Programmation Orientée Objet
TD 4 : Substitution d'objets
Compte-rendu de l'équipe *Lorem Ipsum* (G4E1)

Coordinateur : Tom MOËNNE-LOCCOZ, Tandem 1 : Rémi DEHENNE, Aymeric FERRON,
Tandem 2 : Sébastien DELPEUCH, Aurélien MOINEL

2 octobre 2020

Résumé

Dans le cadre de cette quatrième séance de TD des modules PG202–PG203 à l'ENSEIRB-MATMECA, nous comparons deux méthodes permettant de masquer des informations au client. Au lieu de simplement définir des méthodes d'instance avec des portées différentes, on cherche à séparer les méthodes internes des méthodes à exposer.

1 Découplage entre méthodes internes et méthodes exposées au client

Lors de la séance précédente, nous avons regroupé les classes partageant une même logique au sein de paquetages. Désormais, on souhaite utiliser ces paquetages pour *masquer* certaines méthodes des interfaces `Vehicule` et `Passager`.

Dans le code initial, toutes les méthodes des interfaces `Vehicule` et `Passager` sont déclarées comme publiques. On souhaiterait pourtant que certaines méthodes de ces interfaces deviennent internes au paquetage et donc inacessibles au client. Cependant, le langage Java ne permet pas de déclarer des méthodes de portée *paquetage* dans une interface. En effet, pour une méthode d'instance déclarée ou définie dans une classe, la portée `_paquetage_` est la portée par défaut, lorsqu'aucun autre mot-clé de portée n'est utilisé. En revanche, pour une méthode d'interface, la seule portée autorisée est **`public`** : il n'est donc pas possible de spécifier une portée *paquetage* sur une méthode déclarée dans une interface.

Afin de contourner cette contrainte, on décide de séparer les méthodes d'instance internes de celles qu'on souhaite exposer dans deux interfaces distinctes : les méthodes internes sont conservées dans `Vehicule` et `Passager`, les méthodes à exposer dans de nouvelles interfaces `Transport` et `Usager`. Par conséquent, un objet `Autobus` peut être référencé en tant qu'implémentation de `Vehicule` ou

de `Transport`, et `PassagerStandard` en tant qu'implémentation de `Passager` ou d'`Usager`. Un tel découplage permettra d'implémenter les solutions de masquage détaillées en section 2.

Ainsi, dans le cas de la classe `Autobus`, la nouvelle interface `Transport` contient uniquement la méthode `allerArretSuivant()`, qu'on souhaite accessible par le client. Les autres méthodes telles que `aPlaceDebout()` ou `monteeDemanderAssis()` sont conservées dans `Vehicule` et sont uniquement destinées aux mainteneurs du paquetage.

De ce fait, le code client et les méthodes utilisables par celui-ci doivent être modifiées en conséquence. Par exemple, le prototype de la méthode `monterDans(Vehicule v)` de l'interface `Usager` doit être modifié pour que le paramètre demandé soit une référence de `Transport` (utilisable par le client), et non de `Vehicule` (interne).

Toutefois, l'implémentation de `monterDans()` par la classe `PassagerStandard`, reproduite en Code 1, appelle des méthodes déclarées dans l'interface `Vehicule` et non dans `Transport`. Pour ce faire, il est nécessaire de transtyper la référence de `Transport` en référence de `Vehicule` pour appeler ces méthodes. Néanmoins, cela suppose que toute instance passée en argument de `monterDans()` soit une instance d'une classe qui implémente à la fois `Transport` et `Vehicule`. Même si la classe `Autobus` respecte bien cette condition, cela n'est pas vrai en général pour toute classe concrète : à l'exécution, la machine virtuelle Java doit donc vérifier que le transtypage est valide, c'est-à-dire que la classe concrète peut être référencée comme un `Vehicule`. Dans le cas contraire, une exception `ClassCastException` est levée.

Code 1 : Transtypage d'un Transport en Vehicule

```java
// Fichier  : PassagerStandard.java

@Override
    public void monterDans(Transport t){
        Vehicule b = (Vehicule) t;
        if(b.aPlaceAssise()) {
            b.monteeDemanderAssis(this);
        } else if(b.aPlaceDebout()) {
            b.monteeDemanderDebout(this);
        }
    }
```

Nous pourrions décider de vérifier avant transtypage que la classe concrète référencée par la variable est bel et bien une classe dérivée de `Vehicule`, à l'aide d'un test `if (t instanceof Vehicule)`. Cependant, nous jugeons préférable de conserver le mécanisme d'exceptions afin de mieux repérer les comportements indésirables à l'exécution[1].

