Examen juin 2006

3 heures

documents de cours, TD et TP autorisés

livres, portables et calculatrices interdits

▷ Les deux exercices sont indépendants.

▷ L'ordre des exercices ne correspond pas nécessairement à un niveau de difficulté croissant.

▷ Soyez précis dans vos réponses, notamment dans le respect **strict** du cahier des charges imposé par le sujet. Des informations utiles sont présentes dans tout le texte d'un exercice.

▷ Vous pouvez ajouter toute interface, classe, méthode qui vous paraît utile même si elle n'est pas explicitement réclamée dans le sujet.

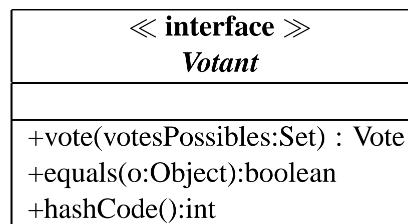
Exercice 1 : Scrutins et Votes

On s'intéresse à la modélisation de scrutins.

Un scrutin est caractérisé par les votants qui y participent et les votes que ceux-ci peuvent exprimer. Ces informations (liste des votants et "votes possibles") sont fixés à la création d'un scrutin. La manière de déterminer le résultat dépend du mode du scrutin.

Un votant ne peut voter qu'une seule fois par scrutin et aucun vote n'est possible une fois que le scrutin a été déclaré *clos*. Evidemment seul les votants inscrits au scrutin peuvent voter. Le choix d'un votant se fait via sa méthode `vote`. Si ce choix est un vote autre que ceux de la liste des votes possibles, le vote sera déclaré *nul* lors du dépouillement (mais il est autorisé).

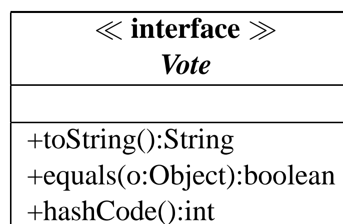
Les votants seront du type :



où la méthode `vote` correspond à la méthode de choix de son vote par le votant.

Votes Il existe différentes votes : les votes OUI ou NON pour un référendum, votes nominatifs pour une élection, etc.

On impose l'interface `Vote` :



et on souhaite disposer de (au moins) 4 classes de vote : le vote OUI, le vote NON, le vote BLANC et le vote nominatif caractérisé par le nom du candidat (ou de la liste). Les 3 premières classes doivent satisfaire le design pattern singleton.

Q 1 . Donnez un code JAVA pour la classe `VoteOUI`

Q 2 . Donnez un code JAVA pour la classe `VoteNominatif`

Scrutin Pour déterminer le vote vainqueur, le type `Scrutin` dispose de la méthode `getVainqueur()` dont le résultat est le vote qui, selon les modalités du scrutin, a remporté le scrutin. Ce résultat vaut `null` si il n'y a aucun vainqueur. Une exception `VoteNonClosException` est levée si cette méthode est invoquée avant la cloture du scrutin. On suppose (à tort on fera l'hypothèse malgré tout) qu'il ne peut y avoir d'égalité à un scrutin.

Dans cet exercice on décide de ne s'intéresser qu'à 2 modes de scrutin différenciés par leur manière de calculer le vote vainqueur :

- le scrutin (type `ScrutinMajoritaire`) à la majorité des suffrages exprimés : est déclaré vainqueur le vote qui aura obtenu plus de 50% des votes autres que les votes blancs ou nul. Il peut bien sûr ni avoir aucun vainqueur.
- le scrutin à majorité relative (type `ScrutinRelatif`) des votants avec au moins 15% des inscrits : est déclaré vainqueur le vote qui a reçu le plus de suffrages pour peu qu'au moins 15% des inscrits au scrutin aient exprimé ce vote.

Evidemment on souhaite avoir la possibilité d'ajouter de nouveaux types de scrutin.

Q 3 . Après avoir lu attentivement la suite de l'exercice, donnez le diagramme UML liant les types `Scrutin`, `ScrutinMajoritaire` et `ScrutinRelatif` qui vous paraît le mieux adapté, en détaillant pour chaque classe les attributs, constructeurs et méthodes définis ou redéfinis.

