

UE Conception Orientée Objet

Devoir Surveillé

durée : 3h – documents écrits autorisés

livres et calculatrice interdits

Dictionnaire de langue (électronique ou non) autorisé

Les exercices sont indépendants.

La clarté des réponses sera prise en compte dans l'évaluation, en particulier pour les diagrammes UML dans lesquels apparaîtront toujours :

- les liens d'héritage/implémentation entre types*
- les noms et types de tous les attributs, ainsi que leur visibilité,*
- les méthodes et constructeurs avec leurs paramètres et leurs types ainsi que les types des valeurs de retour*

Soignez la présentation de ces diagrammes en évitant autant que possible les ratures!

Evidemment vos réponses devront être convaincantes et donc précises!

Exercice 1 : Au restaurant

La classe `Commande` suivante permet de gérer les commandes d'un restaurant :

Commande
numeroCmde : int
numeroTable : int
encaissee : boolean
servie : boolean
contenu : List<Commandable>
+ Commande(numeroTable : int)
+ ajouteElement(element : Commandable)
+ encaisse()
+ estEncaissee() : boolean
+ sert()
+ estServie() : boolean
+ afficheCommande()
+ getPrix() : float

Tous les éléments `Commandable` du restaurant ont une description (une chaîne de caractères) qui correspond à leur nom dans le menu, un prix (un `float`) et une valeur énergétique en calories (un `float`). Ces informations sont fournies à la création.

Parmi les éléments `Commandable` on distingue les éléments de la carte et les menus.

Les éléments de la carte sont :

1. les *boissons* parmi lesquelles on trouve des boissons alcoolisées, caractérisées par leur degré d'alcool et des boissons non alcoolisées divisées en boissons sucrées ou «light»
2. les *plats* dans lesquels on distingue les plats végétariens des plats «carnivores». Ces derniers sont caractérisés par le type de viande : *boeuf*, *porc*, *mouton*, *volaille* (on suppose qu'il n'y a qu'un type de viande par plat).
3. les *desserts* où l'on distingue les desserts à base de fruits des crèmes glacées.

Parmi ces éléments certains sont considérés (à tort ou à raison) comme «diététiques», c'est le cas des boissons light, des plats végétariens et des desserts à base de fruits.

Tous les éléments de carte existent en version XL. Par rapport à la version «normale», la quantité (donc la valeur énergétique) est augmentée de 50% pour un prix augmenté de 25%. La description est la même avec simplement la mention "XL" ajoutée à la fin.

Les menus sont composés d'une boisson, d'un plat et d'un dessert à choisir parmi les éléments de la carte. Le prix d'un menu est obtenu en diminuant de 15% le prix de ses trois éléments. Sa valeur énergétique est naturellement la somme des trois et la description est la concaténation des trois descriptions préfixée de "Menu : ". La version XL des menus n'existe pas, on peut cependant construire un menu à partir d'éléments XL ou non.

Il existe des menus «enfants» dont le prix est diminué de 20% et qui ont en plus un cadeau (d'un type `Cadeau` supposé défini, et gratuit car c'est un vrai cadeau). La boisson de ces menus est forcément sans alcool.

Q 1 . Faites une proposition de modélisation **complète** pour les données présentées ci-dessus.

Vous présenterez votre solution sous la forme d'un ou plusieurs diagrammes UML. Vous pouvez utiliser les 2 pages intérieures de votre copie pour le diagramme UML afin de disposer de suffisamment de place.

Q 2 . La classe `Commande` :

Q 2.1. Donnez le code java de la méthode `encaisse` qui est un modificateur pour l'attribut `encaissee` et lève l'exception `CommandeNonServiceException`, dont vous donnerez le code, si la commande n'est pas servie.

Q 2.2. Donnez le code de la méthode `afficheCommande` qui affiche ligne par ligne chacun des éléments `Commandable` qui la composent avec son prix.

Un exemple d'affichage obtenu est fourni en annexe.

