Programmation Orientée Objet
TD 5 : Nouveaux caractères
Compte-rendu de l'équipe *Lorem Ipsum* (G4E1)

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

9 octobre 2020

Résumé

Dans le cadre de cette cinquième séance de TD des modules PG202–PG203 à l'ENSEIRB-MATMECA, nous introduisons de nouveaux passagers avec différents caractères. Nous cherchons ici à factoriser au mieux le code commun de ces différents passagers par l'utilisation de l'héritage et de la rédéfinition de méthodes.

1 Héritage de classe concrète

Lors de la séance précédente, nous avons implémenté deux solutions différentes permettant de masquer certaines méthodes des interfaces `Vehicule` et `Passager` au client, par l'introduction des relations de type/sous-type. Dorénavant, on souhaite se servir de ces relations pour permettre au client d'instancier de nouveaux passagers aux comportements différents.

Afin de mettre en œuvre ces sous-typages, on souhaite créer de nouvelles classes concrètes héritant de la classe concrète `PassagerStandard`. Pour chaque classe dérivée, cela revient donc à redéfinir les méthodes dont le comportement est spécifique à un comportement de passager, ici `monterDans()` et `nouvelArret()`.

Nous désirons également factoriser le code commun entre nos différents passagers. Par exemple, pour la méthode `nouvelArret()`, le test vérifiant la sortie d'un passager est commun à l'ensemble de ses réalisations. Pour éviter de dupliquer ce code et de devoir le mettre à jour à plusieurs endroits en cas de modification, on souhaite factoriser une partie du code dans la classe mère `PassagerStandard`. Le mot-clé **`super`** fait référence à l'instance de la classe parente et permet, entre autres, d'appeler les méthodes qui y sont définies. Dans notre cas, la logique commune aux différents passagers est implémentée dans la méthode `nouvelArret()` de la classe parente `PassagerStandard`. Cette méthode `nouvelArret()` est par ailleurs redéfinie dans chacune des classes dérivées : dans un premier temps, la méthode parente est appelée à l'aide de la notation

`super.nouvelArret()`. Puis, le code spécifique au comportement du passager est ajouté, comme illustré en Code 1.

Code 1 : Redéfinition de la méthode nouvelArret() pour la classe PassagerStresse

```
// Fichier  : PassagerStresse.java

public abstract void nouvelArret(Vehicule v, int numeroArret){
    super.nouvelArret();
    if(v.aPlaceAssise()) {
        v.monteeDemanderAssis(this);
    } else if(v.aPlaceDebout()) {
        v.monteeDemanderDebout(this);
    }
}
```

Par ailleurs, lors d'un héritage, on souhaite garantir l'encapsulation de la classe parente et s'assurer que les classes dérivées n'accèdent pas aux attributs de leur mère, afin de garantir l'intégrité des données. Pour cela, la portée **`private`** des attributs d'une instance permet de garantir une certaine indépendance entre la classe mère et ses filles, et que les classes tierces (dérivées ou non) ne modifient pas l'état de la classe mère, en créant des états incohérents.

Pour répondre à ce besoin, on utilise un accesseur afin de consulter la valeur d'une donnée sans autoriser sa modification. Toutefois, une telle méthode ne garantit pas nécessairement un niveau d'abstraction satisfaisant : en effet, plus les accesseurs sont proches de l'implémentation et renvoient par exemple la valeur d'un attribut privé, moins le développeur peut modifier l'implémentation sans impacter le code client. Dans le cas de la classe `PassagerStresse`, une instance en position *assis* demande si possible une place *debout* dans les trois arrêts avant sa destination. Au lieu d'accéder à l'attribut `destination` de la classe mère `PassagerStandard` ou de créer un accesseur qui renvoie cette valeur de `destination`, la méthode `getDestination()` renvoie la valeur désirée par le code appelant, à savoir le nombre d'arrêts restants. Ainsi, la représentation des arrêts dans `PassagerAbstrait` peut être modifiée sans « casser » le code des classes dérivées.

