POBJ – Examen Décembre 2011

Durée : 2 heures Tous documents autorisés

PROBLÈME : GESTIONNAIRE D'EXAMEN

On vous confie le développement d'un outil de gestion de copies d'examen qui assure notamment la correction automatique. Votre outil doit être générique, il doit pouvoir gérer différents types d'épreuves, dont des Questionnaires à Choix Multiples (QCM) et des énoncés plus classiques constitués d'exercices découpés en questions et éventuellement exercices plus simples. Au coeur de votre outil se trouve une interface *IEpreuve* qui calcule le nombre de points que reçoit une copie, définie ainsi :

```
package pobj;
public interface IEpreuve {
	public int calcNbPoints(Copie c);
}
```

Une copie est un ensemble de réponses que l'on peut parcourir avec un itérateur. La méthode *getName()* permet d'identifier l'étudiant auteur de la copie de manière unique.

```
package pobj;
public interface Copie extends Iterable<Reponse> {
	public String getName();
}
```

Pour évaluer la réponse fournie par une copie, il faut pouvoir la comparer à une réponse attendue. On définit donc l'interface *Question* avec une méthode *evalQuestion(Reponse r)* qui renvoie un nombre de points dépendant du type de question et de la réponse évaluée :

```
package pobj;
public interface Question {
	public int evalQuestion(Reponse r);
}
```

Gestion des oublis des import ou du package : dans le tableau, il y a une colonne malus. 0 si c'est parfait, 100% si ça manque tout le temps. => l'étudiant perd un point à l'exam (au départ, je proposais -25%, mais si on applique à toutes les questions ça fait cher...). Ne pas oublier de remplir la colonne. Et indiquez le malus (en %) en début de copie

Barême classique à appliquer partout :

-25% s'il y a des petites erreurs de syntaxe : ;, {}, ()...

-25% pour les erreurs de visibilité (attributs public...)

-50% pour les fautes de prog.

On ne paye pas l'encre, il faut que ce soit (à peu près) juste pour avoir des points ;)

1 : Questions concrètes

Pour commencer, nous allons implémenter plusieurs types de questions.

1.1 : Question à choix multiples

Les questions à choix multiples n'attendent qu'une réponse, qui est un choix parmi un ensemble de propositions. A la création d'une question à choix multiples, on définit une réponse juste et une liste de réponses fausses. Pour savoir si la réponse fournie par l'étudiant est juste, on compare son contenu à celui de la réponse fournie à la création de la question. La méthode d'évaluation d'une réponse est la suivante :

- si le candidat ne fournit pas de réponse (i.e. si la référence sur la *Reponse* fournie pointe sur *null*), il ne reçoit aucun point,
- si le candidat fournit la bonne réponse, il reçoit deux points,
- si le candidat fournit une mauvaise réponse, il perd un point.

Q1.1 : Dans le package *pobj*, proposez l'implantation des questions à choix multiples dans une classe *QuestionChoixMultiples.* Ecrivez notamment la méthode d'évaluation spécifique de ces questions.

```
package pobj;
import java.util.List;


public class QuestionChoixMultiples implements Question {
	Reponse bonneReponse;
	List<Reponse> mauvaisesReponses ; //on ne sanctionne pas ArrayList

	public QuestionChoixMultiples(Reponse bonneReponse, List<Reponse> mauvaisesReponses) {
		this.bonneReponse = bonneReponse;
		this.mauvaisesReponses = mauvaisesReponses;
	}
	public int evalQuestion(Reponse r) {
		if(r==null) return 0;
		if(r.equals(bonneReponse))	return 2;
		else return -1;
	}
	@Override
	public String toString() { //non demandé
		return "QuestionChoixMultiple [bonneReponse=" + bonneReponse
				+ ", mauvaisesReponses=" + mauvaisesReponses + "]";
	}
}
```

-50% pour l'oubli du cas null

au moins 50 si evalQuestion() est bonne.

Ne pas pénaliser l'absence des réponses fausses, qui ne servent à rien

1.2 : Question de calcul

Il n'y a qu'une réponse attendue à une question de calcul. De même que précédemment, pour savoir si une réponse est juste, on la compare à une réponse juste fournie à la création de la question. Si la réponse fournie est égale à la réponse attendue, le candidat reçoit un point. Sinon, il ne reçoit rien.

Q1.2 : Dans le package *pobj*, proposez l'implantation des questions de calcul dans une classe *QuestionCalcul*. Ecrivez notamment la méthode d'évaluation spécifique de ces questions.

```
package pobj;
public class QuestionCalcul implements Question {
	Reponse bonneReponse;
	public QuestionCalcul(Reponse bonneReponse) {
		this.bonneReponse = bonneReponse;
	}
	public int evalQuestion(Reponse r) {
		if ((r==null) || !(r.equals(bonneReponse)) return 0;
		return 1;
	}
	@Override
	public String toString() { //non demandé
		return "QuestionCalcul [bonneReponse=" + bonneReponse + "]";
	}
}
```

-25% pour l'oubli du cas null (pas spécifié dans l'énoncé)

1.3 : Question piège

Une question piège est une question (à choix multiples ou de calcul) pour laquelle le nombre de points reçus (positifs ou négatifs) est triplé.