1. Il serait néanmoins possible de « surcharger » la `ClassCastException` levée par Java, en levant une autre classe

Pourtant, ce découplage des méthodes dans deux interfaces, une interne, l'autre exposée au client, ne suffit pas à masquer les méthodes « internes ». En effet, rien n'empêche le client de créer des références de `Vehicule` ou de `Passager`, ou même d'appeler des méthodes « internes » directement sur des références d'`Autobus` et de `PassagerStandard`.

2 Masquage des méthodes internes

On souhaite désormais masquer les méthodes au client et comparer deux solutions, implémentées en parallèle sur des branches Git. La première solution, implémentée par le tandem 2 (Aurélien et Sébastien), utilise une classe abstraite au lieu des interfaces « internes ». La seconde proposition, adoptée par le tandem 1 (Aymeric et Rémi), consiste à appliquer le patron de conception « fabrique ».

2.1 Solution 1 : Modification des portées

La solution la plus évidente consisterait à rendre les méthodes déclarées dans les interfaces `Passager` et `Vehicule` internes au paquetage. Ainsi, même si le client instancie la classe concrète, il ne pourrait pas avoir accès aux méthodes de cette interface. Cependant, comme indiqué en section 1, les méthodes d'une interface sont publiques par défaut, et ne peuvent être déclarées internes au paquetage.

Pour contourner ce problème, il est possible de remplacer l'interface — au sens Java — par une classe abstraite. Théoriquement, une interface est synonyme de *classe abstraite pure* (classe dont aucune méthode n'est définie), mais en Java, ces deux mécanismes sont distincts. Contrairement à une interface, les méthodes d'une classe abstraite sont internes au paquetage par défaut. En revanche, une classe Java ne peut hériter directement de plusieurs classes (abstraites ou non), mais peut implémenter plusieurs interfaces[2].

Ainsi, on définit `Vehicule` et `Passager` comme des classes abstraites de portée paquetage, et dont les méthodes sont également internes au paquetage. Ces méthodes sont alors également internes au paquetage dans la classe dérivée, donc inaccessibles par le client. Une telle solution est modélisée en Figure 1.

d'exception et/ou un autre message, plus précis, lorsque la classe concrète n'est pas dérivée de `Vehicule`.

2. Depuis Java 8, une interface peut également définir des méthodes par défaut, ce qui n'est pas compatible avec la définition de classe abstraite pure. Ces interfaces s'apparentent plus à des *traits*, présents dans d'autres langages (Rust, Racket, C++, etc).

