Programmation Orientée Objet
IG2I, LE4 - SI – 2017-2018

TP 3 – JPA et DAO : Tournées de véhicules

Diego Cattaruzza, Maxime Ogier

Contexte et objectif du TP

L'entreprise nordiste Bière2I est spécialisée dans la livraison des bières. Avec l'arrivée du printemps, elle doit faire face à une grosse demande de la part de ses différents clients. Cette année, la demande pour la bière de printemps est vraiment exceptionnelle et les gestionnaires de la livraison ne sont pas certains d'avoir le temps de pouvoir planifier toutes les livraisons à la main.

Pour améliorer son planning de livraison, Bière2I décide donc de faire appel aux meilleurs étudiants de la région Nord-Pas de Calais pour concevoir des outils informatiques de qualité qui puissent fournir un planning de livraison capable de gérer des quantités de demandes importantes.

L'objectif de ce TP est donc de concevoir un outil pour aider Bière2I dans son travail. Nous nous focalisons en particulier sur les couches données et métier de cet outil. Ce TP propose d'utiliser le concept de *mapping* objet-relationnel afin de mettre en place le modèle objet, à partir d'une base de donnée existante. Il s'agit d'utiliser l'API JPA (*Java Persistence API*), grâce à laquelle nous serons capables de créer le modèle objet en Java à partir du modèle relationnel (base de données). Par ailleurs, nous étudierons une manière de réaliser proprement l'accès aux données, afin qu'il soit possible de continuer à utiliser le même modèle objet, indépendamment du système de stockage de données utilisé ou de l'API pour accéder aux données (JPA, JDBC).

Ce TP permet d'illustrer les notions suivantes :

- le *mapping* objet-relationnel ;
- l'API JPA (*Java Persistence API*) ;
- le *design pattern* DAO (*Data Access Object*) ;
- le *design pattern Factory*.

Bonnes pratiques à adopter

N'oubliez pas de tester votre code au fur et à mesure de votre avancement. Par ailleurs, il vous est très fortement recommandé d'utiliser la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez. Il est également important que votre propre code soit commenté au format Javadoc (commentaire commençant par "/**" au-dessus des définitions des attributs et méthodes). Ceci vous permet (1) de spécifier rigoureusement le comportement d'une méthode au moment où vous l'implémentez, (2) de savoir ce que fait une méthode sans avoir besoin de regarder son code (qui d'ailleurs peut ne pas être accessible), et donc (3) de faciliter le partage et la réutilisabilité du code.

1 Introduction

Monsieur Houblon, le responsable de la logistique chez Bière2I, vous explique que son travail consiste à planifier la livraison d'une quantité de bière à un ensemble de clients. Chaque jour, des chauffeurs partent avec leur véhicule depuis le dépôt de la brasserie, visitent des clients et reviennent au dépôt. Pour utiliser la terminologie de Monsieur Houblon, ils accomplissent une *tournée*.

Le travail de Monsieur Houblon consiste à décider quel sous-ensemble des clients à livrer il faut affecter à chaque chauffeur, de façon à minimiser les coûts de transport (salaire des chauffeurs, coût du carburant) en respectant aussi la capacité de chaque véhicule (nous ne pouvons pas livrer deux clients qui demandent 60 caisses de bouteilles chacun, si le véhicule ne peut en contenir que 100).

Monsieur Houblon vous explique que la brasserie ne possède pas de véhicules, mais elle paie chaque jour des chauffeurs privés, qui, avec leur propre camion livrent les clients. Pour ce service, Monsieur Houblon doit payer un coût proportionnel à la distance parcourue par les chauffeurs pendant leur tournée. Voilà pourquoi il cherche à ce que cette distance soit la plus faible possible. Un exemple de planning de livraison est donné en Figure 1.

2 Modélisation de la solution de l'entreprise de livraison

2.1 Création de la base de données

L'entreprise Bière2I possède déjà une base de données. Elle contient toutes les données concernant le réseau routier, formé par des points (représentant le dépôt et les clients) et des routes (connexions entre deux points du réseau). La base de données contient également les véhicules disponibles pour effectuer les livraisons. Proposer un planning de livraison consiste à :

- affecter les clients aux véhicules ;
- déterminer l'ordre dans lequel un véhicule livre les clients qui lui sont associés.

Le modèle conceptuel de données associé est présenté dans la Figure 2.

