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
17 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 méthodologies permettent d'obtenir des tests maintenables 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. Cela est possible car on sait, avant de l'écrire, ce que le code est censé faire.

Ce que vous devez faire tout au long 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 La couverture de code

Ensemble, tentons 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)`.

Nous allons maintenant réfléchir aux tests que nous devons réaliser afin de garantir le bon fonctionnement de notre méthode. C'est-à-dire, les tests nécessaires [2] pour garantir le fonctionnement de la méthode quelque soit l'entrée passée à la méthode.

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 développeur doit les éviter autant qu'il peut.
2. 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 complets, 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 [3].

Exercice 1 Max - Couverture de code

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 parmi les 2 passées en paramètre.

Exercice 2 Somme d'entiers consécutifs - Couverture de code

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 Anagramme - Couverture de code

« 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. »Wikipedia. [4].

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

Exercice 4 Palindrome - Couverture de code

« 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. »Wikipedia [5].

3. Vous pouvez remarquer que pour chacun de nos exemples, la couverture de code est de 100% ; et que malgré cela, le code ne fait pas toujours ce qu'on souhaitait qu'il fasse.
4. https ://fr.wikipedia.org/wiki/Anagramme
5. https ://fr.wikipedia.org/wiki/Palindrome

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 texte est un palindrome.

Exercice 5 **Nombre occurrences - Couverture de code**

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'occurrences d'une lettre dans un texte.

3 JUnit

Si nous reprenons l'exemple de la méthode calculant la valeur absolue, nous pourrions la développer et la tester comme ceci :

```java
package esi.dev1.td9;

public class Math {

   /**
    * Calcule la valeur absolue d'un nombre.
    *
    * @param message message à afficher.
    * @return l'entier saisi par l'utilisateur.
    */
   static double abs(double x) {
      double solution = x;

      if(x < 0) {
         solution = -x;
      }

      return solution;
   }

   public static void main(String[] args) {
      int entrée;
      int sortie;

      entrée = 4;
      sortie = 4;
      System.out.println("Teste 1 : abs(" + entrée + ") = " + sortie + " ? "
         + (abs(entrée) == sortie));

      entrée = -6;
      sortie = 6;
      System.out.println("Teste 2 : abs(" + entrée + ") = " + sortie + " ? "
         + (abs(entrée) == sortie));
   }
}
```

MonMath.java

Au lancement du code, les deux tests passent.

```
abs(4) = 4 ? true
abs(-6) = 6 ? true
```

Les problèmes avec cette méthodologie sont multiples.

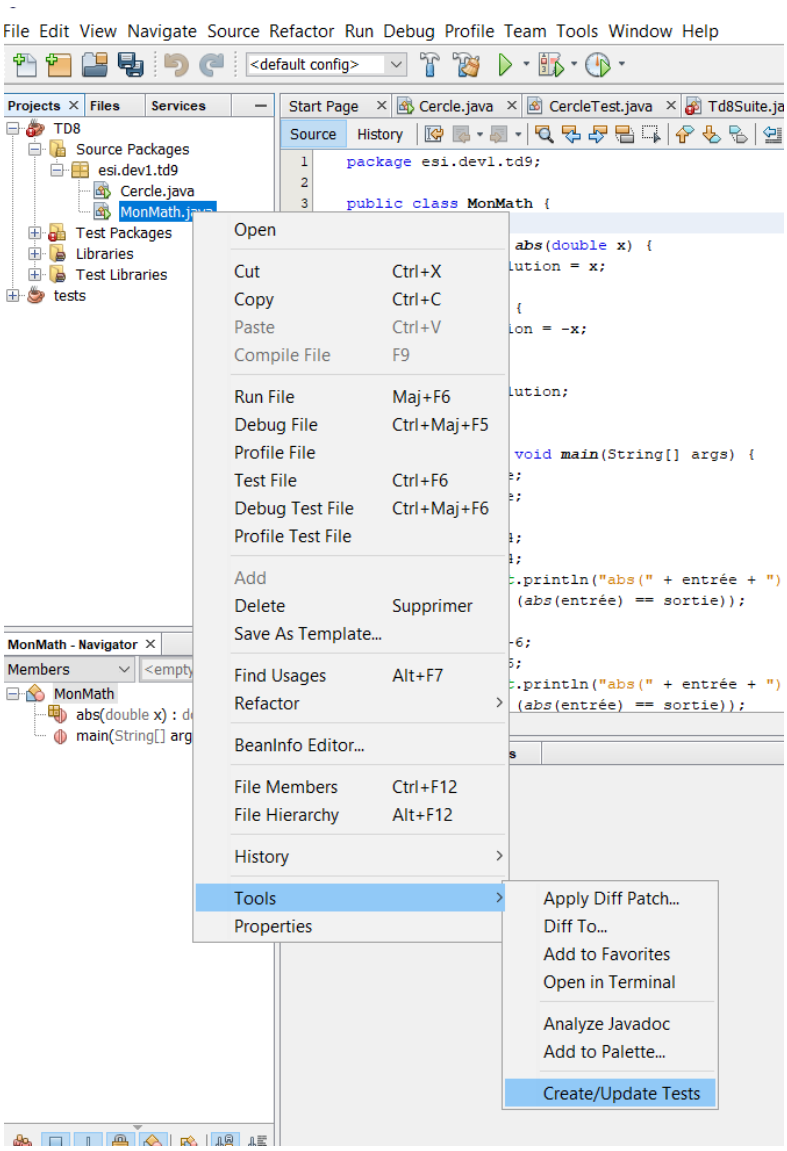
1. Il ne peut y avoir qu'une seule méthode main par fichier Java.
2. On doit avoir accès au fichier pour compléter les tests.
3. Il faut être rigoureux pour que les tests soient lisibles.