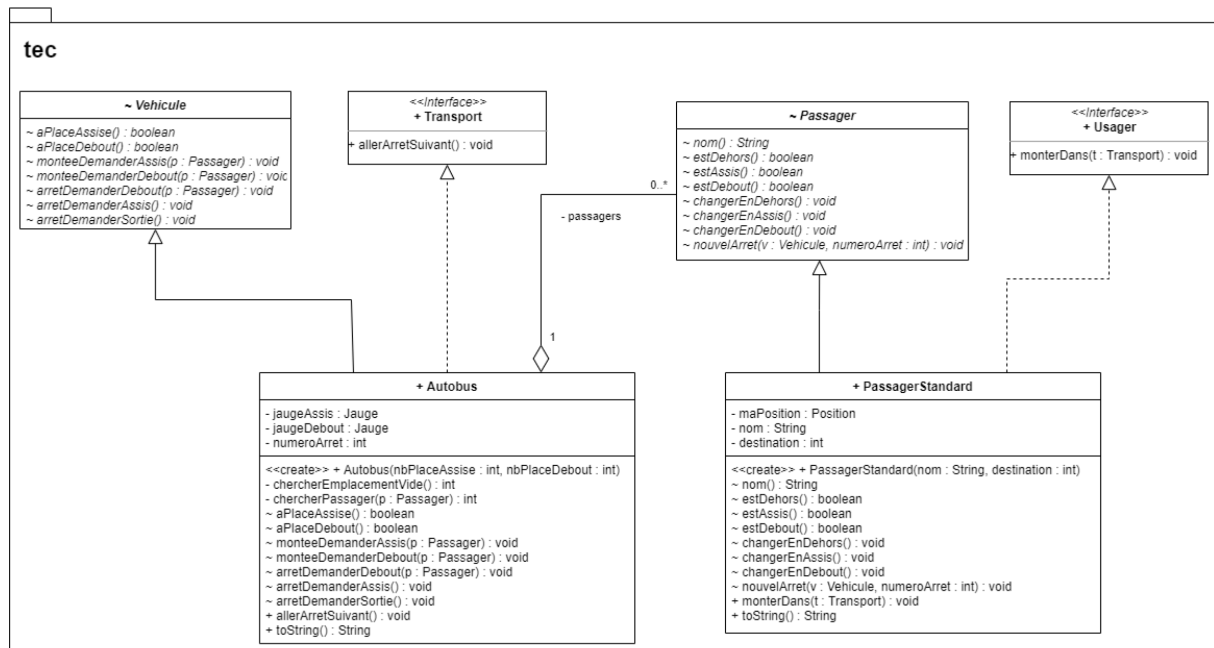

FIGURE 1 : Diagramme de classes (solution abstract)

2.2 Solution 2 : Utilisation d’une fabrique

Une autre approche consiste à masquer au client non pas les méthodes internes, mais les interfaces et les classes concrètes elles-mêmes, via l’utilisation d’une fabrique.

Dans un premier temps, les interfaces à exposer sont déclarées publiques, les autres interfaces et classes étant internes au paquetage `tec`. De ce fait, le client ne peut utiliser les classes concrètes `PassagerStandard` et `Autobus` ou les interfaces internes, seulement les interfaces publiques.

Par conséquent, il n’est plus possible pour le client d’instancier les classes concrètes `PassagerStandard` et `Autobus`. Il s’agit donc de trouver un moyen pour celui-ci d’instancier indirectement des `PassagerStandard`s et des `Usager`s, mais d’interagir uniquement avec des références vers les interfaces publiques `Usager` et `Transport` . Pour ce faire, on utilise le principe de fabrique : une classe, ici `FabriqueTec`, est interne au paquetage `tec` et expose deux méthodes publiques `fairePassagerStandard()` et `faireAutobus()`. Ces méthodes renvoient respectivement une instance de `PassagerStandard` et d’`Autobus`, en appelant leurs constructeurs. Elles servent ainsi de couche d’abstraction pour instancier ces objets et ne pas exposer les classes concrètes au client. Finalement, le client ne peut référencer ces instances que comme des implémentations d’`Usager` ou de `Transport`.

Dans le cadre de ce TD, on décide que les méthodes de la fabrique appartiennent à la classe `FabriqueTec` et non à ses instances. De même, le constructeur par défaut est surchargé et déclaré

privé, afin d'empêcher toute instanciation de `FabriqueTec`. Il s'agit cependant d'un choix d'implémentation de la fabrique, qui ne correspond pas au patron défini par le *Gang of Four* dans *Design Patterns : Elements of Reusable Object-Oriented Software*.

Un `Autobus` peut alors être instancié par le client en appelant la méthode de classe `faireAutobus()` de `FabriqueTec`, et référencé en tant que `Transport`, comme présenté en CODE 2

CODE 2 : Instanciation d'un Autobus par le code client

```
Transport serenity = tec.FabriqueTec.faireAutobus(1, 2);
```