Q1.3.1 : Quel *Design Pattern* pouvez-vous utiliser pour implanter les questions pièges ?

Decorateur : 0 ou 100%

Q1.3.2 : Dans le package *pobj*, proposez l'implantation des questions pièges dans une classe *QuestionPiege*. Ecrivez la méthode d'évaluation spécifique de ces questions.

```
package pobj;
public class QuestionPiege implements Question {
	Question q;
	public QuestionPiege(Question q) {
		this.q = q;
	}
	public int evalQuestion(Reponse r) {
		return q.evalQuestion(r)*3;
```

```
	}
	@Override
	public String toString() {//non demandé
		return "QuestionPiege [q=" + q + "]";
	}
}
```

50% pour la bonne structure (implements + attribut)

50% pour la surcharge/délégation de evalQuestion() (sans oublier le *3 ;))

Beaucoup ont bon à la question tout en s'étant trompé de pattern...

2 : Exercices

Un exercice peut être constitué de questions et d'exercices plus simples.

Q2.1 : Quel *Design Pattern* reconnaissez vous dans l'implantation d'un exercice ?

Composite : 0 ou 100%

Q2.2 : Dans le package *pobj*, fournissez une interface *IExercice* qui définit une méthode *int evalExercice(Iterator<Reponse> reponses).* Cette méthode reçoit en argument un itérateur qui permet de parcourir les réponses aux questions que comporte l'exercice (y compris celles des sous-exercices) et retourne le total du nombre de points obtenus pour un exercice.

```
package pobj;
import java.util.Iterator;
public interface IExercice {
	public int evalExercice(Iterator<Reponse> reponses);
}
```

0 ou 100%

attention à l'oubli du import !! (vous mettez 100 quand même, mais malus = 50% min). Il suffit de quelques autres oublis d'import ailleurs et ça fait 100% de malus.

Q2.3 : Modifiez l'interface *Question* pour qu'elle soit considérée comme un exercice élémentaire.

```
package pobj;
import java.util.Iterator;
public interface Question extends IExercice {
	public int evalQuestion(Reponse r);
}
```

Il est aussi possible de faire de Question une classe abstraite qui contient

```
	public int evalExercice(Iterator<Reponse> reponses) {
		return evalQuestion(reponses.next());
	}
```

-25% pour confusion entre extends et implements

Q2.4 : Donnez une implémentation de la classe *Exercice*. On donnera un moyen d'ajouter de

nouvelles questions ou sous-exercices et on proposera une implémentation de la méthode *int evalExercice(Iterator<Reponse> reponses).*

```
package pobj;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;


public class Exercice implements IExercice {

	protected List<IExercice> exercices = new ArrayList<IExercice>();

	public boolean add(IExercice e) {

		return exercices.add(e);

	}

	public int evalExercice(Iterator<Reponse> reponses) {

		int total = 0;

		for(IExercice exercice : exercices) {

			total += exercice.evalExercice(reponses) ;

		}

		return total;

	}

}
```

Bien vérifier qu'on arrête proprement la récursion dans le composite. Si ce n'est pas le cas, -50%.

50% pour la structure, 50% pour evalExercice qui marche.

Q2.5 : Quelles modifications faut-il apporter aux classes *QuestionChoixMultiples* et *QuestionCalcul,* que l'on va trouver en tant que questions élémentaires des exercices pour qu'elles puissent être évaluées lors du parcours des réponses ? Réalisez ces modifications.

Si à la Q2.3 ils ont fait une classe abstraite, il faut juste ajouter le extends.

S'ils ont gardé une interface, il faut ajouter le implements et la méthode :

```
	public int evalExercice(Iterator<Reponse> reponses) {

		return evalQuestion(reponses.next());

	}
```

25% pour le implements
75% pour la méthode ajoutée

ou bien

100% pour le extends si solution classe abstraite.

Un malin m'a signalé (avec raison) qu'il fallait aussi le faire pour QuestionPiege... ;)

3 : Epreuves

Nous allons maintenant réaliser des implémentations concrètes de l'interface *IEpreuve*. Il y a deux types d'épreuves : les QCM (classe *QCM*) et les exercices. Pour commencer, nous créons une classe abstraite *AbstractEpreuve* qui implémente la méthode *calcNbPoints(Copie c)* de *IEpreuve*.

```
package pobj;
import java.util.Iterator;

public abstract class AbstractEpreuve implements IEpreuve, Iterable<Question>{

	public int calcNbPoints(Copie copie){
		int nbPoints = 0;
		Iterator<Reponse> it = copie.iterator();
		for (Question quest : this){
			nbPoints += quest.evalQuestion(it.next());
		}
		return nbPoints;
	}
}
```

3.1 : Questionnaire à choix multiples (QCM)

Un QCM est une épreuve constituée d'une liste de questions à choix multiples. Evaluer le questionnaire revient à faire la somme des points obtenus à chaque question.

