Encapsulation

Intégration de la partie fonctionnelle

Avant d'effectuer l'intégration, la réalisation de **`Position`** est modifiée pour permettre le partage des instances.

Avec l'intégration, l'arborescence du projet est réorganisée et les classes sont incluse dans le paquetage `tec`.

Afin de vérifier l'intégration, nous vous fournissons un test de recette (la classe `Simple`).

Table des matières

1 Modifications de la réalisation

Les deux tandems effectuent ce travail.

1.1 La classe `Position`

*Si la classe **`Position`** ne construisait pas d'objets constants, combien faudrait-il d'instances pour représenter la position d'un passager dans la classe **`PassagerStandard`** ?*
Comparez avec le nombre d'instances dans le code actuel.

La classe **`Position`** construit des objets constants, il est possible de les partager. Avec le partage, nous n'avons besoin que de trois instances pour représenter les états d'un passager.

Mais, il faut que ce partage soit mis en œuvre par la classe elle-même.

Partage des instances.

Modifiez la classe **`Position`** pour permettre la définition et le partage des trois instances.

Pour respecter la spécification, les trois méthodes **`assis()`** **`debout()`** et **`dehors()`** reste des méthodes d'instance et la première instance fournie doit être dans l'état dehors.

Expliquer vos modifications ?

*A quel moment la classe **`Position`** est-elle instanciée ?*

Modifier en conséquence les classes **`TestPosition`** et **`PassagerStandard`**.

Vérifier les tests unitaires.

Remanier la réalisation de `Position`.

Après les modifications pour le partage des instances, les méthodes **`estAssis()`**, **`estDebout()`** etc… peuvent utiliser les adresses des trois instances à la place de la valeur de l'attribut **`courant`**.

Écrire cette version de la réalisation de **`Position`**.

Expliquer le comportement si les variables stockant les instances sont des variables d'instance.
Pour essayer l'exécution, le code de la méthode de classe doit effectuer une instanciation.

2 Boutez vos neurones

2.1 Les chaînes de caractères littérales

Le début de la documentation de la classe **`String`** reformule une partie des informations sur les chaînes de caractères littérales fournies à la section 3.10.5 de la spécification du langage Java.

A string literal is always of type String. A string literal always refers to the same instance of class String.

La même chaîne littérale correspond toujours à la même instance (même si elle est utilisée dans des classes différentes).

Le code suivant teste cette unicité :

```
String s1 = "kalki";
String s2 = "kalki";

System.out.println(s1 == s2);
System.out.println(s1 == "kalki");
```

Sur des références, l'opérateur `==` teste l'égalité des adresses.

Vérifier que le résultat de l'exécution de ce code est bien conforme à la spécification.

D'après la documentation de la classe **`String`**, ce code est équivalent à :

```
char valeur[] = {'k', 'a', 'l', 'k', 'i'};
String s1 = new String(valeur);
String s2 = new String(valeur);

System.out.println(s1 == s2);
System.out.println(s1 == "kalki");
```

Le résultat de l'exécution du code montre qu'il y a trois instances. Le code n'est pas équivalent il manque le traitement du partage des instances. En fouillant la documentation de la classe **`String`**, vous devriez tomber sur la méthode **`intern()`**.

Reprendre le code précédent en utilisant cette méthode et vérifier le résultat du partage.
*Comparer avec l'utilisation de la méthode **`equals()`**.*

Utilisez cette méthode pour montrer (à travers un exemple) le code que doit générer le compilateur pour respecter la spécification du langage Java.

3 Arborescence des fichiers

Dans l'organisation standard, les fichiers sources et les fichiers compilés sont séparés.

Chaque tandem crée dans son répertoire de travail :

- un répertoire `src` pour les classes principales ;
- un répertoire `tst` pour les classes de test et les classes faussaires ;
- un répertoire `build` pour les fichiers compilés.

Déplacer vos fichiers dans les bons répertoires.

