Substitution d'objets

Ajouter de nouveaux caractères

Avec l'introduction des relations de type/sous-type, ajouter de nouveaux caractères de passager s'effectue simplement. Chaque caractère est une nouvelle classe sous-type (lien «est-un») de `Passager` et `Usager`.

Considérons les deux caractères suivant :

le passager indécis
avec le comportement suivant :

à la montée
demande une place debout ou reste dehors ;

à chaque arrêt
change de place assis/debout.

le passager stressé
avec le comportement suivant :

à la montée
demande une place assise sinon une place debout ou reste dehors ;

à chaque arrêt
à partir de trois arrêts de sa destination, demande une place debout.

Pour réutiliser le code, les deux classes `PassagerIndecis` et `PassagerStresse` se construisent par héritage d'abord sur la classe concrète `PassagerStandard` puis en définisant une classe abtraite.

Chaque tandem prend en charge le développement d'un des caractères (`PassagerStresse` ou `PassagerIndecis`) avec ses tests et le remaniement de l'architecture sur une des deux versions de l'itération précédente.

Table des matières

1 Héritage de la classe concrète `PassagerStandard`
2 La classe abstraite `PassagerAbstrait`
2.1 Définir ce qu'il faut paramétrer
2.2 Remanier la classe `PassagerAbstrait`
2.3 Remanier les classes concrètes.
3 Boutez vos neurones
4 Factoriser les tests
4.1 La classe de tests de `PassagerAbstrait`
4.2 Les classes de tests des classes concrètes de passager

1 Héritage de la classe concrète `PassagerStandard`

Le but de cette partie est de réfléchir à la construire les classes `PassagerStresse` et `PassagerIndecis` par un héritage avec la classe `PassagerStandard`.

Pour respecter l'objectif II («Ajouter une réalisation/une classe sans modifier le code existant sauf l'instanciation») donné dans l'introduction du cours, la classe **`PassagerStandard`** ne doit pas être modifiée dans cette partie.

Pour effectuer ce travail, vous devez pouvoir répondre aux questions suivantes :

- *Quelles méthodes sont à redéfinir dans les classes dérivées ?*
- Si la méthode **`nouvelArret()`** est redéfinie, le test pour la sortie du passager risque d'être dupliqué dans le code des classes dérivées.
  Pour éviter cette duplication, comment peut-on appeller cette méthode dans la classe de base.
- *Expliquer le prototype du constructeur des classes dérivées et comment appeller le constructeur de la classe de base.*
- La réalisation de la classe dérivée **`PassagerStresse`** nécessite l'information sur la destination.
  *Comment est-il possible d'avoir la valeur de la destination dans cette classe dérivée sans modifier la classe **`PassagerStandard`** ?*

Changement du code de la classe de base.

Après la mise en production, le client change la spécification du comportement de la classe **`PassagerStandard`** :

- dans le constructeur, la valeur de la destination est calculée en retirant 4 à la valeur passée en paramètre au constructeur : **`this.destination = destination - 4;`**.
- à la montée, demande une place debout sinon reste dehors ;
- sort un arrêt avant sa destination.

*Quelles modifications faut-il apporter au code des classes dérivées de **`PassagerStandard`** ?*
Concluer sur les problèmes potentielles de ne pas préparer l'héritage par rapport à l'objectif I («changement de la réaliasation sans propagation»), l'objectif II et l'objectif III («pas de duplication de la réalisation par copier/coller»).

Pour préparer l'héritage, il faut soit modifier la classe **`PassagerStandard`**, soit modifier l'architecture des classes pour introduire une classe abstraite.

2 La classe abstraite `PassagerAbstrait`

Pour préparer l'héritage, il est préférable de remanier le code en définissant une classe abstraite **`PassagerAbstrait`**. Les classes de caractères héritent de cette classe abstraite.

La classe abstraite contient le code commun/général et les classes dérivées le code variable particulier à chaque caractère.

Fournir le diagramme de classes.

La classe abstraite **`PassagerAbstrait`** est construite à partir du code de la classe **`PassagerStandard`** :

1. renommer le fichier `PassagerStandard.java` en `PassagerAbstrait.java`,
2. changer le nom de la classe et du constructeur,
3. déclarer la classe abstraite.

2.1 Définir ce qu'il faut paramétrer

Les méthodes `monterDans()` et `nouvelArret()` contiennent aussi du code commun à toutes les réalisations de caractères :

- dans `monterDans()` le traitement de la conversion de type ;
- dans `nouvelArret()` le traitement de l'arrivée à la destination.

Notre objectif est de factoriser ce (peu de) code dans la classe abstraite. Pour cela, nous devons séparer physiquement le code de ces méthodes en deux parties (méthodes) : code commun/général, code variable/particulier.

Les méthodes `monterDans()` et `nouvelArret()` vont contenir le code commun à factoriser. Elles vont être définies dans la classe abstraite `PassagerAbstrait`. Ces deux méthodes ne sont plus à redéfinir dans les classes dérivées.

