Programmation Orientée Objet
TD 2 : Les classes `Autobus` et `PassagerStandard`
Compte-rendu de l'équipe *Lorem Ipsum* (G4E1)

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

18 septembre 2020

Résumé

Dans le cadre de cette deuxième séance des modules PG202–PG203 à l'ENSEIRB-MATMECA, nous nous familiarisons avec les concepts de classes abstraites et d'objets factices, ainsi qu'avec la lecture de diagrammes de classes et de séquence UML.

1 Du paradigme impératif au paradigme orienté objet

Dans le code C initial, nous avons constaté que les fonctions associées à la structure `autobus` se chargeaient de la modification de l'état des passagers standards `ps_standard`, en modifiant directement leurs champs. La structure `autobus` n'était donc pas *indépendante* de l'implémentation de `ps_standard` : tout changement opéré sur `ps_standard` forçait à modifier une partie non négligeable du code de `autobus`, ce qui rendait le code peu évolutif et difficilement maintenable.

1.1 Indépendance de classes

Nous définissons *l'indépendance* par la possibilité de modifier ou remplacer la réalisation d'un code sans impacter le code qui l'utilise. Cela consiste à fournir une couche d'abstraction et à masquer les détails propres à l'implémentation. Bien qu'il eût été possible de modifier le code C de façon à garantir une telle indépendance[1], nous souhaitons ici tirer partie des mécanismes objet proposés nativement par un langage tel que Java.

L'encapsulation permet de fournir une certaine indépendance entre les classes. Toutefois, une classe `A` qui utilise une classe concrète `B` reste directement dépendante de cette dernière : si la classe `B` est

1. De telles méthodes ont été étudiées dans le module PG116 « Atelier Algorithme et Programmation », ainsi que lors les projets de semestre 5 et 6 (PR103 et PR106). Il s'agissait d'imiter les mécaniques objet en C par l'utilisation de pointeurs de fonctions, de pointeurs polymorphes et de fichiers *header* « publics » et « privés ».

supprimée, renommée, ou qu'une de ses méthodes publiques est elle-même renommée, supprimée ou, dans une certaine mesure, modifiée, la classe A ne fonctionnera plus correctement. De même, A est dépendante de B car elle ne peut utiliser qu'une seule classe concrète B à la fois[2], laquelle est définie à la compilation ou à l'exécution selon les langages.

Dans le cadre de ce projet, la classe Autobus dépend de PassagerStandard, car elle possède une liste d'instances PassagerStandard, comme représenté en Figure 1. Par ailleurs, certaines méthodes de PassagerStandard utilisent des paramètres de type Autobus. Par conséquent, les classes Autobus et PassagerStandard dépendent l'une de l'autre. Même si de telles dépendances sont possibles en Java, elles ne sont pas souhaitables car rendent le code difficile à maintenir.

