UE Programmation Orientée Objet

Devoir Surveillé

Lundi 19 mai 2014 – 16h30-18h30
Copie des diapositives de cours autorisée
Une feuille recto-verso de notes personnelles
Dictionnaire de langue (papier ou électronique "dédié") autorisé
Tout autre document interdit

Les exercices sont indépendants. Leur ordre ne préjuge pas de leur difficulté. Il est conseillé de lire entièrement un exercice avant de le traiter.
La javadoc et les méthodes de test ne sont à fournir que si elles sont demandées.

Exercice 1 : Clients et fournisseurs

On va s'intéresser à des **clients** qui proposent des **commandes** à des **fournisseurs**. Les fournisseurs choisissent les commandes qu'ils traitent selon leur propre **stratégie de choix**. Chaque commande a un **identifiant** unique.

Les identifiants et commandes sont modélisés par les classes suivantes du paquetage `marche`.

marche::Identifiant
...
- Identifiant(s : String)
+ genereIdentifiant(): Identifiant
+ hashCode() : int
+ equals(o :Object) : boolean

marche::Commande
- prix : float
- id : Identifiant
+ Commande(prix : float)
+ getPrix() : float
+ getIdentifiant() : Identifiant
+ hashCode() : int
+ equals(o :Object) : boolean

L'identifiant d'une commande est mis à jour lors de sa création. C'est la méthode statique `genereIdentifiant` de `Identifiant` qui fournit à chaque appel un nouvel identifiant garanti unique.

Q 1 . Donnez le code java de la classe `Commande`.

Q 2 . Les clients sont modélisés par des instances de la classe `Client`. Complétez le code de la classe `Client` ci-dessous (ne reprenez pas sur votre copie la javadoc).

```
package marche;
import java.util.*;
public class Client {
  /** collecte les commandes en attente pour ce client en associant une
  *  commande à son identifiant */
  private Map<Identifiant, Commande> commandesEnAttente;
  /** collecte les commandes qui ont été traitees pour ce client */
  private Map<Identifiant, Commande> commandesTraitees;
  /** associe l'identifiant d'une commande au fournisseur qui l'a traitée */
  private Map<Identifiant, Fournisseur> fournisseurCommandes;

  /** crée un Cient, initialement il n'a aucune commande en attente ni traitées */
  public Client() { ... }

  /** fournit les commandes en attente pour ce client
   * @return la liste des commandes en attente pour ce client */
  public List<Commande> getCommandesEnAttente() { ... }

  /** ajoute une nouvelle commande en attente pour ce client
   * @param commande la nouvelle commande */
  public void nouvelleCommande(Commande commande) { ... }
```

CC BY NC SA

```java
  /** indique qu'une commmande en attente est traitée par un fournisseur.
   * Cette commande est retirée des commandes et passe dans les commandes traitées.
   * @param commande la commande traitée
   * @param fournisseur le fournisseur qui traite la commande */
  public void commandeTraiteePar(Commande commande, Fournisseur fournisseur) { ... }

  /** fournit la liste des commandes traitées pour un fournisseur donné
   * @param fournisseur le fournisseur concerné
   * @return fournit la liste des commandes traitées par fournisseur */
  public List<Commande> commandesTraiteesParFournisseur(Fournisseur fournisseur) { ... }
}
```

Les fournisseurs sont des instances de la classe `Fournisseur` :

marche::Fournisseur
- chiffreAffaires : float - commandesTraitees : List<Commande> - strategie : StrategieChoix
+ Fournisseur(strategie : StrategieChoix) + produitCommande(Commande commande) + maintenance() + getChiffreAffaires() : float + augmenteChiffreAffaires(ajout : float) + choisitEtTraiteCommande(client : Client)

La méthode `choisitEtTraiteCommande` sélectionne l'une des commandes encore en attente du client passé en paramètre. Pour effectuer son choix, le fournisseur s'appuie sur la stratégie de choix fournie à sa création. Une *stratégie de choix* est définie par l'interface ci-dessous. Si aucune commande n'a pu être choisie, le fournisseur entre en maintenance (méthode `maintenance`). Lorsqu'une commande est choisie, elle est produite et le chiffre d'affaires est augmenté du prix de la commande. Les données relatives aux commandes traitées sont mises à jour pour le fournisseur et le client.

```java
package fournisseur;
import java.util.List;
public interface StrategieChoix {
  /** choisit une commande dans la liste selon les critères de la stratégie
   * @param commandes la liste des commandes
   * @return la commande choisie
   * @throws PasDeCommandeException si aucune commande ne peut être choisie
   */
   public Commande choisit(List<Commande> commandes) throws PasDeCommandeException;
}
```

Q 3 . Donnez la javadoc et le code java de la méthode `choisitEtTraiteCommande` de la classe `Fournisseur`.

Q 4 . On distingue deux stratégies de choix chez les fournisseurs :

1. ceux qui maximisent leur chiffre d'affaires et choisissent parmi celles proposées la commande dont le prix est le plus élevé,
2. ceux qui n'acceptent que des commandes dont le prix est supérieur à un minimum fixé à la création de la stratégie. Si plusieurs commandes sont possibles ils choisissent l'une d'entre elles au hasard.

Donnez le code java nécessaire à la mise en place de ces deux stratégies.

