Encapsulation

Les classes `Autobus` et `PassagerStandard`

Chaque tandem va prendre en charge la réalisation d'une des classes **`Autobus`** et **`PassagerStandard`**. Le développement des deux classes va s'effectuer en parallèle.

Nous vous fournissons la documentation (diagramme UML, documentation des classes) et les tests unitaires fonctionnels de chaque classe.

Table des matières

1 Description de l'approche orientée objet

Pour développer en parallèle les classes **`Autobus`** et **`PassagerStandard`**, il faut avoir une idée des dépendances entre ces deux classes.

Ces deux classes découlant du découpage de la version en langage C, la lecture de cette version donne une idée des dépendances :

Le code d'`autobus`
: déclenche des traitements de `passager standard`. Un attribut d'`autobus` stocke des `passagers standards`.

  Dans l'approche orientée objet, cette dépendance correspond au lien «a-un»[1] introduit dans le cours. Une instance de la classe **`Autobus`** a des instances de la classe **`PassagerStandard`**.

Le code de `passager standard`
: déclenche des traitements d'`autobus`. Ici, pas d'attribut, l'`autobus` est passé en paramètre de chaque traitement[2].

Conclusion, l'écriture d'autobus ne peut pas se faire sans passager standard et l'écriture de passager standard ne peut pas se faire sans autobus.

Pour changer cette dépendance, nous allons introduire deux abstractions (deux types abstraits de données).

1.1 Programmer avec des abstractions

Nous définissons deux abstractions (types abstraits) **`Vehicule`** et **`Passager`** mises en œuvre par des interfaces Java :

- L'interface **`Vehicule`** contient la déclaration des méthodes de l'encapsulation **`Autobus`**[3].

- L'interface `Passager` contient la déclaration des méthodes de l'encapsulation **`PassagerStandard`**[4].

Les relations entre classes sont maintenant :

- La classe concrète **`PassagerStandard`** réalise/implémente le type abstrait défini par l'interface **`Passager`**. Elle «est un (sous-type de)» **`Passager`**.
- La classe concrète **`Autobus`** réalise/implémente le type abstrait défini par l'interface **`Vehicule`**. Elle «est un (sous-type de)» **`Vehicule`**. La classe **`Autobus`** ne posséde plus de lien «a-un» avec **`PassagerStandard`** mais avec **`Passager`**.

Ces relations sont visualisées sur cet extrait du diagramme de classes.

La définition de liens «est-un (sous-type de)» permet la substitution d'objets.

Ce qui va se révéler important dans les tests unitaires et dans la suite du développement dans l'ajout de nouveaux caractères de passagers.

1.2 Tests unitaires avec objets factices

Pour assurer des tests unitaires du comportement des instances **`PassagerStandard`** et de celui des instances d'**`Autobus`**, il est nécessaire d'utiliser des objets factices («mock object »)[5].

Un objet factice possède les mêmes méthodes que l'objet «réel» mais sa réalisation est dédiée aux tests. Parfois différentes réalisation d'objets factices sont nécessaires pour simuler tous les comportements de la classe à tester.

Pour effectuer votre développement en parallèle, nous vous fournissons les tests unitaires et les deux classes faussaires :

- La classe de test `TestAutobus` qui utilise la classe factice `FauxPassager`.
- La classe de test `TestPassagerStandard` qui utilise la classe factice `FauxVehicule`.

En phase de production/exploitation, une instance d'**`Autobus`** s'éxécute avec des instances de **`PassagerStandard`** (l'objet «réel »). En phase de test, cette instance d'**`Autobus`** s'éxécute avec des objets factices (ici des instances de **`FauxPassager`**).

Cet échange/cette substitution se fait sans modifier le code des classes **`Autobus`** et **`PassagerStandard`** C'est la définition des liens «est-un» qui va permettre la substitution des objets factices/réels.

La classe **`FauxVehicule`** «est-un (sous-type de)» **`Vehicule`**. Et la classe **`FauxPassager`** «est-un (sous-type de)» **`Passager`**.

2 Développement des classes `PassagerStandard` et `Autobus`

Vous pouvez vous répartir les deux classes à réaliser.

La mise en œuvre de ces classes s'effectue à partir :

- De la documentation de `tec` (n'incluant pas la documentation sur les tests).
- Des diagrammes UML suivants : le diagramme de classes et deux diagrammes de séquence indiquant l'enchaînement des envois de messages à la montée d'un passager standard dans un autobus, et à chaque changement d'arrêt.

  Ces diagrammes de séquence traduisent dans une approche orentée objet les interactions de la version en langage C.

- Des fichiers sources Java des classes de tests, faussaires et des classes **`Jauge`** et **`Position`**.

  Le développement de chaque classe n'a pas besoin de tous les fichiers sources (seules les deux interfaces sont communes aux deux développements).

