Examen

3 heures - documents écrits autorisés
jeudi 5 juin 2003

Les deux exercices ne sont pas réellement indépendants, mais il est possible de traiter le second sans avoir répondu à toutes les questions du premier. Il est cependant préférable d'avoir lu l'ensemble du sujet.

Exercice 1 : Expressions logiques

Dans cet exercice nous nous placerons dans le contexte de la **logique trivaluée** dans laquelle il existe trois valeurs : aux habituelles valeurs "*vrai*" et "*faux*" vient s'ajouter la valeur "*inconnu*" permettant d'exprimer l'absence d'information sur la valeur de vérité d'une donnée.
Les opérateurs booléens classiques sont étendus pour prendre en compte la nouvelle valeur ajoutée :

- l'opérateur unaire *non* réalise la négation, la négation de la valeur *inconnu* vaut *inconnu*,
- l'opérateur binaire *et* qui réalise la conjonction ("et" logique) entre deux booléens tri-valués, la table de vérité en est la suivante :

A et B	*vrai*	*faux*	*inconnu*
vrai	*vrai*	*faux*	*inconnu*
faux	*faux*	*faux*	*faux*
inconnu	*inconnu*	*faux*	*inconnu*

Dans la suite du sujet le terme "booléen" fera référence à ce type de valeurs.

Q 1 . Définissez une classe `BooleanConstant` qui permet de représenter les booléens mentionnés précédemment ainsi que les opérateurs "non" et "et" associés.

Vous définirez également une méthode qui permet de "convertir" une donnée de type primitif "`boolean`" vers son correspondant dans ce type.

Définissez une méthode qui retourne un objet `BooleanConstant` à partir d'une chaîne de caractères, le résultat est `null` si la chaîne ne correspond pas à l'un des trois symboles mentionnés correspondant aux constantes.

Nous nous intéressons maintenant aux expressions booléennes (toujours en logique trivaluée). Parmi ces expressions considérés on distinguera les expressions *atomiques* et les expressions *composées*.
Les expressions atomiques sont :

- les constantes *vrai*, *faux* et *inconnu*,
- les variables à valeur booléenne identifiées par un symbole (différents des opérateurs booléens). Si une variable n'est pas définie dans le contexte son évaluation vaut "*inconnu*".

Les expressions composées regroupent :

- les expressions construites à l'aide de prédicats de comparaison numériques $=$, $<$ et $>$, dont le terme gauche sera une variable numérique identifiée par un symbole (différent des opérateurs booléens mentionnés) et le terme droit un entier,
- les expressions contruites par application de l'opérateur de négation sur une expression booléenne, elles sont de la forme `non` *`expression`*,
- les expressions construites comme conjonction de deux expressions booléennes, elles sont de la forme *`expression1`* `et` *`expression2`*.

Ces expressions sont définies par le type abstrait `BooleanExpression` :

```
package expression;

public interface BooleanExpression {
  /** évalue une expression en logique trivaluée selon de contexte donné
   *@param context le contexte de l'évaluation
   *@return le résultat de l'évaluation
   */
  public BooleanConstant evalue(Context context);
}
```

Evaluer une expression booléenne c'est calculer sa valeur en logique trivaluée. Une évaluation dépend de son contexte. Nous définirons un contexte par la donnée de variables et de leurs valeurs associées. Dans la suite nous supposerons qu'il n'existe que des variables numériques et booléennes (tri-valuées), celles-ci sont définies par les interfaces et classes données en annexe.

Q 2 . Définissez une classe `Context` permettant de regrouper l'ensemble des variables et de leur valeur pour un contexte donné. Il faut bien entendu pouvoir accéder à et mettre à jour la valeur associée à une variable. Les identifiants de variables seront des chaînes de caractères (`java.lang.String`) supposées sans espace (pas de vérification demandée).

Q 3 . Donnez le diagramme UML d'héritage des différents types nécessaires à la représentation des expressions booléennes atomiques et composées (il faut bien sûr utiliser le type `BooleanExpression` défini précédemment).

On ne demande pas ici de détailler le contenu des classes (attributs et méthodes, ni les dépendances vers d'autres types que ceux représentant les expressions booléennes).

Q 4 . On s'intéresse maintenant aux expressions construites à partir des comparateurs numériques et décrites précédemment.

Q 4.1. Donnez (en détaillant cette fois) le diagramme UML de tous les types que vous définissez pour votre modélisation de ce cas.

Q 4.2. Donnez ensuite tout le code java nécessaire à l'implémentation de votre solution.

Q 4.3. Définissez une "factory method" permettant la construction des expressions binaires à partir d'une chaîne de caractères. La construction est possible si la chaîne est de la forme "*`symbole opérateur entier`*", où *`opérateur`* est $<$, $>$ ou $=$, dans le cas contraire le résultat vaut `null`. (voir en annexe des extraits de javadoc probablement utiles).

Q 5 . Donnez les codes nécessaires aux deux autres catégories d'expressions booléennes composées.

Q 6 . Définissez une classe `BooleanExpressionFactory` qui représentera, comme son nom l'indique une "factory" pour les expressions booléennes décrites dans ce sujet.