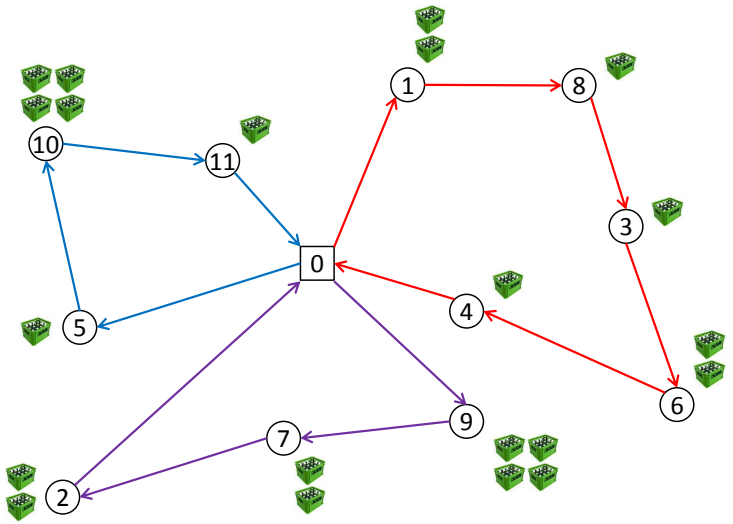

Figure 1: Exemple de planification avec 3 tournées qui partent du dépôt (0) pour livrer les clients.

Monsieur Houblon vous donne quelques indications supplémentaires par rapport au diagramme de classe.

- Les points du réseau sont caractérisés par un identifiant, leur abscisse et leur ordonnée. La classe **Point** sera une classe abstraite.
- Les dépôts sont des points spéciaux du réseau. Tous les véhicules partent d'un dépôt. La classe **Depot** hérite de la classe **Point**.
- Les clients sont des points spéciaux du réseau. Ils sont caractérisés par une demande, et leur position dans la tournée d'un véhicule. La classe **Client** hérite de la classe **Point**.
- Les routes modélisent le réseau routier et relient deux points du réseau. La classe **Route** est une classe d'association entre deux points (départ et arrivée) et est caractérisée par une distance (la distance pour aller du point de départ vers le point d'arrivée).
- Les véhicules transportent des caisses de bières et livrent les clients. La classe **Vehicule** est caractérisée par une capacité maximale (le nombre de caisses que le véhicule peut transporter), la capacité disponible (car une partie de la capacité est utilisée par les clients affectés au véhicule) et un coût de parcours (nous supposons ici que le coût est égal à la distance parcourue).
- Un planning est un ensemble de véhicules avec, pour chaque véhicule, la liste ordonnée des clients à livrer. La classe **Planning** est caractérisée aussi par le coût du planning (qui est la somme de coûts associés aux véhicules).

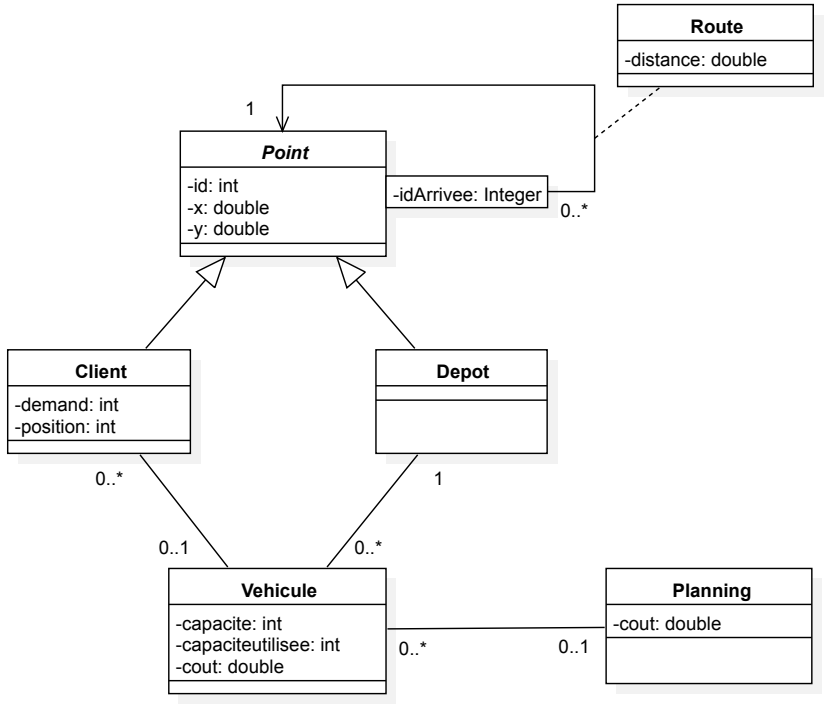

Figure 2: Diagramme de classe pour le système d'information des notre application.

Monsieur Houblon vous explique que chaque jour, le service commercial lui envoie la liste des clients qui doivent être impérativement livrés (avec pour chaque client le nombre de caisses de bière à livrer). Son objectif est alors de constituer un planning livraison qui minimise les coûts de transport (donc la distance totale parcourue par les véhicules).

