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

2020 – 2021
DEV1 – JAVL

DEV1 – JAVL – Laboratoires Java

TD 10 – Les tests unitaires

Un code est souvent modifié. Cela veut dire que le temps passé à le simplifier permet souvent un gain de temps lors d'une modification ultérieure. Il est même parfois conseillé de : écrire un code simplement fonctionnel, de le modifier afin d'améliorer sa lisibilité et sa modularité et, enfin, d'améliorer son efficacité.

Toutes ses modifications du code imposent que le code soit testé. Il doit être testé à chaque modification ce qui serait fastidieux si les tests n'étaient pas automatisés. Heureusement, c'est ce que propose JUnit.

Attention : Pour ce TD, nous vous demandons explicitement de travailler avec Maven.
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

1 Introduction **2**

2 La couverture de code **2**

3 JUnit **4**
3.1 Écrire des tests unitaires 5
3.2 Lancer les tests 7
3.3 Bonnes pratiques 7
3.4 Tester le lancement d'une exception 7

4 Exercices supplémentaires **8**

Version du document
2 octobre 2020

1 Introduction

La démarche que l'on vous propose de suivre dans ce TD permet de limiter au maximum le nombre de *bugs* dans son code [1].

Les tests unitaires ainsi que le développement dirigé par les tests permettent d'obtenir des tests maintenables tout au long de la vie d'un code et d'atteindre l'objectif d'un code (quasi) exempt de *bugs*.

Jusqu'à présent, nous avons écrit du code afin de répondre à une demande. Pour cela, il a fallu :

1. lire et comprendre l'énoncé ;
2. déterminer les entrées et les sorties ;
3. penser votre algorithme ;
4. construire l'algorithme puis l'écrire en Java ;
5. tester votre algorithme.

Ce laboratoire présente le *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.

La démarche est la suivante :

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. construire l'algorithme puis l'écrire en 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 cette méthode. C'est-à-dire, les tests nécessaires [2] pour garantir le fonctionnement de la méthode quelle que soit l'entrée passée à la méthode.

Prenons une valeur positive, 4 par exemple. Le code suivant, qui est pourtant faux, 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 tests nécessaires et suffisants c'est mieux. À défaut, il en vaut mieux trop que pas assez.

Notre test pourrait faire croire que la méthode est correcte. Ce qui 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	-7	7	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 garantit 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)`. Cette méthode calcule la somme des entiers consécutifs de 1 à n.

Exercice 3 Anagramme - Couverture de code

Donnez les entrées et sorties nécessaires à l'élaboration d'une bonne couverture de code de la méthode `boolean estAnagramme(String mot, String candidat)`. Cette méthode vérifie si une chaîne de caractères est une 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. »Wikipedia [4].

Exercice 4 Palindrome - Couverture de code

Donnez les entrées et sorties nécessaires à l'élaboration d'une bonne couverture de code de la méthode `boolean estPalindrome(String mot)`. Cette méthode vérifie qu'un texte est un 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. » 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

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 nbOccurences(char lettre, String texte)`. 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 :

MonMath.java

```java
package esi.dev1.td10;

public class MonMath {

   /**
    * Calcule la valeur absolue d'un nombre.
    *
    * @param message message à afficher.
    * @return l'entier saisi par l'utilisateur.
    */
   public 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));
   }
}
```

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. Il faut être rigoureux pour que les tests soient lisibles.
3. Il faut vérifier à l'écran que chaque ligne affiche un `true` ce qui peut prendre du temps s'il y a des centaines de tests.
4. Le lancement des tests peut prendre du temps puisqu'il faut lancer chaque fichier séparément...

Tous ces problèmes mènent 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 tests 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 permettre d'obtenir cette lisibilité.

Rappel : Pour la suite de ce TD, il est important que vous utilisiez Maven (qui ajoutera automatiquement la librairie de test JUnit à votre projet). Si ce n'est pas encore fait, veuillez donc créer un nouveau projet Java utilisant Maven.

3.1 Écrire des tests unitaires

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 `MonMath.java` dans un nouveau package `esi.dev1.td10`, faite un clic droit sur la classe à tester. Sélectionnez tools et faites *Create/Update tests*.

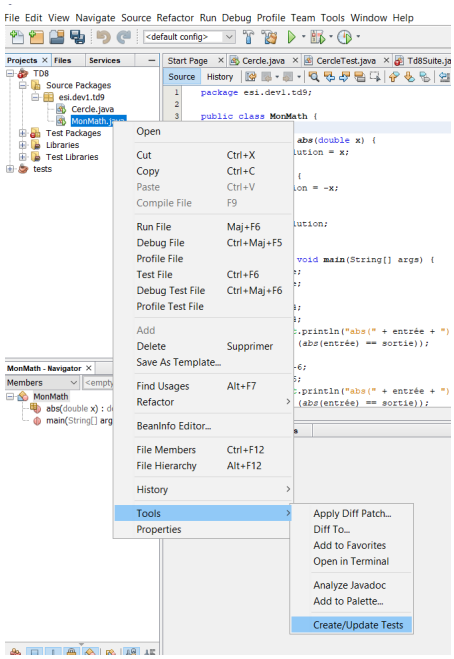

Figure 1 – Demande de création des tests

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

De nouveaux éléments, créés par NetBeans, apparaissent dans votre projet. Remarquez un nouveau dossier nommé *Tests packages* dans lequel se trouve une nouvelle classe `MonMathTest.java`. C'est dans cette classe que vous écrirez vos premiers tests.

Il vous reste à considérer le message laissé par NetBeans et compléter le code.

