

# TD3 : Des classes aux interfaces

<b>I.</b>	<b>Objectifs du TD</b>	<b>1</b>
<b>II.</b>	<b>Point sur le précédent TD</b>	<b>2</b>
<b>III.</b>	<b>Utilisation des interfaces</b>	<b>2</b>
III.1	Une mini-bibliothèque de code qui « expose » des interfaces : un système de location	2
III.2	Implémenter des interfaces : des véhicules empruntables	3
III.2.1	Implements : des voitures électriques louables (ElectricCar)	3
III.2.2	Variable statique et constructeur pour attribuer un identifiant aux vélos	5
III.2.3	Utilisation des interfaces : un système de location de voitures et de vélos	6
III.2.4	Bilan	6
<b>IV.</b>	<b>Construction d'interfaces</b>	<b>8</b>
IV.1	Extraire des interfaces : des véhicules chargeables	8
IV.1.1	Extraire une interface : des objets chargeables	8
IV.1.2	Implémenter une interface : des véhicules chargeables	8
IV.2	Produire une interface : des objets traçables	8
IV.2.1	Des objets traçables	8
IV.2.2	Implémentation du Système de Suivi	9
IV.2.3	Utilisation de votre interface : Des drones traçables	9
<b>V.</b>	<b>Conclusion</b>	<b>10</b>

## I. Objectifs du TD

1. Comprendre les interfaces dans leur usage et en étant capable d'en produire
2. Manipuler des codes plus gros mais bien structurés.

Dans ce TD, vous devez aussi essayer de gagner en autonomie. Les spécifications vous sont données. Vous essayez de faire et si vous rencontrez des difficultés vous suivez les guides.

Dans ce TD, nous allons mettre en place un système de location de véhicules électriques très simplifiés. Ainsi les véhicules électriques deviennent « louables », avec des spécificités pour chaque sorte de véhicule électrique.

Dans ce même TD nous identifions les objets chargeables. Puis, vous créez vous-même un système de traçage, hyper simplifié.

## II. Point sur le précédent TD

Il est grandement préférable que vous utilisiez vos propres codes, quitte à les modifier ou les compléter. Néanmoins vous trouverez [ICI](#) des codes qui correspondent au TD2.

## III. Utilisation des interfaces

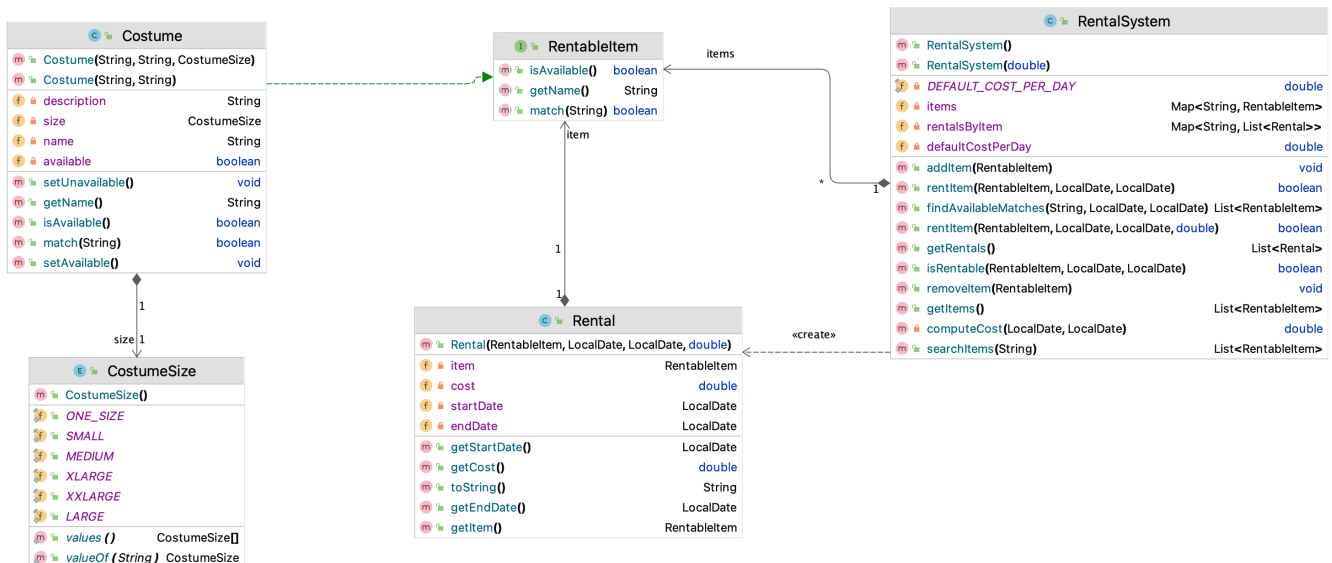
### III.1 Une mini-bibliothèque de code qui « expose » des interfaces : un système de location