Question 1. Monsieur Houblon vous a fourni les scripts de création de la base de données (disponibles sur Moodle). A partir de ces scripts, créez la base de

données. Réalisez la connexion à cette base avec Netbeans. N'oubliez pas de définir un nom d'utilisateur et un mot de passe **non vides** pour l'accès à la base.

2.2 Création de l'unité de persistance

Question 2. Créez un nouveau projet de type *Java Application*, et ajoutez-y une unité de persistance qui permettra de réaliser la connexion à la base de données.

2.3 Création des entités depuis la base de données

Question 3. Créez un paquetage **metier** dans lequel vous ajouterez les entités créées à partir de la base de données, en suivant les indications de la Section 8.1. Parcourez attentivement les entités créées afin de modifier au besoin les structures de données, les constructeurs, les accesseurs et mutateurs, les méthodes **equals()**, **hashCode()** et **toString()** ainsi que les annotations. N'hésitez pas à faire du ménage !

Remarquez en plus que la collection de routes associée à chaque point sera de type **Map** (vous pouvez aller regarder le sujet du TP5 de POO2 de cette année pour plus de détails).

Question 4. Dans la classe **Point**, ajoutez une méthode **public boolean addDestination(Point p, double distance)** qui ajoute la route entre ce point (**this**) et le point **p**, avec une distance **distance**. La méthode renvoie **true** si l'ajout a bien eu lieu, **false** sinon.

Question 5. Dans la classe **Point**, ajoutez une méthode **public double getDistanceTo(int key)** qui renvoie la distance entre le point **this** et le point dont l'id est **key**. Si cette route n'existe pas, la méthode renvoie plus l'infini.

Question 6. Dans la classe **Planning**, ajoutez une méthode **public boolean addVehicule(Vehicule v)** qui permet de rajouter un véhicule au planning. Vous vérifierez bien que le véhicule réfère bien vers ce planning.

Question 7. Dans la classe **Vehicule**, ajoutez une méthode **public boolean addClient(Client c)** qui ajoute le client **c** à la suite des autres clients livrés par le véhicule. Vous testerez bien que l'ajout du client au véhicule est possible par rapport à la capacité et la capacité déjà utilisée. Si le client est ajouté, vous mettrez à jour la capacité utilisée et le coût de transport suite à l'ajout de ce nouveau client. Aussi, le véhicule sur lequel le client est desservi (attribut de type **Vehicule** dans la classe **Client**) doit être mis à jour. Vous mettrez également à jour le coût total du planning. Vous ferez attention à ne pas faire plusieurs fois le même calcul.

Question 8. Dans la classe **Planning** ajoutez une méthode **public void updatePositionClients()** qui parcourt les véhicules du planning et met à jour la position de chacun des clients dans chaque tournée.

Question 9. Ajoutez un paquetage **test** et une classe **Test1** dans laquelle vous ferez un premier test pour vérifier que le modèle objet fonctionne correctement. Vous pouvez vous inspirer du petit jeu de test ci-dessous :

```
Depot d = new Depot(0, 0, 0);
Client c1 = new Client(1, 10, 10, 10);
Client c2 = new Client(2, -10, 10, 5);
Client c3 = new Client(3, 10, -10, 10);
d.addDestination(c1, 14.1);
d.addDestination(c2, 14.1);
d.addDestination(c3, 14.1);
c1.addDestination(d, 14.1);
c1.addDestination(c2, 20);
c1.addDestination(c3, 20);
c2.addDestination(d, 14.1);
c2.addDestination(c1, 20);
c2.addDestination(c3, 20);
c3.addDestination(d, 14.1);
c3.addDestination(c1, 20);
c3.addDestination(c2, 20);
Vehicule v1 = new Vehicule(0, d, 15);
Vehicule v2 = new Vehicule(1, d, 15);
Planning p = new Planning();
p.addVehicule(v1);
p.addVehicule(v2);
if(!v1.addClient(c1)) v2.addClient(c1);
if(!v1.addClient(c2)) v2.addClient(c2);
if(!v1.addClient(c3)) v2.addClient(c3);
p.updatePositionClients();
System.out.println(p.toString());
// Afficher le planning avec ses véhicules et les clients sur les véhicules
// Vérifier cout total des véhicules
// Vérifier capacité utilisée des véhicules
```

3 Le *design pattern* DAO

3.1 *Design pattern*

Un *design pattern* est une solution générale à un problème de conception récurrent dans de nombreuses applications. Les concepteurs de l'application doivent adapter la solution du *pattern* à leur projet spécifique.

