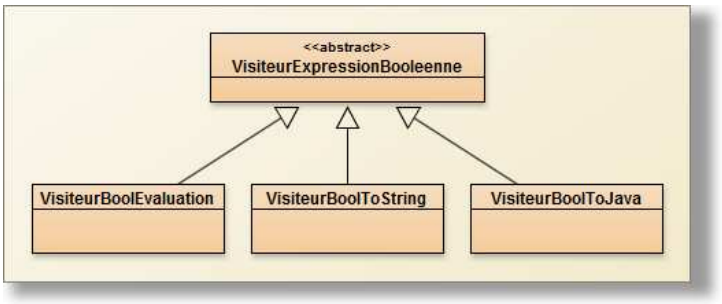
Diagramme de Classes à respecter :



Remarques :

- Cette structure de données est complète vous n'avez rien à y ajouter.
- Pas de variables booléennes.
- 2 constantes seulement Vrai et Faux.
- Un seul opérateur unaire : Non.
- Deux sortes d'opérateurs binaires :
    - Les opérations booléennes
    - Les "Relations" sont des opérateurs entre Expressions Arithmétiques à résultat booléen

Les visiteurs à implanter sont les suivants :



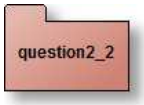
Remarques :

- Pas de visiteur par défaut.
- VisiteurBoolString correspond au visiteur infixe des Expressions Arithmétiques
- Nouveau visiteur : VisiteurBoolTojava. Il s'agit d'obtenir une expression booléenne syntaxiquement correcte pour java

---

Le visiteur "VisiteurBoolTojava"

Complétez le visiteur "VisiteurBoolTojava", "vérifiez" avec la classe de tests fournie.

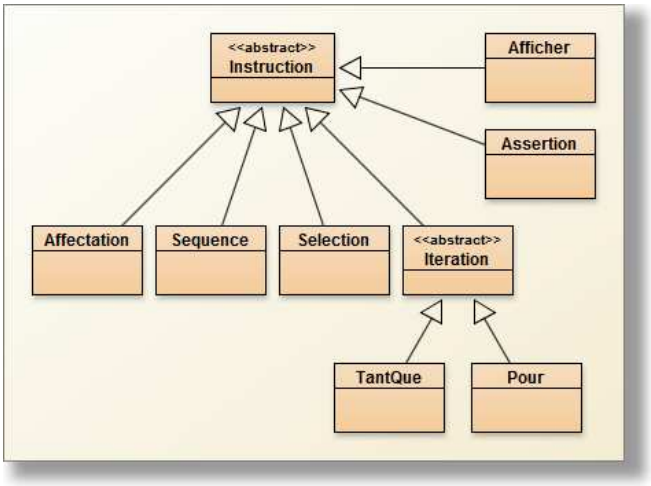


Le visiteur "VisiteurBoolEvaluation"

Complétez le visiteur "VisiteurBoolEvaluation" et proposez une classe de tests de ce visiteur.

WhileL : un (très) petit langage impératif

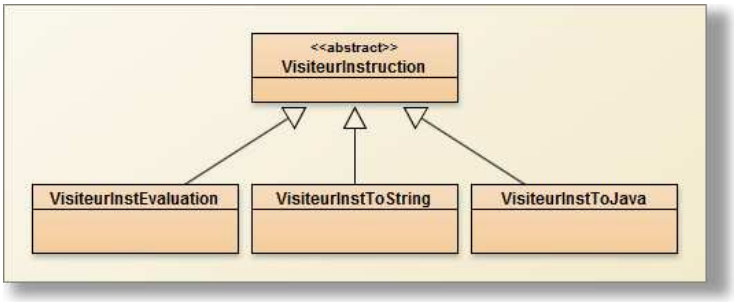
Les instructions de WhileL sont représentées dans le composite :



Remarques :

- L'affectation (= en java) est la seule instruction à modifier le contexte.
- La Séquence est le ';' de java.
- La Sélection est le classique if-then-else (else optionnel)
- Iteration <<abstract>> car il y a plus d'un type de boucle
- Assertion : même idée que assert de java 1.4
- Afficher : même idée que System.out.println() de java