Vous pouvez utiliser `S.o.p` pour `System.out.println`.

Exercice 2 : Contrôle d'accès

(Les types de cet exercice seront définis dans un paquetage `securite` ou un de ses sous-paquetages.)

On s'intéresse à la simulation d'un système de sécurisation d'un bâtiment composé de pièces accessibles par des portes.

Les portes (de type `Porte`) sont équipées par un système de contrôle d'accès par code. Il s'agit d'un code numérique à 4 chiffres¹.

Les utilisateurs (de type `Utilisateur`) du bâtiment doivent saisir un code valide (sur un clavier par exemple) pour pouvoir ouvrir la porte.

Toute tentative, réussie ou non, d'ouverture d'une porte par un utilisateur donne lieu à un enregistrement dans un «journal des accès» dans lequel on distingue si l'accès a été autorisé ou refusé (cf. classe `JournalAcces` en Annexe).

Les portes.

Les portes ont un identifiant inscrit sur un autocollant collé sur la porte. Cet identifiant est représenté par une chaîne de caractères. Il est supposé unique (sans vérification).

Les objets `Porte` disposent tous d'une méthode `accesAutorise` :

```
/** Détermine si un utilisateur peut ouvrir cette porte avec le code
 * fourni. Cette méthode provoque l'ajout d'un enregistrement au
 * journal d'accès, il précise si l'accès a été autorisé ou non.
 *
 * @param user l'utilisateur qui demande à ouvrir la porte
 * @param code le code fourni par l'utilisateur
 * @return true si et seulement si le code fourni par l'utilisateur
 *         a permis d'ouvrir cette porte
 * @see securite.JournalAcces
 */
```

```
public boolean accesAutorise(Utilisateur user, int code)
```

Selon les portes, il y a deux types de points de sécurité :

- soit la porte possède un seul code qui lui est propre, il faut alors fournir ce code pour ouvrir la porte ;
- soit le système de contrôle d'accès est configuré par une liste d'utilisateurs autorisés. Il faut alors fournir le code de l'un des utilisateurs autorisés pour pouvoir ouvrir la porte.

On peut en plus considérer que les portes dont l'accès n'est pas limité correspondent à des portes pour lesquelles tous les codes sont acceptés.

Les utilisateurs. Les utilisateurs possèdent un nom. De plus, chaque utilisateur connaît son propre code et la liste des portes que ce code lui permet d'ouvrir, ainsi qu'un certain nombre de couples (*identifiant, code unique*) qui correspondent aux identifiants des portes dont il connaît le code d'accès.

Dans la simulation considérée ici, on souhaite prendre en compte deux catégories d'utilisateurs. Leurs comportements diffèrent lors de l'ouverture d'une porte. On distingue :

1. les utilisateurs qui ont une bonne mémoire. Quand ils veulent ouvrir une porte, ceux-ci donnent directement le bon code s'il s'agit d'une porte dont il connaît le code d'accès ; sinon il essaie son code personnel.
2. les utilisateurs étourdis. Ils ne tiennent pas compte des portes associées aux codes connus et donc testent successivement chaque code en leur possession, le leur ou celui des portes qu'ils connaissent, jusqu'à en trouver un qui fonctionne.

¹Dans la suite le terme «code» correspondra toujours à cette notion et on représentera les codes par des données de type `int` sans faire de vérification sur la forme du nombre entier utilisé

Evidemment les différences de comportements des utilisateurs sont visibles dans les enregistrements du journal d'accès.

Les objets `Utilisateur` disposent d'une méthode `ouvrirPorte` dont voici la javadoc :

```
/** permet à cet utilisateur de tenter d'ouvrir une porte, il doit
 *   pour cela avoir un acces autorisé à la porte
 * @param porte la porte que cet utilisateur tente d'ouvrir
 * @exception AccesRefuseException si cet utilisateur n'a pas acces à
 *   la porte indiquée
 * @see securite.Porte#accesAutorise(Utilisateur, int)
 */
```

- Q 1 . Quel design pattern est mis en place par la classe `JournalAcces` dont un diagramme est fourni en annexe ?
- Q 2 . Sous la forme d'un diagramme UML détaillé, faites une proposition pour représenter les portes et leur système de sécurité.
- Q 3 . Sous la forme d'un diagramme UML détaillé, faites une proposition pour représenter les différents utilisateurs.
- Q 4 . Donnez le code java correspondant à cette proposition sur les utilisateurs.

