TD 1 : Introduction à la P.O.O.

Objectifs pédagogiques : vision client/fournisseur, encapsulation, responsabilités, construction d'une application simple (sans héritage), diagramme de classes en UML, static, tableaux, ArrayList.

Dans ce premier TD, nous allons concevoir et programmer en Java une application simple dans un domaine métier connu de tous : la gestion de comptes bancaires. Lors d'une phase préliminaire d'analyse du domaine métier, des concepteurs ont élaboré le diagramme de classe suivant:

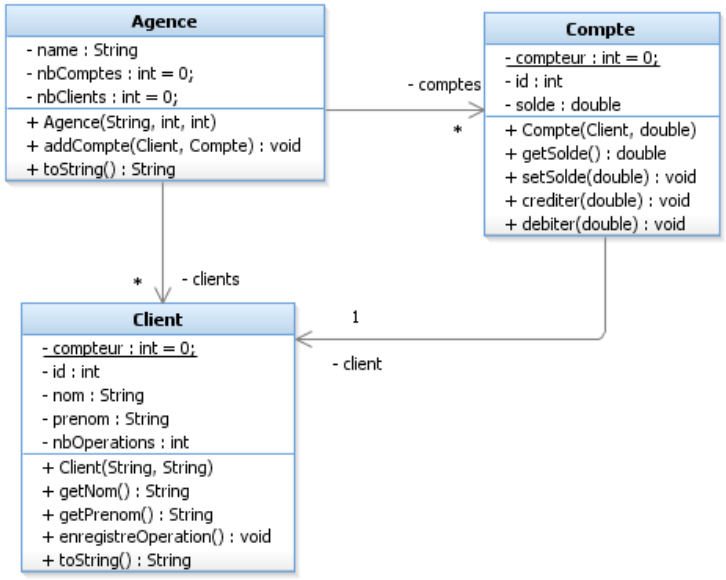

Figure 1: Diagramme UML pour les classes Agence, Compte et Client.

NB: Les associations représentent des attributs des classes. Par exemple, `comptes` est un attribut `private` (identifié par le −) de type `Compte[]` (car multiplicité *) appartenant à la classe `Agence`.

L'association `client` représente un attribut privé typé `Client` (car multiplicité 1) de la classe `Compte`. Attention à ne pas représenter à la fois l'attribut et l'association.

Pour être sûr de ne pas entrer en conflit avec d'autres applications bancaires, nous utiliserons le nom de paquetage `pobj.banque` pour notre implémentation.

Nous donnons ci-dessous l'implémentation de la classe `Client`:

Client.java

```java
package pobj.banque;

public class Client {
  private static int compteur = 0;
  private int id;
  private final String nom;
  private final String prenom;
  private int nbOperations;

  /**
  * Construit un client de nom et prénom spécifiés
  * @param nom le nom du client
  * @param prenom le prénom du client
  */
  public Client(String nom, String prenom) {
    id = compteur++;
    this.nom = nom;
    this.prenom = prenom;
    nbOperations = 0;
  }

  /**
  * Accède au nom du client
  * @return le nom du client
  */
  public String getNom() { return nom; }

  /**
  * Accède au prénom du client
  * @return le prénom du client
  */
  public String getPrenom() { return prenom; }

  /**
  * Enregistre une nouvelle opération effectuée par le client
  */
  public void enregistreOperation() { nbOperations++; }

  /** Méthode spéciale REPRESENTATION TEXTUELLE. */
  @Override
  public String toString() { return "Client(nom=" + nom + ", prénom=" + prenom + ", nbOps="
      + nbOperations + ")"; }

  /** Méthodes spéciales COMPARAISON CLIENTS */
  @Override public int hashCode() {
    int hash = 1;
    hash = hash * 17 + id;
    hash = hash * 31 + nom.hashCode();
    hash = hash * 13 + prenom.hashCode();
    return hash;
  }
  @Override public boolean equals(Object o) {
    if (o == this)
      return true;
    if (!(o instanceof Client))
      return false;
    Client c = (Client)o;
    return (c.nom.equals(nom) && c.prenom.equals(prenom));
  }
}
```

1.1 Structure de la classe et vocabulaire

⇒ Quel est le nom de la classe (relatif/absolu) ?
⇒ Le nom du type correspondant ?
⇒ Quel est son paquetage ?
⇒ Quel est le nom et où doit se trouver le fichier source de la classe ?
⇒ Identifier les différentes parties de la classe, notamment:

- les attributs
- les constructeur(s)
- les accesseurs (ou observateurs)
- les méthode(s) de traitement
- les méthode(s) standard(s)

⇒ Expliquer les mot-clés utilisés dans le code et donner une justification.
⇒ Comment sont calculés les identifiants de chaque client ? Expliquez le principe de ce calcul.