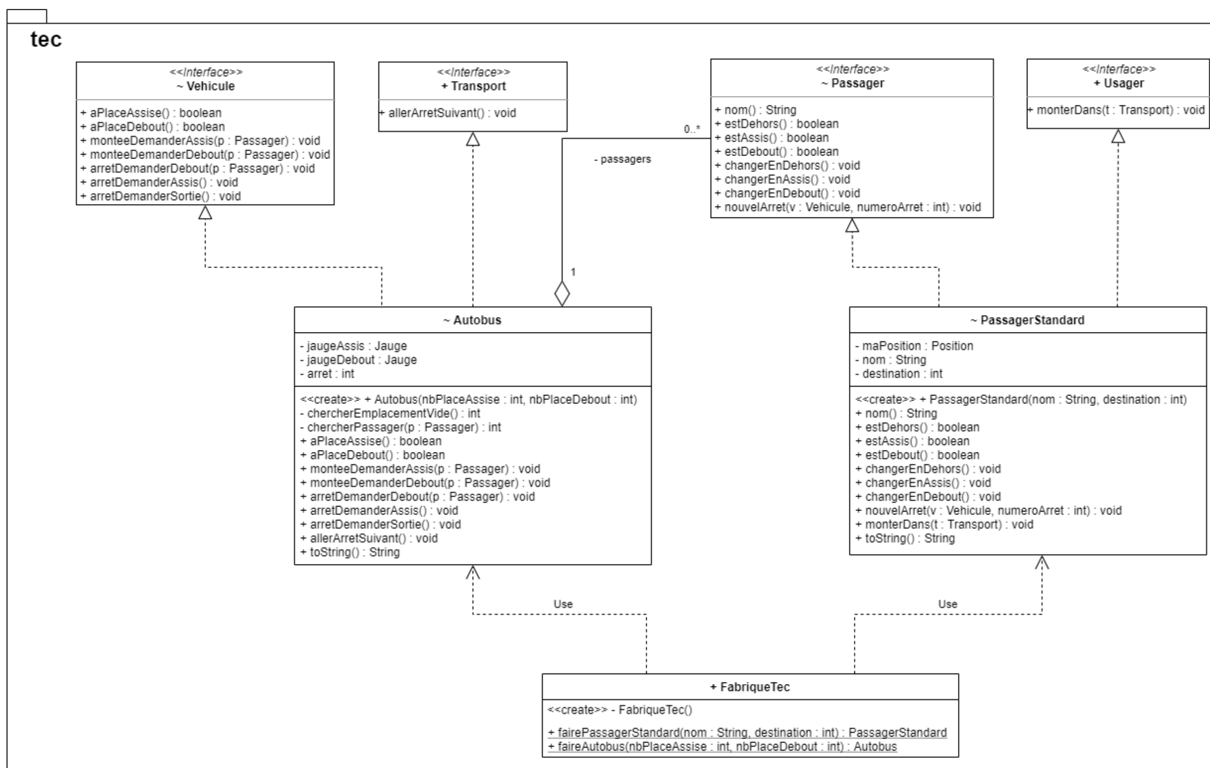

FIGURE 2 : Diagramme de classes (solution interface)

2.3 Comparaison

Finalement, la solution de fabrique semble la plus maintenable sur le long terme. En effet, si l'on souhaitait modifier `Autobus` pour implémenter de la même façon un système de tarification, la première solution impliquerait d'implémenter une interface publique `Tarification` et d'hériter d'une classe abstraite interne `TarificationInterne` dans `Autobus`. Puisque l'héritage multiple n'existe pas en

Java, il faudrait peut-être « fusionner » les classes abstraites `TarificationInterne` et `Vehicule`, alors que la logique de ces deux classes sont *a priori* indépendantes. Cette utilisation d'une classe abstraite pure afin de créer une interface avec des méthodes internes au paquetage relève plus d'une solution de « contournement » (*workaround*) que d'une vraie solution idiomatique en Java. Cette solution est néanmoins rapide à implémenter et facile à comprendre.

Le patron de conception « fabrique » permet d'abstraire et de masquer davantage d'informations. D'une part, les classes concrètes et les interfaces internes ne sont pas exposées au client ; ainsi, le code client n'accède pas directement à la classe ou à son constructeur, qui peuvent donc être modifiés ou renommés sans impacter ledit client. D'autre part, la fabrique permet également d'ajouter de nouvelles classes dérivées en modifiant uniquement le code la fabrique, ou même d'ajouter de nouvelles interfaces aux classes concrètes existantes. Ainsi, nous pourrions ajouter une classe concrète `Train` implémentant `Vehicule` et `Transport` en modifiant simplement le code de la fabrique. On ajouterait ainsi une méthode `faireTrain()` à la fabrique, ou on créerait une nouvelle fabrique `FabriqueTrain` avec une méthode `faireVehicule()` qui instancie un `Train`[3]. De même, cette classe `Train` pourrait, de façon analogue, implémenter une interface `Retard` afin de gérer les différents incidents potentiels (régulation du trafic, grèves, pannes de signalisation, intervention des forces de l'ordre, intervention des démineurs, feuilles sur la voie, etc).

De manière générale, le patron de conception « fabrique » est un patron de conception éprouvé et très utilisé, qui respecte le principe *ouvert/fermé*[4]. Il permet d'abstraire les classes concrètes d'un module pour le code client qu'il utilise, mais également de faciliter la création de certains objets complexes.

Conclusion

Finalement, ce quatrième TD nous a permis de renforcer le principe d'indépendance entre les classes d'un paquetage et le code qui l'utilise. Grâce au patron fabrique, on abstrait à la fois le fonctionnement interne d'une classe *et* la classe elle-même, à l'aide d'interfaces publiques : le code client ne dépend même plus de l'identifiant de la classe, seulement de la fabrique. Enfin, cette séance a permis à certains membres de l'équipe de se familiariser avec le fonctionnement de *branches* Git.