3.2 Motivations

Maintenant que le modèle objet est en place, il semble naturel de mettre en place la couche métier dans laquelle nous devrons définir des méthodes qui nous permettrons d'affecter les clients aux véhicules afin de planifier les tournées de livraison. Il sera donc utile de pouvoir récupérer les données dans la base (par exemple les clients non encore affectés à un véhicule, ou les véhicules par encore utilisés), et de mettre à jour ces données une fois la planification de la livraison réalisée (mise à jour des véhicules avec les clients, la capacité utilisée et leur coût).

Pour ce faire, une première idée pourrait être d'utiliser JPA et d'implémenter des méthodes pour récupérer et mettre à jour les données. Cependant, notre code serait alors très fortement lié à la manière dont la persistance des données est réalisée. Mais que se passerait-il si demain nous décidions d'utiliser JDBC au lieu de JPA ? Ou si même nous décidions de ne plus utiliser une base de données pour stocker les données, mais un simple fichier texte, ou bien des fichiers au format xml ? Eh bien dans ce cas, il faudrait passer en revue toute la couche métier de notre application et la mettre à jour en fonction de la manière dont on gère la persistance des données. Ceci est clairement source d'erreurs, et n'est donc pas du tout une bonne pratique.

Ainsi, l'idée du *design pattern* DAO (*Data Access Object*) est de découpler la couche métier d'une application de la couche d'accès aux données. Ainsi, l'idée est d'isoler la persistance des données dans des objets spécifiques qui s'occupent uniquement de la persistance (stocker de nouvelles données, rechercher des données, les mettre à jour, les supprimer). De l'autre côté, dans la couche métier, on s'occupe uniquement de manipuler les données, sans s'occuper de la manière dont elles sont stockées.

Par ailleurs, dans une application d'entreprise, il est fréquent que la couche des objets métiers soit située sur une machine distante de la machine sur laquelle sont stockées les données. En revanche, la couche d'accès aux données sera elle située à proximité de la machine sur laquelle sont situées les données afin de réduire les temps d'accès aux données. Et par ailleurs, cela permet de bien séparer le travail entre une partie des développeurs qui peuvent s'occuper de l'accès aux données sur la couche d'accès aux données sans s'occuper du fonctionnement de l'application, et des développeurs qui peuvent s'occuper de la logique métier de l'application sans connaître la manière dont sont stockées les données.

3.3 Interface générique Dao

Pour appliquer le *design pattern* DAO, chaque classe métier aura son propre type de DAO (par exemple **EmployeeDAO** et **DepartmentDAO** pour les classes **Employee** et **Department** respectivement). En général, on a par contre une seule instance d'un objet DAO qui est utilisée par toutes les instances d'une classe objet métier, et nous verrons par la suite comment mettre cela en place.

En général, toutes les classes DAO effectuent les mêmes opérations de base. Ces opérations sont appelées CRUD (*Create*, *Retrieve*, *Update* et *Delete*), qui correspondent à la création, le recherche, la mise à jour et la suppression des données.

Ainsi, on peut donc définir un super-type d'objet afin de gérer ces opérations, et utiliser au mieux le polymorphisme.

Question 10. Ajoutez un paquetage **dao** dans lequel vous ajouterez une interface générique **Dao<T>** avec les méthodes suivantes :

- **public boolean create(T obj)** ;
- **public T find (Integer id)** ;
- **public Collection<T> findAll()** ;
- **public boolean update (T obj)** ;
- **public boolean delete (T obj)** ;
- **public boolean deleteAll()** ;
- **public void close()** qui permet de fermer la connexion à la base de données ou le fichier.

3.4 Classe abstraite générique JpaDao<T>

Pour le moment, nous allons utiliser la base de données que nous avons pour stocker les données, et nous allons effectuer les opérations de persistance en utilisant JPA. Pour ce faire, il est alors utile de créer une classe abstraite qui implémente l'interface générique **Dao<T>**. Si l'on souhaite par la suite utiliser un autre type de persistance de données (avec JDBC par exemple), alors on créera une autre classe abstraite spécifique pour le type de persistance.

Question 11. Dans le paquetage **dao** ajoutez une classe abstraite générique **JpaDao<T>** qui implémente l'interface **Dao<T>**. Dans cette classe abstraite, définissez des attributs pour le gestionnaire d'entités et le nom de l'unité de persistance (qui pourra être une constante déclarée **static final**). Ajoutez un constructeur par défaut qui initialise le gestionnaire d'entités. Dans cette classe abstraite générique, implémentez les méthodes **create**, **update**, **delete** et **close**. Pour la mise à jour, vous pouvez utilisez la méthode **merge** de **EntityManager** qui permet que l'entité passée en paramètre soit gérée par le gestionnaire d'entités, et que si cette entité est déjà présente dans la base, elle soit mise à jour.