D'autre part, lors d'une relation de type/sous type, un objet provenant d'une classe dérivée hérite des attributs et méthodes de la classe mère. Même si les attributs de cette classe mère ne sont pas accessibles par la classe fille, ceux-ci doivent être initialisés. Pour cela, le constructeur de la classe dérivée reçoit généralement en paramètres les valeurs nécessaires au constructeur de la classe mère, et, avant toute autre instruction, appelle ce constructeur parent en lui transmettant lesdits paramètres. L'appel à ce constructeur parent est réalisé par le mot-clé **`super`**`()`.

Pourtant, cet héritage de la classe concrète `PassagerStandard` n'est pas une solution maintenable à terme. En effet, si les spécifications des méthodes `nouvelArret()` et `monterDans()` de `PassagerStandard` venaient à changer, la modification de leur code modifieraient également le

comportement des classes dérivées. Par exemple, si le passager standard ne voulait rentrer dans un autobus que si la destination est à plus de quinze minutes à pied, cette modification s'appliquerait également aux `PassagerStresse` et `PassagerIndecis`. Par conséquent, il faudrait retirer l'appel `super.monterDans()` pour chacune des redéfinitions et dupliquer le code de montée précédent dans toutes les classes dérivées partageant le comportement. Un scénario similaire surviendrait si le comportement `nouvelArret()` d'un passager standard était lui aussi modifié. Dans une telle situation, la classe mère partagerait moins de code commun avec ses filles que les classes filles entre-elles.

2 Héritage de type abstrait

On souhaite maintenant séparer le comportement commun aux différents passagers des comportements propres au `PassagerStandard`. Pour cela, nous n'utilisons plus la classe concrète `PassagerStandard` comme classe mère des autres passagers, mais une classe abstraite `PassagerAbstrait`.

2.1 Définition des comportements : appel des méthodes de la super-classe

La classe abstraite `PassagerAbstrait` n'est par définition pas instanciable, puisqu'elle ne correspond pas à une réelle entité, mais sert « de base » aux autres classes.

Elle définit ainsi un certain nombre de méthodes avec un comportement par défaut, qui peuvent être éventuellement redéfinies par les classes dérivées. On décide, selon le besoin, que ces implémentations par défaut puissent ou non être redéfinies : les méthodes devant être communes à toutes les classes dérivées sont marquées comme **`final`** afin d'empêcher une redéfinition.

En revanche, d'autres méthodes peuvent ne pas proposer de comportement par défaut : les méthodes peuvent être simplement *déclarées* et non *définies*, à l'aide du mot-clé **`abstract`**. Ainsi, pour qu'une classe dérivée soit instanciable, elle doit impérativement définir toutes les méthodes abstraites héritées de sa classe mère. Cette solution permet de contraindre chaque classe concrète à définir un comportement propre, lorsqu'un comportement par défaut n'aurait pas de sens. La solution est représentée en Figure 1.