Pour rassurer les votants quant à la confidentialité de leur vote, on décide de publier le code de la méthode `vote` de la classe `Scrutin`. C'est cette méthode que doit invoquer un `Votant` pour participer au scrutin. Le code de la méthode `enregistreVote` de `Scrutin` est également donné :

```
public final void vote(Votant votant) throws VoteImpossibleException {
   if (this.peutVoter(votant)) {
      Vote vote = votant.vote(this.lesVotesPossibles);
      this.enregistreVote(vote);
      this.aVote(votant);
   }
   else {
      throw new VoteImpossibleException ();
   }
}
private final void enregistreVote(Vote vote) {
   this.lesVotes.add(vote);
}
```

Ce code rassure le votant qui peut ainsi vérifier que le vote et le votant ne sont pas mémorisés ensemble par le scrutin.

Q 4 . Quel est l'intérêt et l'importance du qualificatif `final` dans la méthode `vote` (soyez précis et concis dans votre réponse).

Le type `Scrutin` doit également proposer les méthodes :

▷ `tauxParticipation():float` qui retourne à tout moment le taux de participation au scrutin, c'est-à-dire le pourcentage de votants par rapport au nombre d'inscrits au scrutin.

▷ `depouille()` qui lorsqu'elle est appelée :

- clôt le scrutin,
- considère tous les votes enregistrés et comptabilise pour chaque vote son score

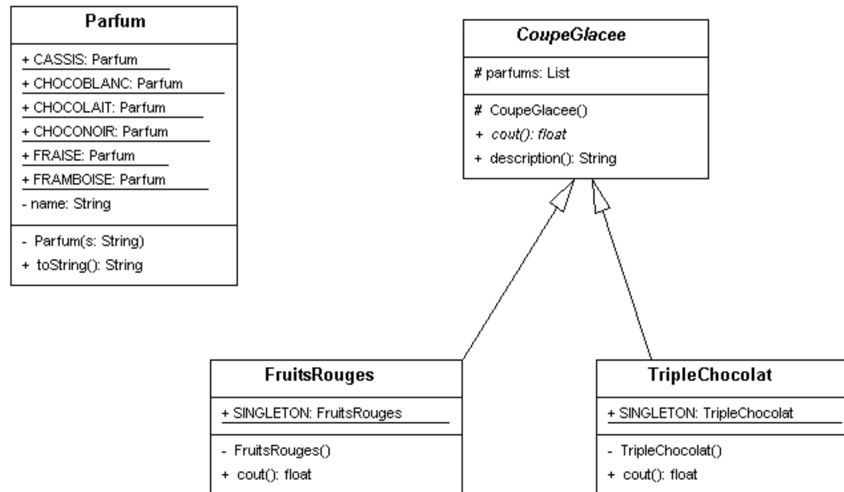
▷ `afficheResultat()` qui affiche chacun des votes avec son score (nombre de voix obtenu), y compris les votes nuls.

Q 5 . Donnez tout le code JAVA des classes `Scrutin`, `ScrutinMajoritaire` et `ScrutinRelatif`.

Exercice 2 : Glaces

Dans le cadre de l'informatisation d'un glacier, il est nécessaire de modéliser les différentes coupes glacées que celui-ci peut proposer. Cela permet notamment la gestion des commandes et factures des clients.

Dans une première version, seules des coupes glacées composées de plusieurs boules de différents parfums sont gérées. Dans cette première version, on trouve le diagramme de classes suivant, auxquels on peut imaginer ajouter d'autres coupes glacées chacune correspondant à un singleton :



Une partie du code associé est fourni en annexe.

La classe `Commande` du logiciel de gestion du glacier comporte la méthode suivante qui permet d'afficher une facture sur le flux de sortie voulu et déterminé par le `PrintWriter` passé en paramètre :

```
public void publieFacture(PrintWriter writer) {
   float total = 0;
   for (Iterator it = this.lesCoupes.iterator(); it.hasNext();) {
     CoupeGlacee coupe =  (CoupeGlacee) it.next();
     writer.println(coupe.description()+" + "+coupe.cout());
     total = total + coupe.cout();
   }
   writer.println("   TOTAL : "+total);
}
```

Pour une commande composée de deux coupes "fruits rouges" et d'une coupe "triple chocolat", la facture ainsi publiée ressemble à :

```
Coupe fraise framboise cassis + 5,50
Coupe fraise framboise cassis + 5,50
Coupe chocolat blanc chocolat au lait chocolat noir +6,00
   TOTAL : 17
```

Evolution du logiciel (décoration). Suite à de nombreuses demandes de ces clients, le glacier décide qu'il sera possible pour chaque client de personnaliser ses coupes par l'ajout d'un ou plusieurs agréments (ou "topping") tels que crème chantilly, coulis chocolat, coulis de fraise etc. Ceci évidemmment contre un modique surcoût dépendant de l'ingrédient. La description de la coupe glacée consommée par le client doit bien entendu se trouver modifier en conséquence.

