Programmation Orientée Objet
TD 1 : Encapsulation – Compte-rendu de l'équipe *Lorem Ipsum* (G4E1)

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

11 septembre 2020

Résumé

Dans le cadre de cette première séance des modules PG202–PG203 à l'ENSEIRB-MATMECA, nous appréhendons les bases de l'écosystème Java et de la programmation orientée objet, à travers l'écriture de tests de classes fournies.

1 Découverte de l'écosystème Java

Nous nous sommes dans un premier temps familiarisés avec les bases de l'écosystème Java. Nous considérons ici que Java est un *écosystème*, non pas simplement un *langage* ou une *norme* comme C. En effet, Java fait référence non seulement à un langage de programmation orienté objet à typage statique, mais aussi à une série d'outils associés : un compilateur (`javac`), un environnement d'exécution (`java`), un outil de documentation (`javadoc`), etc. Différentes plateformes Java sont également proposées, telles que Java SE, Java ME, Java EE, et une implémentation libre OpenJDK.

1.1 Compilation et exécution

Java se distingue de C du fait que le code est plus ou moins indépendant de la plateforme cible. Ainsi, un code compilé n'est pas dépendant d'une architecture et d'un système d'exploitation. Pour ce faire, le compilateur Java `javac` ne génère pas d'instructions machine à partir des fichiers de classes en `.java`, mais du code intermédiaire appelé *bytecode Java* et dont l'extension est `.class`. Ce code ne peut par conséquent pas être exécuté directement : il est exécuté par une machine virtuelle, la JVM, qui permet d'abstraire ces détails d'architecture.

1.2 Bibliothèque standard Java

Java définit également une bibliothèque standard très riche, fournissant des types réutilisables dans tout programme, tels que des exceptions, classes, etc.

Dans le cadre de cette séance, nous nous sommes principalement intéressés à certaines implémentations de chaînes de caractères, `String` et `StringBuffer`, ainsi qu'à `PrintStream` permettant « d'afficher » aisément des données dans un flux de sortie.

1.2.1 Chaînes de caractères muables et immuables

Nous avons dans un premier temps étudié deux types encodant les chaînes de caractères, `String` et `StringBuffer`.

La classe `String` décrit des chaînes immuables, notamment les chaînes littérales (`"Hello world !"`) et peuvent être partagées entre plusieurs instances afin de limiter la consommation de mémoire. En revanche, la concaténation de deux `String` par l'opérateur + nécessite de créer une nouvelle instance et d'y copier les caractères des deux chaînes.

Au contraire, la classe `StringBuffer` fournit des chaînes muables allouées dans le tas, stockées sous forme de vecteurs : il est ainsi possible d'ajouter, d'insérer ou de supprimer des caractères sans avoir à créer de nouvelle instance.

Toutefois, le programmeur est libre de transformer une instance de `String` et `StringBuffer` et réciproquement : pour transformer une chaîne muable en chaîne immuable, on utilise le constructeur `String(StringBuffer buffer)` ou la méthode `toString()` de `StringBuffer`. Le constructeur `StringBuffer(String str)` permet au contraire de créer une chaîne muable à partir d'une chaîne immuable.

Enfin, nous retenons que même si une variable déclarée avec le mot clé **`final`** est immuable et que sa valeur ne peut donc pas être modifiée après son initialisation, cela n'implique pas que l'objet pointé par cette variable est lui-même immuable. Le Code 1 illustre une telle situation.

Code 1 : Exemple d'utilisation d'une variable immuable

```
final StringBuffer s = new StringBuffer("Lorem ipsum");
s.append(" dolor sit amet");          // OK   : modifie l'objet
s = new StringBuffer("Bonne nuit");  // NOK  : modifierait la référence
```

1.2.2 Affichage de données dans un flux avec `PrintStream`

Nous souhaitons désormais utiliser la classe `PrintStream` afin d'afficher un objet `String` dans la sortie standard, dont le code d'exemple est rappelé en Code 2.