3.1 Compilation et exécution.

Les fichiers sources et compilés n'étant plus dans le même répertoire, il faut ajouter des options à la compilation et à l'exécution.

Compilation

A partir de la racine de l'arborescence, la commande de compilation peut s'écrire :
`javac -d build src/*.java`.

Expliquer l'option `-d`.

Vérifier l'emplacement des fichiers compilés.

Pour compiler la classe de test, le compilateur a besoin du fichier compilé de la classe à tester. **`javac -d build -cp build tst/*.java`**.

Expliquer l'option `-cp`.

Exécution des tests

*Donner la commande qui lance l'exécution de la classe **`TestJauge`** à partir de la racine de l'arborescence.*

Vérifier l'exécution de tous les tests.

4 Le paquetage `tec`

Toutes les classes doivent appartenir au paquetage `tec`.

Quelle instruction faut-il ajouter au début de chaque fichier source ?

Pour chaque classe d'un paquetage, il faut préciser sa portée visible ou non à l'extérieur du paquetage. Les seules classes visibles en dehors du paquetage sont les deux intertaces et les classes **`Autobus`** et **`PassagerStandard`**.

Comment déclarer une classe avec une portée interne au paquetage ?

4.1 Compilation et exécution.

Quelle contrainte y-a-t-il sur l'organisation des fichiers des classes compilées ?

Donner le nom complet d'une des classes de test ?

Toutes les commandes de compilation et d'exécution se font à partir de la racine de l'arborescence.

1. *Donner la commande pour compiler les fichiers du répertoire `src` et vérifier l'emplacement des fichiers compilés.*
2. *Donner la commande pour compiler les fichiers du répertoire `tst` et vérifier l'emplacement des fichiers compilés.*
3. *Donner la commande qui lance l'exécution de la classe de test à partir du répertoire du projet*[1].

Fournir un fichier `makefile` à la racine du projet contenant les cibles suivantes :

`srccompile`
: compile les sources du répertoire `src`

`tstcompile`
: compile les sources du répertoire `tst`

`recette`
: exécute le test de recette

`test`
: exécute les tests unitaires

`clean`
: supprime les fichiers du répertoire `build`

5 Rassemblement et test d'intégration

Pour l'intégration, vous devez avoir l'ensemble des sources organisé dans les deux répertoires `src` et `tst`.

Chaque tandem fait ce rassemblement dans son répertoire de travail.

Le scénario d'intégration/de recette de la version en C (`simple.c`) est réécrit dans la classe **`Simple`**

La classe **`Simple`** n'appartient pas au paquetage `tec` mais au paquetage par défaut (sans nom).

Copier le fichier source `Simple.java` dans le répertoire `src`.

Compiler les classes du répertoire `src`, exécuter la classe **`Simple`**.

L'exécution de la classe **`Simple`** doit donner le même résultat que la version en langage C. Ce résultat est donné en commentaire en fin du fichier `Simple.java`

Mais son exécution risque de produire un affichage de la forme :

```
tec.Autobus@1820dda
tec.Autobus@1820dda
tec.PassagerStandard@15b7986
tec.PassagerStandard@87816d
tec.Autobus@1820dda
tec.PassagerStandard@15b7986
tec.PassagerStandard@87816d
tec.PassagerStandard@422ede
tec.Autobus@1820dda
tec.PassagerStandard@15b7986
tec.PassagerStandard@87816d
tec.PassagerStandard@422ede
tec.Autobus@1820dda
```

```
tec.PassagerStandard@15b7986
tec.PassagerStandard@87816d
tec.PassagerStandard@422ede
```

C'est le comportement par défaut de la méthode **`toString()`**.

Pour respecter le format d'affichage de la version C, vous devez adapter le code de la méthode **`toString()`** dans les deux classes **`Autobus`**, **`PassagerStandard`** et **`Jauge`**.

Modifier vos réalisations jusqu'à obtenir la même exécution.

1 Penser à utiliser le nom complet de la classe de test.

Ce document a été traduit de L^AT_EX par H^EV^EA