# tp4 : Les Visiteurs

## Thème :

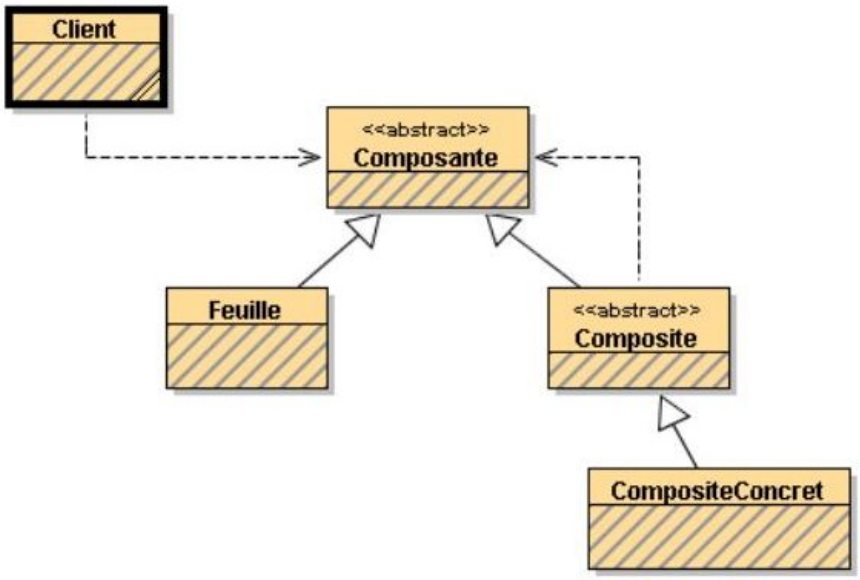
- Le pattern Composite
- Le pattern Visiteur
- le langage WhileL de Hennessy
  *page 47, chapitre 4.3 an Imperative language*
- Le pattern Décorateur

---

preliminaire Le pattern composite : *(aucun programme n'est demandé dans ce préliminaire, mais une lecture attentive facilitera l'exécution du reste du TP)*

---

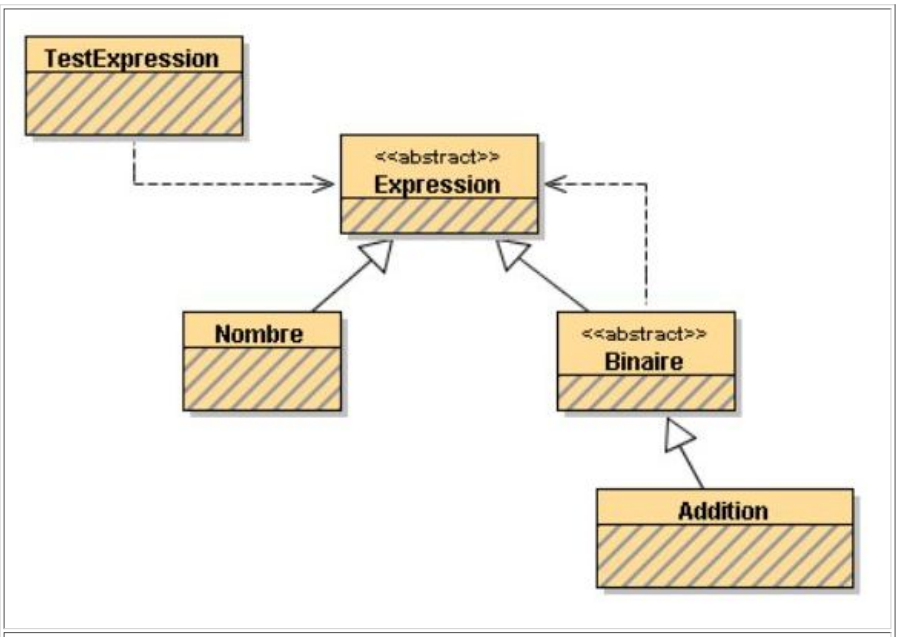
*Regrouper dans une hiérarchie des objets (simples, complexes y compris récursifs).*




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

| informellement | formalisé en : |
|---|---|
| une Composante est un Composite ou une Feuille | Composante ::= Composite \| Feuille |
| Composite est composé de 0 ou plusieurs CompositeConcret | Composite ::= {CompositeConcret} |
| Feuille est un ‘symbole terminal’ (ou 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 | ...