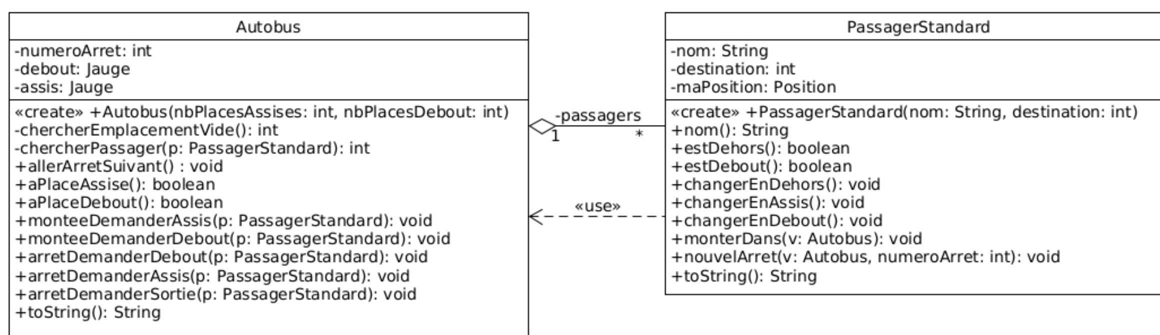

FIG. 1 : Dépendance circulaire entre Autobus et PassagerStandard

Ainsi, nous souhaiterions que la classe Autobus puisse utiliser « tout type de passager » qui respecte une interface de programmation (API) donnée. Dans le paradigme orienté objet, cette généricité est décrite par les notions d'héritage et de méthodes abstraites. D'une part, une classe D peut hériter (ou dériver) d'une classe C afin de modifier des opérations existantes ou d'en ajouter de nouvelles. Une référence vers la classe C permet d'utiliser indistinctement C et D et d'y appeler les méthodes publiques déclarées dans C. D'autre part, une classe peut déclarer des méthodes abstraites, c'est-à-dire des méthodes qui n'ont pas de définition. Une classe qui possède au moins une méthode abstraite (appelée « classe abstraite ») ne peut pas être instanciée, mais peut être dérivée.

De ce fait, une classe abstraite *pure* Passager, c'est-à-dire une classe sans méthode concrète ni attribut, permet de déclarer une interface de programmation (API). Cette API peut être implémentée par différentes sous-classes, ici PassagerStandard et FauxPassager. Une référence de la classe abstraite pure Passager peut alors contenir tout objet PassagerStandard ou FauxPassager, sur laquelle on peut appeler les méthodes publiques de Passager. Toutefois, ce type de polymorphisme induit un surcoût à l'exécution : en effet, lors d'un appel de méthode, il faut identifier la classe de l'ob-

2. En considérant que B est une classe concrète *non dérivable*.

jet référencé par la variable et appeler le code correspondant à cette méthode (le code de la méthode dans `PassagerStandard`, ou bien le code de la méthode dans `FauxPassager`).

Selon le même raisonnement, une classe abstraite pure `Vehicule` est définie, dont héritent `Autobus` et `FauxVehicule`. La situation est modélisée en Figure 2.

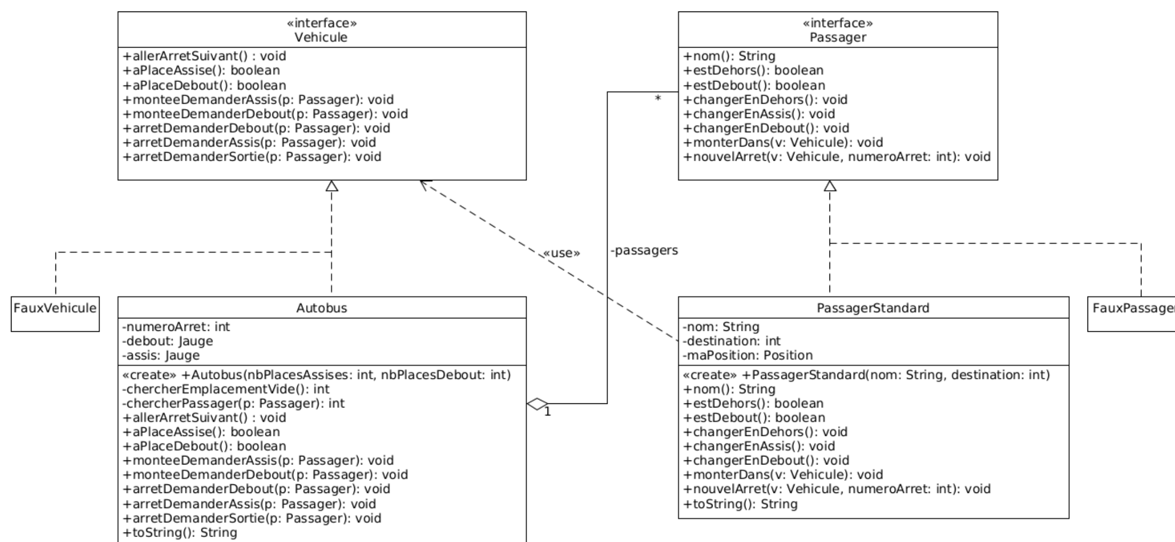

FIG. 2 : Utilisation de classes abstraites pures afin de briser le cycle de dépendance

Ainsi, `Autobus` ne dépend plus de `PassagerStandard`, mais d'une interface `Passager` générique. De même, `PassagerStandard` dépend désormais d'une interface `Vehicule`. Non seulement le cycle de dépendance est supprimé, mais le code est désormais extensible. Un client pourrait éventuellement vouloir ajouter des véhicules `Train`, possédant des caractéristiques similaires à l'`Autobus`, avec quelques particularités, ainsi que des `Voiture`s. On ajouterait sans doute une classe abstraite `TransportCollectif` implémentant `Vehicule` et dont hériteraient `Autobus` et `Train`, tandis que `Voiture` implémenterait directement `Vehicule` et interdirait aux passagers de se lever.