Les visiteurs demandés sont :



Ci-dessous, une spécification de l'interprétation/exécution d'une instruction :

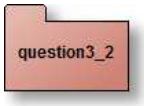
Spécification informelle : Rappel : l'exécution d'une instruction modifie l'état d'une Mémoire M, une variable (par exemple X) est une adresse de la Mémoire M	Spécification formelle : règle d'inférence cf. livre de Hennessy Description d'une règle d'inférence : au dessus du trait les hypothèses à assurer pour affirmer la conclusion en dessous du trait. Exemple la première règle ci dessous peut se lire : au dessus du trait : "si dans le contexte de la mèmoire M l'expression Expr est évaluée dans l'entier N" alors conclusion en dessous du trait : "dans le même contexte M l'affection X = Exp modifie la mémoire tel que maintenant M[X]=N
L'affectation : **X=Exp**. si avant exécution l'état de la Mémoire est M 1. Evaluation de Exp dans un entier N cf. question 1 2. modification de la mémoire en X après exécution l'état de la Mémoire est M1 i.e. maintenant M[X]=N	<M>,Exp -**visite**-> N ——————— <M>, X = Exp -**visite**-> **<**M[X]=N>
La sélection, **si(Bexp)alorsI1 sinon I2** : si avant exécution l'état de la Mémoire est M 1. Evaluation de Bexp dans une valeur Booléenne 2. Evaluation de I1 ou I2 selon la valeur trouvée en 1 après exécution l'état de la Mémoire est M1 ou M2 selon 2. La sélection, **si(Bexp)alorsI1**: si avant exécution l'état de la Mémoire est M 1. Evaluation de Bexp 2. Evaluation de I1 ou "ne rien faire" selon la valeur trouvée en 1	<M>,Bexp -**visite**-> vrai <M>,I1 -**visite**-> M1 ——————— <M>,si(Bexp)alors I1 sinon I2-**visite**-> <M1> <M>,Bexp -**visite**-> faux <M>,I2 -**visite**-> M2 ——————— <**M**>,si(Bexp)alorsI1 sinon I2-**visite**-> <M2>

après exécution l'état de la Mémoire est M1 ou M (inchangée) selon 2.	
La séquence : **I1';'I2** : si avant exécution l'état de la Mémoire est M 1. Evaluation de I1 qui transforme la mémoire M en la mémoire M1 2. PUIS Evaluation de I2 à partir de M1 donc qui qui transforme la mémoire M en la mémoire M2 après exécution l'état de la Mémoire est M2.	<M>,I1 -**visite**-> <M1> <M1>,I2 -**visite**-> <M2> _____ <M>,I1';'I2-**visite**-> <M2>
la boucle **tantque (Bexp) faire I1**: si avant exécution l'état de la Mémoire est M 1. Evaluation de Bexp 2. si Bexp est évaluée à vrai alors Evaluation de **I1';' tantque(Bexp)faire I1** 3. si Bexp est évaluée à faux "ne rien faire"	<M>,Bexp -**visite**-> faux _____ <M>,tantque(Bexp) faire I1-**visite**-> <M> <M>,Bexp -**visite**-> vrai <M>,I1';' tantque(Bexp)faire I1-**visite**-> <M1> _____ <M>,tantque(Bexp)faire I1-**visite**-> <M1>

Complétez les visiteurs "**VisiteurInstEvaluation**" et "**VisiteurInstTojava**". Vérifiez que le code obtenu avec "VisiteurInstTojava" est bien syntaxiquement correct pour java. (cf. classe Classejava du paquetage question3) .

Remarques :

- Servez-vous des règles d'évaluation ci-dessus pour implémenter le VisiteurInstEvalution, par exemple n'utilisez pas l'instruction while de java afin d'implémenter le TantQue de WhileL...
- Le visiteur "VisiteurToString" est complet



La boucle "Pour"

On a ajouté la boucle "Pour" au paquetage question3 sur le modèle de la boucle "for" java (cf. classe "Pour"). Modifiez les visiteurs en conséquence.

Les classes de tests

Vérifiez le bon fonctionnement des classes de Tests.