Programmation Objet

Examen

3 heures - documents écrits autorisés
14 septembre 2002

Exercice 1 : Musique maestro

Un `Orchestre` est composé d'`Instruments`. Parmi les instruments on distingue différentes familles : les `Clavier`, dont le `Piano` ou le `Synthe` sont des exemples, les `Corde`, comme le `Violon` ou la `Guitare-Electrique`, les `Vent`, les `Percussion`. Les instruments à vent se décompose eux mêmes en `Cuivre` et en `Bois`.
Parmi ces instruments certains sont `Electrique`, c'est le cas du `Synthe` et de la `GuitareElectrique`.
S'ils ont des comportements communs : on peut tous les jouer ou les accorder, chaque famille ou instrument d'une famille a ses propres caractéristiques telles que le nombre de cordes et la tonalité de celles-ci, le nombre de touches, etc.

Q 1 . Chacun des mots apparaissant en police `courrier` dans le texte précédent correspond à une entité : classe, classe abstraite ou interface, représentez en utilisant UML les dépendances entre ces entités (composition, agrégation, association, extension, implémentation, etc.) en faisant les choix de conception qui vous paraissent les mieux adaptés. Il est **inutile** de décrire les attributs ou méthodes, seuls les noms des entités et leurs dépendances sont demandés.

Exercice 2 : Epicerie automatique

On souhaite modéliser dans cet exercice une "épicerie automatique" du type distributeur de boissons, sandwiches ou autres.
Les machines qui nous intéresseront ici sont constituées de plusieurs éléments :

▷ un stock contenant les différents articles repérés chacun par une référence,

▷ un monnayeur, à pièces, qui rend la monnaie ou prévient quand cela ne lui est plus possible,

▷ un clavier qui permet au client de sélectionner la référence de l'article qu'il désire obtenir,

▷ une zone d'affichage permettant d'afficher le prix de l'article choisi par exemple.

Toutes les entités utiles à la modélisation de ces épiceries seront placées dans un paquetage `epicerie`. On y trouve notamment la classe `Article` définie ainsi (les prix sont donnés en centimes) :

Article
- nom:String - prix:int
+ Article(nom:String, prix:int) + getPrix():int

Un extrait du fichier définissant la classe `Epicerie` est donné en annexe. Nous allons nous intéresser à la programmation de ses différents composants.

Q 1 . Intéressons nous d'abord au stock. Celui-ci est géré par une instance de la classe `Stock`. Cette classe permet d'associer à une référence, un article et sa quantité en stock. Cette information est stockée dans une table de hachage dont la clef est la référence et la valeur associée l'article et sa quantité.

Q 1.1. Les références sont des objets du type défini par l'interface `Reference`. Sachant que ces objets sont utilisés comme clef dans une table de hachage, donnez pour cette interface la déclaration minimale permettant d'assurer une bonne gestion de telles clefs.

Q 1.2. Faites les déclarations et définitions nécessaires à la gestion des valeurs de la table de hachage sachant que l'on doit pouvoir augmenter ou diminer la quantité d'un article.

Q 1.3. Définissez la classe `Stock` sachant que l'on veut pouvoir :

▷ créer une nouvelle référence pour un article donné,
▷ augmenter le stock pour une référence donnée,
▷ connaître le prix d'une référence,
▷ tester si une référence est disponible en stock,
▷ délivrer un article à partir de sa référence.

Q 2 . Aucun détail n'est fourni sur le fonctionnement de l'afficheur et du clavier. Les seules informations que l'on ait sont que l'on peut invoquer sur l'afficheur la méthode :

```
public void affiche(String msg)
```

qui génère l'affichage du message `msg`, et sur le clavier la méthode

```
public Reference saisieReference()
```

qui est bloquante et retourne la référence saisie par l'utilisateur.
Quelle solution de conception proposez-vous pour les entités `Afficheur` et `Clavier` afin que l'on puisse poursuivre le développement sans plus d'information ? Donnez les définitions de ces entités.

Q 3 . Avant de nous intéresser au monnayeur, examinons la modélisation des pièces acceptées par celui-ci. Celle-ci est réalisée dans le fichier `Piece.java` dont le source est donné en annexe. Il est notamment **important** de noter le comportement contre-intuitif de la méthode `compareTo`.

Q 3.1. L'attribut `valeur` est `public`, pourtant la règle d'or est a priori de déclarer les attributs en `protected` ou `private`. Qu'est-ce qui justifie cette déclaration ici ?

Q 3.2. Quel commentaire vous inspire le couple constructeur, méthode statique `creePiece` ? A quel design pattern cela vous fait il penser ?

Q 4 . Le monnayeur est modélisé par la classe `Monnayeur`. Une partie de son code est fournie en annexe. Un extrait de la `javadoc` de la classe `TreeMap` est également fourni.

Q 4.1. Donnez le code de la classe définissant l'exception mentionnée.

Q 4.2. Donnez un code pour la méthode `argentEnCaisse()`

Q 4.3. Donnez un code pour la méthode `rendMonnaie()`, sachant que le monnayeur cherche à rendre le moins de pièces possible, ce qui revient à dire qu'il rend en priorité les pièces de plus forte valeur possible.

Q 4.4. Donnez un code pour la méthode `attendSous()`. Les règles de fonctionnement sont :

- ▷ d'attendre l'introduction de pièce par l'utilisateur tant que le montant introduit est inférieur au prix, en modifiant les quantités de pièces en caisse pour chaque nouvelle pièce
- ▷ de rendre la monnaie excédentaire si le montant introduit devient supérieur au prix,
- ▷ de rendre le montant introduit en cas d'annulation et de lever une exception `PaiementAnnuleException`,
- ▷ de rendre le montant introduit lorsqu'il est impossible de rendre la monnaie, dans ce cas également une exception `PaiementAnnuleException` est levée.