Sous TD3/CODES DONNES vous chargez les codes associés ou sous forme d'archive [ici](#)

Q.1. Chargez ces codes sur votre machine en respectant les packages et testez-les. Ils doivent s'exécuter sans problème.

Vous trouverez dans cette bibliothèque de codes, un système (*RentableSystem*) qui prend en charge des locations d'items louables (*RentableItem*). Ce système gère des locations (*Rental*). Un exemple de *RentableItem* vous est donné dans le package *costumes* qui définit des déguisements (*Costume*) comme des objets louables. La classe *RentalSystemTest* comporte les tests sur ce système.

- Q.2. Étudiez l'interface *RentableItem*. Quelles méthodes doit-elle implémenter une classe qui implémente cette interface ?
- Q.3. Étudiez la classe *Costume* qui implémente *RentableItem*.
- Q.4. Quelle relation relie la classe *Rental* et l'interface *RentableItem* ? Est-il possible qu'une instance de *Rental* fasse référence à une instance de *Costume* ?
- Q.5. Prenez le temps d'au moins identifier les méthodes exposées (public) par la classe *RentalSystem*.



## III.2 Implémenter des interfaces : des véhicules empruntables

On désire que les voitures et les vélos électriques soient empruntables, mais pas les drones.

### III.2.1 **Implements** : des voitures électriques louables (*ElectricCar*)



Attention tous vos tests sur les voitures électriques doivent continuer à fonctionner. Il ne doit donc pas y avoir de **régression** par rapport à vos codes précédents.

#### Spécifications pour des voitures louables :

- a) Une voiture est disponible à la location à sa construction.
- b) Il est néanmoins possible de la rendre indisponible par exemple lorsqu'elle a été accidentée.
- c) On identifie une voiture par sa plaque d'immatriculation.
- d) On considère qu'une voiture matche une description si
  - son modèle contient la description : par exemple si on recherche une voiture avec pour description « Tesla », toutes les voitures qui ont « Tesla » dans leur modèle correspondront à la description ;
  - Ou si sa plaque d'immatriculation contient la description : par exemple « 06 » matche avec toutes les voitures qui ont « 06 » dans leur plaque.

Si vous êtes débutant et que cette spécification ne vous suffit pas, laissez-vous guider par les questions qui suivent. **Si vous savez quoi faire, sautez à la question Q.11**

Q.6. Commençons par ajouter à la définition de la classe *ElectricCar* **implements RentableItem**

```
public class ElectricCar extends ElectricVehicle implements RentableItem {
```

- Pourquoi avez-vous une erreur ? que dit-elle ? Que manque-t-il à la classe *ElectricCar* pour qu'elle implémente effectivement *RentableItem* ?
- Le système vous propose d'implémenter les méthodes manquantes.
- Il peut générer le squelette de ces méthodes mais pas leur contenu. Laissez-le faire puis passez à la question suivante qui vous guide dans ce que nous voulons faire.