Voici quelques toppings possibles, leur surcoût et la description associée :

topping	*surcoût*	*description*
chantilly	0,50	"chantilly"
sauce chocolat	0,70	"et sa délicieuse sauce chocolat"
coulis fraise	1	"au coulis de fraises fraiches"

Evidemment il est possible pour les gourmands de cumuler plusieurs toppings sur une même coupe glacée. Dans ce cas les surcoûts se cumulent et les descriptions s'enrichissent en conséquence.
Il est évidemment nécessaire de prendre en compte cette évolution au niveau des commandes et factures.
Voici donc un exemple de facture que l'on doit obtenir dans la seconde version :

```
Coupe fraise framboise cassis chantilly + 6,00
Coupe fraise framboise cassis chantilly au coulis de fraises fraiches + 7,00
Coupe chocolat blanc chocolat au lait chocolat noir  +6,00
Coupe chocolat blanc chocolat au lait chocolat noir
       et sa délicieuse sauce chocolat chantilly  +7,50
   TOTAL :   26,50
```

Q 1 . En utilisant le design pattern décorator, proposez une solution élégante à l'ajout des classes permettant la gestion de coupe glacées avec "toppings" pour la seconde version de l'application de gestion du glacier.

Dans un premier temps vous **compléterez** le diagramme UML donné à la figure précédente, puis vous **donnerez tout le code** nécessaire à l'intégration du topping "Chantilly" dans l'application.

Votre solution doit permettre l'ajout de nouveaux toppings sans modification du code (respect du principe *ouvert-fermé*), ce qui n'est par exemple pas possible pour l'ajout de nouveaux parfums avec la conception actuelle. Vous **justifierez** votre respect de ce principe en indiquant brièvement mais clairement ce qu'il faut faire pour ajouter un nouveau topping (*noisettes caramélisées* par exemple) sans écrire de code.

Q 2 . Faut il maintenant modifier la méthode `publieFacture`, si oui comment, si non pourquoi ?

Q 3 . Donnez la ligne de code permettant de créer "*une Coupe chocolat blanc chocolat au lait chocolat noir et sa délicieuse sauce chocolat chantilly*".

Encore une évolution (refactoring). Le succès toujours croissant de notre glacier l'amène à compléter sa carte en proposant également des boissons : des chocolats chauds et des cafés. Les cafés sont de plusieurs variétés possibles : *robusta*, *arabica*, etc.
Ces boissons ont également un coût et une description repris dans le tableau ci-dessous :

boissons	*coût*	*description*
café	1,50	"café" + *nom de la variété de café*
chocolat chaud	2,00	"chocolat chaud"

Le glacier pense tout de suite qu'il sera possible de compléter les cafés et chocolat avec de la chantilly, cela a la même incidence que précédemment pour les glaces.

Q 4 . Indiquez les modifications de conception qu'il faut apporter au modèle pour permettre de gérer à la fois les coupes glacées et les boissons de manière uniforme. C'est-à-dire que, par exemple, les commandes doivent aussi bien pouvoir contenir des coupes glacées que des boissons et qu'une simple et petite modification de code doit permettre de réutiliser la méthode `publieFacture`.

Vous proposerez un nouveau diagramme UML pour gérer les glaces et les boissons. Il n'est pas question ici de respecter le principe ouvert-fermé par rapport à la première version, mais d'y apporter le minimum de modifications nécessaires.

Vous indiquerez également les modifications à apporter à la publication de factures.

Annexe

Sources des classes `CoupeGlacee` et `FruitsRouges`.
Le code de la classe `TripleChocolat` est similaire à la classe `FruitsRouges` aux parfums près.

```
package glace;
import java.util.*;
public abstract class CoupeGlacee {
   protected List parfums;
   protected CoupeGlacee()  {}
   public String description() {
      StringBuffer sb = new StringBuffer("");
      for(Iterator it = parfums.iterator();it.hasNext();) {
         sb.append(it.next().toString());
         sb.append(" ");
      }
      return sb.toString();
   }
   public abstract float cout();
}
----------------------------------------------------------------------
package glace;
public class FruitsRouges extends CoupeGlacee {
   public static final FruitsRouges SINGLETON = new FruitsRouges;
   private FruitsRouges() {
      this.parfums.add(Parfum.FRAISE);
      this.parfums.add(Parfum.FRAMBOISE);
      this.parfums.add(Parfum.CASSIS);
   }
   public float cout() {
      return 6;
   }
}
```