1.2 Développement des classes `PassagerStandard` et `Autobus`

Notre principale tâche a donc consisté à retranscrire le code C impératif en classes Java `Autobus` et `PassagerStandard`, les classes abstraites pures `Vehicule` et `Passager` étant fournies, ainsi qu'à corriger les bugs de `Jauge` et `Position` mis en évidence lors du TD1. Le tandem 1 (Aymeric et Sébastien) s'est chargé des classes `Jauge` et `Autobus`, tandis que le tandem 2 (Tom et Aurélien) a réalisé `Position` et `PassagerStandard`.

De ce fait, les tandems et le coordinateur se sont concertés afin de définir quelques pratiques de pro-

grammation et garantir une certaine homogénéité du code. Nous avons notamment décidé d'utiliser des annotations `@Override` permettant de signaler qu'une méthode d'instance définit une méthode abstraite ou surcharge une méthode concrète d'une classe mère. En plus d'ajouter une indication visuelle dans l'éditeur du programmeur, cette annotation améliorerait le *bytecode* Java produit lors de la compilation.

En revanche, le sujet n'a pas réellement suscité d'interrogation ou de confrontation de points de vue : il s'agissait essentiellement de retranscrire le code C fourni en Java en gardant la même logique algorithmique. Dans un premier temps, le tandem 1 a hésité à utiliser une structure de données `List` proposée par la bibliothèque standard Java pour représenter l'attribut `passagers` d'`Autobus`, avant de s'apercevoir que le sujet imposait l'utilisation de tableaux.

1.3 Tests et objets factices

Enfin, le recours aux classes abstraites pures détaillé en sous-section 1.1 simplifie également l'écriture de tests unitaires. L'objectif d'un test unitaire est de vérifier la correction du code dans la situation testée, indépendemment des autres classes et méthodes. Ainsi, si une classe dépend d'autres classes par une relation de composition, d'agrégation, passage en paramètre… il devient difficile de s'assurer que les tests restent bien « unitaires » et qu'un bug dans une classe tierce ne cause pas un échec des tests unitaires de la classe testée. De même, si la méthode d'instance à tester appelle une méthode d'instance d'une autre classe, il peut être difficile de vérifier si cet appel à la méthode d'instance tierce a été réalisé correctement : en effet, comment vérifier l'état interne d'un objet tiers tout en respectant le principe d'encapsulation ?

Pour ce faire, il est possible de définir des objets dits *factices* : des objets pouvant remplacer les instances d'autres classes à des fins de tests, et qui exposent leur état interne, notamment via des attributs publics. Grâce à l'implémentation générique détaillée précédemment, il est possible de créer de tels objets factices `FauxPassager` et `FauxVehicule` par dérivation de `Passager` et de `Vehicule`. Au lieu d'écrire de « véritables » méthodes pour les objets tiers, ces méthodes enregistrent simplement qu'elles ont été appelées dans une liste `logs` publique. Ainsi, les méthodes de test peuvent vérifier que la classe testée a bien appelé les méthodes de l'objet factice en consultant ces logs.

2 Égalité entre objets

Finalement, le passage au paradigme orienté objet offre un moyen de comparaison efficace entre deux instances de classe quelconque, contrairement à C. Ainsi, en Java, la méthode d'instance `equals()` est implémentée par défaut pour toute sous-classe d'`Object`[3], afin de vérifier l'égalité entre deux

3. Par défaut, toute classe Java hérite d'`Object`.

objets quelconques. En C, il n'existe pas de convention pour nommer une fonction de comparaison, notamment dû au fait qu'une fonction ou un opérateur comme == ne peuvent être surchargés.

En langage Java, l'opérateur == ne permet que de comparer les valeurs des variables, et non de comparer l'état d'un objet référencé par une variable. À cette fin, une classe peut définir une méthode d'instance `equals()` qui renvoie `vrai` si les instances sont considérées « égales » : généralement, deux instances sont égales lorsqu'elles appartiennent à une même classe et que les valeurs de leurs attributs sont identiques[4].

