Examen

3 heures - documents écrits autorisés
samedi 13 septembre 2003

Exercice 1 : Athlétisme et n-athlon

Une compétition d'athlétisme est composée de concours dans différentes `disciplines`. On regroupe en trois catégories ces disciplines : les `lancers` (javelot, disque, poids), les `sauts` (hauteur, longueur, triple, perche) et les `courses` (marathon, 100m, 400m haies, 20km marche, etc). La performance d'un lancer est donnée par une longueur (méthode `Longueur getLongueur()`, celle d'un saut par une distance `Distance getDistance()`et celle d'une course par un temps `Temps getTemps()`[1]. Chaque discipline répond à l'envoi de message `String toString()` qui retourne sous forme d'une chaîne le nom de la discipline et la performance associée.

Q 1 . On décide de représenter la notion de `Discipline`, de `Saut`, `Lancer`, `Course` par des interfaces. On prend comme exemple les classes `Javelot`, `Hauteur` et `400m`.

Donnez le diagramme UML faisant apparaître ces 7 entités, ainsi que les classes `Temps`, `Distance` et `Longueur`, et leurs relations de dépendances ou d'implémentation.

Certaines compétitions sportives consistent à faire concourir des athlètes dans différentes disciplines et de comptabiliser puis de cumuler leurs résultats dans ces différentes disciplines afin d'en déterminer le vainqueur. C'est de cas des n-athlons : décathlon, heptathlon, triathlon, biathlon nordique, pentathlon moderne, etc.
Un n`-athlon` est donc composé de n `épreuves`, chacune de ces épreuves rapporte des points. La manière dont sont comptés les points est propre à chaque épreuve et le score d'une épreuve est accessible par sa méthode `int getScore()`. Le score d'un n-athlon (méthode `int getScore()`) est la somme des scores des épreuves qui le composent et on peut afficher (méthode `void afficher()`) chacune des épreuves avec sa performance.
Le `décathlon` est un n-athlon dont les 10 `épreuves` sont certaines `disciplines` de l'athlétisme (mais toutes les disciplines ne sont pas des épreuves) telles que le *lancer du javelot*, le *saut en hauteur*, le *400m*, etc..

Q 2 . Reprenez (sur un nouveau schéma !) le diagramme précédant et complétez le pour gérer la notion d'épreuve et de n-athlon (en ne faisant à nouveau apparaître que les cas particuliers des 3 épreuves mentionnées ci-dessus).

Q 3 . Donnez le code de la classe modélisant un n`-athlon`.

Exercice 2 : Arbres n-aires

Nous nous intéressons à l'implémentation des arbres n-aires, à arité non bornée. Ces arbres sont tels que tout nœud peut avoir un nombre quelconque de fils. Comme pour les arbres binaires, l'arbre qui ne contient aucun nœud est appelé arbre vide. De plus, dans un arbre, l'unique nœud qui n'a pas de père est appelé racine, et les nœuds qui n'ont pas de fils sont appelés feuilles. A tout nœud on associe une valeur. Dans cet exercice, la valeur asociée à un nœud sera de type `Object`.
Une interface minimaliste qui définit les opérations disponibles sur un arbre n-aire est donné en annexe (interface `ArbreNAire`).
Cette interface ne prévoit pas comment ajouter des nœuds dans un arbre, ni comment obtenir les valeurs associées aux nœuds d'un arbre. Pour notre implémentation, ces opérations se font via un objet `Curseur`, qui indique une certaine position dans un arbre et permet ainsi de se déplacer dans cet arbre. Le principe en est le même que celui des itérateurs pour les listes.
Pour cela, nous avons défini deux interfaces : l'une qui spécifie la notion de `Curseur` et l'autre qui spécifie la notion d'arbre avec curseur (`ArbreACurseur`).
Ces interfaces sont données en annexes.
Les questions suivantes ont pour objectif d'écrire (partiellement) la classe `ArbreNaireACurseur` qui implémente les interfaces `ArbreNaire` et `ArbreACurseur`.

Q 1 . Donner les attributs (variables d'instances) de la classe `ArbreNaireACurseur` permettant de gérer le père et la valeur (quand ils existent) et les fils d'un nœud.

Q 2 . Donner l'implémentation des méthodes de cette classe dues à l'implémentation de l'interface `ArbreNaire`.

[1] Les classes `Longueur`, `Distance` et `Temps` sont supposées définies par ailleurs.

Q 3. Définir une classe `CurseurNaire`[2] et qui implémente l'interface `Curseur`.

Il est important pour pouvoir parcourir avec le curseur l'arbre de mémoriser le noeud pointé par le curseur ainsi que le fils courant (quand il en existe).