Vous avez dû remarquer que pour le moment, nous n'avons pas implémenté de façon générique les méthodes **find**, **findAll** et **deleteAll**. En effet, pour implémenter ces méthodes de manière générique, il est nécessaire de connaître le type des données que l'on recherche ou que l'on veut supprimer. Ainsi, la

première chose que nous pouvons faire est d'ajouter dans la classe **JpaDao<T>** un attribut pour mémoriser la classe des entités avec lesquelles on travaille.

Question 12. Dans la classe abstraite générique **JpaDao<T>**, ajoutez un attribut de type **Class<T>** qui représente la classe des entités avec lesquelles on va travailler. Modifiez le constructeur de **JpaDao<T>** afin que cet attribut soit initialisé (avec un constructeur par donnée).

Question 13. Dans la classe abstraite générique **JpaDao<T>**, implémentez la méthode **find**. Vous pouvez utiliser la méthode **find(Class<T> entityClass, Object primaryKey)** fournie par la classe **EntityManager**.

Jusqu'à présent, nous avons vu que nous pouvions utiliser JPQL pour faire des requêtes sur la base de données. En fait, il existe une autre manière de réaliser des requêtes : en utilisant l'API *Criteria*. Avec cette API, il est possible de créer des requêtes en utilisant des méthodes, et sans écrire de SQL ou JPQL. Par exemple, on peut récupérer la liste des entités de type **Client** de la manière suivante :

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Client> cq = cb.createQuery(Client.class);
Root<Tache> tasks = cq.from(Client.class);
cq.select(tasks);
List<Client> allTasks = em.createQuery(cq).getResultList();
```

Si l'on souhaite récupérer seulement les demandes des clients, on peut procéder de la manière suivante :

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Integer> cq = cb.createQuery(Integer.class);
Root<Client> tasks = cq.from(Client.class);
cq.select(tasks.get("demande"));
List<Integer> allDemandes = em.createQuery(cq).getResultList();
```

On peut également supprimer toutes les entités de type **Client** de la manière suivante :

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaDelete<Client> cd = cb.createCriteriaDelete(Client.class);
int nbDelete = em.createQuery(cd).executeUpdate();
```

Question 14. Dans la classe abstraite générique **JpaDao<T>**, implémentez la méthode **findAll**. Pour ce faire, vous pouvez utiliser l'API *Criteria*.

Question 15. Dans la classe abstraite générique **JpaDao<T>**, implémentez la méthode **deleteAll**. Pour ce faire, vous pouvez utiliser l'API *Criteria*.

3.5 Interfaces pour les classes de la couche métier

Nous avons vu précédemment qu'il était judicieux de proposer une interface générique **Dao<T>** qui définit les opérations de persistances de base, communes à toutes les classes de la couche métier. Cependant, il est fortement probable que pour chacune des classes de la couche métier, il y ait certaines opérations de persistance qui doivent aussi être effectuées.

Afin d'utiliser au mieux le polymorphisme de la programmation orientée objet, il est donc intéressant de proposer une interface pour chacune des classes de la couche métier. Ces interfaces implémentent l'interface générique **Dao<T>** en définissant le type générique (c'est celui de la classe métier), et en définissant potentiellement de nouvelles méthodes de persistance.

Question 16. Dans le paquetage **dao**, ajoutez les interfaces **ClientDao**, **DepotDao**, **RouteDao**, **VehiculeDao** et **PlanningDao**, qui héritent de l'interface **Dao<T>** en spécifiant le type de paramètre de l'interface générique.

Dans l'interface **ClientDao**, définissez une méthode **public Collection<Client> findNotServed()** qui permet de renvoyer tous les clients qui ne sont pas encore livrés par un véhicule. Dans l'interface **VehiculeDao**, définissez une méthode **public Collection<Vehicule> findAllNotUsed()** qui permet de renvoyer toutes les véhicules non encore utilisés (qui ne sont pas affectés à un planning).

3.6 Classes concrètes du DAO

À présent que nous avons tout mis en place pour l'utilisation du polymorphisme, il ne reste plus qu'à implémenter les classes concrètes du DAO. Pour rappel, il faudra créer une classe concrète pour chaque classe de la couche métier.

Question 17. Dans le paquetage **dao**, ajoutez une classe **JpaDepotDao** qui hérite de la classe abstraite **JpaDao<Depot>** et qui implémente l'interface **DepotDao**. Implémentez les méthodes abstraites de **JpaDao<Depot>** et les méthodes définies dans l'interface **DepotDao**.

Question 18. Dans le paquetage **dao**, ajoutez une classe **JpaClientDao** qui hérite de la classe abstraite **JpaDao<Client>** et qui implémente l'interface **ClientDao**. Implémentez les méthodes abstraites de **JpaDao<Client>** et les méthodes définies dans l'interface **ClientDao**.

Question 19. Dans le paquetage **dao**, ajoutez une classe **JpaRouteDao** qui hérite de la classe abstraite **JpaDao<Route>** et qui implémente l'interface

RouteDao. Implémentez les méthodes abstraites de **JpaDao<Route>** et les méthodes définies dans l'interface **RouteDao**.

Question 20. Dans le paquetage **dao**, ajoutez une classe **JpaVehiculeDao** qui hérite de la classe abstraite **JpaDao<Vehicule>** et qui implémente l'interface **VehiculeDao**. Implémentez les méthodes abstraites de **JpaDao<Vehicule>**.

Question 21. Dans le paquetage **dao**, ajoutez une classe **JpaPlanningDao** qui hérite de la classe abstraite **JpaDao<Planning>** et qui implémente l'interface **PlanningDao**. Implémentez les méthodes abstraites de **JpaDao<Planning>**.

3.7 Premier test du DAO

Question 22. Dans le paquetage **test**, ajoutez une classe **Test2** dans laquelle vous ferez un premier test pour vérifier que le DAO fonctionne correctement. Vous pouvez vous inspirer du petit jeu de test ci-dessous :

