Ateliers java - séance 8

Objectifs :

→ Etre capable de sérialiser
→ Comprendre comment implémenter un Assembly

Thèmes abordés :

- Sérialisation

Importez le projet *AJ_seance8*.

Afin de gérer son système de billetterie, une compagnie de chemins de fer décide de créer 5 classes en java : `Trajet`, `Tarif`, `ListeTrajets`, `ListeTarifs` et `Billet` dans le package `tarifs`.

La classe `Trajet` possède 2 `String` `gareDepart` et `gareArrivee` et un `double` `distance` entre les 2 gares. Observez cette classe et complétez là en lisant la suite de ce paragraphe. L'ordre "naturel" des `Trajets` est l'ordre alphabétique sur `gareDepart`, puis sur `gareArrivee`. Un trajet est le même qu'un autre si les gares départs sont les mêmes ainsi que celles d'arrivées.

La classe `ListeTrajets` possède une collection de `Trajets` existants. Cette collection doit être triée par trajet. Elle implémente les méthodes suivantes de façon intuitive :

```
    double getDistance(String gareDepart, String gareArrivee) throws
TrajetInconnuException;

    boolean contient(Trajet t);

    boolean ajouterTrajet(Trajet t);

    boolean supprimerTrajet(Trajet t);

    Iterator<Trajet> trajets();

    Iterator<String> garesDArrivee();

    String toString();
```

La classe `Tarif` définit un `enum` `TypeReduction` dont les constantes sont `TARIF_PLEIN, FAMILLE_NOMBREUSE, SENIOR, VIPO, WEEK_END.` Un tarif possède trois attributs : `priseEnCharge`, `prixAuKilometre` de type `double` et une collection (à définir) qui associe un coefficient réducteur à chacun des types de réduction existants. Ce coefficient doit être strictement supérieur à 0 et inférieur à 1. Ces coefficients sont initialisés à 1 dans le constructeur et peuvent être ensuite modifiés via la méthode `void definirReduction(TypeReduction r, double coeff)`. Les accesseurs pour les attributs primitifs sont définis. La méthode `getReduction` permet d'obtenir le coefficient pour le type de réduction passé en paramètre. La méthode `toString` affiche par exemple :
"Prise en charge : 5 euros
Prix au km : 1.2 euros
TARIF_PLEIN : 1
FAMILLE_NOMBREUSE : 0.8 …"

La classe `ListeTarifs` possède une collection (à définir) de `tarifs` qui stocke les `Tarifs` au cours du temps en associant à chacun sa date de prise d'effet. Cette collection doit être triée par date de prise d'effet. Elle propose une méthode `Tarif getTarif(LocalDate date)` qui permet de connaître le `Tarif` en vigueur à une date donnée (null si aucun), ainsi qu'une méthode `double getPrix(Trajet trajet, TypeReduction typeReduction, LocalDate dateVoyage)` qui calcule le prix d'un trajet à une date donnée

pour une réduction donnée ; il s'agit de sommer le tarif de prise en charge avec le prix relatif à la distance parcourue en prenant en compte la réduction (la réduction ne s'applique pas sur la partie prise en charge). Si le tarif n'est pas défini, il faut lancer une `TarifNonDisponibleException.` Les méthodes suivantes doivent être implémentées :

- `boolean modifierTarif(LocalDate date, Tarif nouveauTarif) throws DateNonAutoriseeException` qui lance l'exception si la date en paramètre est antérieur ou égale à la date du jour. Elle renvoie faux si cette date ne possède pas déjà de tarif défini et vrai sinon.
- `void creerTarif(LocalDate date, Tarif tarif) throws DateNonAutoriseeException, DateDejaPresenteException` qui lance `DateNonAutoriseeException` si la date en paramètre est antérieur ou égale à la date du jour. Elle lance `DateDejaPresenteException` si un tarif est déjà défini à cette date. Si tout va bien elle ajoute un nouveau tarif.
- `toString` pour afficher les différents tarifs.