  L'archive src-passagerStandard.tar contient les sources nécessaires pour le développement de **`PassagerStandard`** et l'archive src-autobus.tar les sources nécessaires pour le développement de **`Autobus`**

Dans votre répertoire de travail :

1. Récupérer l'archive correspondant à votre développement.
2. A l'aide de vos classes de tests **`TestJauge`** et **`TestPosition`**, vérifier la réalisation de **`Jauge`** ou de **`Position`** et modifier la réalisation de ces deux classes si nécessaire.
3. Vérifier la compilation et l'exécution des ces sources en réalisant une version minimale (qui compile) de votre classe à développer.

   Cette version minimale peut se faire de la manière suivante :

   - Définir la classe concrète sans aucun attribut.
   - Pour définir les méthodes d'instance indiquées par la documentation de la classe, copier les méthodes d'instance déclarées dans l'interface [6]. Ces méthodes ont un corps vide ou uniquement une instruction **`return`** avec une valeur de retour qui fait echouer les tests (par exemple **`false`** par une valeur de retour booléenne).
   - Définir un constructeur (prototype dans la documentation de la classe) avec un corps vide.
   - Déclarer que la classe implémente (est sous-type de) l'interface avec le mot clé **`implements`** (voir la documentation de la classe).
4. Exécuter la classe de test, les tests doivent échouer.

Le travail d'écriture de la mise en œuvre peut commencer.

Pour une approche itérative, vous pouvez commenter tous les appels aux méthodes de test dans la méthode **`main`** et enlever ces commentaires au fur et à mesure de l'écriture du code.

Précision sur la réalisation

- Vous devez utilisez un tableau pour stocker les passagers et non pas une **`Collection`**.
- La gestion d'erreurs n'est pas prise en compte, le code ne fait aucun contrôle sur la validité des paramètres, ni sur le dépassement de stockage des passagers.
- La documentation d'une classe correspond aux commentaires commençant par `/**` dans le source (par exemple la classe **`Position`**. La commande **`javadoc`** du `jdk` permet de générer une documentation au format `html` (celle qui est en ligne).
- La méthode **`toString()`** réécrite dans les classes **`Jauge`**. et **`Position`** fournit pour le débogage une chaîne de caractères construite à partir des attributs de la classe.

3 Boutez vos neurones

3.1 Le source de la classe `String`

Le projet OpenJDK[7] met à disposition les sources «open-source» du jdk. Les sources des classes de l'API et de la machine virtuelle sont accessibles sur les dépots OpenJDK. Par exemple le source de la classe

`java.lang.String` pour java 8 se trouve sous le projet jdk8, répertoire jdk, menu browse, répertoire src/share/classes/java/lang.

Quelques questions sur la mise en oeuvre de l'égalité dans cette classe :

- L'opérateur `==` :

  Dans les classes de test utilisant les faussaires la vérification des appels se fait de cette manière : `"monteeDemanderAssis" == getLastLog(faux)`[8]. Ce test d'égalité utilise l'opérateur `==` entre deux chaînes de caractères littérales : celle attendue et celle stockée dans l'instance référencée par la variable **`logs`**. D'après la documentation, toutes les chaînes de caractères littérales sont des instances de la classe **`String`**.

  *Pour employer l'opérateur `==` à la place de la méthode **`equals()`**, quelle hypothèse est faite sur les instances de **`String`** représentant les chaînes de caractères littérales ?*

- la méthode **`equals(Object anObject)`** :

  Comment est calculée l'égalité dans le code de cette méthode ?

  Expliquer l'intérêt de la première ligne du code de la méthode.

  *Expliquer pourquoi l'instruction **`String anotherString = (String) anObject`** est-elle nécessaire ? La conversion de type échouera-t-elle à la compilation ou à l'exécution ?*

1 En UML, c'est une catégorie d'association.

2 Dans ce cas, ce n'est pas un lien «a-un» car il n'y a pas d'attribut. En UML le terme utilisé est dépendance (fonctionnelle).

3 Par rapport à la version en langage C, le paramètre dans ces méthodes de type **`PassagerStandard`** est changé en **`Passager`**.

4 Par rapport à la version en langage C, le paramètre dans ces méthodes de type **`Autobus`** est changé en **`Vehicule`**.

5 voir le cours de PG106

6 Cette classe concrète réalise/implémente le type abstrait défini par l'interface

7 Les distributions Linux installent par défaut les exécutables obtenus à partir du projet OpenJDK et non ceux fournis par Oracle. Pour avoir une idée du problème (licence GPL), vous pouvez consulter l'article de Richard Stallman «Free but Shackled - The Java Trap (2004)» (une traduction existe ici).

8 Extrait de la méthode **`testInteractionMontee()`** dans la classe **`TestPassagerStandard`**.

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