TME 3 : Expressions arithmétiques

Objectifs pédagogiques : hiérarchie simple de classes, pattern Composite.

Dans ce TME, vous allez construire un mini-générateur d'expressions arithmétiques et l'environnement nécessaire à l'évaluation de ces expressions.

Une expression arithmétique est une entité composite constituée d'objets plus simples qui sont eux-mêmes des expressions arithmétiques. Les entités élémentaires qui composent une expression arithmétique sont les constantes, les variables et les opérateurs. Une expression arithmétique est susceptible d'être évaluée dans un environnement.

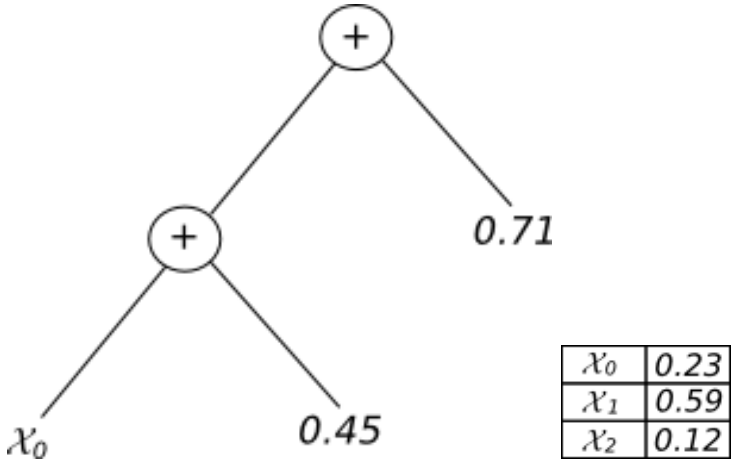

Figure 1: L'expression arithmétique $(X_0 + 0.45) + 0.71$ et un environnement d'évaluation : la variable X_0 vaut 0.23, X_1 vaut 0.59, etc.

La figure 1 représente l'expression arithmétique $(X_0+0.45)+0.71$ et un environnement d'évaluation. Pour évaluer une expression arithmétique, on remplace les variables de l'expression par les valeurs qu'elles prennent dans l'environnement. Par exemple, l'évaluation de l'expression arithmétique de la figure 1 par l'environnement d'évaluation associé donne $(0.23 + 0.45) + 0.71 = 1.39$.

Dans le cadre de ce TME, nous ne intéresserons qu'à des opérateurs binaires (l'addition, la soustraction, la multiplication et la division). Par ailleurs, nous traiterons exclusivement le cas où les constantes et les valeurs des variables sont des nombres réels, représentés en Java par des `double`.

3.1 Pattern Composite et diagramme de classes

Pour représenter les expressions arithmétiques décrites ci-dessus, on va utiliser le Design Pattern *Composite*. Le schéma de réalisation de ce pattern est illustré sur la figure 2.

Dans notre cas, la hiérarchie des classes sera organisée de la façon montrée sur la figure 3.

⇒ **Vous allez réaliser toutes les classes** conformément à ce qui est décrit dans le diagramme, en vous appuyant sur les informations complémentaires fournies dans la suite.

Dans le package pobj.arith (sélection du paquetage dans le panneau de structure à gauche, puis clic droit pour ouvrir le menu contextuel, choisir Nouveau...), créez les classes et interfaces nécessaires à la réalisation du projet.

Vous prendrez soin d'ajouter les commentaires javadoc.

Remarque: pour créer une sous-classe, il faut indiquer le nom complet de la super-classe. A tout moment, on peut utiliser la complétion automatique d'eclipse en tapant CTRL+ESPACE.

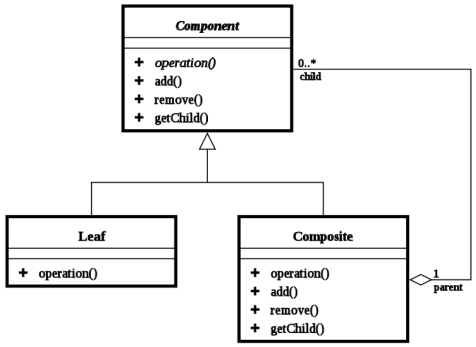

Figure 2

3.2 Classe EnvEval, Interface Expression

Pour évaluer les expressions, on utilise un environnement d'évaluation, qui associe des valeurs aux variables définies par le problème. Cet environnement d'évaluation est implanté dans la classe `EnvEval`. Il contient un tableau dont la taille fixée à la construction correspond au nombre de variables du problème et qui permet d'associer à l'aide de `void setVariable(int indexVariable, double val)` une valeur (un nombre réel) à chacune de ces variables. Une fois l'environnement d'évaluation rempli, on peut lui demander la valeur d'une variable donnée par un entier avec la méthode `double getValue(int indexVariable)`.

⇒ Réalisez cette classe `EnvEval`. Pensez à la munir d'un `toString` permettant des affichages lisibles.

