

Contexte et objectif du TP

Ah le nord ! La pluie, le ciel gris, les gens accueillants, les ch'tis, la bière. Eh oui, la bière ! Beaucoup de bière. De grosses quantités sont consommées dans les bars, les pubs, les restaurants. Mais, vous êtes-vous déjà demandé comment les brasseries font pour livrer toutes ces bières d'une façon efficace ? Nous allons le découvrir dans ce TP.

Monsieur Houblon, chef d'une brasserie de la région Nord-Pas-de-Calais, vous contacte et vous demande de concevoir un outil permettant d'obtenir le planning de livraison de ses bières à ses clients.

L'objectif de ce TP est donc de fournir à Monsieur Houblon un outil lui permettant de livrer ses clients au moindre coût possible, i.e. en parcourant une distance minimale.

Ce TP permet ainsi de revoir les notions suivantes :

- la notion d'héritage ;
- la notion de classe abstraite ;
- l'interface **List** et ses implémentations (**ArrayList** et **LinkedList**) ;

et permet aussi d'illustrer les notions suivantes :

- l'implémentation en Java d'une classe d'association ;
- l'implémentation en Java d'une association qualifiée ;
- l'interface **Set** et ses implémentations (**HashSet** et **TreeSet**) ;
- l'interface **Map** et ses implémentations (**HashMap** et **TreeMap**).

Si vous discutez du problème de Monsieur Houblon avec vos amis Marseillais vous pourrez remplacer *bière* par *pastis* pour illustrer la problématique. De la même manière, vous parlerez de *champagne* avec vos amis de Troyes, et de *bordeaux* avec vos amis Bordelais.

Bonnes pratiques à adopter

Pour réaliser ce TP, il vous est très fortement recommandé d'utiliser la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez. Par ailleurs, il est aussi 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.



Créez vos classes en mettant toujours les attributs en premier, suivis par les constructeurs (avec le constructeur par défaut en premier). Après les constructeurs, insérez les accesseurs et les mutateurs de la classe. Faites suivre les accesseurs et les mutateurs par toutes vos méthodes.

1 Modélisation du réseau routier

Monsieur Houblon, 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 affecte 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.

La Figure 1 présente un exemple de planification des livraisons. Le dépôt est numéroté 0 et les clients sont numérotés de 1 à 11. À chaque client est associé un nombre de caisses de bières à livrer. En supposant que les camions peuvent contenir 8 caisses de bière, la Figure 1 propose un planning de livraison avec 3 tournées. Chaque tournée est représentée par une couleur.

1.1 Création de la classe Point

Nous nous intéressons dans un premier temps à la représentation de la localisation géographique du dépôt et des clients dans la région Nord-Pas-de-Calais.

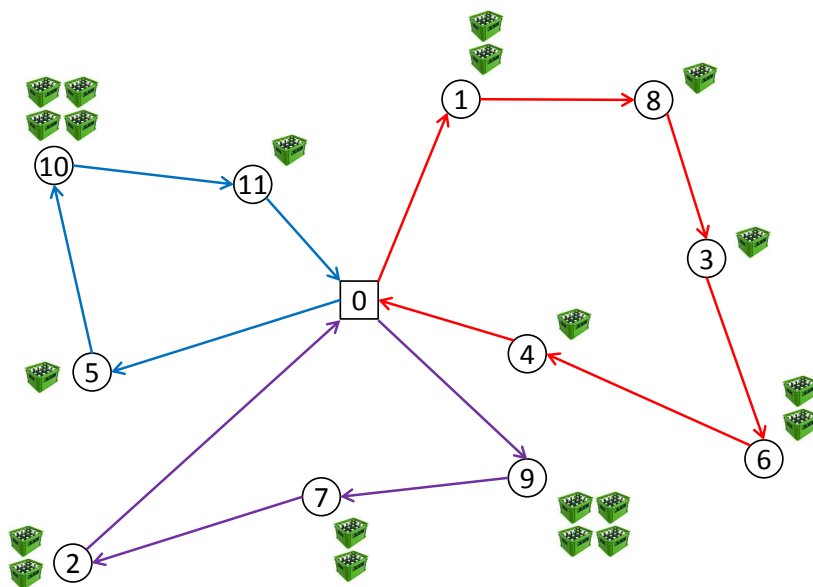


Figure 1: Exemple de planification avec 3 tournées qui partent du dépôt (0) pour livrer les clients.

Chacune de ces localisations peut être considérée comme un point caractérisé par une abscisse et une ordonnée. Comme il est possible que plusieurs clients soient au même endroit, il est préférable de considérer que chaque point possède également un identifiant unique.

Question 1. Après avoir ouvert Netbeans, créez un nouveau projet **Livraison**, un paquetage **reseau** (qui contiendra les classes pour la modélisation de l'infrastructure routière), et écrivez une classe **Point** qui représente un point, dont vous définirez les attributs. Commentez chacun des attributs sous format Javadoc.

Ajoutez un constructeur par données de l'abscisse et de l'ordonnée. Complétez la classe **Point** avec les accesseurs et mutateurs nécessaires et une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur le point (en fait il s'agit d'une redéfinition de la méthode **toString()** de la classe **Object**).