4. Le lancement des tests peut prendre du temps puisqu'il faut lancer chaque fichier séparément...

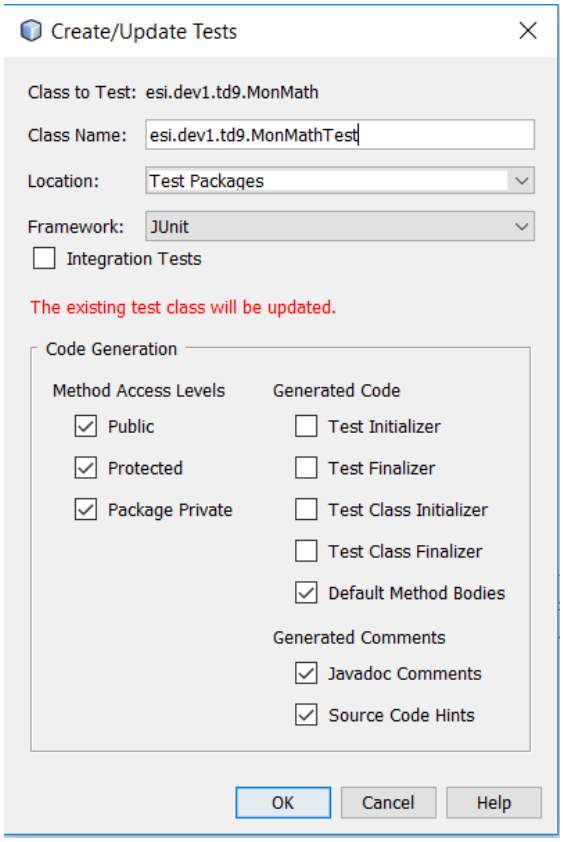
Tous ces problèmes mènes à un problème de maintenabilité des tests et du code.

Notre manière de tester peut être améliorée en sortant les méthodes principales dans des fichiers différents et le dernier point en mettant tous les points dans la même méthode, mais il nous reste à être rigoureux pour obtenir un code maintenable. Nous allons donc utiliser le framework JUnit qui va nous obliger à obtenir cette lisibilité.

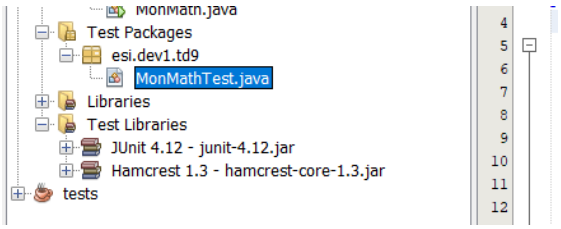
La première chose à faire est de configurer Netbeans. Heureusement pour vous, il fait presque tout lui même (à condition de bien s'y prendre). Après avoir écrit le fichier MATH.JAVA dans un nouveau package ESI.DEV1.TD9, faite un clique droit sur le package. Sélectionnez tools et faite Create/Update tests.



Une nouvelle fenêtre va apparaître, décochez les 4 checkbox en dessous de « Generated Code »et validez.



De nouveaux éléments apparaissent dans votre projet. D'une part des éléments dans les librairies utilisées par netbeans ainsi qu'un nouveau dossier nommé Tests packages. C'est dans ce dernier dossier que vous allez écrire vos tests.



Il vous reste à considérer le message laissé par netbeans.

```
// TODO review the generated test code and remove the default call to fail.
```

Une fois le code nettoyé, vous devriez obtenir ceci :

```
1  package esi.dev1.td9;
2
3  import org.junit.Test;
4  import static org.junit.Assert.*;
5
6  /**
7   *
8   * @author ESI Prof
9   */
10 public class MonMathTest {
11
12    @Test
13    public void testAbs() {
14       System.out.println("abs");
15       double x = 0.0;
16       double expResult = 0.0;
17       double result = MonMath.abs(x);
```

```
18          assertEquals(expResult, result, 0.0);
19      }
20  }
```

MonMathTestEMPTY

La variable x représente l'entrée de la méthode à tester et la variable `expResult` représente le résultat attendu.