Binaire ::= Addition | ...

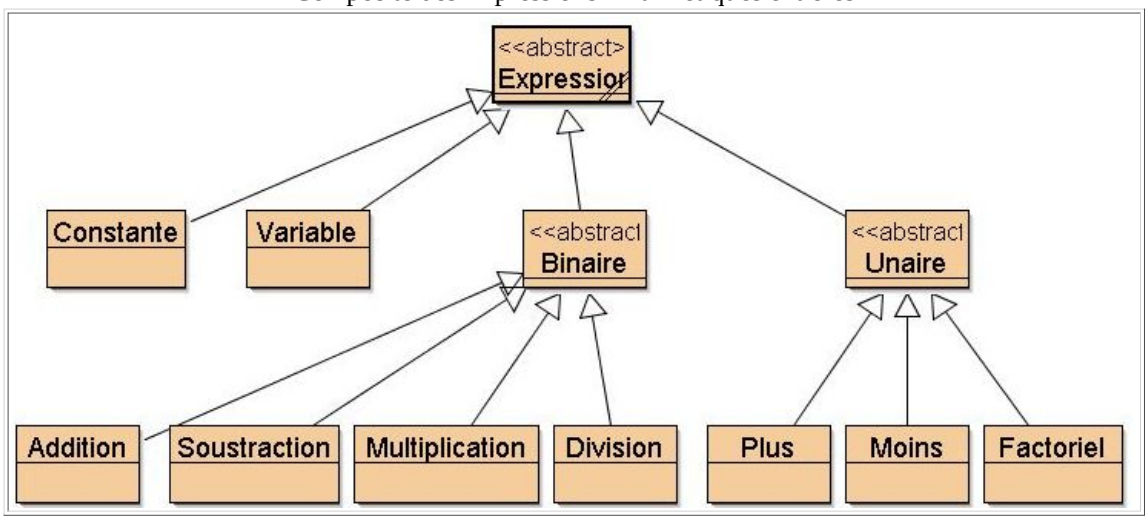
Addition ::= Expression '+' Expression

...

Nombre ::= 'une valeur de type int'

En ajoutant la Multiplication, la Division, et la Soustraction (pourquoi pas aussi les opérations unaires Plus, Moins et Factorielle et la possibilité de désigner un nombre par une Variable ?), on obtiendrait la structure de Données :

Composite des Expressions Arithmétiques entières



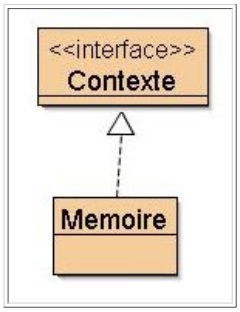

---

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 laquelle 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 de la structure de données par chaque feuille.

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'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" partiquement 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 :

