Héritage et Factorisation

La classe **PassagerStandard** est un exemple d'une réalisation particulière (un caractère) d'un passager/usager.

Nous pouvons envisager d'autres caractères :

le passager stressé — à la montée, demande une place assise ou rien ;
— à partir de trois arrêts de sa destination, demande une place debout ;

le passager lunatique — à la montée, demande une place debout ou rien ;
— change de place (assise-debout) à chaque arrêt.

Pour réaliser ces deux caractères, il suffit de modifier le code des deux méthodes : **monterDans**() du type **Usager** et **nouvelArret**() du type **Passager**.

Le thème est d'étudier une architecture de classes qui permettra d'ajouter des classes de passager/usager avec deux préoccupations :

1. *éviter la duplication de code* ;
2. *ajouter des caractères sans modifier le code existant dans notre paquetage.*

Pour arriver à cette architecture, il va falloir forcément remanier le code déjà écrit.

Les tandems.

Les tandems vont travailler en parallèle sur le même problème :
— remanier le code de la classe **PassagerStandard** ;
— remanier les classes de test pour factoriser le code des tests ;
— réaliser un des deux nouveaux caractères.

Dans le répertoire de référence, vérifier la compilation, les tests et les applications. Ne pas commencer le travail s'il y a un problème, essayer d'abord de le résoudre.

Puis, chaque tandem recopie les sources dans son répertoire de travail.

N'oubliez pas de fournir le diagramme de classe.

Table des matières

1 Pourquoi l'héritage

Voyons les différentes solution pour réaliser les deux nouveaux caractères.

Première solution : Une seule classe. La classe **PassagerStandard** va contenir les trois implantations. Il faut ajouter dans la classe **PassagerStandard** une variable d'instance de type entier. Si la valeur de cette variable vaut :

— 0 c'est le code correspondant au **PassagerStandard** qui est exécuté ;
— 1 c'est le code correspondant au **PassagerStresse** qui est exécuté ;
— 2 c'est le code correspondant au **PassagerLunatique** qui est exécuté.

Le code des deux méthodes **monterDansBus**() et **nouvelArret**() va contenir autant de branchements conditionnels (if/else ou switch) que de valeurs attribuées à cette variable.

Quelle conclusion par rapport à nos deux préoccupations ?

Deuxième solution : Relation type/sous-type.

Chaque caractère est une nouvelle classe qui implante les deux interfaces **Passager** et **Usager**.

Il faut dupliquer le code de la classe **PassagerStandard** par un simple copie/coller dans chaque nouvelle classe. Le code des méthodes **monterDans**() et **nouvelArret**() doit être en partie modifiée.

Conclusion ?

Troisième solution : Héritage.

L'héritage permet de partager l'implantation d'une classe avec ses sous-classes. Chaque caractère est une classe qui hérite de la classe **PassagerStandard** et redéfinit les deux méthodes **monterDans**() et **nouvelArret**().

Conclusion ?

Par contre, il n'est pas forcément évident de savoir dans le code de la classe **PassagerStandard** ce qui est commun à toutes les implantations et ce qui peut être variable. D'ailleurs, il reste du code dupliqué. Lequel ?

La classe **PassagerStandard** n'est qu'un caractère particulier. Elle n'indique donc pas ce qui est commun et ce qui est variable.

Une classe de base pour tous les caractères.

Pour éviter toute confusion entre la partie commune et la partie variable, nous allons introduire une classe de base **PassagerAbstrait**. Chaque classe (**PassagerStandard**, **PassagerLunatique**, **PassagerStresse**) hérite de la classe **PassagerAbstrait**. Elle implante les interfaces **Passager** et **Usager** et factorise le code considéré commun à toutes les sous-classes.

Faire le diagramme de classe.

Passons à la réalisation de cette solution.

2 Une classe de base abstraite

Nous allons construire la classe abstraite **PassagerAbstrait** en fonction du code de la classe **PassagerStandard**.