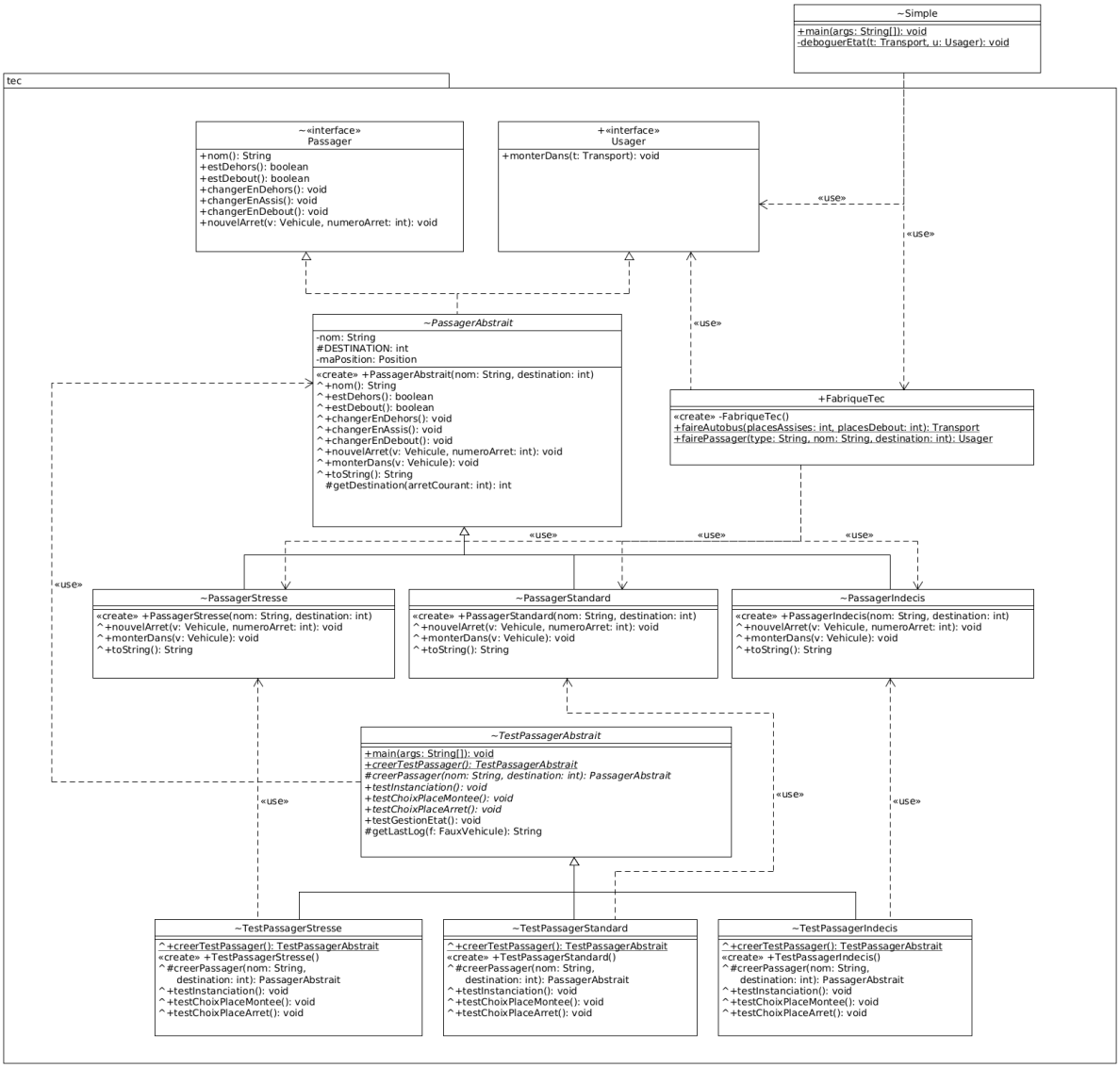

FIGURE 1 : Diagramme de classes UML de conception[1] avant utilisation du « patron de méthode » (solution *fabrique*)

2.2 Définition des comportements : utilisation du patron de méthode

À ce stade, les comportements communs aux passagers et ceux propres au `PassagerStandard` sont bien séparés dans deux classes distinctes. Cependant, le code pourrait être davantage factorisé. En ef-

1. Contrairement à un diagramme d'*implémentation* (proche du langage cible), le diagramme de *conception* relève davantage de la spécification : il est plus ou moins indépendant du langage de programmation utilisé.

fet, les méthodes `nouvelArret()` et `monterDans()` dans les classes dérivées appellent toutes la méthode correspondante dans la classe mère, par un **`super`**`.nouvelArret()` ou **`super`**`.monterDans()`.

