

2.2 La classe PassengerStandard

La classe PassengerStandard est remaniée, de façon à hériter de la classe PassengerAbstrait, et à définir le comportement standard pour les méthodes monterDans() et nouvelArret().

Si on définit la portée de la variable destination en protected, cela signifie qu'elle est accessible dans le paquet *loc*. Ainsi, cette variable de la classe PassengerAbstrait peut être modifiée par d'autres classes du paquet *loc*. L'encapsulation n'est donc pas respectée.

Si le constructeur PassengerStandard() est vide, alors que cette classe hérite de PassengerAbstrait, le code vide va être remplacé par l'appel à super(), sans paramètres. La méthode super() se référant au constructeur de la classe PassengerAbstrait, qui compte deux paramètres, il va y avoir une erreur, indiquant qu'il manque des paramètres dans ce constructeur.

Il est obligatoire de déclencher un constructeur de la classe de base avant celui de la classe dérivée, car les attributs de la classe de base sont initialisés avant ceux de la classe dérivée, et le code du constructeur de la classe dérivée peut utiliser des méthodes de la classe de base.

2.3 Les deux nouveaux caractères

Le code en commun entre les classes PassengerStandard, PassengerLumatique et PassengerStresse est contenu dans les deux méthodes nouvelArret() et monterDans() de PassengerAbstrait :

```
— Dans nouvelArret(), on trouve le code  
if ( destination == numerArret )  
    bus.demanderSortie(this);  
— Dans monterDans(), on trouve le cast  
Bus b = (Bus) r;
```

3 Définir ce qu'il faut paramétrer

Nous allons mettre dans la classe PassengerAbstrait le code dupliqué, afin de le factoriser. Pour ce faire, il faut transférer les méthodes nouvelArret() et monterDans() dans la classe abstraite, et y déclarer deux nouvelles méthodes abstraites choixPlaceMontee() et choixChangerPlace(), que chacune des classes dérivées définira selon le comportement souhaité.

C'est le mécanisme de polymorphisme qui va se charger d'ajuster correctement les méthodes choixChangerPlace() et choixPlaceMontee() vers la bonne classe. Cette liaison est dynamique, elle se fait à l'exécution du programme en fonction de l'instance manipulée pour obtenir l'adresse de la méthode à l'exécution.

→ ainsi grâce au polymorphisme.

on divise la réalisation en 2 parties :
1. quelle est l'instance à utiliser ?
2. quelle est l'instance à utiliser ?

Sur tout permet de forcer la redéfinition (discussion en TD)

3.1 La classe PassengerAbstrait

Les deux nouvelles méthodes choixPlaceMontee() et choixChangerPlace() sont des méthodes abstraites. Si on met un corps vide pour ces deux méthodes, on sera alors obligés de les réécrire dans les classes dérivées (@Override), alors que si on laisse seulement la signature de ces méthodes (méthodes abstraites), on laisse alors le soin à chaque classe dérivée de définir ces méthodes comme elle l'entend, sans la réécrire. Ces deux méthodes ont une portée publique par défaut.

Pour s'assurer que les méthodes monterDans() et nouvelArret() ne soient redéfinies, il faut ajouter le mot-clé final à ces deux méthodes. Elles ne pourront ainsi pas être réécrites.

3.2 Les classes caractères

Nous écrivons maintenant le code des classes PassengerStresse, PassengerLumatique, et PassengerStandard, ainsi que leurs tests.

Ces classes de caractères doivent être déclarées final, pour qu'aucune autre classe n'en hérite. Ces classes de caractères sont les classes de plus bas niveau, c'est-à-dire qu'il n'est pas envisageable qu'il y ait de sous-caractères ou aucune autre classe qui en dérive. Cela empêche aussi tout utilisation inappropriée de ces classes.

4 Factoriser les tests

La classe TestPassengerAbstrait est définie abstraite, et possède une méthode abstraite servant de constructeur, qui est redéfinie dans chaque classe dérivée TestPassengerStandard, TestPassengerLumatique et TestPassengerStresse. Le type de retour utilisé est PassengerAbstrait, puisque d'une part nous voulons seulement tester les PassengersAbstrait, et non les Passengers ou les Usagers, et d'autre part, parce que ces deux derniers sont des interfaces, et non des classes à instancier.

5 Commentaires

5.1 Commentaire de Pierre

Je pense que les deux notions importantes à tirer de ce TD sont l'encapsulation et la factorisation de code. En effet, il y a une marge entre produire un code qui fonctionne, et produire un code bien pensé, propre et concis. C'est cette approche qui nous a été suggérée ici, notamment par le fait de créer beaucoup de classe, mais que chacune ait un rôle bien particulier et propre, l'encapsulation, et par le fait de créer des classes abstraites pour factoriser le code. Le but étant d'avoir un code facilement maintenable pour

protected

gagner du temps par la suite. Ici, on s'est donc abstraits des problèmes techniques du langage en lui-même, pour s'intéresser plutôt aux fondements de la programmation orientée objet.

5.2 Commentaire d'Aurélien

Ce TD consistait à créer la classe abstraite `PassagerAbstrait` ainsi que les classes `PassagerLunatique` et `PassagerStressé`. Bien que le code à écrire n'était pas conséquent, il a fallu bien faire attention lors de la factorisation du code et ainsi remanier correctement le code des fichiers sources ainsi que des fichiers tests. Je me suis également rendu compte dans ce TD que ce n'est pas simple de fournir du code bien factorisé, avec des tests unitaires également factorisés. Mais le fait est qu'une fois le code correctement écrit, il est alors beaucoup plus facile de modifier par exemple une fonction bien précise sans avoir à retoucher tout le code !

5.3 Commentaire de Reda

Dans ce TP, j'ai appris à faire évoluer des classes afin de les rendre plus réalistes (les types de passagers). Grâce au système d'héritage sur les classes abstraites cela se fait plutôt simplement. De plus, j'ai pu découvrir l'utilisation et l'intérêt des exceptions.

5.4 Commentaire de Victor

J'ai trouvé ce TD agréable car nous avons pu voir l'abstraction de plusieurs classes en une classe pour éviter le code commun. Nous nous doutions que nous pouvions faire une telle chose, mais nous avons pu constater que ça fonctionnait bien.

5.5 Commentaire de Lionel

Ce TD était pour moi un approfondissement des notions d'héritage, d'abstraction mais aussi un plus sur la factorisation de code et l'importance des tests unitaires. Cela montre les différentes étapes à respecter lors de l'écriture d'un code.

Code :

- Faire une classe `Lunatique` et `Stressé`
- `Passager` étend `Usager` ??
 Bus étend `Transport` ?? } avec les mêmes méthodes ??
→ `PassagerLunatique` le fait ??
⇒ code raisonnable.