- Attention aux termes (fastidieux !!) - Bien de mettre les corrections.
- Manque EtatPassager ? et le rôle de nos Nandous.

Compte Rendu du TD numéro 3 de l'équipe Eirb'reteau

insuffisant

manque des choses & code bizarre

Pour le 29 septembre

Coordinateur : Reda Boudjeltia
Tandem 1 : Pierre Gaulon, Lionel Adotevi
Tandem 2 : Victor Dury, Aurélien Nizet
TD du 29 septembre 2014

1 Analyse des dépendances entre classes

1.1 Dépendances et développement en parallèle

1.2 Des objets factices pour tester

1.2.1 Exemple et mise en place des tests faussaire

Au début de notre TP, on nous suggère d'appliquer un test avec l'utilisation d'un faussaire sur l'exemple du *TestTelecommande*.
La démarche de compilation à partir du dossier usageDeFaux est la suivante :

```
javac -d build/ src/*.java
javac -d build-tst/ tst/Messages.java
javac -d build-tst/ -cp build-tst/ tst/PorteCharniere.java
javac -d build-tst/ -cp build-tst/:build/  tst/TestTelecommande.java
```

Pour l'exécution :

```
java -cp build/ UnScenario
java -ea -cp build-tst/:build/ TestTelecomande
```

Afin que la classe *TestTelecommande* utilise bien la version faussaire de la classe *PorteCharniere*, il est nécessaire de respecter un ordre précis lors de l'exécution de la JVM par l'intermédiaire de la commande *java* au niveau de l'entrée des classpaths.
Néanmoins, si on inverse cette ordre, il y aura une erreur de la JVM. Qui sera de la forme :

```
.Exception in thread "main" java.lang.NoSuchMethodError: PorteCharniere.<init>(B)V
at TestTelecommande.testAction(TestTelecommande.java:6)
at TestTelecommande.main(TestTelecommande.java:78)
```

ou Kcheck ?

En fait, le compilateur doit compiler tout ce qu'il y a dans build-tst en premier et ensuite tout ce qu'il y a dans build. Ainsi, il trouve d'abord *PorteChaniere* de build-tst avant celui de build et n'aura donc pas à lire celui de build.

1.3 Organisation du développement

Ce qui suit, sera la mise en place des tests avec faussaire des classes *Autobus* et *PassagerStandard*. Chacun des deux tandems fut en charge de ce travail en essayant de développer de la façon la plus homogène possible.

2 *PassagerStandard* et *Autobus*

2.1 Préparation du répertoire de travail

Chaque tadems s'est chargé de préparer le travail, en mettant en place un repertoire de travail organisé rigoureusement contenant une version faussaire de classe qu'ils ne s'occupent pas. Les classes accessibles pour les deux tandems en dehors du paquetage sont, *Autobus* (défini public) et *PassagerStandard* (défini public).

ok

Tandem 1 : En charge de *EtatPassager*.
- src/
  - EtatPassager.java
  - PassagerStandard.java
- tst/
  - Autobus.java (ver. faussaire)
  - Lancertests.java
  - Messages.java
  - TestEtatPassager.java
  - TestPassagerStandard.java

Tandem 2 : En charge de *Autobus*.
- src/
  - Autobus.java
  - JaugeNaturel.java
- tst/
  - PassagerStandard.java (ver. faussaire)
  - Lancertests.java
  - Messages.java
  - TestJaugeNaturel.java
  - TestAutobus.java

ok

Syntaxe !

2.2 Développement et tests

2.2.1 Façons de tester

L'énoncé nous suggérait deux manière de développer les tests. La première, était d'écrire du code puis son test associé (connu sous le nom de *Test Fail first*), et la seconde était d'écrire les tests avant d'implémenter le code (connu sous le nom de *Test Driven Developpement* (T.D.D)).
Par conséquent, nous avons essayé d'utiliser les deux manières afin d'en peser les pours et les contres.

Premièrement, nous avons tenté d'appliquer le style T.D.D le plus souvent possible du fait qu'elle nous était moins connue que l'autre. Ainsi, on s'est aperçu que malgré le fait que ce style de développement de test soit intéressant, il n'en demeure pas moins fastidieux et long à mettre en place. Notons quand même une expérience, lors du développement de la méthode *demanderPlaceAssise*(), nous nous sommes aperçus en développant nos tests qu'ils fonctionnaient alors que la fonction était vide : un problème que nous n'aurions pas pu déceler en développant selon la technique classique.
Ensuite, en raison du temps à consacrer au style de développement T.D.D, nous nous sommes remis à développer les tests après avoir écrit les codes des traitements.
Puis, en comparant les deux types de développement de tests, on peut dire que les deux façons d'écrire les tests sont bonnes mais le style T.D.D serait plus intéressant lorsque le traitement d'une méthode peut s'avérer long et complexe à écrire. En effet, on aura donc dans ce cas, un fil conducteur lors de l'écriture d'un traitement qui est assuré d'être testé unitairement. L'autre méthode, trouve son interêt dans les méthodes simples et courtes.