N'oubliez pas que les assistants Netbeans sont là pour vous aider (clic-droit puis *Insert Code...*). Commentez sous format Javadoc les méthodes que vous venez d'écrire.

Question 2. Écrivez une méthode principale (**public static void main(String[] args)**) afin de tester l'utilisation de la classe **Point**. N'oubliez pas que les assistants Netbeans sont là pour vous aider (tapez "*psvm*" puis *Ctrl* + *espace*, ou encore "*sout*" puis *Ctrl* + *espace*) !

1.2 Création de la classe Route

1.2.1 Classe d'association

Une route correspond au lien physique qui permet de relier deux localisations (deux points). En terme de modélisation UML, une route correspondrait donc à une association entre deux points. Cependant, une route est également caractérisée par sa distance. On peut donc modéliser une route, avec UML, comme une *classe d'association* sur l'association entre deux points. Cette notion de classe d'association permet d'associer des propriétés (ici la distance) à une association, ce qui n'est pas possible autrement. Et cette classe d'association n'a lieu d'être que si l'association existe : si deux points ne sont pas reliés entre eux, alors il n'y a pas de route !

La Figure 2 présente un exemple de classe d'association (sans lien avec le sujet du TP). On considère dans cet exemple qu'un employé peut travailler dans plusieurs entreprises. La classe **Emploi** est alors définie comme une classe d'association car elle permet d'ajouter à l'association entre un employé et une entreprise les informations sur la date d'embauche et le salaire. Ces deux informations ne sont pas propres à l'employé ni à l'entreprise, mais bien à l'association entre les deux ! Et on constate ici encore qu'un emploi ne peut exister que s'il y a effectivement une relation entre un employé et une entreprise (on omet les emplois fictifs...).

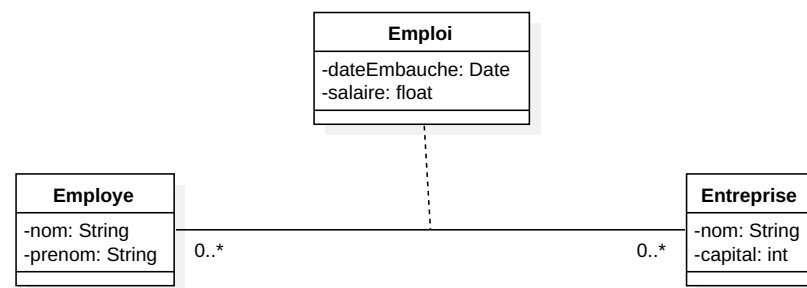


Figure 2: Exemple d'une classe d'association : la classe **Emploi** sur l'association entre **Employe** et **Entreprise**.

Pour implémenter cette notion de classe d'association en Java, on peut créer une classe qui modélise cette classe d'association. On créera par exemple une classe **Emploi** avec ses attributs (**dateEmbauche**, **salaire**), ainsi qu'une

référence sur chaque extrémité de l'association (**Employe** et **Entreprise**). Dans chaque extrémité de l'association (**Employe** et **Entreprise**), on garde une référence sur un ou plusieurs objets du type de la classe d'association (**Emploi**) en fonction de la cardinalité. Ainsi, pour l'exemple de la Figure 2, on pourrait écrire le code suivant :

```
1 public class Emploi {
2     private Employe employe;
3     private Entreprise entreprise;
4     private Date dateEmbauche;
5     private float salaire;
6 }
```

```
1 public class Employe {
2     private List<Emploi> mesEmplois;
3     private String nom;
4     private String prenom;
5 }
```

```
1 public class Entreprise {
2     private List<Emploi> mesEmplois;
3     private String nom;
4     private int capital;
5 }
```

1.2.2 Retour sur le réseau routier

Question 3. Écrivez une classe **Route** qui représente le lien entre deux points (avec une origine et une destination). Ajoutez ses attributs et commentez-les au format Javadoc.

Ajoutez un constructeur par données du point de départ (origine) et du point d'arrivée (destination). Nous considérerons, pour simplifier, que la distance de la route est égale à la distance euclidienne entre les deux points.

Complétez la classe **Route** avec les accesseurs et mutateurs nécessaires et une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur la route.

Question 4. Écrivez une méthode principale (**public static void main(String[] args)**) dans la classe **Route** afin de tester l'utilisation de cette classe.

1.3 Association entre les points

1.3.1 Rappel : collections d'objets

Pour manipuler des collections d'objets en Java, le paquetage **java.util** fournit un ensemble de classes qui permettent de manipuler des vecteurs dynamiques, des ensembles, des listes chaînées, des files et des tables associatives. Les classes relatives aux vecteurs, aux listes, aux ensembles et aux files implémentent une même interface (**Collection**) qu'elles complètent de fonctionnalités propres.

En fait, les collections sont manipulées par le biais de classes génériques implémentant l'interface **Collection<E>**, où **E** représente le type des éléments de la collection. Tous les éléments d'une collection sont donc du même type **E** (ou à la limite d'une classe dérivée de **E**).