Comme cette interface est assez riche (bien qu'elle ne permette pas de modifier les valeurs ni de supprimer des nœuds), donner uniquement :

1. les variables d'instances de cette classe,[3]
2. les implémentations des méthodes permettant de se déplacer (`remonter()` `descendre()`, `allerADroite()`),
3. l'implémentation de la méthode `insererFils(int position, Object valeur)`.

Annexe

L'interface ArbreNaire

```
package arbresNaires ;

import java.util.*;

/** un arbre Naire est un arbre dont chaque noeud peut avoir un nombre
 *  quelconque de fils. De plus, chaque noeud contient une valeur.
 */
public interface ArbreNaire{
    /** teste si un arbre est une feuille, i.e. s'il n'a pas de fils */
    public boolean estFeuille() ;

    /** teste si un arbre est l'arbre vide, i.e. s'il n'a aucun noeud */
    public boolean estVide() ;

    /** renvoie la taille de l'arbre, i.e. son nombre de noeuds */
    public int taille() ;

    /** renvoie la hauteur de l'arbre (0 pour l'arbre vide, 1 pour une feuille)
     */
    public int hauteur();

    /** renvoie le noeud pere de ce noeud ou null si aucun*/
    public ArbreNaire getPere();

    /** renvoie la valeur de ce noeud ou null si arbre vide */
    public Object getValeur();

    /** renvoie la liste des fils  de ce noeud */
    public List getFils();
}
```

L'interface Curseur

```
package arbresNaires ;

/** Un curseur indique une position dans un arbre a.
 *  Si a est vide, la seule opération possible est d'inserer une valeur.
 *  Si a n'est pas vide, le curseur pointe sur un noeud de a.
 *  Lorsqu'un curseur est créé pour un arbre non vide, il pointe sur sa racine.
 */

public interface Curseur {
    /** le curseur remonte vers le pere du noeud courant.
     *  @exception CurseurException quand on est à la racine.
     */
    public void remonter() throws CurseurException ;
```

[2] Il **faudrait** définir cette classe comme interne à la classe `ArbreNaireACurseur` mais ce n'est pas demandé ici car non forcément vu cette année.

[3] avant de faire votre choix, tenez compte du fait que l'on ne demande que de pouvoir "remonter" vers le père d'un nœud et d'"aller à droite" de frère en frère, il n'est pas demandé de pouvoir "remonter" vers le "frère de gauche". Il faut donc pouvoir parcourir du premier au dernier (et seulement dans ce sens) la liste des frères.

```
    /** le curseur descend vers le premier fils du noeud courant.
     *  @exception CurseurException quand on est sur une feuille.
     */
    public void descendre() throws CurseurException ;

    /** le curseur passe au prochain frère du noeud courant
     *  @exception CurseurException quand il n'y a plus de frère à droite
     */
    public void allerADroite() throws CurseurException ;

    /** permet d'obtenir la valeur associée au noeud courant
     *  @return la valeur du noeud courant.
     *  @exception CurseurException quand l'arbre est vide
     */
    public Object valeur() throws CurseurException ;

    /** Si le curseur pointe sur un noeud n,
     *  cette opération ajoute un noeud n', père de n, tel que
     *  Le père de n' est l'ancien père de n.
     *  Si le curseur ne pointe sur aucun noeud (arbre vide),
     *  cette opération crée un noeud n'.
     *  Dans les 2 cas, le curseur pointe sur n' après cette opération.
     */
    public void insererPere(Object valeur) ;

    /** Ajoute au noeud ponité par le curseur un fils à la position donnée.
     *  S'il y a déjà un fils à cette position, on décale ce fils et
     *  tous ses frères droits.
     *  S'il n'y a pas de fils à position-1, le nouveau noeud est créé
     *  en dernière position.
     *  Si position <= 0, le nouveau noeud est créé en tête (position 0)
     *  Après cette opération, le curseur pointe sur le nouveau fils.
     */
    public void insererFils(int position, Object valeur) ;

    /** teste si le noeud pointé par le curseur a au moins un fils */
    public boolean aUnFils() ;

    /** teste si le noeud pointé par le curseur  a un père */
    public boolean aUnPere() ;

    /** teste si le noeud pointé par le curseur  a un frère à sa droite */
    public boolean aUnFrere() ;

}
```

L'interface ArbreACurseur

```
package arbresNaires ;

/** un arbre à curseur est un arbre pour lequel on peut obtenir
 *  un curseur. Ce curseur pointe sur un noeud de l'arbre et permet
 *  de se déplacer dans l'arbre.
 */
public interface ArbreACurseur{
    /** renvoie un curseur sur l'arbre.
     */
    public Curseur curseur()  ;
}
```