5 usages

```
public class ElectricCar extends ElectricVehicle implements RentableItem {
```

Class 'ElectricCar' must either be declared abstract or implement abstract method 'isAvailable()' in 'RentableItem'

Implement methods ↗ ↘ ↻ More actions... ↗ ↘

fr.epu.rentals

```
public interface RentableItem
```

Interface which defines the methods that a rentable item must implement.

Q.7. Le système a ajouté des méthodes :

- Est-ce bien celles que vous aviez identifiées à la question Q.2 ?  
Il y en a 3 ; ce sont les méthodes requises par l'interface *RentableItem*

- Q.8. Reprenez la spécification précédente pour implémenter la disponibilité (*isAvailable*). A nouveau si la spécification ne vous suffit pas, laissez-vous guider, sinon essayez de faire seul.
- Vous définissez un attribut *isAvailable* de type boolean
  - Vous implémentez *boolean isAvailable()* comme *return isAvailable* ;
  - La spécification a) nous dit que cet attribut est initialisé à *true* à la construction de l'objet => mettez à jour le constructeur
  - La spécification b) nous dit que l'on peut modifier cet état dynamiquement => ajoutez un ou des accesseurs en écriture.
- Q.9. Reprenez la spécification précédente pour implémenter l'obtention du nom (*getName*)
- La spécification c) nous dit que l'identifiant est simplement l'immatriculation donc implémenter *getName* en retournant la plaque d'immatriculation
- Q.10. Reprenez la spécification précédente pour implémenter *match*
- La spécification d) nous donne les informations. Vous implémentez *match* un peu comme le match dans *Costume* par un *contains* sur le modèle ou sur la plaque, qui sont deux objets de type String
- Q.11. Tester bien évidemment votre code, en enrichissant votre classe de test (*ElectricCarTest*) ou en créant une autre au choix. Attention vous ne testez que les méthodes que vous avez implémentées à cette étape.

Par exemple

```
final String licensePlate = "AB-123-CD";
final String brand = "Tesla";
final String model = "Model S";

@Test
void testMatch() {
    car = new ElectricCar(batteryCapacity, licensePlate, brand + " " + model);
    assertTrue(car.match("AB-123-CD"));
    assertFalse(car.match("AB-123-CE"));
    assertTrue(car.match("123"));
    assertTrue(car.match("Tesla"));
    assertTrue(car.match("S"));
}
```

### III.2.2 Variable statique et constructeur pour attribuer un identifiant aux vélos

Q.12. Implémentez la spécification ci-dessous pour pouvoir louer les vélos électriques, qui deviennent donc « Rentable ».

#### Spécifications pour des vélos louables

- a) Un vélo est disponible à la location à sa construction.
- b) Il est néanmoins possible de le rendre indisponible par exemple lorsqu'il a crevé (un simple boolean *isAvailable* à vrai ou faux suffit).
- c) On identifie un vélo par un identifiant unique que l'on crée à la construction de l'objet. Il commence par *EB-* puis un numéro : le premier vélo a pour identifiant *EB-1*, le suivant *EB-2* etc. 💡 Si vous avez besoin d'aide pour faire cela, vous trouverez des explications, ci-après.
- d) On considère qu'un vélo matche une description si la description contient l'identifiant du vélo ou si la description contient le nombre de niveaux de consommation d'énergie ou si la description contient « \* ». Attention cette fois-ci nous avons inversé, c'est la description qui doit contenir un de ces éléments. Par exemple, si la description contient « *EB-4 3 levels* », le vélo d'identifiant *EB-4* matchera cette définition mais également les vélos ayant 3 et 4 niveaux d'assistances. Nous avons volontairement simplifié le problème. Mais pour ceux qui s'ennuieraient vous pouvez très bien mettre en œuvre un petit langage 😊 bien plus malin.