1.3.2 Ensemble d'objets

Mathématiquement, un ensemble est une collection non ordonnée d'éléments, aucun élément ne pouvant apparaître plusieurs fois dans un même ensemble. En Java, cette notion est représentée par l'interface **Set** (en anglais ensemble se dit *set*). En conséquence, un **Set** est une collection qui garantit de ne pas contenir deux fois le même objet. Chaque fois qu'un nouvel élément est ajouté dans une collection de type **Set** (avec la méthode **public boolean add(E e)**), l'ajout n'est réalisé que si l'élément (**e**) n'est pas déjà présent dans l'ensemble, autrement dit si l'ensemble ne contient pas un autre élément qui lui soit égal.

Nous rappelons qu'en Java l'égalité entre des éléments est définie par la méthode **public boolean equals(Object o)** (et la méthode **public int compareTo(E e)** pour les objets qui implémentent l'interface **Comparable<E>**). La méthode **equals(Object o)** est définie dans la classe **Object**, mais elle se contente de comparer les adresses mémoires des objets, ce qui est très restrictif. En général, on souhaite se baser sur les valeurs des attributs des objets pour définir la notion d'égalité. Comme toutes les classes héritent par défaut de la classe **Object**, il est possible de redéfinir la méthode **equals(Object o)**.

Ainsi, lorsque l'on souhaite faire un ensemble (**Set**) d'objets de type **E**, il est très fortement recommandé que la classe **E** redéfinisse la méthode **equals(Object o)** (et la méthode **hashCode()** afin de garder une cohérence).

Par ailleurs, en théorie un ensemble n'est pas ordonné (i.e. on n'a pas un premier élément, puis un deuxième, puis un troisième, ...), l'ordre dans lequel les éléments sont ajoutés n'a aucune importance. Mais pour des raisons évidentes d'efficacité des méthodes de test d'appartenance d'un élément à l'ensemble, il est nécessaire que les éléments de l'ensemble soient organisés convenablement. Dans le cas contraire, un tel test d'appartenance ne pourrait se faire qu'en examinant un à un tous les éléments de l'ensemble (ce qui conduirait à une efficacité moyenne en $\mathcal{O}(n)$). Deux démarches différentes ont été employées par les concepteurs des collections, d'où l'existence de deux classes différentes (qui implémentent l'interface **Set**) :

- **HashSet** qui recourt à une technique dite de hachage, ce qui conduit à une efficacité du test d'appartenance en $\mathcal{O}(1)$;
- **TreeSet** qui utilise un arbre binaire pour ordonner complètement les éléments, ce qui conduit à une efficacité du test d'appartenance en $\mathcal{O}(\log(n))$.

La classe **HashSet** se base sur une implémentation avec une table de hachage. Vous pouvez consulter la section 4.1 de ce document si vous ne savez pas ce qu'est une table de hachage. Ainsi, si on souhaite utiliser une collection **HashSet** d'éléments d'une classe **E**, il est nécessaire dans cette classe **E** de redéfinir les méthodes **equals()** et **hashCode()** et que ces méthodes soient compatibles entre elles, i.e. si **e1.equals(e2)**, alors **e1.hashCode() == e2.hashCode()**. L'utilisation d'un **HashSet** offre d'excellentes performances en termes de test d'appartenance d'un élément à l'ensemble. Les méthodes **add()**, **remove()**, **contains()** et **size()** sont en $\mathcal{O}(1)$.

En revanche, si l'on souhaite que l'ensemble soit toujours ordonné selon un certain critère (sans nécessité d'utiliser un algorithme de tri), on privilégie l'utilisation de la classe **TreeSet**. Cette classe se base sur une implémentation avec un arbre binaire, ce qui permet d'ordonner totalement les éléments. Pour cela, si l'on souhaite utiliser une collection **TreeSet** d'éléments d'une classe **E**, il est nécessaire que cette classe **E** implémente l'interface **Comparable<E>**, et implémente donc la méthode **public int compareTo(E o)**. Cette méthode doit comparer l'objet courant (**this**) à l'objet reçu (**o**) et renvoyer un entier dont la valeur exacte est sans importance, mais qui doit être :

- négative si "**this** < **o**" ;
- nulle si "**this** = **o**" ;
- positive si "**this** > **o**".

Là encore, il faut garder la cohérence avec la méthode **equals()**, i.e. **(e1.compareTo(e2)==0) == (e1.equals(e2))**. Avec un **TreeSet**, les méthodes **add()**, **remove()** et **contains()** sont en $\mathcal{O}(\log(n))$.

Ainsi, pour déclarer et initialiser des ensembles on peut écrire :

```
1 Set<E> monEnsemble;
2 monEnsemble = new HashSet<>();
```

```
1 Set<E> monEnsemble;
2 monEnsemble = new TreeSet<>();
```

Puis pour itérer sur l'ensemble, on peut utiliser un itérateur ou une boucle *for...each* de la manière suivante :

```
1 Iterator<E> iter = monEnsemble.iterator();
2 while(iter.hasNext()){
3     E elem = iter.next();
4     // Utilisation de elem
5 }
```

```
1 for(E elem : monEnsemble){
2     // Utilisation de elem
3 }
```

Pour les autres méthodes, c'est à vous de consulter la Javadoc !