Pour contenir le code variable, la classe `PassagerAbstrait` définit deux nouvelles méthodes **abstraites** :

void choixPlaceMontee(Vehicule v)
: Elle permet de paramétrer le choix de la place lors de la montée. Elle est appelée par le code de la méthode `monterDans()`.

void choixPlaceArret(Vehicule v, int arret)
: Elle permet de paramétrer le changement de place à chaque arrêt. Elle est appelée par le code de la méthode `nouvelArret()`.

Ces méthodes doivent être redéfinies par chaque classe dérivée de `PassagerAbstrait`.

Remarque : La classe de base fixe le cadre de variation des réalisations de passager (les deux méthodes abstraites). Dans ce modèle de conception («design pattern»), la méthode `monterDans()` est un patron de méthode («template method») et `choixPlaceMontee()` est une méthode socle («hook method»).

2.2 Remanier la classe `PassagerAbstrait`

1. Déclarer deux méthodes abstraites `choixPlaceArret()` et `choixPlaceMontee()` dans la classe `PassagerAbstrait`.
2. Écrire le code de `monterDans()` qui fait appel à la méthode `choixPlaceMontee()`.
3. Écrire le code de `nouvelArret()` qui fait appel à méthode `choixPlaceArret()`.
4. Définir la méthode `int distanceDestination(int numeroArret)`, elle fournit la distance en nombre d'arrêt avant la destination à partir du numéro d'arrêt passé en paramètre.

Donner la portée des méthodes `distanceDestination()`, `choixPlaceMontee()`, `choixPlaceArret()` ?

Comment éviter la redéfinition des méthodes `monterDans()`, `nouvelArret()`, `distanceDestination()` dans les classes dérivées.

Compiler et corriger les erreurs.

2.3 Remanier les classes concrètes.

Remanier le code des classes concrètes de passager qui ne contiennent maintenant que la redéfinition des deux méthodes `choixPlaceMontee()`, `choixPlaceArret()` et leur constructeur.

À la manière du test de recette `Simple`, réaliser une autre classe servant de test de recette utilisant tous les caractères.

Remarque : Dans la branche avec la fabrique, l'ajout des deux caractères ne doit pas modifier le source de la fabrique.

Fournir le diagramme de classes final.

3 Boutez vos neurones

*Expliquer précisément comment l'appel à la méthode de la classe de base (par exemple **`monterDans()`**) va appeler la méthode (ici **`choixPlaceMontee()`**) redéfinie dans la bonne classe dérivée.*

4 Factoriser les tests

Beaucoup de tests inclus dans la classe **`TestPassagerStandard`** sont des tests pour le code de la classe **`PassagerAbstrait`**

Pour garder la cohérence, nous allons déplacer ces tests dans une nouvelle classe concrète **`TestPassagerAbstrait`**.

À la fin, la classe **`TestPassagerStandard`** contient seulement les tests des méthodes définies dans la classe **`PassagerStandard`** (**`choixPlaceMontee()`** et **`choixPlaceArret()`**).

4.1 La classe de tests de `PassagerAbstrait`

La classe **`PassagerAbstrait`** n'étant pas instanciable, nous allons tester sa réalisation à l'aide des instance de **`PassagerVide`** classe dérivée de **`PassagerAbstrait`** qui redéfinit les deux méthodes **`choixPlaceMontee()`** et **`choixPlaceArret()`** avec un corps vide.

1. Définir une classe concrète **`TestPassagerAbstrait`**.
2. Dans cette classe **`TestPassagerAbstrait`**, définir la méthode privée **`PassagerAbstrait creerPassager(String nom, int destination)`**, elle renvoit une instance de **`PassagerVide`**.

   Pour vous éviter d'écrire la classe **`PassagerVide`**, vous pouvez utiliser le mécanisme de classe anonyme (voir fin de l'annexe D du cours).

   De cette manière :

   ```
   private PassagerAbstrait creerPassager(String nom, int destination) {
      return new PassagerAbstrait(nom, destination) {
        protected void choixPlaceMontee(Vehicule v) {}
        protected void choixPlaceArret(Vehicule v, int arret) {}
      };
    }
   ```

   Le compilateur génére le `.class` d'une classe anonyme avec comme nom : le nom de la classe englobant plus le caractère `$` et un nombre (`TestPassagerAbstrait$1.class`).

3. Déplacer les méthodes de tests de la réalisation de **`PassagerAbstrait`** dans la classe **`TestPassagerAbstrait`**.
4. Dans ces tests, remplacer le type **`PassagerStandard`** par le type **`PassagerAbstrait`** et modifier l'instanciation pour appeler la méthode privée **`creerPassager()`**.
5. Compiler et exécuter la classe **`TestPassagerAstrait`**.

4.2 Les classes de tests des classes concrètes de passager

Les classes **`TestPassagerStandard`**, **`TestPassagerStresse`** et **`TestPassagerIndecis`** contiennent les tests des deux méthodes **`choixPlaceMontee()`** **`choixPlaceArret()`** ((et de l'instanciation pour des initialisations supplémentaires).

Remanier/Écrire les tests dans ces trois classes de tests.

Compiler et exécuter les tests.

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