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 *tec-source.zip* disponibe sur Moodle (resources du projet).

Table des matières

1 Première itération : Tout doit compiler

Voici, les objectifs de cette itération :
— compiler les fichiers sources distribués ;
— ajouter les classes **EtatPassager**, **JaugeNaturel** et leurs tests dans le paquetage `tec` ;
— créer une version minimale des classes à développer.

1.1 Le paquetage `tec`

Après extraction du fichier archive tec-source.tar dans votre répertoire de travail, vous trouverez :
— l'exemple du programme principale fourni par le client `Simple.java` ;
— et dans le répertoire tec, les fichiers sources de départ du paquetage `tec` :
  — deux interfaces publiques,
  — deux interfaces privées au paquetage,
  — la classe définissant l'exception contrôlée,
  — une classe permettant de lancer la suite des tests du développeur.

⇒ *Pour les classes appartenent à un paquetage, le langage Java impose une contrainte physique sur les fichiers. Précisez cette contrainte pour les classes appartenant au paquetage* `tec`

⇒ *Rappelez le but des deux interfaces privées*

Remarque : La classe du programme client n'appartient pas au paquetage `tec`.

1.2 Compilons l'existant

⇒ *En consultant de la documentation du paquetage, donnez les dépendances de compilation entre les interfaces et les classes.*

Vérifier que tous les fichiers sources fournis sont compilés.

1.3 Ajouter les classes au paquetage

Dans la description du sujet, il est demandé d'utiliser :
— des instances de la classe **EtatPassager** dans la réalisation de la classe **PassagerStandard** ;
— des instances de la classe **JaugeNaturel** dans la réalisation de la classe **Autobus**.

La documentation indique que ces classes et leur classe de test sont des classes privées au paquetage `tec`.

Pour `PassagerStandard` recopie dans son répertoire `tec` les fichiers pour la classe **EtatPassager**. Prenez la version qui passe tous les tests.

Pour `Autobus` recopie dans son répertoire `tec` les fichiers pour la classe **JaugeNatuel**. Prenez la verion qui passe tous les tests et avec les exceptions. Modifiez les fichiers sources de ces classes. Compilez et corrigez les erreurs.

1.4 Compilons les nouvelles classes

Version minimale de la classe PassagerStandard ou Autobus.
Elle est créée de la manière suivante :
— La classe fait partie du paquetage `tec`.
— Elle est déclarée publique, pour être instanciée au dehors du paquetage (voir **Simple**).
— Elle réalise les bonnes relations de type/sous-type suivant le diagramme de classe.
— Son constructeur a un corps vide mais avec la bonne liste de paramètres.
— Ses méthodes ont un corps vide ou bien seulement une instruction de retour (`return false` pour une valeur booléenne et `return null` pour une valeur référence)

Corriger les erreurs jusqu'à compilation compléte de votre classe.

Version minimales de la classe PassagerStandardTest ou AutobusTest.

Ses méthodes ont un corps vide.

Remarque : La classe **Simple** sert de test d'intégration.

Corriger les erreurs jusqu'à la compilation compléte des classes.

1.5 Fin de l'itération

Maintenant nous sommes assurés que tout compile et s'exécute. Pour terminer cette itération, vérifiez que les tests des classes **EtatPassager** sont toujours OK.

2 Deuxime itèration : Instanciation et Changement d'état

Pour cette itération, nous allons employer seulement une partie de la spécification :

— Les interactions entre les deux classes concrètes **PassagerStandard** et **Autobus** ne sont pas codées. Il n'y a pas de messages envoyés entre ses deux classes. Les diagrammes de séquence ne sont pas utilisés.

— Les passagers ne sont pas stockés dans la classe **Autobus**.

Ces deux points sont pris en compte dans l'itération suivante.

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).

Le code n'est pas complet (par rapport à la spécification) mais il faut quand même le tester.

Au lieu de coder entièrement votre classe puis d'écrire les tests, il est plus judicieux d'écrire les tests et de les valider au fur et à mesure de l'écriture du code de la classe en développement (c.f. le cours sur le Développement Dirigé par les Tests).