1.3.3 Retour sur le réseau routier

Nous avons défini les points à visiter dans le réseau, ainsi que les routes qui relient ces points. L'idée est maintenant de relier ces objets. Nous avons déjà mémorisé dans la classe **Route** les points extrémités. Il nous reste, dans la classe **Point**, à mémoriser les routes qui permettent de se déplacer vers les autres points.

Dans un premier temps, nous allons juste redéfinir les méthodes **equals** et **hashCode** des classes **Point** et **Route**. Ceci permettra d'avoir des méthodes cohérentes avec l'utilisation de ces classes, notamment quand on utilise des structures de données.

Question 5. Dans la classe **Point**, redéfinissez les méthodes **hashCode()** et **equals()**. N'oubliez pas que les assistants Netbeans sont là pour vous aider (clic-droit puis *Insert Code...*, puis *equals()* and *hashCode()*...).

Question 6. Dans la classe **Route**, redéfinissez les méthodes **hashCode()** et **equals()**. Réfléchissez bien à la manière dont on peut définir l'égalité entre deux routes.

Question 7. Modifiez la classe **Point** afin de prendre en compte la classe d'association **Route**. Il faut donc ajouter un attribut qui permet de mémoriser l'ensemble des routes qui partent d'un point. Dans un premier temps, utilisez une structure de données de type **List**. N'oubliez pas de réaliser l'initialisation dans le constructeur.

Maintenant que nous avons modélisé l'association entre les points, nous allons associer les instances des points entre elles.

Question 8. Dans la classe **Point**, écrivez une méthode **public void ajouterRoutes(Set<Point> mesDestinations)** qui prend en paramètre un ensemble de points. Ces points sont tous les points joignables depuis le point courant (**this**). Cette méthode doit donc remplir la structure de données définie précédemment.

Question 9. Dans la classe **Point**, ajoutez une méthode **public double getDistance(Point p)** qui renvoie la distance entre le point (**this**) et le point **p**. On renverra *infini* (**Double.POSITIVE_INFINITY**) si **p** n'est pas associé au point.

Question 10. Dans la classe **Point**, ajoutez une méthode **public int getNbRoutes()** qui renvoie le nombre de routes partant de ce point (le nombre d'éléments dans la liste).

Question 11. Quel est l'inconvénient d'avoir choisi une structure de données de type **List** ? Vous pouvez vous aider du test *TestAssociationPoints.java* disponible sur Moodle.

Question 12. À présent, modifiez dans la classe **Point** la structure de données pour stocker les routes. Utilisez un type **Set**. Modifiez donc en conséquence le code.

Est-ce que le problème qu'on avait précédemment avec les listes est réglé ? Dans le test *TestAssociationPoints.java*, que pensez-vous du temps de calcul de la somme des distances ?

1.3.4 Tables associatives

Une table associative permet de conserver une information associant deux parties nommées *clé* et *valeur* (chacun étant un objet). Elle est principalement destinée à retrouver la valeur associée à une clé donnée. Les exemples les plus caractéristiques de telles tables sont :

- le dictionnaire : à un mot (clé), on associe une valeur qui est sa définition ;
- l'annuaire usuel : à un nom (clé), on associe une valeur comportant le numéro de téléphone ;
- l'annuaire inversé : à un numéro de téléphone (qui devient la clé), on associe une valeur comportant le nom et, éventuellement, une adresse.

Comme pour les ensembles (**Set**), l'intérêt des tables associatives est de pouvoir y retrouver rapidement une clé donnée pour en obtenir l'information associée (pensez à la recherche d'une définition dans un dictionnaire : vous ne parcourez pas l'ensemble des mots jusqu'à trouver le bon, mais vous utilisez le fait que ces mots soient ordonnés alphabétiquement pour faire une recherche dichotomique). On va donc tout naturellement retrouver les deux types d'implémentation rencontrés pour les ensembles :

- table de hachage : classe **HashMap** ;
- arbre binaire : classe **TreeMap**.

Dans les deux cas, seule la clé sera utilisée pour ordonner les informations. Dans le premier cas (**HashMap**), on se servira du code de hachage des objets formant les clés ; dans le second cas (**TreeMap**), on se servira de la relation d'ordre induite par la méthode **compareTo(E o)**. L'accès à un élément d'un **HashMap** sera en $\mathcal{O}(1)$ tandis que l'accès à un élément d'un **TreeMap** sera en $\mathcal{O}(\log(n))$. En contrepartie de leur accès moins rapide, les clés d'un **TreeMap** seront (comme les **TreeSet**) ordonnées en permanence selon la relation d'ordre induite par la méthode **compareTo(E o)**. Une fois encore, si les clés sont de type **K**, on ne peut que conseiller de redéfinir dans la classe **K** les méthodes **equals(Object o)** et **hashCode()**.

Les classes **HashMap** et **TreeMap** n'implémentent pas l'interface **Collection** mais une autre interface nommée **Map**. Ceci provient essentiellement du fait que leurs éléments ne sont plus à proprement parler des objets mais des "paires" d'objets, i.e. une association entre deux objets.

L'intérêt de garder des valeurs dans une table associative vient du fait qu'on peut les récupérer efficacement par leur clé. Par contre à une clé donnée on ne peut associer qu'une seule valeur, il faut donc bien choisir les clés de sorte qu'elle permettent d'identifier de manière unique les valeurs. Par exemple, on peut utiliser une table associative pour ranger des personnes, et utiliser comme clé le numéro de sécurité sociale ; ou bien encore ranger des comptes bancaires et utiliser comme clé le numéro de compte.