2.1 Implémentation de `equals()` dans la classe `String`

Nous étudions ici plus spécifiquement l'implémentation de la méthode `equals()` pour la classe `String` de la bibliothèque standard, dont le code est reproduit et annoté en Code 1.

Dans un premier temps, les deux variables sont comparées de façon à vérifier en temps constant si leurs valeurs sont identiques. On retourne `vrai` directement si elles référencent le même objet (annotation `[A]`). Puis, on vérifie à l'exécution (en temps constant) si l'objet référencé par `anObject` est une instance de `String` ou une classe dérivée de celle-ci (annotation `[B]`), en retournant `faux` dans le cas contraire : les objets `String` ou dérivés ne peuvent pas être égaux à des instances d'autres classes. Enfin, on compare la longueur des chaînes en temps constant (annotation `[D]`), et en cas d'égalité, on compare linéairement les caractères jusqu'à trouver une différence et retourner `faux` (annotation `[E ]`). Sinon, les chaînes sont égales et `equals()` renvoie `vrai`. Cette méthode privilégie les opérations moins coûteuses d'abord, et ne réalise les vérifications les plus coûteuses que si nécessaire.

Néanmoins, il n'est pas possible d'appeler directement une méthode de `String` sur une variable de la classe générique `Object`, comme réalisé en annotations `[D]` et `[E]`. Ainsi, l'instruction de transtypage `String anotherString = (String)anObject` est nécessaire afin de créer une nouvelle variable `String` à partir d'une variable `Object`. Bien que les classes référencées par ces variables soit différentes, toutes deux référencent le même objet. Il s'agit dans ce cas de *downcasting* : on transforme une référence vers une classe générale en une référence vers une classe plus spécifique. Contrairement à une opération d'*upcasting* (passage d'une référence de classe dérivée à une référence de classe mère), ce *downcasting* peut échouer. En effet, si la classe instanciée est différente de la classe cible du transtypage, une exception est levée à l'exécution[5].

4. L'implémentation par défaut de `equals()` héritée d'`Object` compare les deux objets par référence (avec ==). En redéfinissant `equals()` pour une instance donnée, il est nécessaire de redéfinir `hashCode()` afin que les *hashes* de deux objets `a` et `b` soient égaux lorsqu'`a.equals(b)` est vrai. (Source : Oracle, *Object*. Consulté le 21 septembre 2020. https://docs.oracle.com/javase/10/docs/api/java/lang/Object.html#equals(java.lang.Object))

5. Oracle, *Conversions and Promotions*. Consulté le 20 septembre 2020. https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.5.3#usage

Code 1 : Implémentation d'OpenJDK de la méthode equals() dans la classe String

```
public boolean equals(Object anObject) {
    // [A]
    if (this == anObject) {
        return true;
    }

    // [B]
    if (anObject instanceof String) {
        // [C]
        String anotherString = (String)anObject;
        int n = count;
        // [D]
        if (n == anotherString.count) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = offset;
            int j = anotherString.offset;

            // [E]
            while (n-- != 0) {
                if (v1[i++] != v2[j++])
                    return false;

            }
            return true;
        }
    }

    return false;
}
```

Dans la méthode étudiée, le transtypage est protégé par le test `if (anObject instanceof String)`, qui permet de vérifier que la classe de l'objet à l'exécution avant *downcast* est bel et bien « compatible » avec la classe `String`.

2.2 Représentation mémoire des chaînes littérales

Enfin, nous avons constaté que l'opérateur `==` appliqué à deux chaînes littérales identiques (`"Hola"=="Hola"`) renvoyait `vrai`, et non `faux` comme avec des chaînes non littérales (`new String("Hola")==new String("Hola")`). En réalité, si une même chaîne littérale est affectée à différentes variables dans le code, le compilateur fait pointer toutes ces variables vers une unique

instance de `String`[6]. Cette optimisation permet de limiter le nombre d'instances de `String` et donc la consommation de mémoire, tout en permettant une comparaison des chaînes littérales moins coûteuse qu'avec `equals()`. En effet, une comparaison de références `"Hola"=="Señor Puel"` renvoie `faux` en temps constant, là où `"Hola".equals("Señor Puel")` renvoie le même résultat après comparaison caractère par caractère, en temps linéaire. Par ailleurs, ce partage d'instances ne pose pas de problème dans la mesure où les objets `String` sont immuables.

