Association et hiérarchie de classes

On s'intéresse ici à quelques éléments permettant de définir et planifier des voyages itinérants. On ne s'intéresse ici pour les voyages qu'aux trajets à effectuer, et aux hébergements.

Exercice I *Des trajets*

Pour des raisons de simplification, on se limite ici à des trajets ayant lieu sur une même journée calendaire : pas question de prendre un ferry un soir et d'arriver le lendemain, ni de prendre un train de nuit.
Un trajet a donc lieu à une date, et a un lieu de départ et un lieu d'arrivée (pour simplifier ici, une chaîne de caractères correspondant à un nom de ville ; pour simplifier, on ne traite pas ici des trajets avec même ville d'arrivée que de départ). Chaque trajet a lieu avec un moyen de transport, parmi : voiture, marche, vélo, train, bus.
Les trajets en train et en bus ont une heure de départ et une heure d'arrivée, et un prix déterminé à l'achat du billet de transport.
Pour tous les trajets, on peut connaître leur durée, et leur prix :

- Pour les trajets en train et en bus, la durée se calcule à partir des horaires, et le prix est fixé à l'achat du titre de transport.
- Pour les trajets hors transports en commun (voiture, marche et vélo),
  - la durée peut se calculer via différentes API plus ou moins libres (nous utiliserons ici la classe `OpenMoogleGap` qui est censée interroger un service web pour obtenir ces informations)
  - et le prix se calcule également :
    - le prix pour les marches et les trajets en vélo est calculé par `OpenMoogleMap` (en général le prix est nul, mais le trajet contient parfois des passages de zones payantes, ou l'utilisation de transports comme le bac pour traverser l'estuaire de la Gironde)
    - le prix pour les trajets en voiture se calcule comme la somme du prix de l'essence consommée, et du prix du péage ; `OpenMoogleMap` permet de réaliser les calculs adéquats).

Vous trouverez en fin d'énoncé quelques éléments utiles sur la gestion des dates en Java. Vous trouverez sur moodle la fameuse classe `OpenMoogleGap` qui n'est pas encore implémentée, mais vous supposerez qu'elle l'est !
Pour cet exercice, on placera tous les éléments créés dans un package `trajets`.

Question I.1. Mettez en place (UML et Java) une classe abstraite `Trajet` avec la date du trajet, le type de transport utilisé, un constructeur paramétré, un `toString` pratique à utiliser à des fins de test, et deux méthodes abstraites : `duree` et `prix`, qui retournent respectivement la durée du trajet et le prix du trajet.

Question I.2. Mettez en place la sous-classe de `Trajet` `TrajetTransportEnCommun` (pour les trajets en transport en commun). Un trajet en transport en commun a un horaire de départ et un horaire d'arrivée, il a de plus un prix. Implémentez le constructeur paramétré adéquate, et les deux méthodes `duree` et `prix`.

Question I.3. Mettez en place la sous-classe de `Trajet` `TrajetLibre` (pour les trajets hors transports en commun). Mettez en place le constructeur paramétré adéquat, et implémentez les deux méthodes `duree` et `prix` en utilisant la magnifique classe `OpenMoogleGap`. Cette classe devrait appeler un service web mais répond ici toujours la même information "en dur". Pourquoi ne crée t-on pas d'association (et donc d'attribut au niveau Java) entre `TrajetLibre` et `OpenMoogleMap` ?

Question I.4. Qu'a de particulier la classe `OpenMoogleMapsTravelInfo` ? Etudiez la notion de Record en Java (les étudiants disposant d'une version de Java suffisante peuvent faire quelques essais).

Question I.5. Mettez en place la sous-classe de `TrajetLibre` `TrajetVoiture`. On spécialisera la méthode de calcul du prix pour y ajouter le prix de l'essence, en considérant une moyenne de consommation de 7l pour 100 kms, et un prix de l'essence à 1,97 euros le litre.

Exercice II *Des voyages*

Question II.1. Modélisez et mettez en place en Java la classe `Voyage`, dont une instance représente un voyage itinérant. Un voyage contient une séquence de trajets ordonnés par ordre chronologique. Pour simplifier, on ne considère qu'un trajet par jour au maximum. On peut ajouter un trajet à une séquence (sans vérification particulière, on veillera juste à maintenir l'ordre chronologique). Pour un voyage, on peut calculer les éléments suivants :

- la date de début du voyage
- la date de fin du voyage
- le nombre de trajets du voyage
- la durée totale des trajets (question subsidiaire : si la méthode duree n'est pas présente dans la classe `Trajet`, votre méthode de calcul de durée totale fonctionne-t-elle toujours ?)
- la liste des lieux d'étape, sans doublons, triée par ordre alphabétique (pour trier la liste, on utilisera la méthode de classe `sort` de la classe `Collections`)
- le booléen indiquant si une date prise en paramètre est dans le voyage (inclus entre le début et la fin du voyage)
- la ville étape de la date donnée en paramètre. Si la date donnée en paramètre correspond à une date où il y a un trajet, la ville étape est celle de destination du trajet. Sinon la ville étape est la destination du trajet ayant lieu juste avant la date.

Question II.2. Un voyage référence également des hébergements.

- Un hébergement a un nom (par exemple `hôtel du Capitole`), un lieu (pour simplifier uniquement un nom de ville, par exemple Toulouse), une date d'arrivée et une date de départ.
- On peut calculer le nombre de nuitées de l'hébergement
- On peut savoir si une date donnée est incluse ou pas dans la période d'hébergement.

Modélisez et implémentez en Java les hébergements, ainsi que le fait qu'une instance de voyage maintient sa séquence d'hébergements, triée par ordre chronologique de date d'arrivée.