---

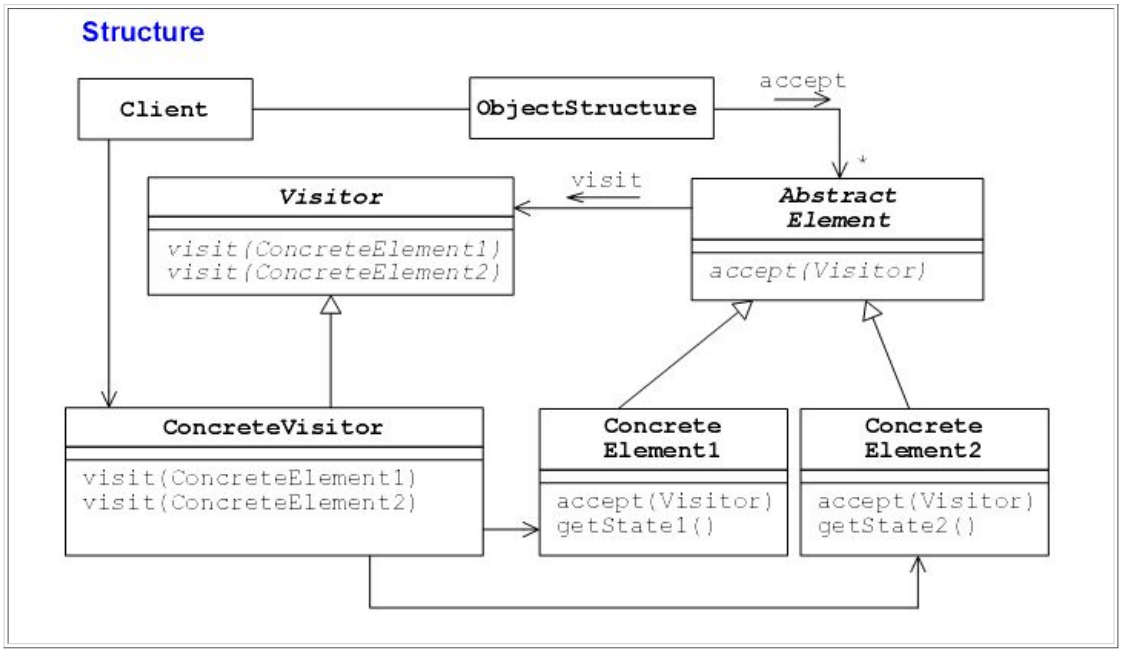
Idée : Pour éviter l'ouverture et la modification de toutes les classes de la structure pour l'implantation d'une nouvelle méthode, on décide d'implanter toutes les méthodes de manipulation de la structure dans des classes externes dites "visiteur". Alors dans chaque classe de la structure on ne trouve 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 );
}
```

Pattern visitor




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

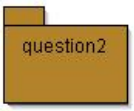
**Question 1)** En vous inspirant de la classe VisiteurInfixe qui est complète, complétez les classes **VisiteurEvaluation** et **VisiteurPostfixe** et proposez les tests des classes de tests appropriés (cf : classe TestsAFaire).

Remarque :

- toutes les classes feuilles contiennent la même méthode :
  public <T> T accepter(VisiteurExpression<T> v){ return v.visite(this); }

Soumettez cette question à JNews.

---

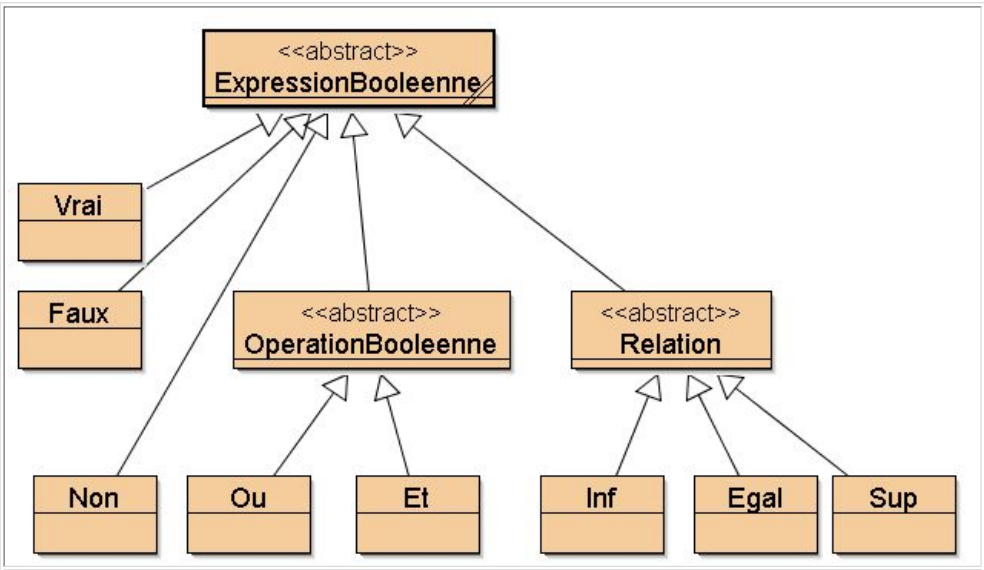


Les Expressions Booléennes

---

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

Diagramme de Classes à respecter :

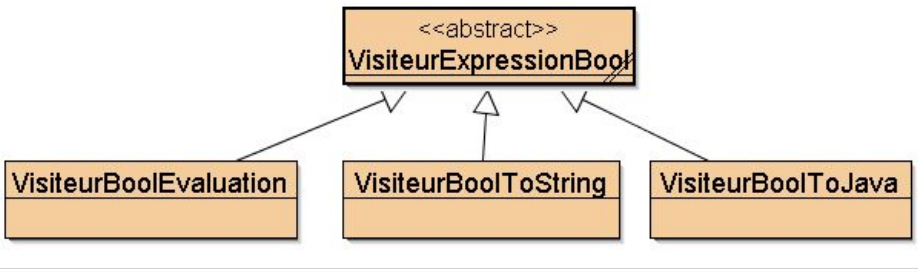



remarques :

- 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

question 2.1) : Complétez le Visiteur "VisiteurBoolToJava", "vérifiez" avec la classe de tests fournie.

question 2.2) : Complétez le "VisiteurBoolEvaluation" et proposez une classe de tests de ce visiteur.

Soumettez cette question à JNews.

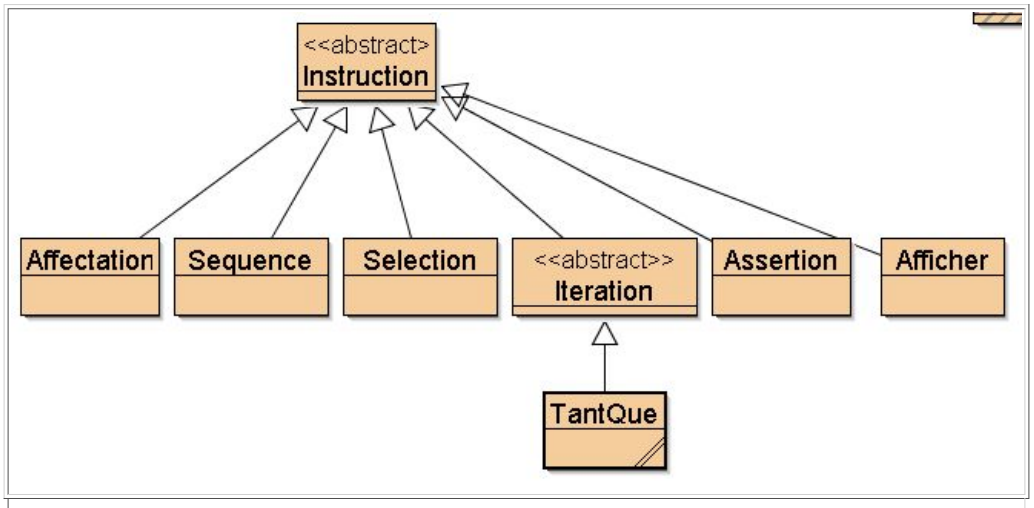
---

WhileL : un (très) petit langage impératif.

Aucun développement n'est demandé pour cette question. Il s'agit d'utiliser le code fourni pour mettre en oeuvre ce qui a été vu à la question précédente, et notamment la classe de test.

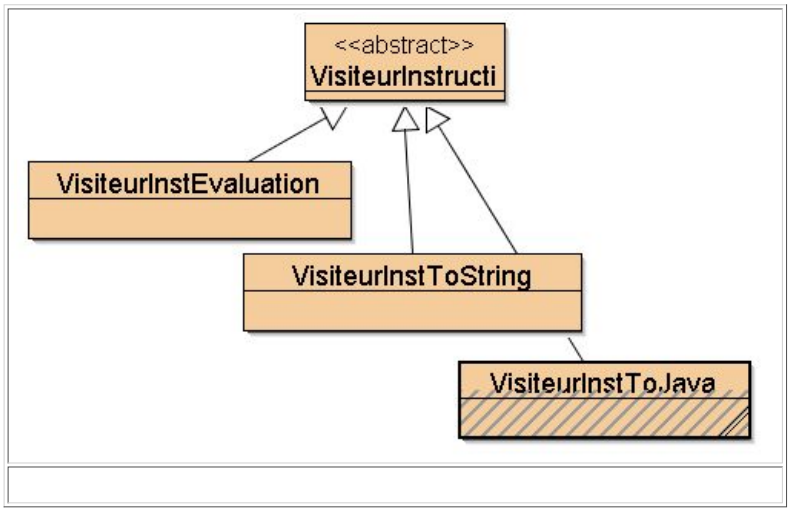
---