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).

2.1 Ordre des tests

2.1.1 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**.

2.1.2 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.

Pour la classe 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.

Pour la classe 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**.

2.2 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).

3 Troisème itération : interaction et stockage des passagers

Cette itérations prend en compte les deux points laissés de coté dans l'itération précédente.

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.

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).

Des faussaires pour tester

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

3.1 Les classes faussaires

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).

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.

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.

3.2 Ordre des tests

PassagerStandard

Ecrire le test de **monterDans** puis de **nouvelArret**

⇒ *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

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

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**).

3.3 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.

4 Quatrième itération : Reste quelques problèmes

Voici les objectifs :
— tester la levée d'une exception dans les cas d'erreur découverts dans le code,
— étudier le masquage d'information du paquetage,
— remanier un peu le code (Ajouter des constructeurs).

4.1 Indiquer et tester les erreurs

Hélas, la spécification est incompléte. Elle ne fournit pas les cas d'erreur à considérer.

A partir du code développé dans les itérations précédente, précisez et compléter les cas d'erreur. Par exemple :
— paramètres d'instanciation invalides,
— état du passager incohérent,
— destination $<$ numéro arrêt
— passager stocké deux fois dans un transport,
— problème de conversion de type.

Ces cas d'erreur doivent être détectés et lever une exception capturée par le programme du client.

Sauf pour les deux méthodes **monterDans**() et **allerArretSuivant**(), il n'y a pas de précision sur la catégorie de l'exception à utiliser.

Voici la réalisation attendue :
— Les instanciations et toutes les méthodes définies par les deux interfaces privées **Bus** et **Passager** lévent forcément une exception non controllée. Vous pouvez vous servir de la classe **IllegalArgumentException** ou **IllegalStateException**
— La spécification des deux méthodes **monterDans**() et **allerArretSuivant**() indique l'exception controllée propagée. Le code de ces méthodes va donc capturer l'exception non contrôlée et léver à la place l'exception **UsagerInvalideException**.

Pour tester la levée de ces exceptions, il est peut-être nécessaire de définir d'autres classes faussaires.

Pour garder vos tests lisibles, vous pouvez écrire les tests des exceptions dans une autre classe de test `PassagerStandardExceptionTest` par exemple.

4.2 Masquage d'information du paquetage

D'après la spécification, certaines classes/interfaces/méthodes ne doivent pas être visibles par le client.

Vérifier ce qui accessible par le client c'est à dire les portées publiques.

Boostez vos neurones ;-)

Vous devriez rencontrer un problème de portée provenant de la construction des interfaces java.

⇒ *Proposer une solution pour éviter ce problème de portée ?*

Désavantage de cette solution ?

Remanier votre code pour mettre en place cette solution.

4.3 Un peu de remaniement de code

1. Ajouter un deuxième constructeur mais sans dupliquer le code d'initialisation :
   - le constructeur **Autobus**(**int nbPlace**) où le nombre de places assises et le nombre de places debout sont égaux au paramètre "nbPlace".
   - le constructeur **PassagerStandard**(**int destination**) où le nom du passager est la concaténation du nom de la classe avec la valeur du parmètre "destination".

   Expliquer l'instanciation avec ce deuxième constructeur (tracer les appels).

⇒ *Dans quel cas faudrait-il tester ce nouveau code ?*

4.4 Fin de l'itération

Fournir (sous forme électronique ou papier) le diagramme de classe complet de l'application.

Rassembler le code. Vérifier la compilation et les tests. Archiver votre délivrable en utilisant la commande tar.

A la fin de cette étape, il reste au moins deux problèmes qu'il faudrat veiller à prendre en compte dans la prochaine version.

⇒ *Expliquer le problème de duplication de code qui reste dans le code*

⇒ *Expliquer le problème de duplication d'instance qui reste dans le code*

⇒ *Avez-vous noté d'autres problèmes ?*

5 Nouvelle architecture

Présenter sur un diagramme de classe la nouvelle architecture obtenue.