UE Conception Orientée Objet

Devoir Surveillé
1h30
Copie des diapositives de cours annotée autorisée.
Dictionnaires de langue autorisés. Autres documents interdits.

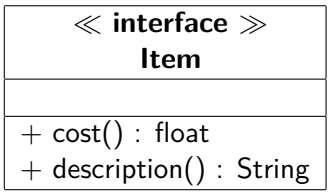
Indiquez sur votre copie *groupe INFO-X* ou *groupe MIAGE-X* où *X* est votre numéro de groupe.

Sauf mention expresse, la javadoc et les tests des classes à écrire ne sont pas demandés.

Les exercices sont entièrement indépendants, ils évaluent des compétences différentes et peuvent être traités dans n'importe quel ordre.

Exercice 1 : Quelques tests

On s'intéresse à un contexte de création de commandes. Les articles qu'il est possible de commander sont de type `Item` et la gestion de stock de ces éléments est prise en charge par des objets de type `Supplier` (*fournisseur*).
Le type `Item` est défini par l'interface :

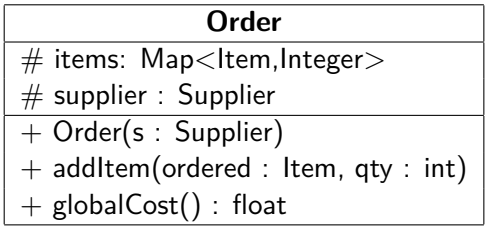


où, sans surprise, `cost()` fournit le coût unitaire d'un `Item`.
Le type `Supplier` est défini par :

```
package order;

public interface Supplier {
  /** tell whether this supplier can supply the item in required quantity
   * @param item the item to supply
   * @param expectedQty the quantity to supply
   * @return <code>true</code> iff this supplier can supply expectedQty of item
   */
  boolean canSupply(Item item, int expectedQty);
  /** decrease the quantity of item available for this supplier
   * @param item the involved item
   * @param qty the quantity to remove
   */
  void decreaseQuantity(Item item, int qty);
}
```

Les commandes sont de type `Order` :



où l'attribut `supplier`, initialisé via le constructeur, désigne le fournisseur auprès duquel est passée la commande.

- `addItem` permet d'ajouter à la commande `qty` éléments `ordered` ;
- `globalCost()` permet de connaître le coût total de la commande, c'est-à-dire le cumul des coûts unitaires de tous les articles commandés.

La cahier des charges stipule que :

- la méthode `addItem` déclenche une exception `NotAvailableException`[1] si le fournisseur ne peut pas fournir l'article commandé avec la quantité demandée ;
- à l'issue de la méthode `addItem`, le coût global de la commande est augmenté du coût des articles ajoutés ;
- l'exécution de la méthode `addItem` provoque l'appel de la méthode `decreaseQuantity` du `supplier` avec les bonnes valeurs de paramètres.

D'autre part, d'autres types de commande sont envisageables. On souhaite ainsi avoir des commandes de type `OrderWithShipping` (qui hérite de `Order`) pour lesquelles le coût global est augmenté des frais d'expédition d'un montant fixe de 10 euros, par exemple.

Q 1. Donnez le code de la suite de tests (sauf l'entête avec les `import` qui seront supposés correctement écrits) qui permet de vérifier toutes les contraintes du cahier des charges ci-dessus pour la méthode `addItem` des classes `Order` et `OrderWithShipping`.

Exercice 2 : Contrats et fournisseurs

Les contrats. Un contrat porte sur une certaine quantité de produits commandé par un client auprès d'un fournisseur. La qualité de la marchandise qui doit être fournie est identifiée par un niveau représenté par un entier. Un contrat est également caractérisé par le prix qu'est prêt à payer le client au fournisseur. Un contrat est représenté par une instance de la classe :

Contract
...
+ Contract(minQuality : int, quantity : int, offeredPrice : float, customer : Customer) + getMinQuality():int + getQuantity() : int + getOfferedPrice() : float

Les fournisseurs. Un fournisseur est un objet du type `Provider`. Un tel objet est caractérisé par son nom (une chaîne de caractères) supposé unique, son niveau de qualité de production (un entier) et son coût unitaire de production[2].
Le type `Provider` dispose (au moins) des méthodes suivantes :

`protected void credit(float amount)` crédite le compte du founisseur de `amount`.

`protected void processContract(Contract contract)` déclenche l'exécution du contrat par le fournisseur ;

`protected boolean isAcceptable(Contract contract)` détermine si un contract est acceptable (`true`) pour ce fournisseur. On ne s'intéressera pas ici à comment est calculé ce critère ;

`public Contract handleNextContract(List<Contract> availableContracts)` permet de gérer le prochain contrat sélectionné par ce fournisseur.

Une exception `AlreadyInvolvedException` est levée si le fournisseur est déjà engagé dans un contrat. On suppose que dans la liste `availableContracts` les contrats sont rangés selon un ordre d'urgence décroissante, on n'a pas ici à se préoccuper de la manière dont ce critère d'urgence est géré.

Dans un premier temps, `handleNextContract` choisit dans la liste `availableContracts` un contrat acceptable pour le fournisseur. Comment est fait ce choix parmi les contrats acceptables dépend du fournisseur.