La classe `PrintStream` définit plusieurs méthodes `println`, qui se différencient par le type de paramètres requis : il s'agit de polymorphisme *ad hoc* ou *surcharge*. La variable `out` du paquetage `System`

Code 2 : Affichage d'un message sur la sortie standard

```
String chaîne = "\t\t Youpi";
final String EXCLAMATION =  " !!!!!!! ";

System.out.println(chaîne + EXCLAMATION + "Cela marche héhé" + EXCLAMATION
    );
```

fait ainsi référence à une instance de `PrintStream` qui écrit sur la sortie standard.

Dans notre exemple, l'argument passé à la méthode `println` est une concaténation de chaînes de caractères (`String`), c'est donc la méthode prototypée `println(String x)` qui est utilisée.

Une fois les bases de l'écosystème Java assimilées, nous avons pu écrire un premier code : des tests unitaires.

2 Un premier programme Java : tests unitaires

Comme pour toute implémentation, il est nécessaire de tester son code afin de détecter des éventuels bugs et s'assurer qu'il réponde aux exigences du client. L'utilisation de tests unitaires permet de répondre à ce besoin.

2.1 Assertions

En langage Java, comme dans de nombreux langages, il est possible de mettre en place ces tests par le biais d'assertions. Celles-ci ont un rôle de vérification, et sont utilisées dans la phase de debug pour comprendre l'origine des erreurs.

Généralement, on cherche à éviter les effets de bord dans des assertions. En effet, de tels effets de bord rendent le code moins lisible et plus difficile à maintenir sur le long terme. Cette pratique est ainsi découragée par la documentation officielle Java, qui stipule que « les expressions contenues dans des assertions doivent être exemptes d'effet de bord : l'évaluation d'une expression ne doit pas modifier un état visible après l'évaluation de cette expression »[1].

Par conséquent, le Code 3 ne suit pas des pratiques de programmation « saines », étant donné que l'assertion modifie par effet de bord la valeur de la variable `estMisAssertion` : l'affectation `estMisAssertion = true` n'est effectuée que si les assertions sont activées. Selon l'usage de

1. Oracle, *Programming With Assertions*. Consulté le 13 septembre 2020. https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html#usage

cette variable, le comportement du programme peut radicalement changer entre une version de test et une version *release*, rendant lesdits tests peu pertinents.

Code 3 : Détection de l'activation des assertions à l'exécution

```
boolean estMisAssertion = false;
assert estMisAssertion = true;

if (!estMisAssertion) {
  System.out.println("Execution impossible sans l'option -ea");
  return;
}
```

2.2 Indépendance entre tests

Pour chaque classe fournie, nous avons défini une classe de test correspondante, laquelle définit une méthode d'instance par cas de test. Afin de garantir davantage l'indépendance entre l'exécution de deux tests, chaque méthode de test est appelée sur une instance propre (voir Code 4). Nous nous assurons ainsi qu'un test ayant échoué ne provient pas d'un effet de bord au sein de l'instance de classe de test, via une modification d'attribut. Toutefois, cela n'empêche pas les effets de bord causés par une variable propre à la classe de test (variable `static`).

Code 4 : Exemple d'exécution de tests pour la classe TestJauge

```
public static void main(String[] args) {
    // --- snippet ---
    new TestJauge().testDansIntervalle();
    new TestJauge().testSuperieurIntervalle();
    new TestJauge().testEgalMaxIntervalle();
    new TestJauge().testDepartNegatif();
    // --- snippet ---
    System.out.println("OK");
}
```

2.3 Test des classes `Jauge` et `Position`

Les différents cas de tests élaborés nous ont permis de mettre en évidence la présence de bugs dans l'implémentation des classes `Jauge` et `Position` par rapport aux spécifications fournies dans la documentation.

2.3.1 Fichier `TestPosition.java`

La réalisation des tests de la classe `Position` a démontré que les méthodes `estDebout()` et `dehors()` étaient incorrectes.