Pour s'affranchir de ces appels redondants, nous mettons en place le patron de conception comportemental « patron de méthode ». Ce patron permet de fixer une partie du code d'une méthode, tout en faisant varier certaines parties en fonction du passager. Plus précisément, les comportements fixés sont définis dans les méthodes`nouvelArret()` et `monterDans()`, qui ne sont plus redéfinissables par les classes dérivées `PassagerStandard`, `PassagerStresse` et `PassagerIndecis` (mot-clé **`final`**)[2]. Ces méthodes sont appelées *méthodes de patron*. On ajoute alors deux nouvelles méthodes abstraites `choixPlaceMontee()` et `choixPlaceArret()` à `PassagerAbstrait`, qui correspondent aux portions de code qui varient d'un type de passager à l'autre. Les classes concrètes ne doivent définir que ces opérations `choixPlaceMontee()` et `choixPlaceArret()`, appelées *méthodes socles*. Celles-ci sont alors appelées par le code de `nouvelArret()` et `monterDans()` du `PassagerAbstrait`. Par conséquent, les méthodes dérivées n'appellent plus les méthodes héritées de la classe mère pour définir leur comportement. Au contraire, ce sont les méthodes de la classe mère qui appellent les opérations des classes dérivées, comme décrit en Code 2. Comme ces méthodes sont vouées à n'être utilisées que par les méthodes de la classe mère, mais qu'on souhaite pouvoir les redéfinir dans les classes dérivées, on utilise la portée **`protected`** qui autorise seulement les classes du paquetage et les classes dérivées à y accéder[3]. Le diagramme de classes en Figure 2 modélise cette nouvelle situation.

Code 2 : Patron de méthode 'monterDans()' de la réalisation du passager stressé

```
// Fichier  : PassagerAbstrait.java
@Override
public void monterDans(Transport t) { // Méthode de patron
    Vehicule v = (Vehicule) t;
    choixPlaceMontee(v);
}

protected abstract void choixPlaceMontee(Vehicule v);


// Fichier  : PassagerStresse.java
@Override
```

2. Dans certains cas, on pourrait autoriser la redéfinition `nouvelArret()` et `monterDans()` pour permettre également de remplacer le code « figé », si un type de passager se comporte radicalement différemment. Néanmoins, cela revient donc à ne plus respecter le *principe de substitution de Liskov*.

3. Dans notre cas, on pourrait décider d'utiliser la portée par défaut *paquetage*, mais ce n'est pas le comportement désiré. En effet, si l'on souhaitait ajouter des classes en dehors du paquetage `tec`, on ne pourrait pas redéfinir `choixPlaceMontee()` et donc pas créer de classe concrète dérivée.

```java
protected void choixPlaceMontee(Vehicule v) { // Méthode socle
    if(v.aPlaceAssise()) {
        v.monteeDemanderAssis(this);
    } else if(v.aPlaceDebout()) {
        v.monteeDemanderDebout(this);
    }
}
```