Dans un second temps, seulement si un contrat a pu être trouvé, cette méthode crédite le compte du fournisseur du montant du contrat et déclenche l'exécution du contrat. Le fournisseur est alors engagé jusqu'à l'exécution du contrat. Le contrat choisi est renvoyé comme résultat, `null` est renvoyé dans le cas où aucun contrat n'a été choisi ;

[1] supposée définie.
[2] Dans cet exercice, par simplification les produis sont indifférenciés.

`isReady()` renvoie `true` dès que le travail lié à l'exécution du contrat a été achevé. Le fournisseur n'est alors plus engagé.

Il peut exister d'autres méthodes auxquelles on ne s'intéresse pas ici.
On identifie plusieurs types de fournisseurs qui se distinguent dans leur manière de choisir les contrats qu'ils acceptent :

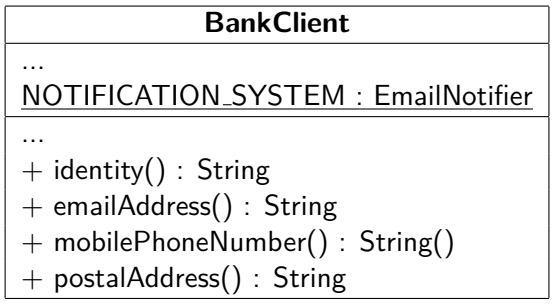
- Certains fournisseurs (`EmergencyFirstProvider`) donnent priorité à l'urgence et choisissent le premier contrat de la liste `availableContracts` qui est acceptable pour eux (on rappelle que dans la liste `availableContracts` les contrats sont, par hypothèse, fournis par ordre d'urgence décroissante) ;
- D'autres fournisseurs (`BestMarginProvider`) choisissent parmi la liste de tous les contrats disponibles celui acceptable pour lequel ils réalisent la marge la plus importante. Ils ne se préoccupent de l'urgence qu'en cas d'égalité de marge entre différents contrats et choisissent alors le plus urgent.

Q 1. Donnez, sous la forme de diagrammes UML clairs et détaillés (liens d'héritage, méthodes, types des attributs, des paramètres, des valeurs de retour), une proposition de modélisation pour représenter les fournisseurs.

Q 2. Donnez le code nécessaire à la mise en œuvre de la méthode `handleNextContract` pour chacun des types de fournisseurs où elle est définie.

Le code des autres méthodes des fournisseurs citées ci-dessus n'est pas demandé.

Exercice 3 : A propos de conception
On suppose définie une classe `BankClient` ainsi[3] :



On fournit les codes suivants :

```
package bank;
public class BankAccount {
  protected float balance;
  protected final BankClient owner;
  protected final String number;

  public BankAccount(BankClient owner, String number) {
    this.owner = owner;
    this.number = number;
    this.balance = 0;
  }

  public void credit(float amount) {
    this.balance += amount;
    BankClient.NOTIFICATION_SYSTEM.notifyOperation(this.owner, "account credited");
  }

  public void withdraw(float amount) throws IllegalStateException {
    if (this.balance < amount) {
      BankClient.NOTIFICATION_SYSTEM.notifyProblem(this.owner, "withdrawal not possible");
      throw new IllegalStateException("withdrawal not possible");
    }
    this.balance -= amount;
    BankClient.NOTIFICATION_SYSTEM.notifyOperation(this.owner, "widthdraw done with sucess");
  }
}
```

[3]Pour des raisons de simplification du sujet, on suppose que les différentes informations sont de type `String`

```java
package bank;
public class EmailNotifier {

  private static final String TRANSACTION_PROBLEM = "transaction problem";
  private static final String TRANSACTION_SUCCESS = "transaction success";

  public void notifyOperation(BankClient owner, String msg) {
    this.sendEmail(owner.emailAddress(), TRANSACTION_SUCCESS, msg);
  }

  public void notifyProblem(BankClient owner, String msgProblem) {
    this.sendEmail(owner.emailAddress(), TRANSACTION_PROBLEM, msgProblem);
  }

  /** send an e-mail with subject and content to given address
   * @param emailAddress the recipient address
   * @param subject the subject of the e-mail
   * @param content the content of the e-mail
   */
  protected void sendEmail(String emailAddress, String subject, String content) {
    // ... code sending the email to emailAddress width given subject and content
  }
}
```

Q 1. Dans le cadre d'une évolution de l'application basée sur ce code, on souhaite avoir la possibilité pour un client de choisir d'être notifié par SMS plutôt que par e-mail (à l'aide une classe `SMSNotifier` qui resterait à écrire sur un modèle qui pourrait être similaire à celui de `EmailNotifier` – travail non demandé).

Q 1.1. Après avoir rappelé rapidement en quoi il consiste, dites pourquoi le code ci-dessus de la classe `BankAccount` ne respecte pas le principe OCP de SOLID.

Q 1.2. Quel autre principe SOLID n'est pas respecté par cette classe et provoque cette violation du principe OCP ?

Justifiez votre réponse.

Q 1.3. Proposez, à l'aide d'un diagramme UML, une correction qui applique ce principe et rétablit du même coup l'OCP.

Q 2. On s'intéresse uniquement maintenant à la classe `EmailNotifier`.

Q 2.1. La dépendance de cette classe vers le type `BankClient` est-elle justifiée ? Quel problème pose-t-elle potentiellement ?

Q 2.2. Quel principe SOLID n'est pas satisfait par cette dépendance ?

Q 2.3. Indiquez comment corriger ce défaut de conception.