Les instructions de WhileL sont représentées dans le composite :




remarques :
Les visiteurs demandés sont :

- l'Affectation (= en java) est la seule instruction qui modifie le contexte.
- la Séquence est le ';' de Java.
- la Sélection est le classique if-else (else optionnel)
- Iteration est <<abstract>> car il y a plus d'un type de boucle
- Assertion : même idée que assert de Java1.4
- Afficher : même idée que System.out.println() de java




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

| | |
|---|---|
| spécification informelle :<br>Rappel : l'exécution d'une instruction modifie l'état d'une Mémoire M,<br>une variable (par exemple X) est une adresse de la Mémoire M | spécification formelle : règle d'inférence cf.livre de Hennessy<br><br>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<br>peut se lire : au dessus du trait : "si dans le contexte de la mémoire M, l'expression Expr<br>est évaluée dans l'entier N" alors conclusion en dessous du trait :<br>"dans le même contexte M l'affectation X = Exp modifie la mémoire tel que maintenant<br>M[X]=N |
| L'affectation : **X=Exp**. si avant exécution l'état de la Mémoire est M<br><br>1. Evaluation de Exp dans un entier N cf. question 1<br>2. modification de la mémoire en X<br><br>après exécution l'état de la Mémoire est M1 i.e. maintenant M[X]=N | <M>,Exp -**visite**-> N<br>———————————<br><M>, X = Exp -**visite**-> <M[X]=N> |
| La sélection, **si (Bexp) alors I1 sinon I2** : si avant exécution, l'état de la Mémoire est M<br><br>1. Evaluation de Bexp dans une valeur Booléenne<br>2. Evaluation de I1 ou I2 selon la valeur trouvée en 1<br><br>après exécution l'état de la Mémoire est M1 ou M2 selon 2.