Première Réalisation du paquetage tec

Cette première version du paquetage `tec` est centrée sur la réalisation de la partie fonctionnelle demandée par le client (grossièerement "faire marcher le programme fourni par le client").

Le paramétrage du paquetage : des passagers munis de caractères différents sera abordé dans une deuxième version du paquetage.

Suivant la description du sujet (Transports En Commun), votre réalisation pratique concerne deux classes **PassagerStandard** et **Autobus** et leurs tests.

La spécification de ces classes est donnée par la documentation du paquetage, le diagramme de classe et les diagrammes de séquence (voir la partie précédente du projet).

Une première distribution est donnée via le fichier archive *src.zip* disponibe sur Moodle (resources du projet).

Table des matières

1 Deuxime itèration : Instanciation et Changement d'état

Objectifs

L'objectif est centré sur l'écriture :
— des constructeurs ;
— des méthodes donnant l'état d'une instance (accesseurs) ;
— des méthodes modifiant l'état d'une instance (modificateurs).

Contraintes de conception

Pour cette itération, nous allons utiliser seulement une partie de la spécification. On considère les contraintes suivantes :
— Pas d'interactions entre les deux classes concrètes **PassagerStandard** et **Autobus** .
— Les passagers ne sont pas stockés dans la classe **Autobus**.

Méthode de travail

— Approche TDD : L'objectif des tests du développeur est de montrer l'adéquation entre le code écrit par le développeur et la spécification fournie par le client. Le principe est d'écrire et d'exécuter ces tests en parallèle avec l'écriture du code.
— Répartir le travail : garder la même répartition du travail que l'itération précédente.

Remarque : Dans les classes **Autobus** et **PassagerStandard**, vous devez redéfinir la méthode **toString**() héritée de la classe **Object**. Elle permettra de déboguer votre code en affichant l'état d'une instance. Elle est d'ailleurs utilisée de cette manière par le programme du client (le format d'affichage est précisé dans le commentaire à la fin du fichier `Simple.java`, vous pouvez vous en servir).

1.1 Ordre des tests

1.1.1 Cohérence des états après instanciation

Ceux sont toujours les premiers tests à effectuer. Chaque cas d'instanciation est codé dans une méthode différente.

— **PassagerStandard**. L'état d'une instance correspond aux valeurs de retour des accesseurs : **estAssis**(), **estDebout**() et **estDehors**.
⇒ *D'après la spécification de cette classe, combien faut-il de cas d'instanciation ?*
Pour réaliser ces états, vous devez utiliser des instances de la classe **EtatPassager**.
— **Autobus**. L'état d'une instance correspond aux valeurs de retour des accesseurs : **aPlaceAssise** et **aPlaceDebout**.
⇒ *D'après la spécification de cette classe, combien faut-il de cas d'instanciation ?*
Pour réaliser ces états, vous devez utiliser des instances de la classe **JaugeNaturel**.

1.1.2 Changement d'état

Chaque méthode de test a la même structure :

1. nouvelle instanciation la classe en test,
2. appel à un modificateur,
3. en utilisant les accesseurs, vérifier avec des assertions la cohérence de l'état de l'instance.

— **PassagerStandard**. Pour une instance de cette classe, les modificateurs sont les méthodes : **accepterPlaceAssise**, **accepterPlaceDebout**, **accepterSortie**.
Les envois de message à un bus ne sont pas codés dans cette itération. Le corps des deux méthodes **nouvelArret** et **monterDans** reste vide.
— **Autobus**. Pour une instance de cette classe, les modificateurs sont les méthodes : **demanderPlaceAssise**, **demanderPlaceDebout**, **demanderChangerEnAssis**, **demanderChangerEnDebout**.

Les envois de message à un passager ne sont pas codés dans cette itération. Le corps des méthodes **demanderSortie** et **allerArretSuivant** reste vide.

Pour les méthodes : **demanderPlaceAssise**, **demanderPlaceDebout**, **demanderChangerEnAssis**, **demanderChangerEnDebout** qui prennent en paramétre un **Passager**, utilisez le mot-clé **null**. Il indique une référence qui ne contient aucune adresse. Si vous faites un envoi de message à travers une référence nulle vous obtenez une exception de la classe **NullPointerException**.

1.2 Fin de l'itération

Si vous rassemblez vos sources dans un même répertoire, la classe **Simple** compile sans erreur (les classes sont toutes construites). Mais l'exécution de cette classe (l'intégration) doit fournir un résultat incorrect (le code est incomplet).

2 Troisème itération : interaction et stockage des passagers

Objectifs

Les objectifs sont :
— réaliser et tester les interactions précisées par les diagrammes de séquence (pour la montée et pour chaque arrêt),
— utiliser un tableau pour stocker les passagers qui rentre dans l'autobus.

Méthode de travail

— Approche TDD : L'objectif des tests du développeur est de montrer l'adéquation entre le code écrit par le développeur et la spécification fournie par le client. Le principe est d'écrire et d'exécuter ces tests en parallèle avec l'écriture du code.
— Utilisation des "mocks"
— Répartir le travail : garder la même répartition du travail que l'itération précédente.

Cette itérations prend en compte les deux contraintes de conception définies dans l'itération précédente.