La méthode `asserEquals` est une méthode qui prend en paramètre :

- ▷ Le résultat attendu.
- ▷ Le résultat obtenu (calculé par la méthode que vous souhaitez tester).
- ▷ Une marge d'erreur.

Le troisième paramètre ne doit être utilisé que si la sortie de la méthode est de type double[6]. L'appel à la méthode `assertEquals` avec un retour de méthode entier s'écrit donc : `assertEquals(expResult, result)`. Si la méthode que vous testez retourne un booléen, il est alors préférable d'utiliser les méthodes `assertTrue` ou `assertFalse`. Ces méthodes prennent un unique paramètre, le résultat, puisque le résultat attendu est défini par la méthode qu'on a choisi d'utiliser. En effet en utilisant `assertTrue`, on s'attend à avoir un résultat positif.

3.1 Tester le lancement d'une exception

Si nous reprenons le code de la méthode périmètre, nous remarquons qu'une exception est lancée dans le cas d'un rayon négatif. En effet, notre plan de test doit ressembler à quelque chose comme ceci :

n° du test	entrées	résultat attendu	note
1	0	0	Extremum du domaine de définition
2	4	25.12	valeur positive (précision 0.01)
3	-5	`erreur`	valeur négative

Nous vous laissons réaliser les deux premiers tests. Pour vérifier le lancement d'une exception, ajoutez le tag `@Test` en spécifiant le type de l'exception attendue.

```
1  package esi.dev1.td8;
2
3  import org.junit.Test;
4  import static org.junit.Assert.*;
5
6  /**
7   *
8   * @author ESI Prof
9   */
10 public class CercleTest {
11     /**
12      * Test de la méthode périmètre avec le lancement d'une exception
13      */
14     @Test(expected=IllegalArgumentException.class)
15     public void testMain() {
16         System.out.println("Avec rayon négatif menant à une exception");
17         Cercle.périmètre(-5);
18     }
19
20 }
```

CercleTest.java

6. Pourquoi ?

Remarquez que dans le cas présent, vous ne devez pas ajouter de méthode `assert`[7].

Exercice 6 Couverture de code - Exceptions

Pour chacun des exercices précédents, vérifiez que vous avez bien pensé aux cas menant à des erreurs.

Exercice 7 Implémentation - Max

1. Écrivez un nouveau fichier Math dans lequel vous allez ajouter la signature de méthode de la méthode `max`. Faite un retour quelconque afin de rendre le code compilable.
2. Développez, avec le framwork JUnit, votre couverture de code écrite précédemment.
3. Lancez votre fichier de test.
4. Développez votre méthode afin que celle-ci complète tous les tests.

Exercice 8 Implémentation - Somme d'entiers consécutifs

1. Écrivez un nouveau fichier Math dans lequel vous allez ajouter la signature de méthode de la méthode `somme`. Faite un retour quelconque afin de rendre le code compilable.
2. Développez, avec le framwork JUnit, votre couverture de code écrite précédemment.
3. Lancez votre fichier de test.
4. Développez votre méthode afin que celle-ci complète tous les tests.

Exercice 9 Implémentation - Anagramme

1. Écrivez un nouveau fichier Math dans lequel vous allez ajouter la signature de méthode de la méthode `estAnagramme`. Faite un retour quelconque afin de rendre le code compilable.
2. Développez, avec le framwork JUnit, votre couverture de code écrite précédemment.
3. Lancez votre fichier de test.
4. Développez votre méthode afin que celle-ci complète tous les tests.

Exercice 10 Implémentation - Palindrome

1. Écrivez un nouveau fichier Math dans lequel vous allez ajouter la signature de méthode de la méthode `estPalindrome`. Faite un retour quelconque afin de rendre le code compilable.
2. Développez, avec le framwork JUnit, votre couverture de code écrite précédemment.
3. Lancez votre fichier de test.
4. Développez votre méthode afin que celle-ci complète tous les tests.

Exercice 11 Implémentation - Nombre d'occurrences

7. Pourquoi ?

1. Écrivez un nouveau fichier Math dans lequel vous allez ajouter la signature de méthode de la méthode `nbOccurances`. Faite un retour quelconque afin de rendre le code compilable.
2. Développez, avec le framwork JUnit, votre couverture de code écrite précédemment.
3. Lancez votre fichier de test.
4. Développez votre méthode afin que celle-ci complète tous les tests.

4 Exercices supplémentaires

Exercice 12 Implémentation - PGCD

Selon le principe TDD, implémentez la méthode `pgcd(int a, int b)` qui retourne le pgcd de deux nombres.

Exercice 13 Implémentation - PPCM

Selon le principe TDD, implémentez la méthode `ppcm(int a, int b)` qui retourne le ppcm de deux nombres.

Exercice 14 Implémentation - Chiffrement par décalage

Selon le principe TDD, implémentez la méthode `césar(String texte, int décalage)` qui permet de retourner un texte chiffré selon le code de César [8].

8. https ://fr.wikipedia.org/wiki/Chiffrement_par_d%C3%A9calage