Polytech Lille IS4
Contrôle de Programmation Par Objets

7 janvier 2021

Tous documents papiers autorisés. Lire le sujet entièrement avant de répondre aux questions. Répondre en respectant l'ordre des questions et en rappelant leur numéro. Programmer vos solutions en Java. Vous pouvez utiliser ce qui a été spécifié dans une question même si vous n'y avez pas répondu. Ne vous préoccupez pas des packages ni des `import` *de bibliothèques, et il ne doit pas être question de quelconque* `main`*.*

On s'intéresse au développement par objets d'un logiciel de gestion de parkings payants. Un parking est constitué d'emplacements réservés appartenant à des résidents et offre un certain nombre d'emplacements banalisés que des visiteurs (abonnés, occasionnels, ...) peuvent occuper passagèrement (en payant le temps d'occupation). Il est équipé de bornes automatiques à ses passages d'entrée et de sortie pour supporter sa gestion.

1 Configuration de base

On part du diagramme de classes UML de la Figure 1 :

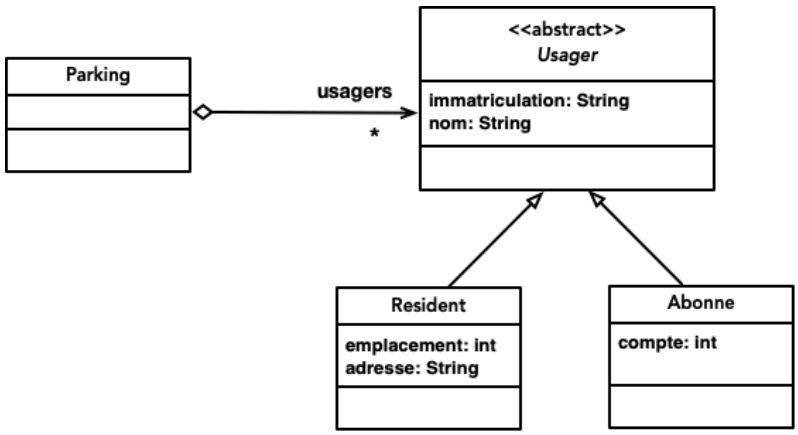

FIGURE 1 – Parking

Un `Parking` a des *`Usager`* réguliers connus du système : association de rôle `'usagers'` entre les classes `Parking` et *`Usager`*.

1.1 Hiérarchie de classes *Usager*

Usager est la sur-classe (abstraite) des véhicules connus du `Parking` :

- soit parce qu'ils appartiennent à des résidents (sous-classe `Resident`) avec un numéro d'`emplacement` réservé et l'adresse du résident
- soit (sous-classe `Abonne`) parce qu'ils ont un abonnement pour occuper le parking à hauteur d'un `compte` d'heures pré-payées (qui se décrémente à chaque occupation)
- ou pour tout autre cas (cette hiérarchie de classes n'est pas exhaustive et doit rester extensible).

Tout *Usager* est décrit par son numéro de plaque d'`immatriculation` et le `nom` de son propriétaire.

Question (3 points)

Programmer cette hiérarchie de classes avec leurs attributs.

Précision : il n'est pas demandé de programmer de méthodes ni de constructeurs à ce niveau.

1.2 Parking

On choisit de représenter l'association de rôle `usagers` entre `Parking` et *Usager* par une Map : $immatriculation \rightarrow Usager$ qui doit permettre leur accès ordonné par `immatriculation`.

1.2.1 Question (1 point)

Déclarer cette Map dans la classe `Parking`.

1.2.2 Question (5 points)

Programmer les méthodes nécessaires pour afficher tous les *usagers* du parking, ordonnés par numéro d'immatriculation et devant chaque numéro :

- pour un résident : la chaîne `"résident"`, son `nom` et son numéro d'`emplacement` réservé
- pour un abonné : la chaîne `"abonné"`, son `nom` et son `compte` d'heures.

1.2.3 Question (3 points)

Programmer par streaming et lambdas une méthode `'int abonnementsEpuises()'` dans la classe `Parking` qui retourne le nombre d'`Abonne` dont le `compte` d'heures est tombé à 0.