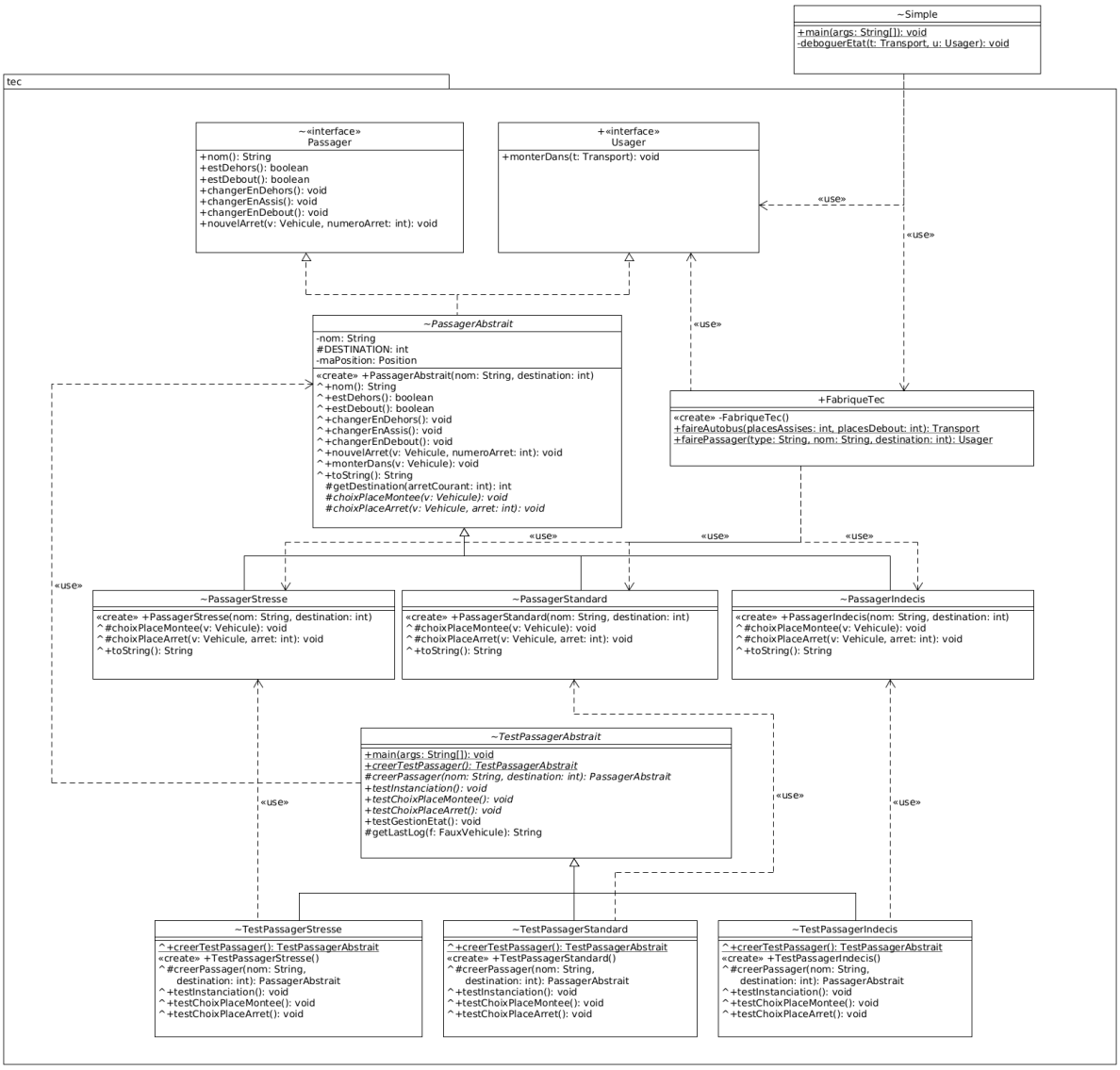

FIGURE 2 : Diagramme de classes UML de conception avec utilisation du « patron de méthode » (solution *fabrique*)

L'utilisation de ce patron de conception comportemental peut néanmoins susciter une interrogation :

comment les méthodes de patron définies dans la classe mère peuvent-elles appeler une méthode définie dans une classe fille ?

Pour ce faire, la méthode socle à appeler par la méthode de patron n'est connue qu'à l'exécution. Par des mécanismes d'aiguillage, la machine virtuelle Java exécute le code de la méthode socle qui correspond à la classe concrète de l'instance[4].

Par exemple, pour l'instance de `PassagerStresse` référencée par une variable `PassagerAbstrait`, la méthode `monterDans()` appellerait *à l'exécution* le code de `choixPlaceMontee()` définie dans `PassagerStresse`. Cela n'aurait pas pu être déterminé à la compilation : en effet, il n'est pas possible de connaître *à la compilation* la classe concrète de l'objet (`PassagerStresse`, `PassagerStandard` ou `PassagerIndecis`) lorsqu'une variable référence la classe mère (`PassagerAbstrait`). Ce mécanisme correspond au « polymorphisme d'exécution », ou « envoi dynamique de méthode » (*Dynamic Method Dispatch*).

Cette flexibilité permise par la relation de type/sous-type a donc un impact sur les performances du code à l'exécution. Nous étudierons en détail la notion de *types paramétrés* dans un projet ultérieur, un mécanisme permettant de réaliser certaines vérifications de type à la compilation plutôt qu'à l'exécution. Ce mécanisme est également survolé en section 3 afin de contourner certaines limitations du langage Java.

3 Factoriser les tests

De façon analogue, la technique de patron de méthode a été utilisée afin de factoriser les tests communs aux passagers, l'exécution de ces tests et de déclarer des tests spécifiques à une classe concrète.