2.2.2 Traitements des Classes

Tout d'abord, il faut savoir que que cette parti fut fastidieuse à traiter. Cela est dû principalement au fait que nous avons tous trouvé la doc ambigue. Néamoins, des solutions ont été trouvées.
En ce qui concerne le tandem 1, en charge de *PassagerStandard*. L'équipe s'est chargé de développer des tests pour des méthodes tels que *MonterDans*() et *NouvelArret*().
Pour *MonterDans*(), il y a eu des problèmes situé au niveau des valeurs retournée par les faussaires *aPaceAssise*() et *aPlaceDebout*(). En effet, au début, les accesseurs de la classe faussaire ne renvoyaient que la valeur *false* alors que nous avons dû modifier selon la m'éthode *placeAssise*. Le problème fut résolut en mettant une condition sur l'attribut *placeAssise* (== 0).
De plus, notons que cette idée est venu en regardant le code en C, et qu'après ce problème nous avons préter plus d'attention au code C d'origine, ce qui nous à permis de gagner en temps, en particulier pour le tandem 2.
Pour *nouvelArret*(), la fonction de test fonctionnait bien, aucun problème a été rencontré.
Ensuite, dans le développement de ses tests, le tandem 2 devait gérer plusieurs instances de *PassagerStandard*() stockée dans un tableau. A part ça, le protocole de développement fut le similaire à celui du tandem 1. Comme susdit, nous nous somme souvent inspirer du code C d'origine lors de ce développement contenu du fait que nous avions pas réussit à bien comprendre la documentation. A partir de là, peu de problème furent rencontré. Notons, que le tandem 2 a porté un intêrt particulier aux tests de type TDD.

3 Commentaires

3.1 Commentaire de Pierre

Je trouve que ce TD a été plus difficile que le précédent dans la mesure où il fallait gérer les classes faussaires, et adapter nos méthodes selon la documentation, qui n'est, à mon goût, pas très explicite. C'est pour ça que j'ai souvent dû me référer à la version en C du code pour comprendre le fonctionnement des différentes classes, leurs attributs, et les méthodes à développer. Par rapport à la méthode qui consistait à développer d'abord les tests et ensuite le code de la fonction, j'ai essayé de m'y tenir, mais je trouve que pour de petites fonctions comme celles-ci, on perd plus de temps par rapport à la méthode classique.

3.2 Commentaire de Victor

Cette partie fut fort intéressante. Nous avons compris l'utilité des tests, et de les implémenter avant d'implémenter les fonctions, afin de mieux réfléchir à ce que nous voulons vraiment faire. Maintenant que la documentation est claire dans nos têtes, et que le langage Java commence à nous être familier, nous sommes de plus en plus rapides à écrire. Le travail en équipe est satisfaisant, car nous travaillons ensemble, et il y a un débat constant sur l'homogénéité du code, ainsi que sur la compréhension des consignes.

3.3 Commentaire de Reda

A travers ce TP, en tant que coordinateur, je me suis rendu compte de l'importance de l'écoute des autres pour synchroniser les tandems et résoudre des problèmes de répartition de taches. En effet, il n'a pas était rare pour l'écriture du rapport d'avoir de longue conversation sur le développement des tests. D'autre, il m'a fallu prendre connaissance du sujet et de tenter de le comprendre au même titre que les equipes en charge du code, de comprendre ce qu'ils fesaient sans avoir a y toucher, mais aussi de réflechir au solution de certains problème avec eux. Ainsi, cela m'a permis de bien connaitre les deux partis développer, et parfois de transmettre une solution trouver par un tandem à l'autre tandem dans le but de le débloquer. Pour finir, ce fut une expérience très enrichissante.

3.4 Commentaire de Aurélien

Ayant été coordinateur du TD2, ce TD3 est le premier TD où j'ai pu réellement coder en java avec mon adjoint du tandem. Lors du dévelopement de la classe *Autobus*, nous avons choisi de d'abord coder les tests unitaires puis ensuite le code des méthode (TDD), c'était la première fois que je coder réellement de cette manière, et j'ai pu comprendre son intêret. Parfois nous avons mis beaucoup de temps à écrire quelques lignes de code car nous avons mis un temps conséquent à réfléchir sur les tests unitaires, ainsi que les problèmes rencontré avec la classe faussaire. En fin de compte, j'ai pu bien comprendre la méthode de développement en tandem que nous avons suivi et j'ai apprécié m'investir dans ce TD.

⇒ les Tests ne sont pas conformes à la spécifications.

3.5 Commentaire de Lionel

Dans le cadre de ce TP, j'ai eu l'occasion de découvrir une nouvel façon de développer des tests et de me rendre compte des avantages que cela peu apporter dans le cadre de gros projet. En effet, j'ignorait l'existance des tests avec faussaire, alors qu'ils n'en demeurre pas moins intéréssant lorsque l'equipe de développement est divisé. Ensuite, la méthode de développement TDD, qui est assez particulière m'a permis de voir une autre façons d'écrire un traiment de code plus sûre et testable unitairement.

4 Conclusion

A travers ce TP, nous avons découvert de nouvelle façons de produire des tests qui peuvent être utile lors de gros projet tel que notre PFA. ¿ solution?
Ainsi, tester avec un faussaire, nous permet d'effectuer des tests malgré le manque d'une classe. Cela peut se montrer très intéressant lors d'une phase de développement d'un projet quand une équipe a besoin d'utiliser une partie en état de développement par une autre équipe ou qui n'a pas encore été développée.
Puis, nous avons vu deux façons distincts de développer des tests. L'une fut plus traditionnelle alors que la seconde fut plus fastidieuse à mettre en oeuvre et à pris un temps de réflexion non negligeable. Nous en avons conclut, qu'avant de se lancer dans un type de test ou un autre, il est préférable de peser la complexité d'écriture des traitements à tester.

|| surtout de faire des tests peut importe la méthode

- Quid de EtatPassager ??
- code : Autobus : * demanderSortie utilise le Mock ??? !!!
  ↳ mettre à jour les jauges, et le tableaux !!!
  * changement du prototype de ~~aller~~arrêtSuivant(…) ?? PassagerStandard.