Programmation Objet

Examen

3 heures - documents écrits autorisés
12 septembre 2002

Exercice 1 : Hall of Fame

Le but de cet exercice est d'écrire une classe permettant de gérer les scores de joueurs pour un jeu donné. Les informations que l'on souhaite mémoriser pour un joueur, désigné par son nom, sont son meilleur score sur l'ensemble des parties, le nombre de parties jouées et la moyenne des scores de toutes les parties jouées.
Les jeux dont il est question se jouent seul et le joueur se voit délivrer en fin de partie un score (entier). Plus celui-ci est faible meilleur est le résultat du joueur. Le démineur ou le Mastermind sont des exemples de tels jeux.

Toutes les unités définies dans cet exercice appartiendront au paquetage `hall`.

Q 1 . Les jeux vont être définis par une interface `Jeu`, qui ne possédera qu'une méthode appelée `joue` qui prend en paramètre un nom de joueur (une chaîne de caractères) et retourne le score du joueur pour une partie du jeu. Définissez le code de cette interface.

Q 2 . La classe `Resultat` permet de gérer les informations pour un joueur.

Q 2.1. Définir les attributs nécessaires à la gestion de l'état des objets de la classe `Resultat`.

Q 2.2. Définir un constructeur qui prend comme paramètre le premier score effectué par le joueur au jeu concerné et initialise les attributs.

Q 2.3. Définir la méthode `getMeilleur` qui retourne le meilleur score du joueur.

Q 2.4. Définir le code de la méthode dont la signature est :

```
public void nouveauScore(int score)
```

et qui permet d'ajouter un nouveau score aux résultats d'un joueur.

Q 3 . La classe `HallOfFame` permet de gérer les résultats pour l'ensemble des joueurs pour un jeu donné. Elle mémorise pour chaque joueur, désigné par son nom (une chaîne de caractères) un objet de la classe `Résultat`.

Q 3.1. L'état des instances de cette classe est défini par le jeu concerné et la table de hachage des résultats pour celui-ci. Faites les déclarations nécessaires ainsi qu'un constructeur, prenant comme paramètre un jeu, et initialisant les attributs définis.

Q 3.2. Définissez une méthode `joueurJoue` qui prend en paramètre un nom de joueur, fait jouer une partie à celui-ci et met à jour ses résultats.

Q 3.3. Définissez le code de la méthode dont la signature est :

```
public void meilleurMeilleur()
```

et qui affiche le nom du (ou de l'un des) joueur ayant le meilleur de tous les scores pour le jeu.

Exercice 2 : Le MasterMind

Dans cet exercice nous allons modéliser le jeu du Mastermind.
Rappelons brièvement les règles de ce jeu, a priori bien connu. Le joueur doit découvrir une combinaison secrète de *taille* couleurs (ordonnées et éventuellement avec répétition) par propositions successives de combinaisons réponse. Après chaque essai, le joueur reçoit des informations sur la qualité de sa proposition sous la forme de pions noirs et blancs : un pion noir signifie qu'une couleur se trouve à la même position dans sa proposition et dans la combinaison secrète, un pion blanc signifie qu'une couleur est présente dans sa proposition et dans la combinaison secrète mais à des positions différentes. A chaque couple position/couleur ne peut correspondre qu'au plus un pion blanc ou noir et les noirs ont priorité.
Ainsi si, pour *taille=4*, on a :

position	1	2	3	4
secret	JAUNE	BLEU	JAUNE	ROUGE
proposition	ROUGE	BLEU	JAUNE	BLEU

l'information fournie pour la proposition est : 1 pion noir (pour la couleur `JAUNE` à la position 3), 1 pion noir (pour la couleur `BLEU` en position 2, la couleur `BLEU` en position 4 de la proposition n'est pas prise en compte) et 1 pion blanc pour le pion `ROUGE` en position 1 de la proposition et 4 du secret. La couleur `JAUNE` à la position 1 du secret ne se voit attribuer aucun pion car le `JAUNE` de la proposition du joueur est déjà utilisé par la position 3 pour un pion noir.

Le joueur dispose de l'historique de toutes ses propositions et des informations qui en ont découlées et dispose d'un nombre d'essais limité, appelé *hauteur* pour trouver la combinaison secrète.
Toutes les unités définies dans cet exercice appartiendront à un paquetage `mastermind`.