3.1 Factorisation des méthodes de tests

La classe abstraite `TestPassagerAbstrait` définit le test commun `testGestionEtat()`, qui vérifie que lorsqu'un passager change de position, les méthodes `estAssis()`, `estDebout()` et `estDehors()` renvoient des valeurs cohérentes.

Cette méthode de test est toutefois redéfinissable par les classes dérivées, afin de permettre à un type de passager de se comporter différemment. Par exemple, un usager en fauteuil roulant n'occuperait jamais de place assise dans le bus : on pourrait souhaiter « interdire » sa méthode `changerEnDehors()`, soit en levant une exception, soit en ne modifiant pas sa position.

4. Oracle, *Method Invocation Expressions*. Consulté le 12 octobre 2020. https://docs.oracle.com/javase/specs/jls/se15/html/jls-15.html#jls-15.12

Les autres méthodes de tests sont simplement déclarées abstraites, et doivent être définies par les classes concrètes `TestPassagerStandard`, `TestPassagerIndecis` et `TestPassagerStresse` .

Toutefois, le code de la méthode `testGestionEtat()` de `TestPassagerAbstrait` doit instancier un `PassagerStandard`, un `PassagerIndecis` ou un `PassagerStresse` selon la classe de test concrète. Le patron de méthode permet à nouveau de résoudre ce problème : on déclare une méthode abstraite `creerPassager()` dans `TestPassagerAbstrait`, qui est implémentée par les classes dérivées (`TestPassagerStandard` instancie et renvoie un objet `PassagerStandard`, `TestPassagerIndecis` un `PassagerIndecis`, etc). Cette méthode est alors appelée par le code de `testGestionEtat()` de `TestPassagerAbstrait` pour chaque objet à créer.

Par ailleurs, on décide que la méthode `creerPassager()` renvoie une référence de `PassagerAbstrait` et non des interfaces `Usager` et `Passager`. En effet, on souhaite utiliser à la fois des méthodes déclarées dans `Passager`, celle déclarée dans `Usager` et celle propre à `PassagerAbstrait`. En référençant l'instance par un `Usager` et `Passager`, il faudrait systématiquement réaliser un *down-cast* pour appeler les méthodes sur une référence de `PassagerAbstrait`, qui pourrait échouer à l'exécution si l'instance implémente `Passager` ou `Usager` mais n'hérite pas de `PassagerAbstrait`. D'un point de vue plus conceptuel, on souhaite que `TestPassagerAbstrait` serve de « base » aux tests des classes filles de `PassagerAbstrait`, pas à l'ensemble des implémentations possibles de `Passager` et `Usager`.

3.2 Factorisation de l'exécution des tests

Enfin, la déclaration des méthodes de tests au niveau de `TestPassagerAbstrait` permet également d'exécuter la même suite de tests sur chacune des classes dérivées de `TestPassagerAbstrait`.

En utilisant de nouveau le patron de méthode, on souhaiterait définir une méthode de classe `main()` au niveau de `TestPassagerAbstrait` qui exécute les tests sur une instance de `TestPassagerStresse`, `TestPassagerStandard` ou `TestPassagerStandard` selon le fichier exécuté. Nous pourrions alors déclarer une méthode statique abstraite `creerTestPassager()` au niveau de `TestPassagerAbstrait` afin de créer des instances de `TestPassagerStresse`, `TestPassagerStandard` ou `TestPassagerStandard`. Selon la classe, `creerTestPassager()` créerait une instance du type souhaité.

Cependant, le langage Java ne permet pas de déclarer de méthode statique abstraite ni d'appeler une méthode statique d'une classe dérivée depuis la classe mère : cette implémentation n'est donc pas possible. À la place, on adopte l'approche opposée étudiée en sous-section 2.1 : chaque classe dérivée définit une méthode `main()` qui appelle une méthode d'exécution des tests `runTests()` de la classe mère.

Cette méthode `runTests()` nécessite par conséquent une référence qui indique quelle « sorte de » passager tester : on pourrait par exemple lui passer en argument un objet `TestPassagerStresse`, `TestPassagerStandard` ou `TestPassagerStandard`. Pourtant, pour garantir l'indépendance entre les tests, on a imposé lors du TD 1 que chaque test s'exécute sur une instance de test propre[5]. À la place, notre méthode `runTests()` prend en paramètre « toute classe `T` qui hérite de `TestPassagerAbstrait` ».