Cette classe devra satisfaire le design pattern singleton et disposera d'une méthode :

```
public BooleanExpression build(String t)
```

qui à partir de la chaîne `t` construit l'objet "expression booléenne" correspondant. On supposera que la chaîne de caractères est bien formée et qu'elle correspond effectivement à l'une des expressions booléennes décrites précédemment.

L'opérateur "*et*" doit être considéré comme moins prioritaire que l'opérateur de négation.

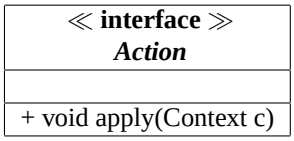
Voici donc quelques exemples de chaînes de caractères admissibles :
`"vrai"`, `"x>4"`, `"non x<10"`, `"x=4 et non y>5"`, `"non non x=4 et y=12"` (équivaut à "(non (non x=4)) et y=12").
Une chaîne telle que `"varBool"` provoquerait la construction d'une expression atomique correspondant à une variable booléenne dont l'identifiant serait `varBool`, on ne s'occupe pas ici de savoir si cette variable est définie, cela n'a d'importance qu'au moment de l'évaluation.

Exercice 2 : Chaînage avant simplifié.

L'algorithme de chaînage avant est utilisé dans les systèmes experts pour calculer toutes les déductions possibles à partir d'une base de connaissance. La base de connaissance regroupe à la fois des règles, dans la base de règles, et des faits, dans la base de faits.
Les faits peuvent être considérés comme des variables dans une première approximation dont nous nous contenterons ici. Nous ne manipulerons que des variables entières et booléennes. Ici une base de faits correspondra à une instance de la classe `Context` définie dans l'exercice 1.
Les règles sont des structures de la forme :

SI *condition* ALORS *action*

où *condition* est une expression booléenne (trivaluée) comme définie dans l'exercice 1 et *action* consiste en une affectation de valeur à une variable (du moins nous nous contenterons de cela ici).
On peut *exécuter*, ou *appliquer*, une règle dès que sa condition est satisfaite, c'est-à-dire vaut "*vrai*". Dans ce cas, on réalise l'action décrite dans la seconde partie de la règle et on modifie donc éventuellement la base de faits (c-à-d l'objet "`Context`"). On supposera que l'application d'une action ne peut pas contredire les informations déjà connues du contexte.
La partie *action* d'une règle est décrite par des objets de type `Action` dont voici le diagramme UML :



On définit une base de règles comme une collection[1] de règles. Dans une base de règles, une règle peut être active ou non.

Q 1 . Donnez le code d'une classe permettant de définir une règle. Vous lui donnerez les méthodes que vous jugerez nécessaires. Cette question doit peut-être être traitée en simultanée avec la suivante.

Q 2 . Codez une classe permettant de définir une base de règles.

Cette classe possèdera au moins la méthode :

```
public void chainageAvant(Context c)
```

qui réalise l'algorithme de chaînage avant donné en annexe pour cette base de règles et le contexte donné en paramètre.

Annexe

Valeurs

```
package expression;

public interface Value {
  public int intValue() throws UnsupportedOperationException;
  public BooleanConstant booleanValue() throws UnsupportedOperationException;
}

public class ValueAdapter implements Value {
  public int intValue() throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
  }
  public BooleanConstant booleanValue() throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
  }
}

public class NumericValue extends ValueAdapter {
  private int value;
  public NumericValue(int value) {
    this.value = value;
  }
  public int intValue() throws UnsupportedOperationException {
    return value;
  }
}
public class BooleanValue extends ValueAdapter {
  private BooleanConstant value;
  public BooleanValue(BooleanConstant value) {
    this.value = value;
  }
  public BooleanConstant booleanValue() throws UnsupportedOperationException {
    return value;
  }
}
```

Algorithme de chaînage avant

```
activer toutes les règles
tant qu'il existe des règles actives et
          des règles dont la condition est satisfaite
     choisir une de ces règles, l'appliquer et la désactiver
     désactiver les règles dont la condition vaut "faux"
fin tant que
```

[1] N'y voyez pas nécessairement un sens Java.

Extraits de javadoc

`java.lang.String`

`String trim()` Returns a copy of the string, with leading and trailing whitespace omitted.

`int indexOf(String str)` Returns the index within this string of the first occurrence of the specified substring.

`String substring(int beginIndex)` Returns a new string that is a substring of this string.

`String substring(int beginIndex, int endIndex)` Returns a new string that is a substring of this string.

`java.util.StringTokenizer`

`public StringTokenizer(String str, String delim, boolean returnDelims)`
Constructs a string tokenizer for the specified string. All characters in the delim argument are the delimiters for separating tokens.

If the `returnDelims` flag is `true`, then the delimiter characters are also returned as tokens. Each delimiter is returned as a string of length one. If the flag is `false`, the delimiter characters are skipped and only serve as separators between tokens.

`int countTokens()` Calculates the number of times that this tokenizer's `nextToken` method can be called before it generates an exception.

`boolean hasMoreTokens()` Tests if there are more tokens available from this tokenizer's string.

`String nextToken()` Returns the next token from this string tokenizer.