Malheureusement, l'autocollant de l'identifiant de la porte se détache parfois. Cela correspond alors à des objets `Porte` dont l'attribut identifiant vaut "". Dans ce cas, l'utilisateur qu'il ait une bonne mémoire ou non doit procéder de manière systématique.

Il applique alors plusieurs stratégies les unes après les autres jusqu'à en trouver une qui fonctionne :

1. il essaie tous les codes des portes dont il connaît le code,
2. puis il essaie son code personnel,
3. finalement, chaque utilisateur connaissant d'autres utilisateurs (on ajoute un attribut `listeConnaissances` de type `List<Utilisateur>` et son accesseur aux objets `Utilisateur`), il peut les appeler et leur demander de venir essayer d'ouvrir la porte. Ces connaissances appliquent alors leur comportement pour `ouvrirPorte` (et n'appellent donc pas leurs propres connaissances).

On doit pouvoir envisager la mise en œuvre d'autres stratégies pour l'ouverture de porte tout en respectant le principe ouvert-fermé.

La méthode invoquée par les objets utilisateurs dans ce cas là est : `ouvrirPorteInconnue` dont la signature est identique à celle de `ouvrirPorte`.

- Q 5 . Utilisez le design pattern *chaîne de responsabilités* pour implémenter l'application successive de ces trois stratégies. Donnez le diagramme de classe correspondant à votre solution puis son code java (celui des types présentés dans votre solution et de la méthode `ouvrirPorteInconnue` de `Utilisateur`).

Annexe

Exemple affichage commande

Pour les éléments commandés, les textes affichés correspondent à leur *description*.

Commande 12 pour table 42

=====

Plat viande1 = 12,50

Boisson soda2 = 1,50

Dessert glace5 XL = 2,40

Menu - Boisson biere0 : Plat viande3 : Dessert fruit12 = 13,50

Menu - Boisson soda1 XL : Plat vegetarian4 XL : Dessert glace1 XL = 16,60

Menu enfant - Boisson soda5 : Plat viande1 : Dessert glace10 : cadeau = 12,00

=====

total commande = 46,50

=====

servie, non encaissée.

Journal des accès

securite::JournalAcces
+ INSTANCE : JournalAcces
...
- JournalAcces()
+ enregistreAccesAutorise(u : Utilisateur, p : Porte)
+ enregistreAccesRefuse(u : Utilisateur, p : Porte)

Les méthodes `enregistreAccesRefuse` et `enregistreAccesAutorise` sont invoquées selon que l'accès a été refusé ou autorisé.

Exemple de journal généré par la classe `JournalAcces`. Les horaires correspondent aux dates d'invocation des méthodes `enregistreAccesRefuse` et `enregistreAccesAutorise` :

```
9:50:42 memory tente ouverture porte 5 : *** accès autorisé ***
10:00:12 etourdi tente ouvertue porte 2 : /\ accès refusé /\
10:00:14 etourdi tente ouvertue porte 2 : /\ accès refusé /\
10:00:16 etourdi tente ouvertue porte 2 : *** accès autorisé ***
10:12:54 memory tente ouverture porte 4 : *** accès autorisé ***
10:23:04 memento tente ouverture porte 2 : *** accès autorisé ***
10:25:10 lunatique tente ouvertue porte 7 : /\ accès refusé /\
10:25:12 lunatique tente ouvertue porte 7 : /\ accès refusé /\
10:25:14 lunatique tente ouvertue porte 7 : /\ accès refusé /\
...
```