Pour déclarer et initialiser des tables associatives on peut écrire :

```
1 Map<K,V> maMap;
2 maMap = new HashMap<>();
```

```
1 Map<K,V> maMap;
2 maMap = new TreeMap<>();
```

Pour insérer un objet dans une table associative vous pouvez utiliser la méthode **put(K key, V value)**, où **key** est la clé de la valeur **value**. Si la clé **key** existe déjà dans la table associative, la valeur **value** va remplacer l'ancienne valeur associée à la clé.

Pour récupérer une valeur associée à une clé **key** vous pouvez utiliser la méthode **get(K key)** qui renvoie la valeur **value** associée à **key**.

En théorie, les types **HashMap** et **TreeMap** ne disposent pas d'itérateurs. Mais on peut par exemple effectuer les parcours suivants :

```

1 // entrySet() renvoie un objet de type Set<Map.Entry<K, V>>
2 for(Map.Entry<K, V> entry : maMap.entrySet()) {
3     K key = entry.getKey();
4     V value = entry.getValue();
5     // Utilisation de key et value
6 }

```

```

1 // keySet() renvoie un objet de type Set<K>
2 for(K key : maMap.keySet()) {
3     // Utilisation de key
4 }

```

```

1 // values() renvoie un objet de type Collection<V>
2 for(V value : maMap.values()) {
3     // Utilisation de value
4 }

```

Pour connaître les autres méthodes offertes par ces objets, reportez-vous à la Javadoc !

1.3.5 Association qualifiée

Une association qualifiée met en relation deux classes sur la base d'une clé (appelée qualificateur). Quand une classe est liée à une autre classe par une association, on peut restreindre la portée de l'association à quelques éléments ciblés en utilisant ce qualificateur. Un qualificateur est un attribut ou un ensemble d'attributs dont la valeur sert à déterminer l'ensemble des instances liées à une instance via une association.

La Figure 3 propose d'illustrer ce concept, sans lien avec le sujet du TP. Dans cet exemple, au lieu d'avoir une association avec des cardinalités multiples entre les objets Banque et Personne, on utilise le qualificateur *numeroCompte*. Ceci permet de préciser qu'en utilisant un numéro de compte, on est capable d'identifier de manière unique la personne cliente de la banque. Dans la version sans association qualifiée (association avec des cardinalités multiples entre les objets Banque et Personne), il était possible que plusieurs personnes aient le même numéro de compte. Avec l'association qualifiée, il n'est plus possible que plusieurs personnes possèdent le même numéro de compte dans une banque. Ceci permet donc d'éviter de nombreuses erreurs, et permet aussi de faire des recherches de façon plus efficace.

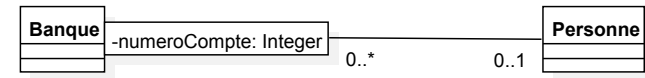


Figure 3: Exemple d'une association qualifiée : le qualificateur est le numéro de compte.

Pour mettre en œuvre ce concept en Java, on utilise une table associative, dans laquelle la clé est le qualificateur, et la valeur est l'objet cible de l'association. Ainsi, dans la classe **Banque** on aura le code suivant :

```

1 public class Banque {
2     private Map<Integer, Personne> mesComptes;
3 }

```

1.3.6 Retour sur le réseau routier

Nous avons défini les points à visiter dans le réseau, ainsi que les routes qui relient ces points. L'idée est maintenant de relier ces objets. Nous avons déjà mémorisé dans la classe **Route** les points extrémités. Il nous reste dans la classe **Point** à mémoriser les routes qui permettent de se déplacer vers les autres points. Pour ce faire, nous proposons de suivre le diagramme UML présenté dans la Figure 4. Ainsi, nous utilisons une association qualifiée pour associer deux points, en plus de la classe d'association **Route**. Pour cette association qualifiée, nous nous basons sur l'identifiant du point de destination. Ce qualificateur permet bien de qualifier un point de manière unique, car l'identifiant est unique.

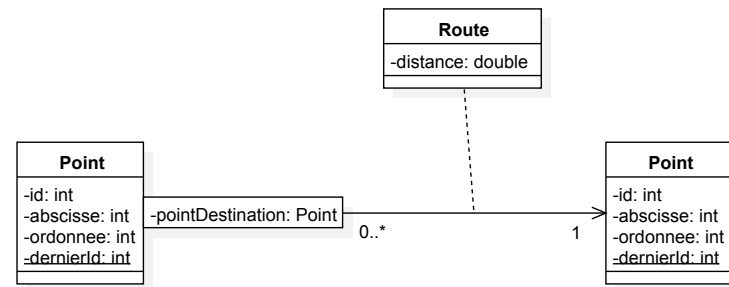


Figure 4: Association entre les points : classe d'association **Route** et association qualifiée.

Question 13. Modifiez la classe **Point** de telle sorte que la structure de données pour stocker les routes soit de type **Map**. Modifiez en conséquence le code, en particulier la méthode **getDistance**.