La classe `Billet` possède un `Trajet`, un `LocalDate dateVoyage`, un `TypeReduction typeReduction` et un `double prix`. Les accesseurs sont définis. Deux constructeurs sont définis : l'un prend le trajet, le type de réduction et la date en paramètre et l'autre prend uniquement le trajet et le type de réduction en paramètre considérant la date du jour pour date du voyage. Le prix se calcule sur base de ces paramètres. La méthode `toString` renvoie :

```
"Voyage de Wavre à Mons
Distance : 76 km – TARIF_PLEIN
Prix : 12 euros"

Ou
"Voyage de Wavre à Mons
Distance : 76 km – Réduction : SENIOR
Prix : 10 euros"
```

Les paramètres sont testés avec l'interface `Util`.

1) Les objets possédés par `ListeTarifs` et `Billet` sont tous des attributs. Modifiez le code pour en tenir compte. Attention, la méthode `clone()` de `Tarif` ne peut pas se contenter d'appeler `super.clone()`.

2) Utilisez un assembleur (`Assembly` – confer ci-dessous) pour construire votre application. En particulier, l'utilisation de cet assembleur fait en sorte que les classes `ListeTrajets` et `ListeTarifs` sont traitées comme des singletons.

3) La méthode `Tarif getTarif(LocalDatedate)` de `ListeTarifs` renvoie le `Tarif` en vigueur lors de la `date`. Pour l'instant, elle le fait en parcourant les dates dans l'ordre croissant, qui est l'ordre naturel défini par la classe `LocalDate`. Modifiez le code afin que cette méthode parcoure les dates dans l'ordre décroissant pour la détermination de la clé de la `SortedMap`. Pour cela, on peut utiliser la méthode `reverseOrder` de `Collections`. Il faut passer le `Comparator` renvoyé par cette méthode au constructeur de la `TreeMap<LocalDate, Tarif>`. Il faut aussi modifier la méthode `getTarif(...)` pour tenir compte du changement.

4) Lorsque des changements sont apportés à la collection des `Trajets` ou à celle des `Tarifs`, il faut pouvoir en tenir compte dans chacun des points de vente de `Billets`. Il faut donc pouvoir sérialiser et désérialiser ces 2 collections. Pour cela, il faut :
    a. Rendre sérialisables les classes `Trajet` et `Tarif` ;
    b. La classe `ListeTrajets` doit contenir une méthode `boolean serialize(String nomFichier)` qui permet de sauvegarder le `SortedSet<Trajet>` dans un fichier. Cette méthode renvoie `false` s'il existe déjà un fichier de ce nom non accessible en écriture. Pour cela, vous devez d'abord créer un `Path` correspondant au fichier comme expliqué dans la théorie.
    c. De manière similaire au point ci-dessus, la classe `ListeTarifs` doit contenir une méthode `boolean serialize(String nomFichier)` qui permet de sauvegarder la `SortedMap<Calendar, Tarif>` dans un fichier.

d. Les classes `ListeTrajets` et `ListeTarifs` doivent aussi contenir une méthode `static boolean deserialize(String nomFichier)` qui permet de récupérer la collection sauvegardée dans un fichier. Cela se fera via la méthode `readObject(...)` d'un `ObjectInputStream` construit sur base d'un `InputStream`. Cette méthode se charge d'initialiser l'objet correspondant de l'`Assembly`. Cette méthode renvoie `false` si le fichier n'existe pas ou s'il n'est pas disponible en lecture.

e. Finalement, déclarez un nom de fichier à utiliser par défaut en champ static dans la classe ListeTrajets et écrivez des méthodes `boolean serialize()` et `static boolean deserialize()`qui permettent, respectivement, de sauvegarder dans ce fichier par défaut et de récupérer les trajets à partir du fichier par défaut. Faites de même pour la classe `ListeTarifs` !.