```
Depot d = new Depot(0, 0);
DepotDao depotManager = new JpaDepotDao();
depotManager.deleteAll();
depotManager.create(d);
Client c1 = new Client(10, 10, 10);
Client c2 = new Client(-10, 10, 5);
Client c3 = new Client(10, -10, 10);
ClientDao clientManager = new JpaClientDao();
clientManager.deleteAll();
clientManager.create(c1); clientManager.create(c2);
clientManager.create(c3);
d.addDestination(c1, 14.1); d.addDestination(c2, 14.1);
d.addDestination(c3, 14.1);
c1.addDestination(d, 14.1); c1.addDestination(c2, 20);
c1.addDestination(c3, 20);
c2.addDestination(d, 14.1); c2.addDestination(c1, 20);
c2.addDestination(c3, 20);
c3.addDestination(d, 14.1); c3.addDestination(c1, 20);
c3.addDestination(c2, 20);
depotManager.update(d);
clientManager.update(c1); clientManager.update(c2);
clientManager.update(c3);
Vehicule v1 = new Vehicule(d, 15);
Vehicule v2 = new Vehicule(d, 15);
VehiculeDao vehiculeManager = new JpaVehiculeDao();
vehiculeManager.create(v1);
vehiculeManager.create(v2);
Planning p = new Planning();
PlanningDao planningManager = new JpaPlanningDao();
vehiculeManager.deleteAll();
planningManager.deleteAll();
p.addVehicule(v1);
p.addVehicule(v2);
planningManager.create(p);
if(!v1.addClient(c1)) v2.addClient(c1);
if(!v1.addClient(c2)) v2.addClient(c2);
if(!v1.addClient(c3)) v2.addClient(c3);
p.updatePositionClients();
clientManager.update(c1);
clientManager.update(c2);
clientManager.update(c3);
System.out.println(p.toString());
```

Vérifiez qu'à la fin de ce test, les données sont bien enregistrées en base, et notamment que les clients sont bien affectés à un véhicule, et que les capacités

des véhicules sont bien respectées.

Question 23. Que se passe-t-il si à la fin du jeu de test précédent vous ajoutez le code ci-dessous :

```
ClientDao clientManager2 = new JpaClientDao();
clientManager2.create(c1);
System.out.println(tacheManager.findAll().size());
```

Devrait-on avoir 3 ou 4 clients dans la base de données ? Combien y en a-t-il ? Expliquez pourquoi. Peut-on se satisfaire de ce comportement dans une application ?

4 Le *design pattern* Singleton

Vous avez dû remarquer à la fin de la section précédente que notre code a un gros problème : il est possible de créer autant d'instances que l'on souhaite des objets DAO, ce qui génère autant de contextes de persistance. Or, il serait très intéressant de pouvoir assurer qu'on ne peut avoir qu'une seule instance de chacune des classes DAO.

Considérons la classe suivante :

```
public class MyClass {
    // attributs
    public MyClass () {...}
    // méthodes
}
```

En fait, tant que cette classe possède un constructeur avec une visibilité **public**, il sera toujours possible de créer autant d'instances que l'on souhaite. Mais il est possible en Java de définir un constructeur avec une visibilité **private** :

```
public class MyClass {
    private MyClass () {...}
}
```

A ce moment là, la classe **MyClass** est le seul type d'objet qui peut faire appel au constructeur, et donc construire des instances de **MyClass**. Mais donc en dehors de **MyClass**, on ne peut plus avoir aucune instance ... ce qui pose problème. En fait, si vous y réfléchissez bien il est toujours possible d'obtenir des instances de **MyClass**. Pour cela, on peut utiliser une méthode statique (aussi appelée méthode de classe) :