A partir de la description des deux nouvelles classes **PassagerStresse** et **PassagerLunatique**, il est possible de factoriser toute l'implantation de **PassagerStandard** sauf une partie du code contenue dans les deux méthodes **monterDans**() et **nouvelArret**(). Le code de ces deux méthodes doit être modifié dans chaque sous-classe. Pour obliger à fournir du code à ces méthodes, nous allons les déclarer comme méthodes abstraites dans la classe **PassagerAbstrait**. La classe de base devient une classe abstraite.

Occupons nous de la construction de la classe **PassagerAbstrait** :

1. copier le fichier `PassagerStandard.java` en `PassagerAbstrait.java`,

2. changer nom de la classe et du constructeur,
3. déclarer la classe abstraite,
4. supprimer le corps des deux méthodes **monterDans**() et **nouvelArret**() et les déclarer comme méthode abstraite. Ne pas oublier le **;** après la déclaration de chacune de ces deux méthodes.

La compilation ne doit pas indiquer d'erreurs.

Avant de réaliser les nouveaux caractères, nous allons remanier le code de la classe **PassagerStandard** :

— faites hériter la classe **PassagerStandard** de **PassagerAbstrait**
— supprimer toutes les variables d'instances et toutes les méthodes (sauf bien sûr les deux méthodes **monterDans**(), **nouvelArret**() ;-)
— réécrire le constructeur de +PassagerStandard+, il faut appeler le constructeur de la classe de base.

Pour pouvoir redéfinir la méthode **nouvelArret**() dans les sous-classes, il faut avoir accès à la valeur de la variable qui stocke la destination.

Comment la rendre accessible aux classes dérivées ?

Dans quel cas est-il préférable de définir une méthode qui renvoie sa valeur ? Quel est le modificateur d'accès de cette méthode ?

Compiler les deux classes.

Après ces modifications les tests de la classe **PassagerStandard** doivent toujours être valides (Tiens ! les tests servent à quelquechose).

Remarque1 : Pour le langage java, il n'est pas nécessaire d'implanter les interfaces à chaque classe dérivée car **PassagerAbstrait** les implante déjà.

Remarque2 : Les tests du développeur jouent le rôle de test de non régression. Ce qui marchait avant doit toujours marcher après.

Question : Pourquoi l'appel au constructeur de la classe de base est-il nécessaire ? Si le corps du constructeur de **PassagerStandard** est vide, vous obtenez un message d'erreur. Expliquez ce message d'erreur.

3 Définir ce qu'il faut paramtrer

Pour réaliser les deux nouveaux caractères, il suffit de faire hériter les classes de **PassagerAbstrait** et de redéfinir le méthodes **monterDans**(), **nouvelArret**(). Pour chaque caractère, donner le code de ces deux méthodes.

Vous remarquez que tout le code n'est pas à changer, une partie est à dupliquer. Cette partie doit d'ailleurs être présente dans toutes les classe dérivée. Comment factoriser ce code commun obligatoire dans la classe de base ?

Dans la classe **PassagerAbstrait**, nous définissons deux nouvelles méthodes :

void choixPlaceMontee(Bus b) Elle permet de paramètrer le choix de la place lors de la montée. Elle est appelée par le code de **monterDans**().

void choixChangerPlace(Bus b, int arret) Elle permet de paramètrer le changement de place à chaque arrêt. C'est le code de **nouvelArret**().

Les méthodes **monterDans**() et **nouvelArret**() ne sont plus abstraites. Elles contiennent le code commun et l'appel à une des méthodes **choixPlaceMontee**() et **choixChangerPlace**(). Les deux nouvelles méthodes vont contenir le code variable à définir dans les classes dérivées.

1. Déclarer deux méthodes abstraites **choixChangerPlace**() et **choixPlaceMontee**() dans la classe **PassagerAbstrait**. Pourquoi les déclarer abstraites ?
2. Ces deux méthodes sont-elles privées, publiques ou protégées ?

3. Modifier le code de **monterDans**() pour faire appel à **choixPlaceMontee**().
4. Modifier le code **nouvelArret**() de pour faire appel à **choixChangerPlace**().
5. Redéfinir ces méthodes abstraites dans **PassagerStandard**.
6. Compiler et tester.