Pour les **étudiants en difficulté sur le point c)**, vous utilisez une **variable statique donc partagée** par toutes les instances de *ElectricBike* que vous incrémentez à chaque construction d'un *ElectricBike*.

Dans la classe *ElectricBike*

```
private final static String PREFIX = "EB-";
private static int nextIdentifier = 1;

public ElectricBike(double batteryCapacity, double[] energyConsumptionLevels) {
    super(batteryCapacity);
    this.energyConsumptionLevels = energyConsumptionLevels;
    this.identifier = PREFIX + nextIdentifier;
    nextIdentifier++;
    isAvailable = true;
}
```

Attention cependant la variable étant partagée, il est nécessaire que dans vos tests, vous puissiez la remettre à 1 de l'extérieur de la classe en utilisant une méthode statique. Par exemple :

Dans le test, appel à la méthode statique *resetIdentifier*.

```
@BeforeEach
void setUp() {
    ElectricBike.resetIdentifier();
}
```

Et dans la classe *ElectricBike*

```
/**
 * Resets the identifier counter to 1.
 * This method is useful for testing.
 */
protected static void resetIdentifier() {
    nextIdentifier = 1;
}
```

### Q.13. Testez votre classe

Par exemples

```
final double[] energyConsumptionLevels = new double[]{energyConsumptionPerKilometer, 0.5, 0.8};

@org.junit.jupiter.api.Test
void testIdentifiers() {
    bike = new ElectricBike(batteryCapacity, energyConsumptionLevels);
    assertEquals("EB-1", bike.getName());
    ElectricBike bike2 = new ElectricBike(batteryCapacity, energyConsumptionLevels);
    assertEquals("EB-2", bike2.getName());
    ElectricBike bike3 = new ElectricBike(batteryCapacity, energyConsumptionLevels);
    assertEquals("EB-3", bike3.getName());
    assertEquals("EB-1", bike.getName());
}
```

### III.2.3 Utilisation des interfaces : un système de location de voitures et de vélos

A présent vous pouvez utiliser telle que la classe *RentalSystem* pour louer des vélos et des voitures électriques.

Q.14. Vous devez donc définir un système de location et le tester avec des vélos et des voitures électriques.

- Recopiez la classe *RentalSystemTest* qui vous a été donnée dans une classe *VehiculeRentalSystemTest* et placez la copie dans votre répertoire de tests des *vehicules*.
- Modifier les codes pour manipuler des voitures et des vélos.

### III.2.4 Bilan

A cette étape, votre projet devrait avoir la même structure que la figure ci-dessous<sup>1</sup>.

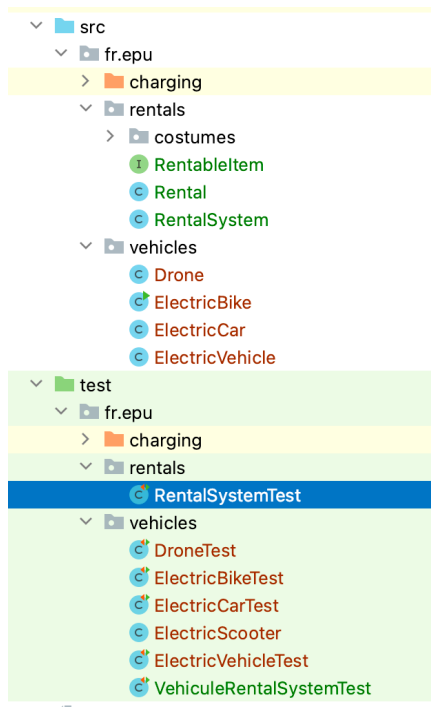
Vous avez actuellement plus de 10 classes. Vous en aurez beaucoup plus dans votre projet en PS5. Organisez vos codes sinon vous ne parviendrez plus à les maîtriser.