Exécutez à nouveau le test *TestAssociationPoints.java*. Qu'observez-vous ? Quel est l'intérêt d'avoir utilisé une structure de données de type **Map** ?

Question 14. Que se passe-t-il si dans la classe **Point** vous utilisez le code de hachage suivant :

```

1  @Override
2  public int hashCode() {
3      return 0;
4  }

```

Est-ce valide ? Qu'observez-vous si vous exécutez à nouveau le test *TestAssociationPoints.java* ? Proposez une explication.

Question 15. Modifiez la méthode **toString()** de la classe **Point** pour obtenir des informations sur les routes qui partent du point.

1.4 Héritage et classes abstraites

En fait, toutes les tournées de livraison partent d'un même endroit : le dépôt de la brasserie. Le dépôt est un point particulier dans la région Nord-Pas-de-Calais.

Lors des tournées de livraison, il faut livrer des caisses de bière à des clients. Les clients sont donc des points spéciaux : en plus de leurs coordonnées, ils sont caractérisés par une demande représentant le nombre de caisses de bière à livrer.

Question 16. Créez une nouvelle classe **Depot** qui hérite de **Point**. Complétez la classe avec une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur le dépôt.

Question 17. Créez une nouvelle classe **Client** qui hérite de **Point**. Ajoutez les attributs, les constructeurs, les mutateurs et les accesseurs nécessaires. Complétez la classe avec une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur le client.

Question 18. Nous faisons à présent les observations suivantes :

- les classes **Depot** et **Client** héritent de la classe **Point** ;
- dans ce projet nous n'utiliserons que des dépôts et des clients.

Modifiez le code en conséquence des observations faites ci-dessus (si nécessaire, vous pouvez aller regarder les classes **Personne**, **Pilote**, **Voiture** codées lors du TP 2).

Question 19. Testez vos classes **Depot** et **Client** pour vérifier que tout marche bien. Créez une méthode principale dans chaque classe et testez les codes suivants.

Dans la classe **Depot** :

```

1  Depot d = new Depot(0,0);
2  Client c1 = new Client(5,5,10);
3  Client c2 = new Client(-5,5,10);
4  Client c3 = new Client(-5,-5,10);
5  Client c4 = new Client(5,-5,10);
6  Set<Point> mesClients = new HashSet<>();
7  mesClients.add(c1);
8  mesClients.add(c2);
9  mesClients.add(c3);
10 mesClients.add(c4);
11 d.ajouterRoutes(mesClients);
12 System.out.println(d);

```

Dans la classe **Client** :

```

1  Client c1 = new Client(5,5,10);
2  Client c2 = new Client(-5,5,10);
3  Client c3 = new Client(-5,-5,10);
4  Client c4 = new Client(5,-5,10);
5  Set<Point> mesClients = new HashSet<>();
6  mesClients.add(c2);
7  mesClients.add(c3);
8  mesClients.add(c4);
9  c1.ajouterRoutes(mesClients);
10 System.out.println(c1);
11 System.out.println(c2);
12 System.out.println(c3);
13 System.out.println(c4);

```

2 Modélisation du planning de livraisons

2.1 Création de la classe Tournée

Nous avons maintenant tous les éléments pour modéliser le déplacement des véhicules. Nous allons donc créer des tournées. Une tournée est une suite de livraisons chez les clients qui part du dépôt et termine au dépôt. Elle est caractérisée par la distance parcourue (depuis le départ du dépôt jusqu'au retour au dépôt) et le nombre de caisses de bière livrées. De plus, une tournée est

effectuée par un camion qui a une capacité limitée (en nombre de caisses). Une tournée est donc également caractérisée par une capacité. Le nombre de caisses livrées dans la tournée ne doit pas dépasser la capacité.

Question 20. Ajoutez un paquetage **planning**. Ajoutez-y une nouvelle classe **Tournee**. Ajoutez ses attributs, parmi lesquels vous aurez un attribut de type **Depot** et un attribut qui représente les clients livrés lors de cette tournée. Justifiez le type de collection utilisé pour ce dernier attribut. Ajoutez un constructeur par données du dépôt et de la capacité. Ajoutez les accesseurs et mutateurs nécessaires.

Question 21. Ajoutez une méthode **public boolean ajouterClient(Client client)** qui ajoute le client dans la collection des clients livrés (on suppose que ce client est ajouté à la fin de la tournée, avant de retourner au dépôt). Cette méthode renvoie **true** si l'ajout a bien eu lieu, et **false** sinon. Notez bien qu'il n'est pas possible d'ajouter un client si cela engendre un dépassement de la capacité. Cette méthode doit mettre à jour la distance de la tournée ainsi que le nombre de caisses livrées. Faites attention au fait qu'une tournée commence et termine toujours au dépôt. Il faut donc considérer les routes qui vont du dépôt au premier client et du dernier client au dépôt. N'hésitez pas à faire un dessin.

Question 22. Ajoutez une méthode **public boolean ajouterClient(Set<Client> ensembleClient)** qui ajoute tous les clients de l'ensemble **ensembleClient** à la tournée (en mettant à jour la distance parcourue et la quantité livrée). Cette méthode renvoie **true** si tous les ajouts ont bien eu lieu, et **false** sinon (certains clients n'ont pas pu être ajoutés).