<br><br>La sélection, **si (Bexp) alors I1**: si avant exécution, l'état de la Mémoire est M<br><br>1. Evaluation de Bexp<br>2. Evaluation de I1 ou "ne rien faire" selon la valeur trouvée en 1<br><br>après exécution l'état de la Mémoire est M1 ou M (inchangée) selon 2. | <M>,Bexp -**visite**-> vrai<br><M>,I1 -**visite**-> M1<br>———————————<br><M>,si (Bexp) alors I1 sinon I2-**visite**-> <M1><br><br><M>,Bexp -**visite**-> faux<br><M>,I2 -**visite**-> M2<br>———————————<br><**M**>,si (Bexp) alorsI1 sinon I2-**visite**-> <M2> |
| La séquence : **I1';'I2** : si avant exécution l'état de la Mémoire est M<br><br>1. Evaluation de I1 qui transforme la mémoire M en la mémoire M1<br>2. PUIS Evaluation de I2 à partir de M1<br>donc qui transforme la mémoire M1 en la mémoire M2<br><br>après exécution, l'état de la Mémoire est M2. | <M>,I1 -**visite**-> <M1><br><M1>,I2 -**visite**-> <M2><br>———————————<br><M>,I1';'I2-**visite**-> <M2> |
| la boucle **tantque (Bexp) faire I1**: si avant exécution, l'état de la Mémoire est M<br><br>1. Evaluation de Bexp<br>2. si Bexp est évaluée à vrai alors Evaluation de **I1';' tantque (Bexp) faire I1**<br>3. si Bexp est évaluée à faux "ne rien faire" | <M>,Bexp -**visite**-> faux<br>———————————<br><M>,tantque (Bexp) faire I1-**visite**-> <M><br><br><M>,Bexp -**visite**-> vrai<br><M>,I1';' tantque (Bexp) faire I1-**visite**-> <M1><br>———————————<br><M>,tantque (Bexp)**visite**-> <M1> |

Question 3.1) Regardez les visiteurs "**VisiteurInstEvaluation**" et "**VisiteurInstToJava**". Vérifiez que le code obtenu avec "VisiteurInstToJava" est bien syntaxiquement correct pour Java. (cf. ClasseJava du paquetage question3) .

Remarques :
- Les règles d'évaluation ci-dessus servent pour implémenter le VisiteurInstEvaluation : par exemple, on n'utilise pas l'instruction while de java pour implémenter le TantQue de WhileL...
- Le visiteur "VisiteurToString" est complet

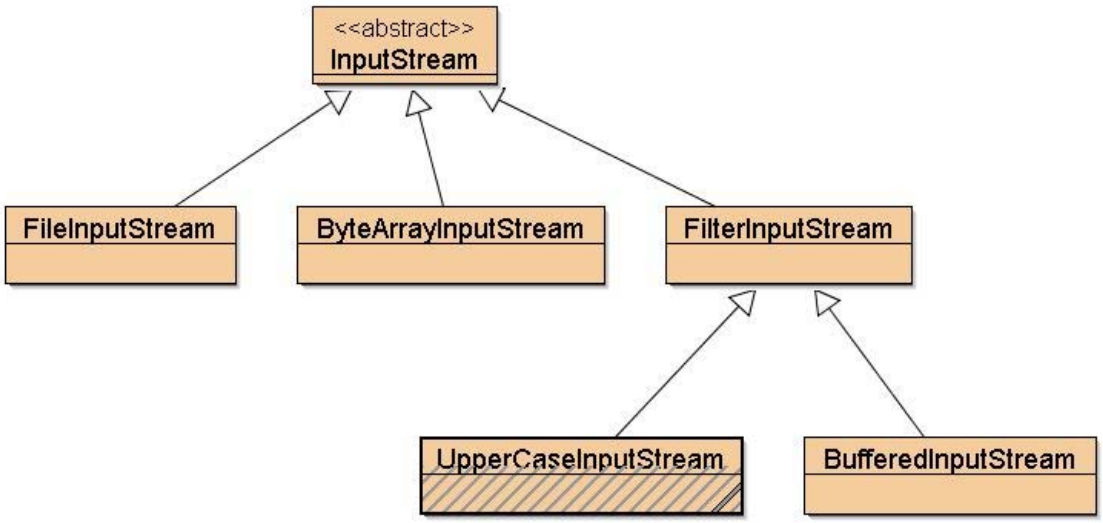