Q 1 . Définissez la classe `Couleur` qui permettra de définir **exclusivement** 6 constantes de nom `ORANGE`, `BLEU`, `JAUNE`, `ROUGE`, `VERT`, `MARRON` (`Couleur` correspond donc à un "type énuméré"). Les 6 instances de ce type ne seront définies que par un attribut `nom` de type `String`. Définissez également les méthodes `toString` (qui retourne le nom) et `equals` de cette classe.

Q 2 . Vous trouverez en annexe la définition de la classe `Combinaison`. Celle-ci permet de définir des instances utilisées pour représenter des combinaisons de couleur (secret ou proposition du joueur). Les informations fournies au joueur en réponse à sa proposition sont représentées par un objet de la classe `Reponse` dont un squelette est donné en annexe. Compléter la méthode `compare` de cette classe.

Q 3 . Nous allons maintenant nous intéresser à la classe `MasterMind`.

Q 3.1. L'état des instances de cette classe est caractérisé par les entiers *taille* et *hauteur*, la combinaison secrète et le tableau de l'historique des propositions du joueur. Faites les déclarations nécessaires.

Q 3.2. Le constructeur de cette classe initialise la taille et la hauteur, définissez le.

Q 3.3. (La lecture de l'exercice 1, à défaut de sa réalisation, est nécessaire à la compréhension de cette question). Il sera possible de gérer le tableau d'honneur des intances de la classe `MasterMind` par des instances de la classe `HallOfFame` de l'exercice 1. La classe `MasterMind` sera donc conforme à l'interface `Jeu` de la question de l'exercice 1.
On supposera définie la méthode :

```
private Combinaison propositionJoueur()
```

qui attend la saisie par l'utilisateur d'une proposition de combinaison et retourne cette proposition.
Faites les déclarations nécessaires pour la conformité de la classe `MasterMind` à l'interface `Jeu` et donnez notamment le code de la méthode `joue` qui commencera par initialiser la combinaison secrète puis lira les propositions du joueur jusqu'à ce qu'il ait trouvé le secret ou épuisé le nombre d'essais autorisé. Après chaque proposition, le nombre de pions noirs et blancs correspondant est affiché. Le score du joueur est le nombre d'essais qui lui auront été nécessaires pour trouver le secret.

Annexe

La classe `Combinaison`

```
package mastermind;

public class Combinaison {
    private int taille;
    private Couleur[] couleurs;
    public Combinaison () {
        this(4);
    }
    public Combinaison (int taille) {
        this.taille = taille;
        couleurs = new Couleur[taille];
    }
    public int getTaille() { return taille; }
    public void setCouleur(int position, Couleur couleur) {
        couleurs[position] = couleur;
    }
    public Couleur getCouleur(int position) {
        return couleurs[position];
    }
    /** initialisation aléatoire des couleurs de la combinaison */
    public void init() {
        ... méthode supposée définie ...
    }
    /** retourne l'objet Reponse permettant de comparer cette
      * combinaison et celle passée en paramètre.
      * @param lAutre la combinaison à laquelle se comparer
      * @return la réponse associée
    */
```

```
    public Reponse compare(Combinaison lAutre) {
        return new Reponse(this, lAutre);
    }

    public String toString() {
        String result = "";
        for(int i = 0; i <taille; i++) {
            result = result + " "+ couleurs[i];
        }
        return result;
    }
}// Combinaison
```

La classe Reponse

```
package mastermind;

public class Reponse {

    private int nbNoirs = 0;
    private int nbBlancs = 0;

    public Reponse(Combinaison lUne, Combinaison lAutre){
        compare(lUne,lAutre);
    }

/** calcule le nombre de bien placés (noirs) et de mal placés
     * (blancs) entre les deux combinaisons fournies en paramètre, par
     * mise à jour des attributs nbNoirs et nbBlancs
     * on suppose que les combinaisons sont de même taille sans
     * vérification
     * @param lUne une combinaison
     * @param lAutre une combinaison
     /
    private void compare(Combinaison lUne, Combinaison lAutre) {
        ... A COMPLETER ...
    }

    public int getNbBlancs() { return nbBlancs; }
    public int getNbNoirs() { return nbNoirs; }
    public String toString() {
        return nbNoirs+" noir(s) - "+nbBlancs+" blanc(s)";
    }
}// Reponse
```