Haute École de Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2018 – 2019
DEV1

DEV1 – Laboratoires Java I

TD 9 – Les tests unitaires

Un code est souvent modifié, cela veut dire que le temps passé à le simplifier est souvent gage d'un gain de temps lors d'une modification ultérieure. Il vous sera même peut-être conseillé par votre professeur de

1. faire un code fonctionnel ;
2. de le modifier afin d'améliorer sa lisibilité et sa modularité ;
3. et enfin, d'en améliorer son efficacité.

Les codes sources et les solutions de ce TD se trouvent à l'adresse :

`https://git.esi-bru.be/dev1/labo-java/tree/master/td09-tests-unitaires`

Table des matières

Version du document
10 septembre 2018

1 Introduction

La démarche que l'on vous demande de suivre dans ce TD permet d'avoir un code sans bug [1], grâce aux tests unitaires ainsi que le développement dirigé par les testes. Ces méthodoloties permettent d'obtenir des tests maintenanables tout au long de la vie d'un code.

Jusqu'à présent, nous vous avons demandé d'écrire du code afin de répondre à une demande. Pour cela, vous avez dû :

1. lire et comprendre l'énoncé ;
2. déterminer les entrées et les sorties ;
3. penser votre algorithme ;
4. traduire l'algorithme en pseudo-code et/ou Java ;
5. tester votre algorithme.

Dans ce laboratoire, vous allez faire ce que l'on appelle du Développement dirigé par les tests. Pour faire court, il s'agit d'écrire les tests avant d'avoir écrit le code. En effet, on sait, avant de l'écrire, ce que le code est censé faire.

Ce que vous devez faire tout au lond de ce TD est :

1. lire et comprendre l'énoncé ;
2. déterminer les entrées et les sorties ;
3. penser aux tests que vos méthodes devront passer ;
4. écrire les tests ;
5. penser votre algorithme ;
6. traduire l'algorithme en pseudo-code et/ou Java ;
7. tester votre algorithme [2].

2 La couverture de code

Nous allons tenter de comprendre ce qu'est une bonne *couverture de code* en reprenant notre fonction valeur absolue qui est définie de la façon suivante :

$$|x| = x \text{ si } x \geq 0, -x \text{ sinon.}$$

Soit la méthode `abs` qui a pour signature `public static double abs(double x)` puisque la fonction absolue s'applique sur des réels.

Nous devons maintenant réfléchir aux tests que nous devons réaliser afin de garantir le fonctionnement de notre méthode. C'est-à-dire, les tests nécessaires [3] pour garantir le fonctionnement de la méthode quelque soit l'entrée autorisée de la méthode. Réfléchissons-y ensemble.

Prenons une valeur positive, 4 par exemple. Le code suivant, qui n'est pas la valeur absolue, passe avec succès notre test puisqu'il va retourner la valeur d'entrée.

```
public static double abs(double x) {
   return x;
}
```

1. Un code sans bug est difficile à obtenir, mais un développer doit les éviter autant qu'il peut.
2. Certains codes ont été écrit lors d'un TD précédent. L'ordre des étapes n'est parfois pas respecté.
3. Avoir les testes nécessaires et suffisants c'est mieux. À défaut, il en vaut mieux trop que pas assez.

Si l'objectif n'est pas de trouver un code pour lequel les tests fonctionnent, il montre bien que notre batterie de tests est insuffisante.

Ajoutons un second test en essayant l'entrée -4. Le code suivant, qui n'est pas la valeur absolue, passe avec succès nos deux tests.

```
public static double abs(double x) {
   return 4;
}
```

Si nous prenons toutes nos observations en considération, nous obtenons le plan de tests suivant

n° du test	entrées	résultat attendu	note
1	4	4	nombre positif
2	-6	6	Un nombre négatif

Même si notre exemple est aberrant, nous souhaitons ici mettre en évidence la difficulté d'obtenir des tests complet, que le processus d'élaboration des tests unitaires demande de la réflexion et qu'il ne garantie pas à 100% le bon fonctionnement du code. Il est donc essentiel de prendre le temps nécessaire afin de fournir une couverture de code aussi complète et exacte que possible [4].

Exercice 1 Couverture de code - Max

Donnez les entrées et sorties nécessaires à l'élaboration d'une bonne couverture de code pour la méthode `max(int a, int b)` qui permet d'obtenir la plus grande valeur parmis les 2 passées en paramètre.

Exercice 2 Couverture de code - Somme d'entiers consécutifs

Donnez les entrées et sorties nécessaires à l'élaboration d'une bonne couverture de code pour la méthode `somme(int n)`qui permet de calculer la somme des entiers consécutifs de 1 à n.

Exercice 3 Couverture de code - Anagramme

Une anagramme est une construction fondée sur une figure de style qui inverse ou permute les lettres d'un mot ou d'un groupe de mots pour en extraire un sens ou un mot nouveau [5].

Donnez les entrées et sorties nécessaires à l'élaboration d'une bonne couverture de code de la méthode estAnagramme qui permet de vérifier si un text est un anagramme.

Exercice 4 Couverture de code - Palindrome

Le palindrome est un texte ou un mot qui reste identique qu'on le lise de gauche à droite ou de droite à gauche. Nous considérons ici la version stricte dans le sens où l'on prend en considération les signes diacritiques (accents, trémas, cédilles) ainsi que les espaces [6].

Donnez les entrées et sorties nécessaires à l'élaboration d'une bonne couverture de code de la méthode estPalindrome qui permet de vérifier si un text est un palindrome.

Exercice 5 Couverture de code - Nombre d'occurances

4. Vous pouvez remarquer que dans nos exemples, la couverture de code est à chaque fois de 100%, que malgré cela le code ne fait pas toujours ce qu'il doit faire.
5. https ://fr.wikipedia.org/wiki/Anagramme
6. https ://fr.wikipedia.org/wiki/Palindrome

Donnez les entrées et sorties nécessaires à l'élaboration d'une bonne couverture de code de la méthode `public static int nbOccurance(char lettre, String text)`. Cette méthode permet de compter le nombre d'occurances d'une lettre dans un text.

3 JUnit

3.1 Tester le lancement d'une exception

4 Exercices supplémentaires