---

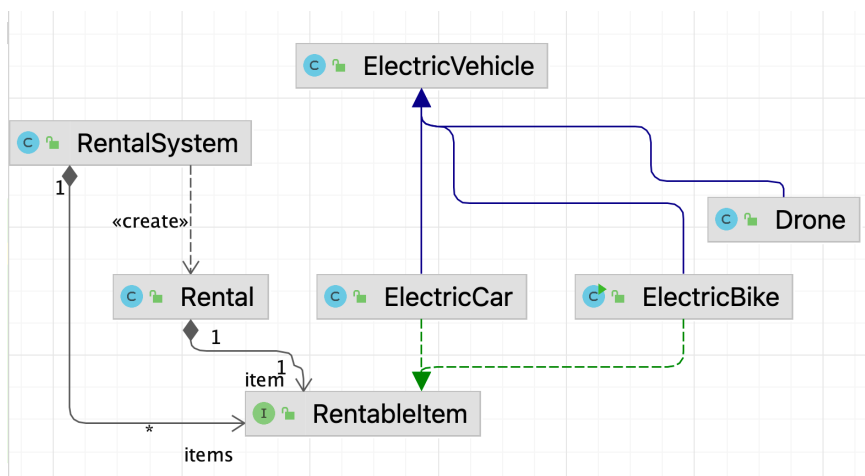
<sup>1</sup> Nous avons placé le package *charging* comme exclus pour ne pas être gêné dans les tests, mais ce n'est pas votre cas.

Il faut absolument que vous vérifiez les points suivants, et si ce n'est pas le cas demandez de l'aide mais surtout ne laissez pas vos codes diriger, ils ont besoin de vous 😊 :

- Tous vos sources sont dans le folder **src**, en bleu dans la figure ci-dessous.
- Tous vos tests sont dans le folder **test**, en vert dans la figure ci-dessous.
- Vos classes relatives aux véhicules sont dans le package *vehicles* et de même pour les tests associés.
- Vos classes relatives aux locations sont dans le package *rentals*.
- Etc.



Le modèle suivant présente vos classes, vous devez avoir la même visualisation au placement près des classes. Trois classes héritent de *ElectricVehicle* : *Drone*, *ElectricBike*, *ElectricCar*. Deux classes implémentent *RentableItem* (pas Drone).



## IV. Construction d'interfaces

Vous reprenez les codes donnés ou votre propre code qui se trouve dans le package `fr.epu.charging`.

### IV.1 Extraire des interfaces : des véhicules chargeables

Les stations de charge peuvent être connectés uniquement aux véhicules électriques tels que nous les avons définis. Mais le fournisseur de ce service souhaite ne plus dépendre de notre définition et **inverser la dépendance** en exposant une interface que les objets qui se connecteront devront implémenter. C'est la mission qu'il vous a confiée.

#### IV.1.1 Extraire une interface : des objets chargeables

- Q.15. Vous analysez votre code pour déterminer quels sont les méthodes qu'utilise `ChargeStation` pour connecter un **véhicule électrique**, puis vous les placez dans une interface, par exemple `ChargeableItem`.

L'objectif est de pouvoir remplacer toutes les références à `ElectricVehicle` par une référence à cette interface. Ainsi la station de charge ne dépend plus de notre implémentation, mais impose, elle, le type qu'elle attend. On parle alors d'**inversion de dépendance**.



A la fin de cet exercice, dans le package `fr.epu.charging` plus aucune classe ne doit faire référence aux véhicules.

#### IV.1.2 Implémenter une interface : des véhicules chargeables

- Q.16. Reprenez votre code pour rendre les véhicules chargeables.

**Vous n'avez plus rien à faire tous vos tests doivent continuer à passer.**

### IV.2 Produire une interface : des objets traçables

Dans notre système, nous avons besoin de suivre la position de divers objets, tels que des drones, des voitures, ou d'autres actifs. Un objet traçable est un objet qui peut être suivi et qui peut nous fournir sa position actuelle. Par exemple, un drone en vol peut nous donner sa position en temps réel.