À partir d'un objet `Class`, on peut appeler le constructeur sans paramètre par la méthode `newInstance()`[6], ou en appelant la méthode `newInstance()` définie sur un objet `Constructor` de type paramétré, obtenu par application de `getConstructor()` sur l'objet `Class` (voir Code 3).

Code 3 : Échantillon du code d'exécution des tests

```
// Fichier : TestPassagerAbstrait.java
protected static <T extends TestPassagerAbstrait> void runTests(Class<T>
    psgCl) throws /* --snip-- */ {
     /* --snip-- */
     Constructor<T> construct = psgCl.getConstructor();
     /* --snip-- */
     construct.newInstance().testInstanciation();
     /* --snip-- */
     construct.newInstance().testGestionEtat();
     /* --snip-- */
   }
```

Enfin, une alternative que nous avons également implémentée cette semaine consisterait simplement à utiliser le mécanisme d'introspection, afin de déterminer à l'exécution toutes les méthodes d'une classe dont le nom commence par `test`, pour les appeler. Toutefois, cette approche « contournerait » l'objectif du TD, à savoir de se familiariser avec les notions de classes abstraites du TD et d'héritage. Néanmoins, l'ajout d'une méthode de test à une classe fille ne nécessite plus, avec cette solution, de déclarer la même méthode comme « abstraite » dans sa classe mère, de l'appeler dans `runTests()` et de la définir dans les autres classes filles. Les tests spécifiques à une classe fille seraient ainsi locaux à cette classe et indépendants vis-à-vis de la classe mère et des classes sœurs.

5. Cette précaution est de notre point de vue superflue, car les classes de test ne définissent pas d'attribut, ce qui limite l'existence d'effets de bord. Dans tous les cas, il est toujours possible d'ajouter des effets de bord en créant des attributs `static`, ce qui ne règle pas le problème que l'on souhaite empêcher.

6. Cette méthode est cependant dépréciée. Source : Oracle, *Class*. Consulté le 12 octobre 2020. https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Class.html#newInstance()

Conclusion

Finalement, ce TD nous a permis de manipuler des classes abstraites afin de définir à la fois des comportements par défaut, rédéfinissables ou non, et des comportements non définis que les classes concrètes dérivées doivent redéfinir.

Enfin, certaines contraintes propres au langage Java nous ont permis d'étudier les mécanismes de types paramétrées et d'appels de constructeurs dynamiques.

Commentaires

Sébastien (Coordinateur) : D'une part TD m'a permis de mieux comprendre le fonctionnement d'une classe abstraite surtout au moment de l'implémentation des tests. Je n'avais pas bien saisi le fait qu'une méthode abstraite devait être définie dans les classes filles et représentait le code qui change entre les différentes classes filles, alors que les méthodes non abstraites permettent de factoriser du code commun. D'autre part, ce TD s'est différencié par la partie explicative qui a occupé une bonne partie du TD. Cette partie m'a permis de bien comprendre ce qui était attendu et certains concepts de cours.

Rémi (Tandem 1) : En plus des concepts abordés dans ce TD, j'ai pu me rendre compte de certaines limites du langage Java en matière de méthodes statiques : même si les classes sont elles-mêmes des objets `Class`, une méthode statique ne peut pas appeler des méthodes statiques redéfinies dans une classe fille, de même qu'il n'est pas possible de déclarer des méthodes statiques abstraites.

Par conséquent, j'ai dû chercher une solution pour contourner cette contrainte, qui m'a permis de manipuler des types paramétrés sur des objets de `Class`. Toutefois, le code devient ainsi moins lisible, et souffre des problèmes évoqués en sous-section 2.1.

Finalement, j'ai trouvé que le maintien des deux branches en parallèle était laborieux, chronophage et sans grand intérêt, d'autant plus que le copier/coller de code entre les branches est une très mauvaise pratique et qu'une fusion ou qu'un `cherry-pick` entraîne des conflits « en cascade ».