```
public class MyClass {
    private MyClass () {...}

    public static MyClass getInstance () {...}
}
```

Ainsi, en dehors de **MyClass**, on peut récupérer une instance de la manière suivante :

```
MyClass.getInstance()
```

Si on remet les choses ensemble, on peut écrire le code suivant dans **MyClass** afin que la classe puisse être instanciée dans d'autres classes :

```
public class MyClass {
    private MyClass () {...}

    public static MyClass getInstance () {
        return new MyClass();
    }
}
```

Et là, vous vous dites certainement qu'on a pas réglé le problème. En fait, si désormais on souhaite restreindre le nombre d'instances de **MyClass**, on peut procéder de la manière suivante :

```
public class MyClass {
    private static MyClass instance;

    private MyClass () {...}

    public static MyClass getInstance () {
        if(instance == null) {
            instance = new MyClass();
        }
        return instance;
    }
}
```

Le problème de limiter le nombre d'instances d'un type d'objet est souvent rencontré par les programmeurs. Ce qui vient d'être présenté est en fait le *design pattern* singleton.

Question 24. Mettez en place le *design pattern* singleton dans les classes **JpaClientDao**, **JpaDepotDao**, **JpaPlanningDao**, **JpaRouteDao** et **JpaVehiculeDao**. Vérifiez dans le test de la classe **Test2** que le problème est résolu !

Question 25. À présent, que se passe-t-il si vous effectuez le test suivant :