Plus précisément, les instances de chaînes littérales sont stockées dans une zone mémoire nommée `String Pool`, localisée dans le tas de la machine virtuelle Java, comme représenté en Figure 3. Si une chaîne de caractères littérale n'est pas encore présente dans la `String Pool`, une nouvelle instance y est allouée. Dans le cas contraire, l'adresse de l'instance `String` déjà allouée est retournée.

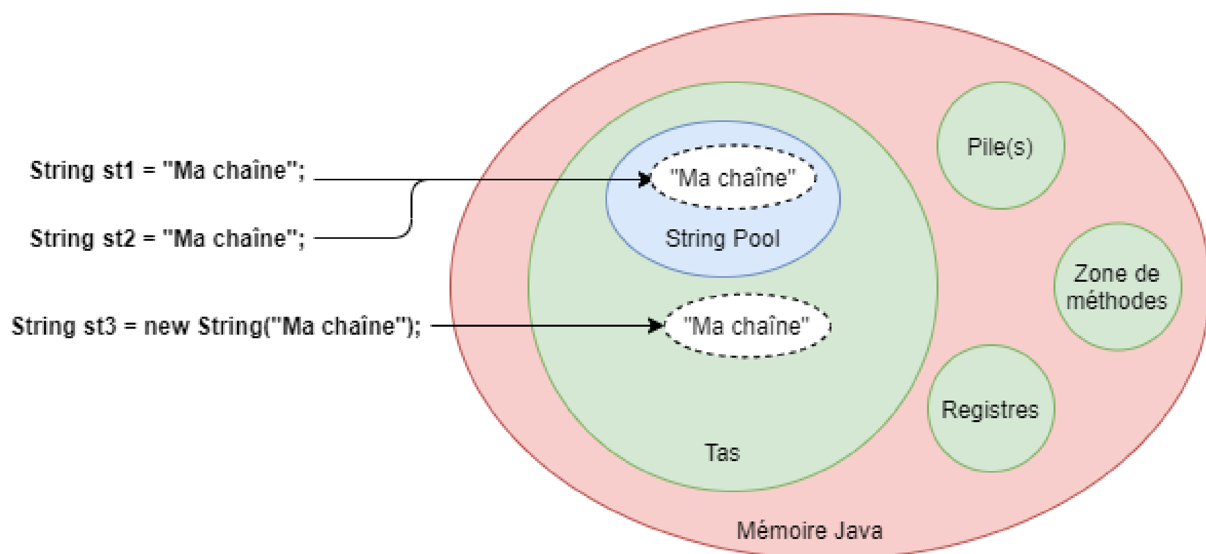

FIG. 3 : Zones mémoire impliquées dans l'instanciation de chaînes de caractères[7]

Conclusion

Finalement, ce deuxième TD nous a permis de mettre en pratique les concepts de classes abstraites pures et d'héritage abordés en cours, tout en découvrant le concept d'objets factices. Certains membres de l'équipe se sont également familiarisés avec la lecture de diagrammes UML.

6. On parle de patron de conception « Poids-Mouche », permettant de réduire le nombre d'objets à allouer si ceux-ci sont identiques.
7. Schéma inspiré de DZone, *How do I compare strings in java*. Consulté le 22 septembre 2020. https://dzone.com/articles/how-do-i-compare-strings-in-java

Commentaires

Rémi (Coordinateur) : Lors de cette séance, j'ai endossé le rôle de coordinateur. Pas de difficulté majeure à ce propos, le sujet pouvant être aisément scindé en deux équipes. Le diagramme UML a facilité le travail de répartition des tâches et de coordination des équipes, afin d'identifier les tâches et leurs dépendances.