Tout d'abord, la méthode `toString()` associée à la classe `Position` révèle que la méthode `dehors()` crée une nouvelle instance dans l'état `DEBOUT` et non `DEHORS`. De plus, la méthode `estDebout()` renvoie `vrai` si le passager est assis, `faux` sinon. Par conséquent, il est difficile de vérifier l'état d'une instance, ce qui peut faire échouer des tests de méthodes correctement implémentées, ou faire réussir des méthodes fausses (cf. Code 5).

Code 5 : Débogage de code

```
Position pos = new Position();
pos = pos.debout();
pos = pos.dehors(); // Cette méthode est erronnée : elle produit le même
    comportement que debout()

// L'assertion suivante ne produit pas d'erreur puisque la vérification op
    érée est également fausse.
assert !pos.estDebout() : "Ne devrait pas être debout aprés avoir été mis
    dehors !";
```

2.3.2 Fichier `TestJauge.java`

L'implémentation de `Jauge` fournie s'est également avérée incorrecte.

D'une part, la jauge est censée être verte si le nombre de passagers est compris entre 0 et le nombre maximum de passagers, et rouge si le nombre maximal de passagers est atteint ou dépassé. En pratique, la méthode `estVerte()` renvoie « vrai » même si le nombre de passagers dépasse la limite fixée.

D'autre part, la méthode `incrementer()` soustrait le nombre maximal de passagers au nombre de passagers. Cette méthode est donc entièrement fausse.

Conclusion

Finalement, ce premier TD nous a permis de mettre en pratique certains concepts de base du paradigme orienté objet abordés en cours, tout en découvrant (ou redécouvrant) l'écosystème Java. Nos niveaux de connaissance du langage Java et de l'orienté objet étant disparates au sein de l'équipe, les

étudiants les plus aguerris ont principalement laissé leurs collègues manipuler et les ont assistés tout au long de la séance.

Commentaires

Aurélien (Coordinateur) : Ayant déjà manipulé le langage Java lors de ma scolarité, l'intérêt de ce TD a principalement résidé dans ma capacité à comprendre et à répondre aux problèmes rencontrés par l'équipe, plutôt que de me familiariser avec les attentes du sujet sur des notions que je maîtrisais déjà.

Rémi (Tandem 1) : La réalisation des tests unitaires m'a semblé un peu laborieuse : sans framework de tests unitaires comme JUnit, une certaine quantité de code est dupliquée pour chaque cas de test. Le code en amont est tellement erronné qu'il est parfois difficile de savoir si les tests eux-mêmes sont corrects, d'autant plus que l'exécution s'arrête à la première assertion fausse et que nous pouvons pas *en principe* corriger le code source.

J'aurais souhaité améliorer l'exécution des tests en permettant d'afficher ceux qui ont échoué sans s'arrêter au premier échec, le tout sans dupliquer davantage de code (en évitant de recopier des blocs `try-catch` pour chaque méthode de test). Cela aurait été assez peu pratique dans ce cas : il aurait pour cela fallu utiliser des *callbacks*, soit en utilisant une interface et une implémentation par méthode de test, soit en définissant les tests dans des lambda expressions. Ces deux implémentations auraient probablement rendu le code moins lisible qu'à l'origine et puisqu'il ne s'agissait pas de l'objectif du TD, nous avons préféré garder notre gestion des tests basique !

Sébastien (Tandem 1) : C'était pour moi mon premier contact avec le Java, ce TD m'a permis de découvrir comment compiler, créer une classe… De plus ce premier TD nous a permis de mettre l'équipe en place. Finalement le codage en tandem fut très productif puisque nous avions associé un membre connaissant le Java et un membre ne le connaissant pas.

Aymeric (Tandem 2) : Étant un cas contact au COVID-19, j'ai tenté de suivre le TD depuis chez moi grâce à un outil audio déployé par notre coordinateur. J'ai pu participer au code et au rapport grâce à des outils de composition à distance.

Tom (Tandem 2) : Durant ce TD j'ai programmé pour la première fois en Java. J'y ai appris à compiler et à mettre en place à travers une classe le principe d'encapsulation étudié en cours. Grâce à l'aide de notre coordinateur (Aurélien Moinel) et d'un logiciel de conférence audio, j'ai pu coopérer sans trop de difficulté avec mon tandem à distance.