```
Planning p = new Planning();
Vehicule v1 = new Vehicule();
Vehicule v2 = new Vehicule();
PlanningDao planningManager = JpaPlanningDao.getInstance();
VehiculeDao vehiculeManager = JpaVehiculeDao.getInstance();
planningManager.deleteAll();
vehiculeManager.deleteAll();
vehiculeManager.create(v1);
vehiculeManager.create(v2);
p.addVehicule(v1);
p.addVehicule(v2);
planningManager.create(p);
for(Vehicule v : vehiculeManager.findAll()) {
    System.out.println(v);
}
```

Combien de véhicules sont affichés ? Combien y a-t-il de véhicules dans la base de données ? Expliquez pourquoi. Si le nombre de véhicules n'est pas conforme à ce que vous attendez, proposez une manière de régler ce problème.

5 Le *design pattern* Factory

Pour le moment, nous n'avons implémenté des classes DAO que pour la persistance avec JPA. Mais avec les évolutions de notre application, peut-être serions nous amenés à implémenter d'autres classes de DAO (pour la persistance avec JDBC, ou l'utilisation de fichiers xml). Et si nous souhaitons utiliser ces autres classes pour obtenir une instance d'une classe DAO, alors il faudra passer en revue tout le code et modifier les appels pour récupérer une instance d'une classe DAO. Et si l'on souhaite avoir un code plus générique qui permette l'utilisation de différents types de persistances, alors il nous faudra utiliser à chaque fois une structure à choix multiple (**switch**). Ce n'est clairement pas performant car on risque de faire des erreurs, et ce type de code est difficile à maintenir.

En fait, ce problème de l'instanciation de classes concrètes qui implémentent une interface est très souvent rencontré. Le *design pattern* Factory apporte une solution à ce type de problème. Il s'agit en fait d'utiliser un objet (on parle de fabrique (*factory*)) qui sera chargé de l'instanciation des objets. Ainsi, l'instanciation est centralisée en un seul endroit ce qui est beaucoup plus facile en terme de maintenance du code.

D'ailleurs, rappelez-vous que précédemment, nous avions commencé par créer des objets de type **ClientDao**, **DepotDao**, **PlanningDao**, **RouteDao** et **VehiculeDao** en faisant appel au constructeur des classes concrètes **JpaClientDao**, **JpaDepotDao**, **JpaPlanningDao**, **JpaRouteDao** et **JpaVehiculeDao**. Puis par la suite, nous avons décide d'appliquer le *design pattern* singleton et de modifier la création d'instance. Heureusement, nous n'en étions qu'à la phase de test, mais si plus tard de nouveaux changements sont à faire, il est préférable d'utiliser une fabrique pour centraliser la création des instances.

Question 26. Dans le paquetage **dao**, ajoutez une classe **DaoFactoryJpa** avec cinq méthodes statiques : **getClientDao()**, **getDepotDao()**, **getPlanningDao()**, **getRouteDao()** et **getVehiculeDao()**. Modifiez la visibilité des méthodes statiques des classes **JpaClientDao**, **JpaDepotDao**, **JpaPlanningDao**, **JpaRouteDao** et **JpaVehiculeDao** qui permettent de récupérer l'instance : il faut à présent éviter que ces méthodes soient appelées directement, mais que l'on passe par la fabrique. Modifiez en conséquence le test, et vérifiez que tout fonctionne correctement.

Question 27. À présent, pour utiliser encore une fois le polymorphisme, et centraliser la création des fabriques de DAO, nous pouvons utiliser une fabrique de fabrique ! Dans le paquetage **dao**, ajoutez une classe abstraite **DaoFactory**. Cette méthode contient cinq méthodes abstraites **getClientDao()**, **getDepotDao()**, **getPlanningDao()**, **getRouteDao()** et **getVehiculeDao()**. En conséquence, la classe **JpaDaoFactory** hérite de cette classe abstraite (et on ferait de même pour des classes **JdbcDaoFactory** ou **XmlDaoFactory**). Ajoutez une énumération **PersistenceType** avec les noms possibles des types de persistance. Dans la classe abstraite **DaoFactory**, implémentez une méthode **public static DaoFactory getDaoFactory (PersistenceType type)**. Modifiez en conséquence le code de test, et vérifiez que tout fonctionne correctement.

6 Création du planning

Maintenant que toute la partie persistance des données est en place, nous pouvons aisément mettre en place une méthode pour créer le planning pour livrer les clients non affectés à un véhicule. La règle à suivre pour l'optimisation est de considérer l'ensemble des clients non encore affectés à un véhicule, et de choisir tous les véhicules non encore affectés à un planning.

Ensuite, il faut affecter les clients aux véhicules du planning, avec pour objectif principal de minimiser la distance totale parcourue.

Question 28. Créez un paquetage **algo**, et ajoutez-y une classe **Solveur**, avec une méthode statique **resoudre**. Implémentez cette méthode de manière à affecter les clients présents dans la base de données et qui ne sont pas encore affectés. Dans le paquetage **tests**, ajoutez une classe **TestRoutage** afin de vérifier que le planning proposé fonctionne correctement (les données doivent être correctes et enregistrées dans la base de données).

7 Bonus

7.1 Gestion des exceptions

Jusqu'ici nous n'avons pas parlé de la gestion des exceptions dans les méthodes de persistance des données. En fait les méthodes des DAO peuvent lancer des exceptions puisqu'elles effectuent des opérations d'entrées/sorties. Mais ces exceptions ne doivent pas être liées à un type de DAO particulier si on souhaite pourvoir changer facilement de type de DAO.

Ainsi, une solution est de créer une ou plusieurs classes d'exceptions, indépendantes du support de persistance. On peut par exemple créer une classe **DaoException** (qui hérite de la classe **Exception**). Dans les méthodes des DAOs, il faut attraper les exceptions particulières et relancer des exceptions de type **DaoException**.

Question 29. Mettez en place la gestion des exceptions dans la couche DAO de votre application.

7.2 Un autre type de persistance : JDBC

Question 30. Afin de mieux voir l'intérêt du *design pattern* DAO, mettez en place le DAO pour la persistance des données avec JDBC. Vérifiez ensuite que la modification du code de la méthode **resoudre** est très simple.

8 Quelques notions utiles

8.1 Création des entités depuis une base de données

Après avoir créé une unité de persistance, il est possible d'utiliser les assistants Netbeans pour créer des entités depuis une base de données. Pour cela, il suffit de suivre les étapes suivantes :

1. ouvrez la fenêtre *Projects* (*Ctrl + 1*) ;
2. clic-droit sur le nom du paquetage dans lequel vous souhaitez créer l'entité, puis cliquez sur *New* et sélectionnez *Other ...* ;
3. dans la catégorie *Persistence*, choisissez un fichier de type *Entity Classes from Database...* puis cliquez sur *Next*;
4. dans la fenêtre qui s'est ouverte, rentrez les informations sur le nom de la connexion à la base de données, les tables pour lesquelles vous souhaitez créer des entités ;
5. cliquez sur le bouton *Next* ;
6. sélectionnez les options nécessaires (une option utile est '*Generate Named Query Annotations for Persistent Fields*) ;
7. cliquez sur le bouton *Finish* pour créer les entités.

Vous pouvez alors regarder le code généré par Netbeans, et vous rendre compte que les entités ont été générées automatiquement, avec toutes les annotations.

Il reste juste à relire ce code généré et effectuer les modifications nécessaires afin que le code soit bien conforme à ce que vous souhaitez. En particulier, on peut s'interroger sur l'utilité des accesseurs et mutateurs, sur les constructeurs, ainsi que sur la définition des méthodes **equals()** et **toString()**.

References

[1] Freeman, E., Freeman, E., Bates, B., Sierra, K., *Head First Design Patterns, O'Reilly, 2004*

[2] Keith, M. and Schincariol, M., *Pro JPA 2: Second Edition - A definitive guide to mastering the Java Persistence API, APress, 2013*