- Prévoyez de quoi ajouter un hébergement à un voyage (on ne garantira pas pour l'instant qu'il n'y a pas plus d'un hébergement par jour, autrement dit les nuitées des hébergements ne se chevauchent pas ; on garantira juste que les hébergements sont ordonnés par date d'arrivée croissante).

Dans la suite on supposera que les hébergements n'ont pas de recouvrement, bien que cela ne soit pas assuré par la méthode d'ajout d'hébergement.

Question II.3. Écrivez dans la classe `Voyage` une méthode qui retourne l'hébergement pour une date donnée (et null si aucun hébergement prévu à cette date, ou si la date est en dehors de la période du voyage).

Question II.4. Écrivez dans la classe `Voyage` une méthode permettant de vérifier qu'un voyage a des hébergements cohérents avec les trajets : pour toute la durée du voyage, il y a un hébergement chaque jour au bon endroit.

- Cette méthode retournera une liste d'erreurs, chaque erreur étant définie par une date, et un motif d'erreur parmi les deux motifs suivants : hébergement au mauvais endroit ou absence d'hébergement.
- On utilisera un algorithme simpliste. Pour toutes les dates entre le début et la fin du voyage, on regarde la ville étape, et on vérifie qu'à cette date, il y a un hébergement adéquate dans cette ville étape. On peut ne pas s'intéresser à la date de fin de voyage, puisque le voyage est fini ...

Exercice III *Questions facultatives*

Question III.1. Rajoutez à la méthode précédente la détection des erreurs d'hébergement liées aux hébergements en dehors de la période de voyage.

Question III.2. Écrivez dans la classe `Voyage` une méthode permettant de vérifier que la séquence des trajets est cohérente du point de vue des dates et des lieux : en maintenant la sequence par ordre chronologique, le trajet à l'étape n+1 a pour départ l'arrivée du trajet à l'étape n. Cette méthode retourne la liste des dates de trajets incohérents (dates pour lesquelles un trajet est prévu à partir d'un lieu non atteint lors du trajet précédent).

Question III.3. Écrivez une méthode dans la classe `Voyage` retournant la liste des erreurs détectées pour ce voyage (erreurs d'hébergement ou erreur de trajet), chaque erreur étant définie par une date, et un motif d'erreur parmi les quatre motifs suivants : hébergement au mauvais endroit, absence d'hébergement, hébergement hors voyage lieu du trajet de départ incohérent.

Question III.4. Reprenez la méthode `ajoutHebergement` afin de garantir qu'il n'y a pas plus d'un hébergement par jour, autrement dit les nuitées des hébergements ne se chevauchent pas

Question III.5. L'accès à `OpenMoogleGap` est très fréquent, et néanmoins probablement coûteux. On y accède systématiquement pour calculer les données sur les trajets, afin, d'avoir un calcul prenant en compte les conditions de circulation les plus récentes. Proposez une solution pour ne pas y accéder plus d'une fois par jour.

Question III.6. Affinez les trajets en voiture :
- le trajet est lié à une voiture, dont on définit les caractéristiques de carburant et de consommation, éléments que l'on prend en compte dans les calculs, on peut aussi réfléchir aux carburants alternatifs ...
- Vérifier que lors d'un voyage, on n'abandonne pas la voiture en cours de route et que plus globalement, il n'y a pas de trajet en voiture au départ d'un point A si la voiture n'est pas arrivée en A. Pour cela, pour faire simple, on extrait la liste des trajets en voiture ordonnée par ordre chronologique, et on vérifie que le départ de l'étape n+1 est au même endroit que l'arrivée de l'étape n.

Question III.7. Réfléchissez à la prise en compte des trajets avec départ et arrivée au même endroit.

Quelques éléments sur les dates en Java,
reportez-vous à la documentation de l'API pour plus de détails

- Pour représenter une date, on utilisera une instance de la classe `LocalDate`, du package `java.time` (classe présente depuis Java 8 [1]).
- Pour obtenir une instance de `LocalDate`, on utilisera la méthode de classe (`static`) `now` s'il s'agit de la date du jour, ou l'une des surcharges `of` sinon.
- Pour représenter un horaire, on utilisera une instance de la classe `LocalTime`, du package `java.time`.
- Pour obtenir une instance de `LocalTime`, on utilisera la méthode de classe (`static`) on utilisera l'une des surcharges parmi les méthodes `of`.
- Pour calculer la durée entre deux horaires, on utilisera la méthode `until` de la classe `LocalTime`, qui prend en paramètre un autre horaire (postérieur au premier si on veut une durée positive) et une unité temporelle (puisqu'on veut la durée en minutes, on choisira `ChronoUnit.MINUTES`). N'hésitez pas une fois encore à aller lire la documentation qui est limpide.

1. Globalement le package `java.time` est arrivé avec Java 8, et a considérablement simplifié la gestion du temps avec Java.

— Pour savoir si une date est avant une autre, on utilisera la méthode `isBefore` de la classe `LocalDate`, qui prend une date en paramètre et retourne vrai si et seulement si la date receveur est strictement antérieure à (strictement avant) la date reçue en paramètre. Évidemment la méthode `isAfter` existe aussi ...
— Pour obtenir toutes les dates comprises entre une date (incluse) et une autre (exclue [2]), on utilise depuis Java 11 la méthode `dateUntil` de la classe LocalDate, qui prend en paramètre une date, et qui retourne les dates comprises entre la date receveur et la date paramètre (exclue). Ces dates sont retournées sous formes de flux, que l'on transformera en liste grâce à la méthode `toList()`. Les plus chanceux d'entre vous étudieront les flux Java au prochain semestre.

2. On note avec gourmandise qu'on écrit incluse et exclue.