Actuellement, le modèle de données est relativement simple, et les classes réalisées ont des comportements très différents. En effet, chaque interface n'est implémentée que par deux classes : un objet métier et un objet factice. À l'avenir, il pourrait pourtant être nécessaire d'enrichir le code existant en ajoutant de nouvelles méthodes à l'interface, et de nouveaux objets métiers, tels que d'autres véhicules. Comme expliqué précédemment, nous pourrions ajouter une classe `TGV` avec un comportement proche de celui d'`Autobus`, ainsi qu'une classe `Voiture` bien plus différente. Ainsi, afin de limiter la duplication de code entre `TGV` et `Autobus`, il serait pertinent de définir une classe abstraite `TransportCollectif`, classe fille de `Vehicule` et classe mère de `TGV` et d'`Autobus`. `Voiture` implémenterait quant à elle directement `Vehicule`.

Toutefois, cet héritage « en cascade » n'est pas nécessairement une solution adéquate. Que faire si l'on souhaite implémenter un transport `Covoiturage`, qui partage à la fois des comportements de `Voiture` (comme l'impossibilité de se lever), de `TransportCollectif` (notion d'arrêts) et de `TGV` (réservation obligatoire) ? Doit-on définir une classe abstraite `Transport` mère de `Voiture` et `TransportCollectif`, avec des méthodes par défaut ? Doit-on dupliquer du code ? Au lieu d'un simple héritage, nous souhaiterions pouvoir *composer* une classe avec différents comportements : peut-on se lever ? doit-on réserver ? comment fonctionnent les arrêts ? Il serait alors sans doute préférable de préférer la composition d'attributs à l'héritage (*composition over inheritance*, ou patron stratégie), en utilisant par exemple des traits.

Sébastien (Tandem 1) : J'ai travaillé en *pair programming* avec Aymeric. C'est la première fois que je trouve cette technique aussi utile, étant physiquement distant d'Aymeric, je ne pouvais absolument pas toucher à son clavier. En somme, j'ai pu me concentrer uniquement sur la réflexion sur le code et donner des consignes à Aymeric. Lorsque nous n'étions pas d'accord ou que l'un des points n'était pas clair, nous en discutions et au besoin nous allions vérifier dans le cours. Cela s'est donc avéré très efficace puisque nous n'avions presque plus de code à réaliser à la maison. De plus, nous avons pu approfondir certains concepts de POO comme les variables d'instance, les constructeurs et la différence entre les méthodes privées et publiques. Lorsque nous avions une question nous pouvions la poser à Rémi, notre coordinateur, qui s'est avéré de très bons conseils.

Aymeric (Tandem 1) : J'ai travaillé en *pair programming* avec Sébastien depuis mon domicile puisque je suis confiné. Nous avons pu interagir facilement avec un Discord grâce auquel j'ai partagé mon écran sur lequel je codais. Lorsque nous avions des questions par rapport à des subtilités du langage, nous

demandions à notre coordinateur, Rémi. J'ai pu continuer de progresser en Java en découvrant les variables d'instance, les constructeurs et les méthodes publiques et privées. D'autre part, j'ai également appris la notion de type et de sous-type.

Aurélien (Tandem 2) : J'aurais souhaité pouvoir utiliser l'annotation `@NotNull` dans son analyse statique tel que proposé par l'IDE IntelliJ IDEA (`org.jetbrains.annotations.NotNull`). Chose que je n'ai pas retrouvé avec le compilateur intégré. Néanmoins, j'ai apprécié l'aspect « conception » abordé dans la mise en place de l'indépendance entre les classes. Cependant, j'aurais aimé pouvoir implémenter les objets factices lors de la phase de test pour éviter une simple retranscription de fichiers C en langage Java. Enfin, la facilité d'utilisation des classes abstraites m'a permis de me rendre compte de l'ensemble des opérations cachées par le langage Java, par rapport à l'implémentation d'un tel comportement en C.

Tom (Tandem 2) : Ce TD a été très satisfaisant dans la mesure où nous nous sommes servis des spécificités de Java en tant que langage orienté objet pour résoudre des problèmes rencontrés dans le module PG116-Atelier Programmation (langage C). Par exemple, l'usage de classes abstraites pures nous a permis de résoudre les problèmes de dépendance entre types de manière naturelle et rapide. Ainsi, je trouve bien plus fastidieux en C de garantir l'indépendance de l'implémentation d'un type abstrait de données avec le code qui l'utilise. En outre, l'usage de classes factices a permis de mettre en place des tests unitaires vérifiant effectivement des méthodes de façon isolée et sans dépendance.