#### IV.2.1 Des objets traçables

- Q.17. Isoler votre implémentation des objets traçables dans un package par exemple `fr.epu.tracking`
- Q.18. Créez une interface nommée **`Trackable`**. Cette interface doit déclarer une méthode **`getPosition`** qui ne prend pas d'argument et renvoie un objet de type **`Position`**. Tous les





objets que nous voulons pouvoir suivre dans notre système devront implémenter cette interface.

- Q.19. **Classe Position** : Créez une classe nommée **Position**. Cette classe doit avoir deux attributs, **x** et **y**, qui représentent les coordonnées en deux dimensions. De plus, cette classe devrait avoir des méthodes pour récupérer les valeurs **x** et **y**. Vous pouvez également ajouter une méthode **toString** pour donner une représentation lisible de la position.

#### IV.2.2 Implémentation du Système de Suivi

Maintenant que nous avons défini le concept d'objets traçables à l'aide de l'interface **Trackable** et de la classe **Position**, nous allons créer une classe **TrackingSystem** qui permettra de suivre ces objets dans un système de suivi.

- Q.20. Créez une classe nommée **TrackingSystem** dans le package **fr.epu.tracking**. Cette classe doit avoir les propriétés suivantes :
- Une variable d'instance de type liste  de **Trackable** pour stocker les objets que nous voulons suivre ;
  - Une méthode **addTrackableObject** qui prend un objet **Trackable** comme argument et l'ajoute à la liste d'objets traçables du système ;
  - Une méthode **getTrackableObjectPosition** qui prend un indice en tant qu'argument et renvoie la position actuelle de l'objet traçable correspondant à cet indice dans la liste. Assurez-vous de gérer les cas où l'indice est hors limites ou où l'objet n'a pas de position en renvoyant un **Optional<Position>** ;
  - Une méthode **getAllTrackablePositions** qui renvoie une liste de toutes les positions actuelles des objets traçables dans le système. Cette méthode doit ignorer les objets qui n'ont pas de position ;
  - Une méthode **getNumberOfTrackedObjects** qui renvoie le nombre d'objets traçables actuellement suivis dans le système.

-  Pour manipuler des Listes vous avez besoin de manipuler les classes associées c'est-à-dire,
- Faire savoir au programme de quelle classe il s'agit en important l'interface **List** et la classe **ArrayList** avant le début de la classe, après la déclaration du package :
    - `import java.util.List;`
    - `import java.util.ArrayList;`
  - Utiliser l'interface **List** en précisant entre chevrons les types de objets dans la liste, ici **Trackable**  
`private List<Trackable> trackedObjects;`
  - Créer un objet de type **ArrayList** qui implémente l'interface **List**  
`public TrackingSystem() {  
 trackedObjects = new ArrayList<>();  
}`

#### IV.2.3 Utilisation de votre interface : Des drones traçables

Nous voulons rendre les drones traçables. Il vous suffit donc d'implémenter la méthode **getPosition** puis nous pourrons les suivre avec notre système de suivi.

Pour faire simple, en vol, la position est une position générée au hasard. Par exemple, en utilisant le code qui suit :

```
@Override
public Position getPosition() {
    return new Position(Math.random(), Math.random());
}
```

Q.21. Complétez la classe Drone avec les méthodes **takeOff**, **getPosition** et **returnToBase** de la classe **Drone**.

- La méthode **takeOff** met le drone en vol (boolean *isFlying* à vrai par exemple)
- **getPosition** renvoie la position actuelle du drone s'il est en vol (une position aléatoire comme ci-dessus) ou sa base (position fixe) s'il ne vole pas,
- et la méthode **returnToBase** devrait ramener le drone à sa base en arrêtant son vol.

Q.22. Tester votre système de suivi en lui ajoutant des drones, etc.

## V. Conclusion

À l'issue de ce TD vous maîtrisez les bases de java et nous l'espérons plus largement celles de la POO.