⇒ **Créez ensuite une interface Expression**, dont l'unique méthode spécifie qu'une expression peut être évaluée par un environnement d'évaluation. Cette méthode a pour signature `double eval(EnvEval e)`.

3.3 Constante, Variable et OperateurBinaire

Les constantes sont des expressions qui contiennent simplement un réel entre 0 et 1. Leur évaluation renvoie leur valeur quel que soit l'environnement d'évaluation.

⇒ Créez la classe `Constante`. On la munira d'un attribut de type double déclaré `final`.

Les variables sont des expressions qui sont choisies dans une liste finie. Pour représenter ce choix, on leur affecte simplement leur rang dans la liste. Si le rang vaut « n », l'affichage de la variable donnera « Xn ». Leur évaluation renvoie la valeur de la variable de même rang dans l'environnement d'évaluation.

⇒ Créez la classe `Variable`. On la munira d'un attribut de type int (le rang) déclaré `final`.

Les opérateurs binaires sont des expressions qui s'appliquent à deux sous-expressions. On considère 4 types d'opérateurs binaires (l'addition, la soustraction, la multiplication et la division). Leur évaluation renvoie le résultat de l'opération correspondante appliquée aux valeurs renvoyées par l'évaluation des deux sous-expressions.

⇒ Créez la classe `OperateurBinaire`. Munissez-la de deux fils typés `Expression` et d'un attribut stockant le type de l'opérateur tous déclarés `final`. Pour la gestion des différents types d'opérateurs binaires, appuyez-vous sur un `enum` Java (voir les explications un peu plus bas).

⇒ Assurez-vous que l'ensemble de ces classes `implements Expression` et sa méthode `eval`.

Enumérations `enum` en Java.

Une énumération java (introduite depuis Java 1.5) permet de représenter un type de données qui prend ses valeurs dans un domaine fixe. Le tutoriel

