Université Paul Cézanne - Licence de Math-Info 2^e année

I4 : Programmation Objet - Examen 2^e session
Durée : 1h30 - Aucun document autorisé

Important : dans toutes les classes que vous écrirez, tout attribut devra être déclaré privé. Vous ne rajouterez que les méthodes d'accès et de modification d'attribut qui seront nécessaires au bon fonctionnement de la classe.

On veut modéliser les expressions booléennes afin de pouvoir afficher et donner le résultat de ces expressions. On définit donc d'abord l'interface suivante :

```
interface ExpressionBooleenne
{
    int evalue() ;
    void affiche() ;
}
```

Une *expression booléenne* se définit formellement comme étant soit la *constante* VRAI ou FAUX, soit une *variable booléenne*, soit l'*expression unaire NON*, soit une *expression binaire*. Syntaxiquement, l'*expression unaire NON* est composée de l'opérateur unaire NON suivi d'une *expression booléenne* entre parenthèses. Une *expression binaire* est composée d'une *expression booléenne* suivi d'un *opérateur binaire*, par exemple OU ou ET, suivi d'une *expression booléenne*, le tout entre parenthèses. Les classes que nous définirons seront telles qu'elles permettront par exemple leur utilisation suivante :

```
public class TestExpression
{
   public static void main(String[] Args)
   {
      Variable v = new Variable(true) ;
      ExpressionBooleenne e = new Ou(new Non(v), Constante.FAUX) ;

      e.affiche() ; System.out.println() ; // affiche (NON(VRAI) OU FAUX)
      System.out.println(e.evalue()) ; // affiche : false
      v.setValeur(false) ;
      System.out.println(e.evalue()) ; // affiche : true
   }
}
```

1. *Ecrire la classe* `Variable` *implémentant l'interface* `ExpressionBooleenne` *et définissant en plus :*
- un constructeur à un paramètre de type `boolean` permettant d'initialiser la valeur de la variable booléenne,
- le modifieur `void setValeur(boolean)` permettant de changer la valeur de la variable.

2. *Ecrire la classe* `Non` *implémentant l'interface* `ExpressionBooleenne` *et définissant notamment un constructeur à un paramètre de type* `ExpressionBooleenne`.

3. *Ecrire la classe abstraite* `ExpressionBinaire` *contenant notamment :*
- deux attributs mémorisant des expressions booléennes,
- un constructeur prenant deux expressions booléennes en paramètre,
- la méthode abstraite `operation` prenant 2 valeurs de type `boolean` en paramètre et retournant une valeur de type `boolean`,
- l'implémentation de la méthode `boolean evalue()`.

4. *Ecrire la classe concrète* `Ou` *qui hérite de* `ExpressionBinaire` *et représente le OU logique. En quoi la classe* `Et` *(qui représente le ET logique) s'en différencie-t'elle ?.*

5. *Ecrire la classe* `Constante` *qui définit deux constantes, VRAI et FAUX, et dont voici les caractéristiques :*
- elle a les mêmes méthodes que `Variable`, qui effectuent exactement les mêmes opérations, sauf que le modifieur `void setValeur(boolean)` n'existe pas. On fera donc en sorte de réutiliser intelligemment la classe `Variable` sans réécrire le contenu des méthodes qui sont identiques.
- elle définit les deux attributs de classe `VRAI` et `FAUX`, qui sont de type `Constante` et qui sont respectivement initialisés avec les valeurs `true` et `false`.
- on ne peut pas créer d'autres instances de `Constante`. `VRAI` et `FAUX` sont les seules instances qui existeront.

6. *Ecrire la classe* `ExpBoolFactory` *qui permet de créer des expressions booléennes grâce à trois méthodes de classe qui ont le même nom* `expbool` *mais un nombre différent de paramètres :*
- la première méthode a un paramètre de type `String` qui retourne une variable (une instance de `Variable`) valuée à `true` si le paramètre est "VRAI", une variable valuée à `false` si le paramètre est "FAUX" et `null` dans les autres cas.
- la deuxième méthode a deux paramètres. Le premier est censé être "NON" et le deuxième est soit "VRAI", soit "FAUX", soit une expression booléenne. Cette méthode retourne l'expression booléenne représentant la négation de l'expression représentée par le deuxième paramètre.
- la troisième méthode a trois paramètres. Le premier et le troisième peuvent être soit "VRAI", soit "FAUX", soit une expression booléenne. Le deuxième est soit "OU" soit "ET". Cette méthode retourne l'expression booléenne représentant le OU ou le ET entre les expressions représentées par le premier et le troisième paramètres.

Ainsi plutôt que d'écrire `ExpressionBooleenne e = new Ou(new Non(v), Constante.FAUX);`, on pourra écrire

```
ExpressionBooleenne e = ExpBoolFactory.expbool(ExpBoolFactory.expbool("NON", v), "OU", "FAUX");
```

7. On veut maintenant pouvoir définir autrement les expressions binaires. L'idée est de créer chaque expression binaire en passant en paramètre l'opération qu'elle effectue au moment de sa création (plutôt que de définir une nouvelle classe par héritage pour chaque type d'opération binaire comme jusqu'à présent). Pour ceci, on définit les classes suivantes :

```
interface OperateurBinaire
{
   boolean applique(boolean e1, boolean e2);
   String getName();
}
class OperateurOu implements OperateurBinaire
{
   public boolean applique(boolean e1, boolean e2) { return e1 || e2;}
   public String getName() { return "OU"; }
}
```

Ecrire la classe concrète `ExpressionBinaire2` *implémentant* `ExpressionBooleenne` *et contenant notamment un constructeur prenant deux expressions booléennes ainsi qu'un opérateur binaire en paramètre.* Cette classe devra alors permettre de définir ainsi l'expression booléenne déjà donnée en exemple :

```
ExpressionBooleenne e = new ExpressionBinaire2(new Non(v), new OperateurOu(), Constante.FAUX);
```