Assembly d'une application Java

Toute application Java se doit d'être JUnit testable. Ceci pose une contrainte sur la manière dont on peut se permettre d'écrire une application. Regardons par exemple le singleton classique :

```
public class MaClasseQuiOuvreUneConnexionBD {

     public static final INSTANCE = new MaClasseQuiOuvreUneConnexionBD();
     private Connection con;

     MaClasseQuiOuvreUneConnexionBD() {
          this.con=…
     }

     …

}
```

Il sera très difficile de tester par JUnit tout code qui utilisera `MaClasseQuiOuvreUneConnexionBD`. En effet, chaque fois que l'on écrit `MaClasseQuiOuvreUneConnexionBD.INSTANCE`, on se retrouve avec une instance qui ouvre cette connexion à la base de données de production. Le problème ne réside pas dans le fait que l'on utilise un objet concret plutôt qu'une interface. En effet, il est très simple de définir un objet `MockMaClasseQuiOuvreUneConnexionBD` qui `extends` `MaClasseQuiOuvreUneConnexionBD` et qui `override` toutes les méthodes y compris le constructeur pour obtenir une classe qui évite d'ouvrir une connexion BD avec le comportement souhaité pour le JUnit test. Le problème tient dans le fait qu'il est écrit `new MaClasseQuiOuvreUneConnexionBD()` directement dans le code, et que, de ce fait, on est restreint à une instance de ce type particulier sans possibilité d'y mettre une instance de `MockMaClasseQuiOuvreUneConnexionBD` à la place.

Pour éviter cet écueil, on va sortir toutes les opérations `new` concernant les types spécifiques de l'application. On va les placer dans ce qu'on appelle un **assembleur**. L'idée est d'utiliser des assembleurs différents lorsque l'on est en production et lorsque l'on fait des tests unitaires.

L'assembleur est donc une classe, dont les méthodes publiques retournent des instances des entités de l'application. L'assembleur lui-même devant être facilement mockable pour les tests unitaires, ses méthodes publiques ne sont pas statiques de manière à permettre une redéfinition partielle ou complète par héritage. Dans le restant du code, chaque fois que l'on faisait un `new` d'une entité de l'application, il faut maintenant faire un appel à l'assembleur, plus exactement à l'instance de l'assembleur puisque les méthodes ne sont pas statiques. Pour éviter de devoir se propager une référence à cette instance partout dans le code, l'assembleur définit une méthode statique qui retourne l'instance, à la manière d'un singleton classique. Cependant l'assembleur lui-même devant être mockable, on ne peut pas l'écrire complètement comme un singleton classique puisqu'on se retrouverait dans le cas décrit ci-dessus. À la place, on considère que l'assembleur doit être instancié une fois au démarrage de l'application, et que c'est le constructeur de l'assembleur qui définit alors l'instance statique.

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

```
    public static Assembly getInstance() {
        return instance;
    }

    public Assembly() {
        Assembly.instance=this;
    }
    …}
```

Et dans le `main` on commence par :

```
new Assembly();
```

Chaque `new` d'une entité de l'application est remplacé par un appel à une méthode de l'assembleur :

```
Assembly.getInstance().appelMethode()
```

Si on veut créer un autre assembleur pour un test unitaire, on crée un mock en étendant cette classe et en redéfinissant ses méthodes :

```
public class MockAssembly extends Assembly {
    public MockAssembly() {
        super();
    }

    @Override
    …
}
```

Le test unitaire est simplement initialisé par :

```
new MockAssembly();
```

Il existe deux types de méthodes de base pour un assembleur :

1. Celles qui retournent des singletons
2. Celles qui retournent des nouvelles instances

Les singletons sont implémentés par l'usage : chaque appel de méthode retourne toujours le même objet. Ceci est trivialement réalisé par des attributs :

```
public class Assembly {
    private MonObjet monObjet;
    private static Assembly instance;

    public static Assembly getInstance() {
        return instance;
    }

    public Assembly() {
        Assembly.instance=this;
        this.monObjet=new MonObjet();
    }

    public MonObjet getMonObjet () {
        return monObjet;
    }
}
```

Les nouvelles instances sont implémentées sous forme de factory :

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

     public static Assembly getInstance() {
          return instance;
     }

     public Assembly() {
          Assembly.instance=this;
     }

     public MonAutreObjet createMonAutreObjet (String param) {
          return new MonAutreObjet(param);
     }
}
```