2 Occupation d'un parking

Un parking a des emplacements réservés aux résidents (classe `Resident` précédente) et un certain nombre d'emplacements libres qui peuvent être utilisés par les abonnés (classe `Abonne` précédente) ou par tout autre véhicule moyennant paiement. Le nombre d'emplacements libres est géré au travers d'une variable `'int nbLibres'` à ajouter à la classe `Parking`.

Un parking a des passages d'entrée et de sortie équipés de bornes optiques représentées par la classe `BorneOptique` donnée ci-dessous (il n'est pas demandé de la programmer), capables de "scanner" les plaques d'immatriculation :

```
class BorneOptique {
   String scan() throws IllisibleException
}
```

La méthode `'scan()'` retourne le numéro d'immatriculation du véhicule ou provoque une exception `IllisibleException` si la plaque n'a pu être "scannée".

On représente les passages par la hiérarchie de classes suivante :

```
abstract class Passage {
   Parking park; // le parking
   BorneOptique borne; // la borne optique du passage
}

class PassageEntree extends Passage {
   void entree() ...
 }
```

développée dans la question 2.1 ci-dessous.

```
class PassageSortie extends Passage {
   String identificationManuelle()
   double sortie() ...
}
```

développée dans la question 2.2 ci-dessous.

2.1 Entrée de véhicules

Le parking enregistre dans une `Map` *horaires* : *immatriculation* → *Horaire* les horaires d'entrée des véhicules occupant le parking :

```
class Parking {
   Map<String, Horaire> horaires;
   ...
}
```

La classe `Horaire` est donnée ci-dessous (il n'est pas demandé de la programmer) :

```
class Horaire {
   static Horaire now() // horaire actuel
   int diff(Horaire x) // nombre d'heures écoulées entre this et x
}
```

Question (4 points)

Programmer la méthode `'void entree() ...'` de la classe `PassageEntree` qui :

- "scanne" par sa borne optique la plaque d'immatriculation du véhicule se présentant
- propage l'exception `IllisibleException` si la plaque n'a pu être "scannée" et l'opération est terminée
- sinon fait appel à une méthode `'void entreeVehicule(String immatriculation)...'` à programmer dans la classe `Parking` pour :
  - vérifier que le véhicule peut entrer :
    - c'est toujours possible pour un résident (place réservée)
    - dans tous les autres cas il faut vérifier qu'il reste des places libres
    - s'il n'y en a pas une exception `FullException` sera provoquée et propagée par le passage d'entrée
  - si tout s'est bien passé, enregistrer l'`immatriculation` avec son horaire d'entrée (utiliser la méthode `now()` de la classe `Horaire`) dans la `Map` *horaires* et mettre à jour le nombre de places libres, le cas échéant.

2.2 Sortie de véhicules

Question (4 points)

Programmer la méthode `'sortie()'` de la classe `PassageSortie` qui :

- "scanne" par sa borne optique la plaque d'immatriculation du véhicule se présentant
- si la plaque n'a pu être "scannée" (`IllisibleException`, il se peut en effet qu'entre temps la plaque d'immatriculation soit devenue illisible depuis l'entrée du véhicule), faire appel à la méthode `'String identificationManuelle()'` de la borne. Cette méthode est donnée (il n'est pas demandé de la programmer), elle retourne le numéro d'immatriculation du véhicule par appel au gardien du parking
- retourne le montant dû par appel à une méthode
  `'double sortieVehicule(String immatriculation)'`
  à programmer dans la classe `Parking` qui :
  - retourne le montant dû :
    - pour un `Resident` : 0
    - pour un `Abonne` : le nombre d'heures d'occupation est décompté de son `compte` d'heures si celui-ci est suffisant. Sinon le complément d'heures $\times$ $TARIF\_HORAIRE$ est dû ($TARIF\_HORAIRE$ étant le tarif horaire à définir en `static` dans la classe `Parking`).
    - pour toute autre occupation : $TARIF\_HORAIRE$ $\times$ nombre d'heures écoulées entre ses horaires d'entrée et de sortie.
  - supprime l'immatriculation de la `Map` *horaires* et met à jour le nombre de places libres le cas échéant.