A priori, pour tester une instance de la classe **Autobus** il nous faut une instance de la classe **PassagerStandard** mais avec une réalisation complète. Et pour le tester une instance de la classe **PassagerStandard**, il nous faut une instance de la classe **Autobus** avec aussi une réalisation complète. Mais comme le développement s'effectue en parallèle, nous n'avons pas le code complet. C'est ce code justement qui est encore en cours d'écriture.

Puisque nous n'avons pas les bons objets pour tester le code en développement, nous allons construire des classes pour remplacer les instances de **Autobus** ou de **PassagerStandard**. Cette technique est nommée en anglais "mock object" ou objet faussaire (bouchon).

2.1 Les classes faussaires

Pour tester certaines méthodes d'**Autobus**, nous devons substituer les instances de **PassagerStandard** par des instances d'une autre classe nommée **FauxPassager** (et inversement pour le test de **PassagerStandard**).

Le code d'une classe "faussaire" doit permettre de simuler simplement les états des objets "réels" [1]. en fonction des tests à écrire. Parfois plusieurs faussaires sont utilisées pour simuler tous les comportements

Les fichiers sources des classes faussaires sont fournis dans le fichier archive *usageDeFaux.zip* disponibe sur Moodle (resources du projet).

— Pour tester la classe **PassagerStandard**, vous avez quatres classes faussaires **FauxBusVide**, **FauxBusPlein**, **FauxBusAssis**, **FauxBusDebout**.

— Pour tester la classe **Autobus**, vous avez une seule classe faussaire **FauxPassager**.

On recopie dans son répertoire de travail les fichiers des classes faussaires dont on a besoin.

Ces classes faussaires sont données à titre d'exemple d'un code dit "simpliste" pour les tests (Ne coder pas plus qu'il n'est nécessaire).

2.1.1 Le faux passager

La classe faussaire **FauxPassager** va permettre de tester la classe **Autobus**. C'est une classe privée du paquetage `tec`.

Cette classe est instanciée dans les méthodes de test de la classe **AutobusTest**.

⇒ *A quel condition une instance de cette classe peut être substituée à une instance de la classe* ***PassagerStandard****.*

Elle va permettre de vérifier le comportement de **Autobus** avec les trois états possible d'un passager (fixé par trois constantes).

Le changement d'état est fait directement dans les méthodes de test en modifiant la valeur de la variable **status**.

Contrairement aux classes "réelles", les interactions avec la classe **Autobus** ne sont pas écrites mais le code permet de tracer l'appel à certaines méthodes intervenant dans ces interactions. Le nom de dernière méthode appelée est contenu dans la variable **message**.

Compiler cette classe dans le paquetage tec.

2.1.2 Les faux bus

Les quatre classes faussaires **FauxBus**$*$ vont permettre de tester la classe **PassagerStandard**. Ceux sont des classes privées au paquetage `tec`.

Ces classes sont instanciées dans les méthodes de test de la classe **PassagerStandardTest**.

⇒ *A quel condition une instance de cette classe peut être substituée à une instance de la classe* ***Autobus****.*

Les instances de ces quatres classes faussaires sont des objets constants.

Contrairement aux classes "réelles", les interactions avec la classe **PassagerStandard** ne sont pas écrites mais le code permet de tracer l'appel à certaines méthodes intervenant dans ces interactions. Le nom de dernière méthode appelée est contenu dans la variable **message**.

Compiler cette classe dans le paquetage tec.

2.2 Ordre des tests

— **PassagerStandard**. Ecrire le test de **monterDans** puis de **nouvelArret**

1. il ne s'agit pas de (re)faire le travail des autres développeurs

⇒ *comment passer l'objet receveur en paramètre d'une méthode*
Pour faciliter le test, les méthodes **demander**∗ des classes faux effectuent un appel aux modificateurs de la classe **PassagerStandard**.

— **Autobus**. Vous devez prendre en compte le stockage du passager. Utilisez un tableau. Compléter le code et les tests pour les méthodes **demander**∗ puis écrire le code pour **demanderSortie** et **allerArretSuivant**.

Remarque : Pour les interactions entre les classes en développement, les tests vérifient que les appels sont conforme à la spécification. En gros, ils vérifient la trace du code car ils ne peuvent pas vérifier le résultat global (il manque du code). Les faussaires sont donc écrits pour fournir le nom de la méthode appelées (variable message).

Le test vérifie l'égalité entre deux chaînes de caractères celle prévue et celle obtenue. Comment calculer l'égalité entre deux chaînes ?

La vérification du résultat global se fait grâce aux tests d'intégration (ici grâce à la classe **Simple**).

2.3 Fin de l'itération

Quand la réalisation des classes est terminée, rassemblez tous vos sources dans un seul répertoire. Compiler le programme principal. Comparer votre exécution avec le commentaire donné à la fin du fichier `Simple.java`.

Corriger votre réalisation jusqu'à trouver la même exécution.

2.4 Bonus-Objets Mock – Mockito `http://site.mockito.org/`

En se basant sur la spécification des objets Mock du framework Mockito, modifiez les précédents tests des faussaires pour utiliser un mock de l'Environment. Pensez à : *m*ocker (when...)

3 Nouvelle architecture

Présenter sur un diagramme de classe la nouvelle architecture obtenue.