Substitution d'objets

Remaniement

Le contenu des deux interfaces **`Passager`** et **`Vehicule`** ne correspond pas au découpage (à la structuration) proposé dans la version en langage C.

Le fichier en-tête `__internes.h` a pour but de cacher des informations de réalisation au client (structures de données mais aussi certaines fonctions).

Les méthodes suivantes ne devraient pas être accessibles au code client :

- **Dans l'interface `Passager`** les méthodes : **`changerEnDehors()`**, **`changerEnAssis()`** **`changerEnDebout()`**, et **`nouvelArret()`** mais aussi les méthodes **`nom()`**, **`estDehors()`**, **`estAssis()`** **`estDebout()`**.
- **Dans l'interface `Vehicule`** les méthodes : **`aPlaceAssise()`**, **`aPlaceDebout()`**, **`monteeDemanderAssis()`**, **`monteeDemanderDebout()`**, **`arretDemanderDebout()`**, **`arretDemanderAssis()`**, et **`arretDemanderSortie()`**.

Pour respecter ce découpage de la version de départ, il faut masquer ces méthodes au code client (représenté par la classe **`Simple`**).

Table des matières

1 Séparer les méthodes avec des accès différents.

Nous allons jouer sur la visibilité/accéssibilité des méthodes au client. Le masquage va être réalisé au niveau des interfaces plutôt que des méthodes.

1.1 Des interfaces internes au paquetage.

Pour différencier l'accès à ces méthodes, nous préférons séparer les méthodes (comme dans la version C) plutôt que d'utiliser le mécanisme de portée. Cette séparation est effectuée en utilisant des interfaces.

Les interfaces **`Passager`** et **`Vehicule`** vont devenir internes au paquetage et contiennent la déclaration des méthodes inaccessibles au client. Nous introduisons deux interfaces publiques **`Usager`** et **`Transport`** qui contiennent la déclaration des méthodes accessibles au client.

1.2 Remaniement

Comme précédemment, il y a deux parties à remanier. Chaque tandem se charge d'une des deux parties.

La partie véhicule est remaniée en :

- Une interface publique **`Transport`** contenant la méthode utilisée par le client : **`allerArretSuivant()`**.
- L'interface **`Vehicule`** est interne au paquetage et contient les méthodes utilisées dans le paquetage.
- Les classes concrètes **`Autobus`** et **`FauxVehicule`** ont un lien «est-un» avec ces deux interfaces.

La partie passager est remaniée en :

- Une interface publique **`Usager`** contenant la méthode utilisée par le client : **`monterDans(Transport t)`**.
- L'interface **`Passager`** devient interne au paquetage et contient les méthodes utilisées dans le paquetage.
- Les classes concrètes **`PassagerStandard`** et **`FauxPassager`** ont un lien «est-un» avec ces deux interfaces.

Puisque nous utilisons des interfaces, nous profitons du cadre de l'héritage multiple de type.

Expliquer pourquoi il est nécessaire d'effectuer la conversion ***`Vehicule b = (Vehicule) t;`*** *dans le code de la méthode* ***`monterDans(Transport t)`*** *même si l'instance contenue dans le paramètre* **`t`** *possède bien les méthodes attendues.*

Cette conversion de type peut-elle échouer ?
En cas d'échec, cette échec est-il détecté à la compilation ou à l'exécution ?

Rassemblez les fichiers modifiés. Compilez, corrigez les erreurs, faites passer les tests.

1.3 Boutez vos neurones : un problème de portée

Il est toujours possible au client (ici la classe **`Simple`**) d'utiliser les méthodes déclarées dans les deux interfaces internes au paquetage.

Montrer de quelle manière le client peut accéder à ces méthodes. Expliquer les raisons de cet accès.

2 Les classes concrètes `PassagerStandard` et `Autobus`.

Le mécanisme de portée de Java offre deux manières de rendre ces méthodes internes inaccessibles en dehors du paquetage. Elles dépendent de la visibilité publique ou non publique des classes concrètes **`PassagerStandard`** et **`Autobus`**.

Il y a deux solutions pour résoudre ce problème de portée. Chaque tandem se charge de la réalisation d'une solution.

Pour préparer ce développement, chaque tandem crée un répertoire nommé `abstract` pour la première solution et `interface` pour la deuxième solution et copie tous les fichiers sources du remaniement précédent dans ce répertoire.

2.1 Ces deux classes concrètes sont publiques.

Le client instancie les deux classes (par exemple dans la classe **`Simple`**). Mais il ne doit pas avoir accès aux méthodes déclarées dans les interfaces internes aux paquetage.

Pour cela, nous allons gérer l'accès au niveau de la portée des méthodes.

Pour déclarer ces méthodes internes au paquetage, il faut remplacer nos deux interfaces **`Passager`** et **`Vehicule`** par des classes abstraites Java.

1. Remplacer ces deux constructions **`interface`** internes au paquetage par une construction classe abstraite (**`abstract class`**).
2. Dans ces deux classes abstraites, les méthodes doivent être déclarées abstraites (**`abstract`**) et avec une portée paquetage.
3. Modifier l'héritage des classes concrètes ainsi que la portée des méthodes héritée des classes abstraites.
4. Vérifier la portée des classes abstraites (interne au paquetage), des classes concrètes (publique) et des méthodes.

2.2 Ces deux classes concrètes sont internes au paquetage `tec`.

Le client ne peut plus instancier ces classes (par exemple dans la classe **`Simple`**). Les méthodes internes peuvent rester avec une portée publique.

Nous pouvons garder la construction interface pour **`Passager`** et **`Vehicule`**.

Nous fournissons le service d'instanciation grâce à la classe publique **`tec.FabriqueTec`** :

1. Elle contient deux méthodes de classe **`fairePassagerStandard()`** et **`faireAutobus()`**.
2. Chaque méthode de cette classe effectue l'instanciation d'une des classes concrètes internes au paquetage.

   Donner le prototype de ces deux méthodes.

3. La classe **`tec.FabriqueTec`** ne doit ni être instanciée, ni servir de classe de base.

   De quelle manière assurez-vous ces deux contraintes ?

4. Ne pas oublier de modifier la portée des classes concrètes.

2.3 Boutez vos neurones : Comparer ces deux solutions

Fournir d'abord les deux diagrammes de classes.

Pour chacune des deux solutions, montrer en quoi ladite solution est adaptée (ou pas) à des modifications ultérieures du code.

Pour vous aider, voici deux exemples de modification ultérieure du code :

- L'ajout de la classe concrète **`PassagerIndecis`**. (Employer la technique du copier/coller à partir de la classe **`PassagerStandard`**, et faites de même pour la classe de test).
- L'idée de séparer les méthodes utilisées pour la montée (les méthodes utilisées dans **`monterDans()`**, comme **`monteeDemander*`**) des méthodes utilisées pour la descente (les méthodes utilisées avec le code de **`nouvelArret()`**, comme **`arretDemander*`**). Cette séparation se ferait dans deux classes/interfaces différentes.

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