- comment sont fait les tests
- quelles sont les erreurs détecté / quel correctifs
- problème de portée

- proto des méthode de test ??
  TestePassager
- Manque de Messages / TIN.java
  testToString() ??
- Harmoniser la façon de coder ! des fonctions.

Compte Rendu du TD numéro 2 de l'équipe Eirb'reteau

lusufisant +

Pour le 29 septembre

Coordinateur : Aurélien Nizet
Tandem 1 : Reda Boudjeltia, Lionel Adotevi
Tandem 2 : Victor Dury, Pierre Gaulon
TD du 26 septembre 2014

## 1 Introduction

Ce TD était le premier TD ou nous avons commencé à coder en langage java. Il a eu pour but de nous familiariser avec les tests unitaires en java, les paquetages tec ainsi que les technique d'encapsulation. Nous avons alors suivi ce TD en tandem ou par toute l'équipe lorsque c'était conseillé.

## 2 Les tests unitaires

Dans cette partie il s'agissait de coder des tests unitaires pour tester le bon fonctionnement des deux classes : **JaugeNaturel** et **EtatPassager**.
Pour exécuter ces tests unitaires et activer les assertions, il faut utiliser l'option **-ea**, qui signifie *enable assertions*. Si cette option est oubliée lors de l'exécution, dans le code de la classe **LancerTests**, on rentre dans une condition qui ne passe pas par les assertions et termine le programme en affichant un message d'erreur.

### 2.1 Écriture du premier test

En premier lieu, on souhaite tester l'état des objets juste après instanciation dans les deux classes **JaugeNaturel** et **EtatPassager**. Pour ce faire, il nous faut tout d'abord déclarer la variable qui contiendra les instances à tester puis instancier la classe. Ceci ce fait en une ligne de code, pour la classe **JaugeNaturel** on a donc codé :

```
JaugeNaturel aJauge = new JaugeNaturel(67899,100);
```

La partie de gauche de l'affectation concerne la déclaration d'une variable pour contenir une instance de le classe **JaugeNaturel**, et la partie de droite, celle avec *new* concerne l'instanciation en elle-même.
Afin de réaliser le test, il faut pouvoir envoyer un message aux instances à tester. Pour une méthode *methode*() et une instance *instance* cet envoi se fait de la



dangereux : instruction à effet de bord (modification de variable) non garantit. comportement fonctionnel non garantit

pourquoi ?

forme *instance.methode*().
Les fichiers *.class* à tester contiennent du code intermédiaire récupérable en utilisant la commande *javap* sur un *.class* avec l'option -c. Ces fichiers contiennent également des informations sur les méthodes et attributs des classes concernées, et pour pouvoir récupérer les informations sur les attributs déclarés *private*, il faut utiliser l'option -p (-private) de la commande *javap*. De plus, il est possible d'utiliser la commande *javap* sur plusieurs fichiers de type .class, afin d'analyser leurs méthodes en une seule fois.
Pour que le test s'effectue lors de l'exécution du programme, il nous a fallu appeler la méthode *testDansIntervalle*() (et *testExterieur*()) dans la classe *LancerTests*, on a donc écrit dans le premier cas : *aTestJauge.testDansIntervalle*();, où *aTestJauge* est une instance de la classe **TestJaugeNaturel**.

Après le premier test réalisé dans chacune des deux classes à tester, il nous a fallu coder les tests dans des méthodes différentes pour tous les autres cas de figure. Ces méthodes sont toutes appelées par une méthode nommée *lancer*() qui décompte tous les tests effectués et respecte un certain affichage.

### 2.2 Boutez vos neurones

#### 2.2.1 Classe LancerTests

Le code au début de la méthode *main*() détecte l'option -ea en affectant un booléen *estMisAssertion* à *vrai* dans une assertion. Ainsi, si les assertions ne sont pas activées, le booléen gardera sa valeur de départ, c'est à dire *faux*, et nous ne rentrerons pas dans la partie du code effectuant les tests. C'est dangereux de faire une affectation dans un test, car une affectation est toujours vraie.

#### 2.2.2 Classe EtatPassager

En observant la documentation et le .java de **EtatPassager**, on peut voir que les attributs de la classe **EtatPassager** sont définis constants (écrits en majuscule et précédés de *final*). Par conséquent, une instance de la classe **EtatPassager** est constante. Cependant, de ce fait, chaque méthode de la classe en question devra renvoyer une nouvelle instance lors d'un changement d'état tel qu'un assis/debout.
D'autre part, l'avantage de manipuler des objets constants permet de bien maitriser la mémoire utilisée à tout instant.

pas de modificateur d'état.

conséquence ?