Exercice 2 : Gestion d'heures complémentaires

Chaque enseignant à l'université effectue un certain nombre d'heures d'enseignement dans une année. Suivant le statut de l'enseignant, certaines de ces heures peuvent-être considérées comme *complémentaires*. Les heures complémentaires sont payées séparément à l'enseignant. Les volumes horaires sont exprimés en heures entières et le prix d'une heure complémentaire est de 45 euros.

Pour un enseignant, son nom et son nombre total d'heures effectuées dans une année sont fixés à sa création. Ces données peuvent être consultées mais pas modifiées.

D'autre part on veut pour un enseignant pouvoir accéder à son volume d'heures complémentaires (méthode `heuresComplementaires()`) et sur le paiement correspondant à ces heures (méthode `paiement()`).

Il y a trois types d'enseignants :

- **les enseignants universitaires** : seules les heures assurées au delà d'une charge statutaire de 192h sont complémentaires,
- **les intervenants extérieurs** : toutes les heures effectuées sont complémentaires,
- **les étudiants de troisième cycle** qui assurent des enseignements : ce sont des enseignants extérieurs mais ils ne peuvent faire plus de 96h complémentaires.

  De plus, les étudiants, puisqu'ils n'ont pas d'employeur principal, subissent des charges supplémentaires et voient donc le paiement de leurs heures complémentaires diminué de 18%.

Q 1 . Donnez sous la forme de diagrammes UML **détaillés** une proposition de modélisation utilisant l'héritage pour représenter les différents types d'enseignants.

Pour chaque classe vous préciserez les attributs, constructeurs et les méthodes implémentées avec leurs paramètres et valeur de retour.

Q 2 . Donnez le code java de la classe à la racine de votre arbre d'héritage.

Q 3 . Donnez le code java de la classe qui permet de représenter les étudiants de troisième cycle.

Exercice 3 : Taquin

Le *taquin* est ce jeu formé d'un rectangle de n sur p emplacements carrés. Tous ces emplacements sauf un, que nous appellerons "*trou*" par la suite, sont occupés par une tuile.
Une tuile peut glisser verticalement ou horizontalement, à la seule condition qu'elle se trouve à côté du *trou*. On peut considérer que le déplacement d'une telle tuile située à côté du *trou* consiste en fait à déplacer le trou et ainsi à échanger cette tuile avec le *trou*. Il y a donc au maximum quatre directions possibles pour le déplacement du *trou* : vers le haut, le bas, la gauche ou la droite.
Dans la configuration initiale les tuiles forment un *motif particulier*, dans lequel le *trou* occupe la position en "bas à droite". Chaque tuile "porte" donc une partie de ce motif. Les tuiles peuvent donc être numérotées de manière ordonnée en accord avec ce motif particulier, la tuile en haut à gauche du motif reçoit le numéro 0, les tuiles sont ensuite numérotées de manière incrémentale de gauche à droite et de haut en bas (voir Figure 1).

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	T

Figure 1: Numérotation des tuiles pour un taquin 5×4 dans la configuration intiale du *motif particulier* . Le *trou* est symbolisé par T.

Pour le jeu les tuiles sont mélangées, ce qu'on obtient en réalisant une séquence aléatoire de déplacements. Le but du jeu est alors de retrouver le motif particulier initial par une suite de déplacements du trou.

On s'intéresse dans cet exercice à la modélisation de ce jeu. Les types demandés par la suite sont à définir dans un paquetage `taquin`.
Les tuiles sont définies par leur numéro dans la configuration du motif initial. Un objet particulier constant de type `Tuile` appelé `TROU` est défini dans cette classe, il reçoit par convention le numéro -1.

Q 1 . Donnez le diagramme UML détaillé d'une classe `Tuile` correspondant à cette description. Il doit être possible d'accéder au numéro attribué à la tuile lors de sa construction.

La classe `Taquin` représente un jeu de taquin. Une instance de cette classe est définie par la largeur et la hauteur du taquin et est caractérisée par les tuiles qui le composent, rangées dans un tableau. Elle dispose également d'attributs représentant la ligne et la colonne du trou.

Q 2 . Donnez le code java permettant de définir les attributs de la classe `Taquin` ainsi que le code java du constructeur de cette classe qui prend en paramètre la largeur et la hauteur du taquin et qui ordonne les tuiles, et donc les numérote, selon le motif particulier comme décrit précédemment (à la création le taquin est dans la configuration initiale, le trou est situé en bas à droite).

Q 3 . Définissez un type énuméré `DirectionDeplacement` qui définit les quatre directions de déplacement possibles pour le *trou*.

Q 4 . On suppose définie la méthode `deplacement` de la classe `taquin` dont la documentation est :

```
/** Réalise, quand il est possible, le déplacement du trou dans
  *  la direction de deplacement indiquee
  * @param depl la direction du déplacement du trou à réaliser
  * @exception DeplacementImpossibleException si le déplacement
  * n'est pas possible dans la direction demandée.
  */
```

Q 4.1. Donnez le code de la classe `DeplacementImpossibleException` du paquetage `taquin`.

Q 4.2. Définissez une méthode `melanger` dans la classe `Taquin` qui consiste à mélanger le taquin en réalisant 100 déplacements possibles du *trou* dans une direction tirée aléatoirement à chaque déplacement (les tentatives de déplacements impossibles ne sont pas comptabilisées dans les 100). Vous trouverez en annexe un extrait de la documentation de la classe `java.util.Random`.

Annexe : documentation classe `java.util.Random`

Random() Creates a new random number generator.

int nextInt(int n) Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.