1.2 Classe `Compte`

La classe `Compte` garde en référence un client, elle contient le solde du compte.

⇒ **Proposez une implémentation Java conforme à ce qui apparaît dans le diagramme de classes**. Concernant l'identifiant, utilisez le même procédé que pour la classe `Client`. On demande de comptabiliser les opérations sur le compte : on doit incrémenter le compteur d'opérations du Client à chaque crédit ou débit sur son compte.

1.3 Classe `Agence` : Implémentation à base de tableaux

L'agence bancaire gère directement le fichier clients et la base des comptes. Dans une première approche, nous utiliserons une implémentation de ces bases sous la forme de tableaux (révision de la syntaxe associée). On passera donc au constructeur de l'agence, en plus de son nom, les limites en nombre de clients et de comptes.

La création de comptes et de fiches client est appelée de l'extérieur, l'agence se contente d'enregistrer le nouveau compte et le client associé tant que les nombres maximum ne sont pas dépassés.

Lorsqu'un même client possède deux comptes, on fera comme s'il s'agissait de deux clients différents.

⇒ **Proposez une implémentation de cette fonction d'ajout de compte** (en vous assurant que le code de la question 4 tournera).

1.4 Représentation mémoire et types de copies

⇒ Donnez les affichages issus des instructions suivantes.

BanqueTest.java

```
package pobj.banque.test;
import pobj.banque.Agence;  // ou alors, d'un seul coup: import pobj.banque.*;
import pobj.banque.Client;
import pobj.banque.Compte;
public class BanqueTest {
  public static void main(String[] args) {
    Agence agence = new Agence("Banque Impopulaire",5,3);
    Client client = new Client(args[0],args[1]);
```

```
    System.out.println("Nom client: "+client.getNom());
    System.out.println("Prénom client: "+client.getPrenom());
    Compte compte1 = new Compte(client,1000);
    Compte compte2 = new Compte(client,2000);
    agence.addCompte(client,compte1);
    agence.addCompte(client,compte2);
    System.out.println(compte1.getSolde());
    System.out.println(compte2.getSolde());
    compte1.crediter(200);
    compte2.debiter(300);
    System.out.println(compte1.getSolde());
    System.out.println(compte2.getSolde());
  }
}
```

⇒ Construisez une représentation schématique des objets en mémoire à la fin de l'exécution du code.

1.5 Classe Agence : Implémentation à base d'ArrayList

⇒ Modifiez votre implémentation pour utiliser des tableaux dynamiques (classe `ArrayList`, cf. annexe) et ainsi ne pas limiter le nombre de clients et de comptes gérés par l'agence.

1.6 Evolution de la structure

⇒ Construisez une classe `Banque` gérant toutes les agences.

⇒ Ajoutez des fonctions de transfert de compte d'une agence à l'autre.

Avec l'implémentation proposée jusqu'ici, comment fait-on pour trouver tous les comptes d'un client? Cela est-il efficace ?

⇒ Proposez une structure de données plus adéquate pour gérer les opérations bancaires courantes.

Annexe : Utilisation d'ArrayList

La classe `ArrayList` du paquetage `java.util` permet de manipuler des tableaux de taille dynamique. Voici un extrait de programme illustrant les principales méthodes disponibles. Consultez la documentation en ligne (documentation de l'API) pour plus d'informations. `java.util.List` est une interface implémentée (entre autres) par `Vector`, `ArrayList`, `LinkedList` ...

```
List<String> tab = new ArrayList<String>(); // contient des chaînes
System.out.println(tab.size()); // affiche 0 : la taille est vide
tab.add("hello"); // ajoute la chaîne "hello" (position 0)
System.out.println(tab.size()); // affiche 1 : un élément
String s = tab.get(0); // récupère le premier élément
System.out.println(tab); // affiche [hello]
tab.add("world"); // ajouté en position 1
System.out.println(tab.get(0)); // affiche le premier élément
for (String elt : tab) { // parcourir tous les éléments
  System.out.println("Element : " + elt); // affiche l'élement courant
}
tab.remove(0); // retire le premier élément
System.out.println(tab.size()); // taille 1 ("world" en position 0)
System.out.println(tab); // affiche [world]
tab.clear(); // retire tous les éléments
System.out.println(tab.size()); // taille 0
```