Aymeric (Tandem 1) : Au cours de ce TD, j'ai pu améliorer ma compréhension des classes mères et des classes filles. En raison du sens des flèches sur le modèle UML, j'avais une vision inversée de la situation. Pour moi, comme `PassagerStresse` contenait l'implémentation des méthodes déclarées dans `PassagerAbstrait`, il était plus logique que `PassagerAbstrait` soit la fille de `PassagerStresse`. Aurélien et Rémi m'ont alors expliqué qu'il fallait voir la situation dans l'autre sens : il faut voir `PassagerAsbtrait` comme une mère possédant un certain nombre de méthodes implémentées et quelques autres déclarées. Cette classe donne alors naissance à deux filles qui

héritent des fonctions implémentées dans la mère mais qui ont alors leur propres particularités, i.e. les fonctions déclarées dans leur mère et implémentées chez elles.

Dans un deuxième temps j'ai réalisé les tests concernant `PassagerIndecis`. Grâce à ce travail, j'ai mieux compris l'interêt des classes faussaires que je n'avais pas encore eu l'occasion de manipuler. En effet, elles permettent d'utiliser rapidement des objets sans avoir à les instancier. Cela a un double interêt : permettre un gain de temps en utilisant directement un objet qui correspond à un contexte particulier et pouvoir rendre l'implémentation des tests indépendante des classes qui interagissent avec la classe que l'on teste. Cependant, un objet faussaire n'a pas le comportement exact d'une instance de l'objet qu'il imite. On doit alors se contenter de vérifier si les méthodes sont bien appelées avec les `logs` puisque l'objet que l'on teste n'est pas modifié par l'objet faussaire.

Aurélien (Tandem 2) : Cette séance de POO a été l'occasion d'aborder l'héritage entre objets, mais plus particulièrement d'optimiser la factorisation de code induit par cet héritage. Pour cela, l'introduction du patron de méthode a été une bonne approche, cependant celui-ci peut s'avérer limitant dans certains cas. En effet, si l'on considère que tout passager salut le conducteur à la montée de l'autobus, et que l'on décide de créer une classe `PassagerMalpoli`, alors on se retrouve dans le cas où il ne faut pas définir la salutation du conducteur comme comportement par défaut. On peut donc se demander s'il ne vaut pas mieux réaliser l'appel `super.monterDans()` pour chaque classe dérivée, plutôt que de devoir redéfinir le comportement de salutation du conducteur (ou autre comportement plus complexe) pour chaque classe dérivée l'implémentant. On se rend compte de ce problème dans l'implémentation actuel, puisque `PassagerStandard` et `PasssagerStresse` partagent le même comportement à la montée mais pas `PassagerIndecis`, il faut donc dupliquer le code entre `PassagerStandard` et `PassagerStresse` ou autoriser la redéfinition du comportement par défaut dans les sous-classe.

Par ailleurs, il m'a semblé assez laborieux et redondant de développer le même code sur deux branches en parallèle. Je n'ai pas saisi l'intérêt de ce double développement, à moins d'une utilisation pour un futur TD ?

Tom (Tandem 2) : Je crois avoir eu plus de mal sur ce TD que sur tous les autres. Au final, je pense avoir compris le fonctionnement de chaque méthode de factorisation mais je n'arrive pas à m'en faire une carte mentale. Autrement dit, je saurais sûrement justifier l'usage d'une méthode *a posteriori* mais je ne me sentirais pas capable en vue d'un projet d'en choisir une. J'espère avoir le temps de m'y replonger avant le prochain TD pour ne pas accumuler le retard. Aussi, je souhaiterais souligner que les diagrammes de classes UML n'ont pas vraiment été étudiés en cours et cela me manque vraiment lors de ces TD. Enfin, j'ai eu beaucoup de mal à suivre l'explication faite en début de TD, passée la première demi-heure. Quelques concepts où je n'ai pas réussi à suivre la cadence des explications m'ont fait décrocher et la seconde partie de l'explication m'a donc beaucoup moins apporté.