3. En effet, au lieu de modifier une unique fabrique, on préfère définir une fabrique par classe concrète (`FabriqueAutobus`, `FabriqueTrain`), qui héritent toutes d'une même interface (`FabriqueVehicule`), afin de découpler davantage les différentes.

4. Principe orienté objet selon lequel une classe doit être « ouverte à l'extension et fermée aux modifications ».

Commentaires

Tom (Coordinateur) : Avoir passé beaucoup de temps à clarifier les mécanismes de portées au TD précédent a vraiment porté ses fruits durant ce TD. Dans l'ensemble, le sujet m'a semblé plus compliqué que les précédents. Il m'aurait donc été très dur de le suivre sans le travail effectué en amont. Nous avons longtemps réfléchi ensemble sur ce qui était demandé. Cela m'a été très bénéfique et j'ai ainsi pu avoir une vision claire de la majorité des subtilités apportées par le TD dès la fin de la séance. En tant que coordinateur, j'ai essayé de partager au mieux le fruit de notre réflexion à Sébastien qui était l'unique élève en distanciel. Cependant, cela s'est fait en décalage de notre discussion de groupe dont il a donc été grandement exclu. Pour des raisons principalement acoustiques, nous n'étions pas en mesure de l'ajouter naturellement à notre discussion. Il serait intéressant de discuter avec l'enseignant de méthode permettant de diminuer le bruit dans la classe, sans trop affecter la capacité des groupes à discuter.

Rémi (Tandem 1) : Cette séance s'est démarquée des précédentes : nous avons passé une partie non négligeable de la séance à modéliser sur papier les solutions proposées par le sujet, afin de comprendre et d'expliquer leur fonctionnement avant de les implémenter. Heureusement que la plupart de l'équipe était présente, sinon cela aurait été particulièrement compliqué.

Par ailleurs, il me semble qu'une alternative à la première solution aurait pu consister à « composer plutôt qu'à hériter » : au lieu de d'hériter de `Vehicule` et d'implémenter `Transport` dans `Autobus`, on aurait pu ajouter un attribut à `Autobus` référençant un `Transport`. `Vehicule` serait alors simplement une implémentation de `Transport`, qui définirait à la fois des méthodes internes et la méthode publique `allerArretSuivant()`. Toutefois, il aurait tout de même fallu modifier la classe `Autobus` , ajouter une nouvelle interface et utiliser une fabrique pour bénéficier des mêmes avantages que la seconde solution du sujet.

Aymeric (Tandem 1) : Au cours de ce TD, nous avons longuement étudié la théorie avant de se lancer dans le code. Ce moment était très intéressant car il a éclairci des points qui n'étaient pas forcément clairs. Une fois la structure du code bien en tête, il a été facile de coder. Rémi a commencé par m'expliquer comment fonctionnaient les branches sur Git, puis après en avoir créé une, j'ai créé la fabrique et modifié les portées.

Aurélien (Tandem 2) : Cette séance a été l'occasion d'aborder des concepts intéressants liés à la POO tels que le masquage d'information ou le patron de conception « fabrique », mais aussi de se confronter aux choix de conception propres au langage Java (portée des méthodes d'interfaces, absence d'héritage multiple…). Il a également été intéressant de confronter nos points de vue concernant l'implémentation des solutions, cela a été l'occasion d'éclaircir certains points pour l'équipe. Concernant l'organisation, je pense que si l'équipe avait majoritairement été en distanciel comme la séance précédente, il aurait été difficile de se répartir le travail, cela pourrait éventuellement poser problème

pour les futures séances de cet acabit.

Sébastien (Tandem 2) : Ce TD fut assez différent des autres, tout d’abord nous avons dû réfléchir à ce que nous allions mettre en place avant de commencer à coder. Étant donné que j’étais le seul membre de mon équipe à distance, j’avais du mal à participer aux réflexions. Cependant, mon équipe a pris le temps de me réexpliquer les différents concepts. J’ai donc pu saisir les concepts de fabrique et de classe abstraite plus précisément. Pour finir sur les conditions du TD, le fait d’être à distance par rapport au reste de l’équipe reste toujours complexe. Cependant, la mise en place d’un BBB « général » fut une bonne expérience, nous pouvions entendre les différentes consignes ce qui a rendu le TD plus facile à comprendre. Le passage dans les différents BBB des groupes est efficace. Il reste encore quelques soucis, par exemple pendant les phases de réflexion de l’équipe j’ai eu du mal à intervenir. De plus, la salle de TD est très bruyante ce qui est assez désagréable via BBB.