Q3.1 : Dans le package *pobj*, écrivez la classe *QCM.* Vous donnerez un moyen d'ajouter des questions à choix multiples pour pouvoir remplir un questionnaire.

```
package pobj;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class QCM extends AbstractEpreuve {
   List<Question> questions = new ArrayList<Question>() ; //on ne sanctionne pas ArrayList

   public boolean add(QuestionChoixMultiple q) {
      return questions.add(q);
   }

   @Override
   public Iterator<Question> iterator() {
      return questions.iterator();
   }
}
```

-50% pour l'oubli du extends

-50% pour l'oubli de l'iterateur

3.2 : Epreuve composée d'exercices

Pour pouvoir évaluer une copie contenant les réponses à des exercices, il faut créer un type d'épreuve adéquat, que l'on appelle *EpreuveExercice* et qui contient au moins un *Exercice*.

Q3.2 : Proposez une implémentation de la classe *EpreuveExercice*. Si vous utilisez un pattern

particulier, indiquez-le.

Adapter (pas bien réalisé ci-dessous...)

```
package pobj;
import java.util.Iterator;
public class EpreuveExercice extends Exercice implements IEpreuve {
	private String name;
	public EpreuveExercice(String name) {
		this.name = name;
	}
	public int calcNbPoints(Copie c) {
		return evalExercice(c.iterator());
	}
	@Override
	public String toString() { //non demandé
		String s = "Epreuve "+ name +" :\n";
		for(IExercice ex : exercices) {
			s += ex.toString() + "\n";
		}
		return s;
	}
 }
```

Bonus : +25% si adaptateur mentionné

-25% si mentionne un pattern qui n'est pas celui utilisé.

Il y a des solutions sans étendre AbstractEpreuve (faire une classe qui contient une liste d'IExercice, renvoyer la somme des evalExercice() de chaque...) => j'ai mis les points

Rions un peu : il y en a un (bon, par ailleurs), qui m'a proposé un Proxy « qui vérifie qu'il y a bien un exercice avant de déclencher le calcul »... ;)

4 : Evaluateur

L'évaluateur est l'objet qui se charge d'évaluer un paquet de copies correspondant à une épreuve. Il mémorise le nombre de points obtenus par chaque copie (un entier) en l'associant au nom de l'étudiant. Il ramène ensuite le nombre de points de chaque étudiant à une note (qui est un double) en faisant en sorte que la moyenne des notes soit égale à 10. Pour cela, il calcule le nombre moyen de points sur l'ensemble des copies et il mutiplie tous les nombres de points par 10 divisé par ce nombre. Enfin, l'évaluateur permet de consulter les notes, sachant qu'on consulte généralement l'ensemble des notes par ordre alphabétique des copies concernées.

Q4.1 : Dans une classe *Evaluateur* du package *pobj*, écrivez le code de la méthode *evaluer()* --dont vous préciserez les paramètres-- qui reçoit une épreuve et une liste de copies et se charge de stocker les nombres de points associés au nom de l'étudiant.

```
package pobj;
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;


public class Evaluateur {
	Map<Copie,Integer> points = new HashMap<Copie,Integer>();
	Map<Copie,Integer> notes = new HashMap<Copie,Double>();// peut être envoyée en valeur de retour de noter()
	public void evaluer(EpreuveExercice epreuve, List<Copie> copies) {
		for(Copie copie : copies) {
			points.put(copie, epreuve.calcNbPoints(copie));
		}
	}
}
```

On peut aussi utiliser les noms comme clef, plutôt que les copies...

Q4.2 : Dans la classe *Evaluateur*, ajoutez le code de la méthode *getMoyenne()* --dont vous préciserez les paramètres-- qui renvoie le nombre moyen de points sur l'ensemble des copies.

```
	public int getMoyenne() {
		int moy = 0;
		for(Integer note : notes.values()) {
			moy += note;
		}
		return moy / notes.size();
	}
```

Diverses syntaxes possibles, ne pas pénaliser les syntaxes lourdes, ils se pénalisent tout seuls en ayant plus à écrire...

-50% pour les fautes de prog.

Q4.3 : Dans la classe *Evaluateur*, ajoutez la méthode *noter()* --dont vous préciserez les paramètres-- qui se charge de stocker les notes associées au nom de l'étudiant dans une autre collection.

```
	public void noter() {
		int moy = getMoyenne();
		for(Copie copie : points.keySet()) {
			notes.put(copie, points.get(copie)*10/moy);
		}
	}
```

Q4.4 : Dans la classe *Evaluateur*, ajoutez une méthode *afficher()* qui envoie sur la sortie standard la liste des notes triées par ordre alphabétique.

```
public void afficher() {
        List<Copie> copies = new ArrayList<Copie>();
        for(Copie copie : notes.keySet()) {
                copies.add(copie);
        }
        Collections.sort(copies);
        for(Copie copie : copies) {
                System.out.println(notes.get(copie));
        }
}
```

Nombreuses variantes possibles, plus ou moins élégantes...