Question 23. Ajoutez une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur la tournée et une méthode principale dans la classe. Testez votre classe avec (mais pas seulement) le code suivant :

```
1 Depot d1 = new Depot(0, 0);
2 Client c1 = new Client(5, 5, 10);
3 Client c2 = new Client(-5, 5, 10);
4 Client c3 = new Client(-5, -5, 10);
5 Client c4 = new Client(5, -5, 10);
6 Set<Point> ensPoint = new HashSet<>();
7 ensPoint.add(c1);
8 ensPoint.add(c2);
9 ensPoint.add(c3);
10 ensPoint.add(c4);
11 d1.ajouterRoutes(ensPoint);
12 ensPoint.add(d1);
13 c1.ajouterRoutes(ensPoint);
14 c2.ajouterRoutes(ensPoint);
15 c3.ajouterRoutes(ensPoint);
16 c4.ajouterRoutes(ensPoint);
17 Set<Client> ensClient = new HashSet<>();
```

```
18 ensClient.add(c1);
19 ensClient.add(c2);
20 ensClient.add(c3);
21 ensClient.add(c4);
22 Tournee t = new Tournee(d1, 50);
23 t.ajouterClient(ensClient);
24 System.out.println(t);
```

2.2 Planning

La brasserie que nous avons le plaisir d'aider a tellement de clients qu'un seul véhicule n'est pas suffisant pour livrer toutes les caisses de bière (car les véhicules ont une capacité limitée). On a donc besoin de créer plusieurs tournées. Pour ce faire on crée une nouvelle classe **Planning**. Ses attributs sont un ensemble de tournées, la distance totale parcourue, le nombre total de caisses de bières livrées, ainsi que la capacité des camions.

Question 24. Créez dans le paquetage **planning** une nouvelle classe **Planning** avec ses attributs et un constructeur par donnée de la capacité. Justifiez le choix de la collection que vous utilisez pour stocker les tournées. Ajoutez les accesseurs et les mutateurs nécessaires. Ajoutez une méthode **private boolean ajouterTournee(Tournee t)** qui ajoute une tournée à l'ensemble des tournées. Cette méthode mettra à jour les attributs pour la distance totale et le nombre total de caisses livrées. Ajoutez une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur le planning et une méthode principale pour tester la classe.

Question 25. Ajoutez dans votre classe **Planning** une méthode **public void planificationBasique(Depot depot, Set<Client> clients)**. Cette méthode commence par créer une tournée vide, que nous nommerons *tournée courante*. Puis, elle itère sur l'ensemble des clients et les ajoute à la *tournée courante*. Lorsque l'ajout d'un client à la *tournée courante* n'est pas possible, alors la *tournée courante* est ajoutée au planning (avec la méthode **ajouterTournee**). Puis une nouvelle *tournée courante* vide est créée et le client qui n'avait pas pu être ajouté précédemment lui est ajouté. Il suffit ensuite de répéter ce processus. On s'assure bien à la fin que la dernière *tournée courante* est ajoutée au planning.

3 Planification de la livraison

Question 26. Créez un nouveau paquetage **planification** dans lequel on aura les objets qui permettent de fabriquer les plannings de la brasserie. Dans ce paquetage, créez une nouvelle classe **ROUTAGE**. Elle aura les attributs suivants : un attribut de type **Depot**, un ensemble de clients (**mesClients**) contenant les clients à livrer ; un attribut de type **Planning**, contenant le planning de la

livraison ; et un entier représentant la capacité des véhicules (on suppose que cette capacité est identique pour tous les véhicules qui font les livraisons).

Ajoutez un constructeur par la donnée de la capacité des véhicules. Le constructeur initialise le dépôt au point de coordonnées $x = 0$, $y = 0$. Ajoutez un constructeur par la donnée d'un dépôt et de la capacité des véhicules.

Question 27. Ajoutez dans la classe **Routage** la méthode suivante qui permet d'initialiser l'ensemble des clients afin de réaliser un premier test :

```
1  private void creationClientsTest1() {
2      Client c0 = new Client(-99.7497, 12.7171, 4);
3      Client c1 = new Client(61.7481, 17.0019, 10);
4      Client c2 = new Client(-29.9417, 79.1925, 17);
5      Client c3 = new Client(49.321, -65.1784, 18);
6      Client c4 = new Client(42.1003, 2.70699, 7);
7      Client c5 = new Client(-97.0031, -81.7194, 8);
8      Client c6 = new Client(-70.5374, -66.8203, 20);
9      Client c7 = new Client(-10.8615, -76.1834, 1);
10     Client c8 = new Client(-98.2177, -24.424, 11);
11     Client c9 = new Client(14.2369, 20.3528, 13);
12     mesClients.clear();
13     mesClients.add(c0);
14     mesClients.add(c1);
15     mesClients.add(c2);
16     mesClients.add(c3);
17     mesClients.add(c4);
18     mesClients.add(c5);
19     mesClients.add(c6);
20     mesClients.add(c7);
21     mesClients.add(c8);
22     mesClients.add(c9);
23 }
```

Question 28. Écrivez une méthode **private void initialiserRoutes()** qui initialise les routes qui partent du dépôt et de chaque client. Rappelez-vous de la méthode **ajouterRoutes()** que vous avez codée dans la classe **Point**.