Question 3.2) On a ajouté la boucle "Pour" au paquetage question3, sur le modèle de la boucle "for" Java (cf. classe "Pour"). Et on a modifié les visiteurs en conséquence.

Question 3.3) Vérifiez le bon fonctionnement des classes de Tests.

Ne soumettez pas cette question à JNews.

---


## I/O JAVA et pattern décorateur




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


**.1)** Proposez la classe UpperCaseInputStream (en grisé ci dessus), un des décorateurs possibles de FilterInputStream, qui transforme en Majuscule tous les caractères du fichier transmis en paramètre dans le constructeur.

Les 3 méthodes à écrire doivent évidemment faire appel à leur équivalent dans la superclasse.
Consultez la javadoc des 2 méthodes `read()` à écrire pour savoir ce qu'elles doivent faire exactement.

Ci-dessous, une utilisation possible dans une classe de test :

```
public void testUpperCase_README_TXT() throws Exception
{
    InputStream is = new BufferedInputStream(
                         new FileInputStream(
                             new File( "README.TXT" ))); // déclaration à décorer ...
    int c = is.read();
    while( c != -1 ) {
      assertTrue( "erreur !, '" + Character.valueOf((char)c) + "' ne semble pas être une majuscule ...",
                  Character.isUpperCase((char)c) || (char)c==' ' );
```

```
            c = is.read();
        }
        is.close();
    }
```

Ne pas oublier ce cas :

```
byte[] tab = new byte[16];
int result = is.read( tab, 0, tab.length );
```


**.2)** Créez une instance de la classe UpperCaseInputStream, enrichie des fonctionnalités offertes par la classe **PushbackInputStream**.

Vérifiez, en proposant un test supplémentaire de la classe de test UpperCaseInputStreamTest, le bon fonctionnement de cette instance ainsi décorée....