## 3 Un paquetage tec

### 3.1 Compilateur et dépendance de compilation

La compilation de la classe LancerTests est suffisante pour créer les classes LancerTests, TestJaugeNaturel, JaugeNaturel, TestEtatPassager et EtatPassager.

### 3.2 Arborescence du projet

Nous décidons de créer deux répertoires. Un répertoire **src** où nous mettons nos fichiers sources, et un répertoire **build** où nous mettons nos fichiers compilés. Le seul problème est qu'après chaque compilation, nos fichiers compilés se retrouvent dans le dossier *src*, et nous devons déplacer manuellement les classes dans le répertoire *build*.

### 3.3 Inclure les classes dans le paquetage

Afin d'implémenter un paquetage, il faut inclure au début de chaque fichier source *package tec*; pour dire à chaque classe qu'elle appartient à ce paquetage. Ensuite, toutes les classes sauf **JaugeNaturel** ont une portée limitée au paquetage, car aucun mot-clé n'est utilisé avant *class*. Par contre, la classe **JaugeNaturel** est déclarée publique (*public class*), et est donc accessible en dehors du paquetage.

### 3.4 Compilation

La compilation avec $-d$ demande au moins deux arguments. Le premier va déterminer le dossier dans lequel les fichiers compilés vont être placés, puis les autres déterminent les fichiers à compiler. La commande : *javac −d build src/* .java* compile toutes les classes écrites dans le répertoire *src*, et place les classes compilées dans un sous-répertoire *tec* du répertoire *build*. Le sous-répertoire *tec* a été créé automatiquement par la commande, en prenant le nom du paquet inscrit au début de chaque fichier source.

### 3.5 Exécution des tests

Maintenant, pour exécuter une classe à partir d'un répertoire parent, il faut utiliser l'option $-cp$ comme *classpath* (chemin de classe en français). Ainsi, la commande pour lancer les tests est *java − ea − cp build/ tec.LancerTests*. On peut voir que l'option $-cp$ demande en argument le chemin du paquet dans lequel figure la classe à exécuter.

### 3.6 Boutez vos neurones

Lorsque nous essayons d'exécuter la classe LancerTests, en mettant comme nom de fichier *tec.LancerTests.class*, un message d'erreur apparait: "Impossible de trouver ou charger la classe principale tec.LancerTests.class". La machine virtuelle cherche la classe "class" dans le paquet *tec.lancerTests*.

## 4 L'encapsulation jauge naturel

Le travail concernant cette partie a été réalisé dans un répertoire différent, afin de garder en place la première réalisation, pour l'utiliser dans la suite du projet.



### 4.1 Méthode d'accès à la valeur d'un attribut

Nous avons ajouté la méthode *public int valeur*() dans la classe **JaugeNaturel**, qui permet d'obtenir la valeur de l'attribut *valeur* d'une instance de cette classe, d'où son nom de *getter*. Ainsi pour tester les méthodes *incrementer*() (ou *decrementer*()) de la classe **JaugeNaturel**, il suffit alors de sauvegarder le retour du *getter valeur*() dans une variable juste après une instanciation de la classe **JaugeNaturel**, d'utiliser la méthode *incrementer*() (ou *decrementer*()) sur cette instance, puis de comparer le nouveau retour de *valeur*() avec l'ancien.

Cela prendra par exemple cette forme dans notre réalisation :

```
JaugeNaturel aJauge = new JaugeNaturel(10, 1);
long old = aJauge.valeur();
aJauge.incrementer();
assert aJauge.valeur() == (old + 1);
```

### 4.2 Changement de la réalisation de JaugeNaturel

Maintenant, nous décidons de changer le type de l'attribut *valeur*. Ça sera une instance de la classe **java.math.BigDecimal**. Le lien *a-un* signifie qu'une instance de la classe **JaugeNaturel** stocke une instance de la classe **java.math.BigDecimal**. Ainsi au niveau du code, il nous faut changer le type de l'attribut *valeur* lors de sa déclaration, et procéder à l'instanciation de cet attribut dans le constructeur *JaugeNaturel*(). C'est la traduction du lien *a-un*.

```
private java.math.BigDecimal valeur;
private final java.math.BigDecimal max;
public JaugeNaturel(long vigieMax, long depart) {
  valeur = new java.math.BigDecimal(depart);
  max = new java.math.BigDecimal(vigieMax);
}
```