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

Une fois le code ajouté, vous devriez obtenir ceci :

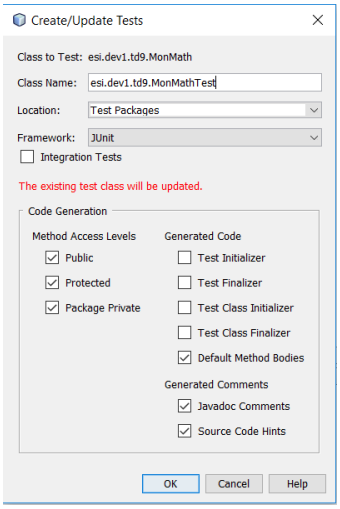

FIGURE 2 – Choix des tests qui seront créés automatiquement

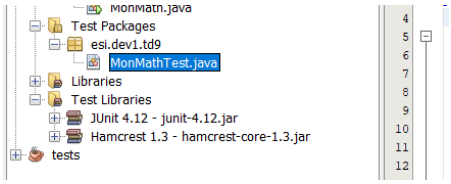

FIGURE 3 – Montre l'arborescence créée

MonMathTestEMPTY

```java
package esi.dev1.td10;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

/**
 *
 * @author ESI Prof
 */
public class MonMathTest {

    @Test
    public void testAbs() {
        double x = 4.0;
        double expResult = 4.0;
        double result = MonMath.abs(x);
        assertEquals(expResult, result, 0.001);
    }
}
```

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 pseudo-réel, dans ce cas `double`[6]. L'appel à la méthode `assertEquals` avec un retour

6. Pourquoi ?

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(result)` ou `asserFalse(result)`. Ces méthodes prennent un unique paramètre puisque le résultat attendu est défini par le nom même de la méthode utilisée.

3.2 Lancer les tests

Pour lancer les tests, il suffit de choisir `Run/Test Project (Alt+F6)` dans le menu. Vous pouvez également lancer un seul fichier de test via un clic droit sur le fichier puis `Test File`.

NetBeans affiche alors un message indiquant le nombre de tests lancés/réussis.

En cas d'erreur, il génère un petit rapport reprenant pour chaque test lancé :

- Son nom ;
- La valeur attendue (expResult) ;
- La valeur obtenue.

Un bon réflexe en cas d'erreur

Lorsqu'un test rate c'est parfois parce que le test a **mal été écrit** et pas à cause de la méthode testée. Soyez prudent !

3.3 Bonnes pratiques

En cas d'erreur, il faut pouvoir identifier rapidement ce qui n'a pas été et pourquoi. Pour cela, le choix du nom de la méthode de test est crucial. Comme il ne sera jamais appelé explicitement, il peut être très long ! Voici quelques exemples possibles pour le test de la valeur absolue d'un nombre positif.

- `testAbs_nombrePositif`
- `absQuandPositifAlorsInchangé`
- `testAbs_LaValeurDUnNombrePositifEstLuiMeme`
- `abs_NombrePositif_ResteInchangé`

On peut aussi veiller à rendre le code du test le plus lisible possible. Voici une écriture possible pour le test de la valeur absolue d'un nombre négatif.

```
@Test
public void abs_QuandNégatifAlorsInversé() {
   assertEquals(6.0, Math.abs(-6), 0.001);
}
```

3.4 Tester le lancement d'une exception

Voici le code d'une méthode qui calcule le périmètre d'un cercle.

Cercle.java

```
1  public static double périmètre(double rayon) {
2     if(rayon < 0) {
3        throw new IllegalArgumentException("Le rayon doit être positif: "+rayon);
4     }
5     return 2 * Math.PI * rayon;
6  }
```

Nous remarquons qu'une exception est lancée dans le cas d'un rayon négatif. Un plan de tests devrait ressembler à :

n° du test	entrées	résultat attendu	note
2	4	25.12	valeur positive (précision 0.01)
1	0	0	plus petite entrée acceptée
3	-5	`erreur`	valeur négative

Nous vous laissons écrire les deux premiers tests.

Comment vérifier le lancement d'une exception dans le cas du troisième test ? Il faut cette fois utiliser la méthode `assertThrows` plutôt que `assert`. Cette méthode vérifie qu'une exception est bien déclenchée, elle prend en paramètre le type de l'exception attendue et la méthode que l'on souhaite tester.

Ecrivez le code de la méthode de test ainsi :

CercleTest.java

```
@Test
public void périmètre_rayonNégatif_IAException() {
    assertThrows(IllegalArgumentException.class, () -> MonMath.périmètre(-5));
}
```

Si, lors de l'exécution de l'appel de méthode `MonMath.périmètre(-5)` une exception de type `IllegalArgumentException` n'est pas déclenchée, le test échouera.

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 Développement guidé par les tests

Pour les exercices 1 à 5 – c'est-à-dire pour `max`, `somme`, `estAnagramme`, `estPalindrome` et `nbOccurrences` – pour lesquels vous avez écrit la couverture de code :

- ▷ écrivez la signature de la méthode et retournez une valeur quelconque afin de rendre le code compilable ;
- ▷ développez, avec JUnit, votre couverture de code écrite précédemment. Il s'agit d'écrire plusieurs tests ;
- ▷ lancez l'exécution des tests. Cette exécution va signaler des erreurs (puisque la méthode retourne une valeur quelconque) ;
- ▷ corrigez la méthode afin qu'elle complète les tests.

4 Exercices supplémentaires

Exercice 8 Implémentation - PGCD Plus grand commun diviseur

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

Exercice 9 Implémentation - PPCM Plus petit commun multiple

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

Exercice 10 Implémentation - Chiffrement par décalage

Selon le principe vu précédemment, implémentez la méthode `césar(String texte, int décalage)` qui permet de retourner un texte chiffré selon le code de César [7].

7. https://fr.wikipedia.org/wiki/Chiffrement_par_décalage