Question 29. Ajoutez dans la classe **Routage** une méthode **public void planificationBasique()**. Cette méthode doit faire appel à la méthode de planification basique de la classe **Planning**.

Question 30. Ajoutez une méthode statique **public static void test()**. Cette méthode crée un objet de type **Routage** (avec une capacité de 50 caisses de bière, valeur donnée par Monsieur Houblon). Puis, cette méthode fait appel à la méthode **creationClientsTest1()** puis à la méthode **initialiserRoutes()**. Enfin, on peut faire appel à la méthode **planificationBasique()** afin de planifier

les tournées pour livrer les clients de l'ensemble de test. Puis vous afficherez les informations sur le nombre de tournées, la distance totale et la quantité livrée.

Question 31. Écrivez une méthode principale (**public static void main(String[] args)**) dans la classe **Routage** dans laquelle vous faites appel à la méthode statique **test()**. Quels résultats obtenez-vous en termes de nombre de tournées et de distance totale ? Le nombre de caisses livrées doit être de 109 (si vous n'en avez pas perdu en route ...).

4 Quelques notions utiles

4.1 Table de hachage

Une table de hachage est une organisation des éléments d'une collection qui permet de retrouver facilement un élément de valeur donnée (par exemple une instance d'une classe). Pour cela, on utilise une méthode (**hashCode()**) dite "fonction de hachage" qui, à la valeur d'un élément (une instance d'une classe), associe un entier. Un même entier peut correspondre à plusieurs valeurs différentes (plusieurs instances différentes). En revanche, deux éléments de même valeur (deux instances égales selon la méthode **equals()**) doivent toujours fournir le même code de hachage.

Pour organiser les éléments de la collection, on va constituer un tableau de N listes chaînées (nommées souvent seaux). Initialement, les seaux sont vides. À chaque ajout d'un élément à la collection, on lui attribuera un emplacement dans un des seaux dont le rang i (dans le tableau de seaux) est défini en fonction de son code de hachage *code* de la manière suivante :

$$i = \text{code} \% N$$

S'il existe déjà des éléments dans le seau, le nouvel élément est ajouté à la fin de la liste chaînée correspondante. On peut récapituler la situation par le schéma de la Figure 5. Les deux éléments du seau de rang i sont donc tels que leur code de hachage modulo N est égal à i .

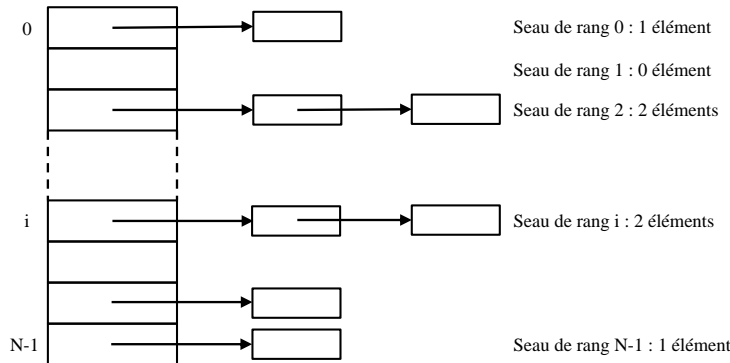


Figure 5: Présentation d'une table de hachage : un tableau de taille N dont chaque élément est une liste chaînée.

Comme on peut s'y attendre, le choix de la valeur (initiale) de N sera fait en fonction du nombre d'éléments prévus pour la collection. On nomme "facteur de charge" le rapport entre le nombre d'éléments de la collection et le nombre de

seaux N . Plus ce facteur est grand, moins (statistiquement) on obtient de seaux contenant plusieurs éléments ; mais plus ce facteur est grand, plus le tableau de références des seaux occupe de l'espace en mémoire. Généralement, on choisit un facteur de l'ordre de 0,75.

Bien entendu, la fonction de hachage joue également un rôle important dans la bonne répartition des codes des éléments dans les différents seaux. Pour retrouver un élément de la collection (ou pour savoir s'il est présent), on détermine son code de hachage *code*. La formule $i = \text{code} \% N$ fournit un numéro i de seau dans lequel l'élément est susceptible de se trouver. Il ne reste plus qu'à parcourir les différents éléments du seau pour vérifier si la valeur donnée s'y trouve (avec la méthode **equals()**). Notez qu'on ne recourt à la méthode **equals()** que pour les seuls éléments du seau de rang i . Cette remarque est très importante car cela implique que si deux éléments **e1** et **e2** sont égaux (**e1.equals(e2)** vaut **true**), alors il est nécessaire que leurs codes de hachage soient égaux (**e1.hashCode()==e2.hashCode()** vaut **true**).

Avec Java, les tables de hachage sont automatiquement agrandies dès que leur facteur de charge devient trop grand (supérieur à 0,75). On retrouve là un mécanisme similaire à celui de la gestion de la capacité d'un tableau dynamique. Certains constructeurs d'ensembles permettent de choisir la capacité et/ou le facteur de charge.

References

- [1] Delannoy, C., *Programmer en Java*, Eyrolles, 2014