### 4.3 Influence du changement de la réalisation sur le code client

Le type de l'attribut *valeur* ayant changé, il faut alors changer le type de retour de la méthode *valeur*(), qui est logiquement le même, puisque cette méthode retourne *valeur*. Le changement de retour de cette méthode provoque alors le changement du code de la méthode *testDepassementValeur*(), puisqu'on y stockait la valeur dans une variable, donc du même type que *valeur*. C'est pourquoi il a fallu changer le type de cette variable. De plus, la comparaison de *BigDecimal* est complètement différente de la comparaison classique d'entiers (longs), avec les opérateurs <,>,==,<=,>=. Il a donc fallu réécrire les comparaison en fonction de ce nouveau type, et donc en utilisant les méthodes de la classe **java.math.BigDecimal** prévues à cet effet.

Les tests qui n'utilisaient pas le *getter valeur*, n'ont eu à subir aucune réécriture, il sont restés indépendants de la réalisation de ce code, donc plus facilement maintenables.

# 5 Commentaires

## 5.1 Commentaire de Pierre

Pour commencer, je pense que le travail qu'on a réalisé est perfectible. Il peut être amélioré dans le sens où, pour effectuer tous les tests, on a souvent copié/collé la même portion de code, en se contentant de changer les valeurs des booléens dans les *assert*, et les valeurs des entiers dans les constructeurs. On aurait pu faire une méthode avec comme paramètre les valeurs des entiers et les valeurs des booléens attendus pour ces entiers. Cela nous aurait permis de factoriser le code et de gagner en temps de debug. Enfin je pense que la partie sur l'encapsulation est le point abordé le plus important de ce TD, car il est souvent négligé, et pourtant une mauvaise encapsulation peut entraîner un code très difficilement maintenable, et donc une perte de temps énorme. Déjà que changer la réalisation d'un test à cause d'un changement de type n'était pas drôle, alors c'est difficilement imaginable sur un code plus conséquent.

## 5.2 Commentaire de Lionel

Tout comme dans le précédent TD, j'ai pu éclaircir à l'aide de ce TD, les points que j'avais laissé obscur comme les options de compilation avec les dossiers src et build, ainsi que les paquetages. Ce sont des choses que j'ai utilisé, et utilise encore avec NetBeans mais étant donné le peu de temps que j'avais pour réaliser mon projet de stage, je n'avais jamais pris la peine de bien comprendre ce qu'il y avait derrière. Ainsi donc, ce TD et le précédent ne font que confirmer mes acquis et m'apprendre plus encore sur le fonctionnement de Java.

## 5.3 Commentaire de Reda

A travers ce TD, j'ai appris le fonctionnement des test unitaires et des assertions en Java dans un premier temps, puis à mieux utiliser les arborescences dans le compilateur javac afin de différencier les dossiers distinct les *.class* des *.java*. Par la suite j'ai pris connaissance des méthodes de type getter et setter.

## 5.4 Commentaire de Victor

Ce TD a été important pour moi, car il m'a permis de commencer à coder en Java. Les premières classes et méthodes nous ont pris du temps à écrire, mais le fait de l'avoir commencé avant l'heure de TD nous a permis d'être à l'aise pour faire les exercices. Avoir revu la compilation et l'exécution, avoir découvert les paquetages et l'encapsulation, et avoir utilisé tout cela pour effectuer des tests a été intéressant, et je pense que je suis prêt pour le prochain TD.

## 5.5 Commentaire d'Aurélien

Tout d'abord ce TD m'a permis de me familiariser encore plus avec le vocabulaire employé pour coder en java. Il nous a fourni des indications qui serviront par la suite à écrire du code plus propre et correctement segmenté par l'apprentissage des techniques de test unitaires, d'encapsulation et de paquetage. De plus étant coordinateur, j'ai pu lors de ce TD réaliser qu'il n'était pas toujours facile de gérer une équipe de 5 personnes lorsque la deadline est relativement proche du démarrage du TD. Néanmoins nous avions préparé le TD légerement en avance ce qui nous a permis de bien nous organiser et ainsi de terminer ce TD sans trop de problème. Bien que n'ayant pas perftionné le code, j'ai bien pris conscience qu'il était par exemple préférable de factoriser le code effectuant les test unitaires. En conclusion ce TD suit parfaitement le premier TD et m'a réellement permis de me familliariser d'avantage avec le java tout en restant accesible aux personnes comme moi n'ayant jamais fait de java avant.