Q 5 . Donnez un code pour la méthode `run()` de la classe `Epicerie`. Son fonctionnement est sans fin (sauf en cas d'extinction bien sûr) et se déroule ainsi :

1. affichage d'un message de demande de saisie de référence,
2. attente de la saisie,
3. affichage du prix de l'article choisi si disponible,
4. attente du paiement,
5. si pas d'annulation du paiement, l'article est délivré.

Annexes

Extrait du fichier `Epicerie.java`

```
/**
 * Epicerie.java
 */
package epicerie;
public class Epicerie {
    public Epicerie (){
        init();
    }
    protected Stock stock;
    protected Clavier monClavier;
    protected Afficheur monAfficheur;
    protected Monnayeur monnayeur;

    protected void init() {
       // ... initialise stock, monClavier, monAfficheur et monnayeur
       // ... code supposé fourni
```

```
    }
    public void on() { run(); }
    public void off() { System.exit(0); }

    public void run() {
        // ... à compléter
    }
}// Epicerie
```

Le fichier `Piece.java`

```
/**
 * Piece.java : les pièces d'un monnayeur avec une valeur en centimes
 */
package epicerie;
class Piece implements Comparable {
    public final int valeur;
    private Piece (int valeur){
        this.valeur = valeur;
    }
    public static Piece creePiece(int valeur) {
        if ((valeur == 1) || (valeur == 2) || (valeur == 5) ||
            (valeur == 10) || (valeur == 20) || (valeur == 50) ||
            (valeur == 100) || (valeur == 200)) {
            return new Piece(valeur);
        }
        else return null; // la gestion par une exception eut été préférable
    }
    public boolean equals(Object o) { return valeur == ((Piece) o).valeur; }
    public int hashCode() { return new Integer(valeur).hashCode(); }
    /** définit une relation d'ordre : ordre inverse sur les valeurs.
     * Une pièce est donc supposée d'autant plus petite que sa valeur est grande.
     * -1 si this.valeur > o.valeur, 0 si égales, +1 sinon
     */
    public int compareTo(Object o) {
        Piece autre = (Piece) o;
        return new Integer(autre.valeur).compareTo(new Integer(this.valeur));
    }
}// Piece
```

La classe `Monnayeur`

```
/**
 * Monnayeur.java (on suppose les pièces fournies valides)
 */
package epicerie;
import java.util.*;
public class Monnayeur {
    public Monnayeur (){
        initMonnaie();
    }
    /** (!!!à gérer!!!) true si il est possible de rendre la monnaie,
     * devient faux dès qu'une tentative de rendre la monnaie a échoué
     */
    private boolean peutRendreMonnaie = true;
    /** table de hachage dont les clefs (instance de Piece) sont triées et qui associe
     * pour chaque pièce le nombre de pièces de cette valeur présentes dans le
     * monnayeur */
    private TreeMap monnaie;

    protected void initMonnaie() {
        monnaie = new TreeMap();
        // ... initialisation de la monnaie ici
    }

    public void modifierMonnaie(int valeur, int nbPieces) {
        int nouveau = ((Integer) monnaie.get(Piece.creePiece(valeur))).intValue()+nbPieces;
        monnaie.put(Piece.creePiece(valeur),new Integer(nouveau));
    }
```

```
    /** retourne le montant d'argent contenu dans le monnayeur
     * @return le montant d'argent contenu dans le monnayeur
     */
    public int argentEnCaisse() {
        // ... à compléter
    }
    /** attend que le client fournisse les pièces suffisantes pour payer le prix
     * @exception PaiementAnnuleException si le paiement est annulé ou
     * impossible (dans le cas où le monnayeur ne peut plus rendre la
     * monnaie)
     */
    public void attendSous(int prix) throws PaiementAnnuleException {
        // ... à compléter
    }
    /** rend la monnaie en utilisant le moins de pièces  possible
     * @param aRendre le montant de la monnaie à rendre
     * @return la somme qu'il reste à rendre (et donc 0 si il est
     * possible de rendre toute la monnaie)
    */
    protected int rendMonnaie(int aRendre) {
        // ... à compléter
    }
    /** méthode bloquante tant que l'utilisateur n'a pas introduit une
     * pièce dans le monnayeur,
     * @return la piece introduite
     * @exception PaiementAnnuleException si l'opération est annulée par l'utilisateur
     */
    protected Piece attendPiece() throws PaiementAnnuleException {
        // ... le code qui convient est supposé être ici...
    }
}// Monnayeur
```

Extrait (adapté) de la `javadoc` de la classe `TreeMap`

Le fonctionnement global est le même que pour la classe `HashMap`. Seuls les détails utiles pour l'exercice et qui distinguent une instance de `TreeMap` d'une instance de `HashMap` sont donnés ici.

```
java.util
Class TreeMap

java.lang.Object
  |
  +--java.util.AbstractMap
        |
        +--java.util.TreeMap
All Implemented Interfaces:
        Cloneable, Map, Serializable, SortedMap
public class TreeMap
    extends AbstractMap
    implements SortedMap, Cloneable, Serializable

Red-Black tree based implementation of the SortedMap interface. This
class guarantees that the map will be in ascending key order, sorted
according to the natural order for the key's class (defined by
compareTo(Object o)), or by the comparator provided at creation time,
depending on which constructor is used.

methods :

  public boolean containsKey(Object key)
  public Collection values()
  public Object get(Object key)
  public Object put(Object key, Object value)
  public Set keySet()
    Returns a Set view of the keys contained in this map. The set's
    iterator will return the keys in ascending order (according to compareTo).
```