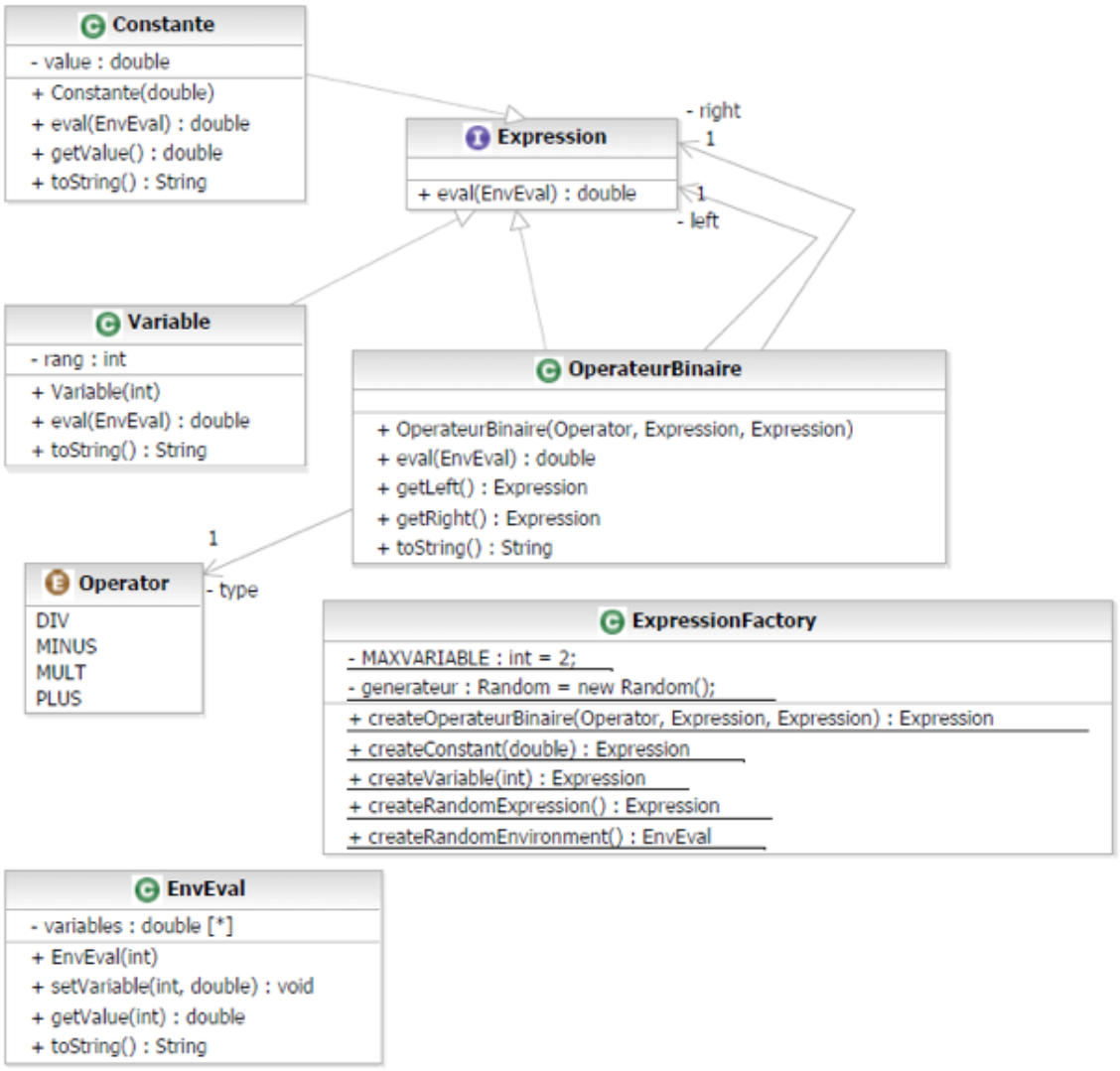

Figure 3

`http://download.oracle.com/javase/tutorial/java/javaOO/enum.html` en donne un bon aperçu. L'exemple reproduit ici en donne la syntaxe.

```
public enum Direction {
  NORTH, SOUTH, EAST, WEST
}
```

On peut ensuite déclarer une variable de type Direction et s'en servir. Cf. cette méthode `main()` de test:

```
public static void main(String[] args) {
  Direction dir = Direction.SOUTH;
  switch (dir) {
    case NORTH :
    System.out.println("NORTH Impossible !");
    break;
    case SOUTH :
    System.out.println("OK");
    break;
```

```
    default :
    System.out.println("Ni North, Ni SOUTH, Impossible !");
  }
  System.out.println("direction : " + dir);
  System.out.println("Les directions " + Arrays.toString(Direction.values()));
  System.out.println("La troisième direction :" + Direction.values()[2]);
}
```

La sortie obtenue est :
OK
direction : SOUTH
Les directions [NORTH, SOUTH, EAST, WEST]
La troisième direction :EAST

3.4 Production d'expressions : classe ExpressionFactory

Afin de cacher la façon dont sont implémentées les expressions, vous allez centraliser dans une classe munie de méthodes `static` des opérations permettant de construire des expressions.

⇒ Construisez cette classe `ExpressionFactory` et les trois méthodes suivantes.

```
/**
* Un constructeur pour des expressions binaires usuelles: +,-,*,/
* @param op le type de l'opérande, {@link Operator}, PLUS,MOINS,MULT,DIV
* @param left operande gauche
* @param right operande droite
* @return une expression binaire
*/
public static Expression createOperateurBinaire (Operator op, Expression left, Expression
right) {
/**
* Un constructeur d'expressions constantes.
* @param constant sa valeur
* @return une constante
*/
public static Expression createConstante(double constant) {
/**
* Un constructeur de variables, identifiées par un entier compris entre 0 et MAXVARIABLES.
* La demande de création de variables d'indice plus grand entraine un accroissement de
* MAXVARIABLE (attribut static).
* @param id l'indice de la variable
* @return une Variable
*/
public static Expression createVariable (int id) {
```

⇒ Restreignez ensuite la visibilité des constructeurs de vos classes représentant des expressions, pour être de visibilité package (pas de modificateur de visibilité).

On assure ainsi qu'un client ne peut construire des expressions que via la Factory. On est en train ici de réaliser une version simplifiée du Design Pattern *Factory*, qui sera vu en détail plus tard dans l'UE. Comme on le voit, l'objectif est que le client n'ait besoin de connaître que l'interface `Expression` et la Factory mais pas les classes implémentant `Expression`. Cela permet de facilement de modifier notre implémentation si besoin est.

⇒ Ajoutez ensuite une opération permettant de créer un `EnvEval` aléatoire, dont le nombre de variables sera passé en paramètre au programme principal. Les valeurs des variables sont prises entre 0 et 1.

⇒ Pour finir, implantez dans la Factory une méthode permettant de construire une expression aléatoire, qui ne contienne pas de divisions. Les opérandes des opérateurs binaires seront eux-mêmes

générés aléatoirement. Les variables auront un indice aléatoire qui n'excède pas MAXVARIABLE.

NB : Le générateur de nombres aléatoires (instance de `java.util.Random`) pourra être stocké comme attribut `static` de la Factory.

NB : Aléatoire + appels récusifs = problèmes potentiels d'expressions de profondeur infinie !!

⇒ Pour traiter correctement le problème bornez la profondeur de récursion, par exemple à l'aide d'un paramètre supplémentaire « profondeur » décrémenté à chaque appel récursif. Codez aussi une méthode `createRandomExpression` sans arguments qui crée des expression de profondeur max 3.

⇒ Testez l'ensemble en évaluant des expressions aléatoires sur des environnements aléatoires avec des affichages. Vous placerez la classe portant ce `main` dans un package `pobj.arith.test` différent du package `pobj.arith` contenant les autres classes.

3.5 Rappel : création d'une archive et remise du TME (Impératif)

Copiez-collez dans votre mail le code de votre méthode de génération aléatoire d'expressions. Donnez l'ordre de grandeur de la valeur des expressions que vous générez (faites un vingtaine de lancements).