La classe de base fixe le cadre de variation des implantations de passager (les deux méthodes abstraites). Vous pouvez maintenant développer des deux caractères sans modifier le code existant et avec un code factorisé.

Remarque : 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").

Réaliser une des deux classes **PassagerStresse** et **PassagerLunatique**.

Pourquoi faut-il mieux déclarer les autres méthodes de la classes **PassagerAbstrait** comme finales ?

Pourquoi faut-il mieux déclarer les classes **PassagerStandard**, **PassagerStresse** et **PassagerLunatique** comme finales ?

L'étape suivante est de tester le comportement de ces nouvelles classes. Une partie des tests sont valides pour ces trois classes. Voyons comment éviter de dupliquer le code des tests.

4 Factoriser les Tests

Beaucoup de tests inclus dans la classe **PassagerStandardTest** sont valables pour tous les caractères de passager. Certains ne sont valables que pour l'implmentation de **PassagerStandard**. Appliquons le même principe : construire une classe abstraite pour factoriser.

Construction de la classe abstraite.

1. Définir une classe abstraite **PassagerAbstraitTest**
2. Faire hériter la classe **PassagerStandardTest** de cette classe.
3. Compiler et exécuter la classe **PassagerStandardTest**. Noter le nombre de tests exécutés.
4. Déplacer toutes les méthodes de test valables pour toutes les implémentations dans la classe abstraite. Dans la classe **PassagerStandardTest** ne reste que les tests spécifiques à l'implantation de **PassagerStandard**.

   Pourquoi doit-il y avoir toujours le même nombre de test ?

Pour réaliser les classes de test **PassagerStresseTest** et **PassagerLunatiqueTest**, il suffit de définir une classe qui hérite de la classe **PassagerAbstraitTest** et de réaliser dans chaque classe les tests spécifiques.

Eh ben non, mauvaise pioche :-). Dans la classe **PassagerAbstraitTest**, le code des méthodes instancie toujours la classe **PassagerStandard**. Il faut définir ce qu'il faut paramètrer.

Définir ce qu'il faut paramètrer.

Nous avons besoin d'une méthode abstraite dans la classe **PassagerAbstraitTest** qui sera redéfinie dans les sous-classes avec l'instanciation de la bonne sous-classe.

1. Dans la classe **PassagerAbstraitTest**,
   — définir la méthode
   abstract protected PassagerAbstrait creerPassager(**String nom**, **int destination**);
   Pourquoi prendre comme type de retour **PassagerAbstrait** et non pas **Passager** ou **Usager**
   — dans cette classe changer toutes les occurences de **new PassagerStandard** par **creerPassager** et le type **PassagerStandard** par **PassagerAbstrait**

Compiler. Ne pas oublier de déclarer la classe comme abstraite.

2. Redéfinir cette méthode dans la classe **PassagerStandardTest**.
3. Compiler et tester **PassagerStandard**. Tous les tests doivent passer.
4. Ajouter les tests sur les exceptions.
5. Compiler et tester **PassagerStandard**. Tous les tests doivent passer.

Remarque : Dans ce modèle de conception ("design pattern"), la méthode **creerPassager**() est une méthode de fabrication ("factory method").

La classe **PassagerAbstraitTest** a pour but de vérifier le code contenu dans la classe **PassagerAbstrait** et la classe **PassagerStandardTest** celui contenu dans la classe **PassagerStandard**.

Pour suivre ce principe, vous pouvez réécrire les tests contenus dans **PassagerStandardTest** en vous servant uniquement de l'appel à la méthode **choixPlaceMontee**() et non à **monterDans**().

Ecrire les tests des classes **PassagerLunatique** et **PassagerStresse**.

Remarque : L'écriture des tests dans **PassagerAbstraitTest** est plus générale. Ils doivent être valables quelquesoit le comportement des sous-classes. En particulier, la valeur exacte de la variable message n'est pas toujours connue. Il faut parfois modifier le test par rapport à sa version de **PassagerStandardTest** vérifier que message n'est pas égal à (au lieu de vérifier l'égalité à).

5 Nouvelle architecture

Présenter sur un diagramme de classe la nouvelle architecture obtenue.