Exercice de TDD: String Calculator Kata

GLO-2003: Introduction aux processus du génie logiciel

Hiver 2022

1 Consignes

Le but de cet exercice est de mettre en pratique le Test Driven Development. En guise de rappel, voici les «Lois du TDD »selon Robert C. Martin dans Clean Code[2] (traduction extraite de Wikipedia) :

1. Vous ne pouvez pas écrire de code de production tant que vous n'avez pas écrit un test unitaire qui échoue, c'est-à-dire qu'il n'est permis d'écrire du code de production que si un test unitaire est en échec.
2. Vous ne pouvez pas écrire plus d'un test unitaire que nécessaire pour échouer, et ne pas compiler revient à échouer, c'est-à-dire qu'il n'est permis d'écrire qu'un nouveau test unitaire en échec à la fois, et un test unitaire qui ne compile pas est déjà un test en échec.
3. Vous ne pouvez pas écrire plus de code de production que nécessaire pour que le test unitaire actuellement en échec réussisse, c'est-à-dire qu'il n'est permis d'écrire que du code de production permettant directement de faire passer le test unitaire précédent, ni plus ni moins.

On préfère aussi des tests précis avec idéalement un seul `assert` par test afin de ne tester qu'un seul concept par test.

Vous devez réaliser l'exercice une étape à la fois, sans regarder les étapes suivantes et en respectant les principes du TDD. Nous vous recommandons d'utiliser la structure `arrange-act-assert` pour organiser vos test, voir wiki.c2.com/?ArrangeActAssert pour plus de détails.

Cet exercice peut être réalisé seul ou en équipe de 2 à 3 personnes. Nous vous conseillons de réaliser l'exercise en `Pair Programming`. Nous vous recommandons aussi de créer un `repository` GitHub pour partager le code entre les membres de l'équipe et de faire un commit par étape. Vous pouvez utiliser le langage de votre choix, mais l'exercice doit être fait en TDD. Une solution sera proposée en Java sur le `repository` de l'exercise.

Le code de base se trouve ici : glo2003/Exercice-TDD-string-calculator

2 Conseils

Cet exercice demande de manipuler des chaînes de caractères. La fonction `String.split` pourrait vous être utile pour séparer une `String` selon un délimiteur. Ci dessous vous trouverez un exemple d'utilisation de `String.split` avec un `regex`.

`,|\Q***\E` veut dire que l'on va séparer sur une virgule ou sur `***`. Vous pouvez consulter regexplanet.com pour plus de détails sur les `regex`. Un petit indice, vous pouvez construire un `regex` en concaténant des chaînes de caractères.

```
String aString = "a***b,c";
String[] split = aString.split(",|\\Q***\\E");
```

Les `Streams` Java pourraient aussi vous être utiles pour manipuler des listes.

```
import java.util.List;

List<Integer> xs = List.of(1, 2, 3, 4, 5, 6);
int sum = xs.stream() // [1, 2, 3, 4, 5, 6]
            // Ne conserve que les elements ou le predicat est vrai
            .filter(x -> x % 2 == 0) // [2, 4, 6]
            // Applique une fonction lambda sur tous les elements de la liste
            .map(x -> x + 1) // [3, 5, 7]
            // Combine tous les elements de la liste en sommant
            .reduce(0, Integer::sum); // 0 + 3 + 5 + 7 = 15
```

Voici un exemple d'utilisation de JUnit5 :

```
package com.github.glo2003;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;


public class CalculatorTest {

    Calculator calculator;

    @BeforeEach
    void setUp() {
        calculator = new Calculator();
    }

    @Test
    void whenEmptyString_thenReturnsZero() {
        int result = calculator.add("");

        assertEquals(0, result);
    }

    @Test
    void whenNonNumericValues_thenThrowInvalidInput() {
        assertThrows(InvalidNumberFormatException.class,
                () -> calculator.add("4,a"));
    }

}
```

3 Kata

Le but de ce kata est de concevoir une calculatrice calculant la somme de nombres passés dans une chaîne de caractères. Ce kata est inspiré du `String Calculator Kata` de Kata-Log[1]. Le code de base se trouve ici : glo2003/Exercice-TDD-string-calculator. Nous vous conseillons de faire un commit par étape complétée.

3.1 Étape 1

Vous devez ajouter une méthode avec la signature suivante à la classe `Calculator` :

```
int add(String numbers)
```

Cette méthode doit pouvoir additionner jusqu'à deux entiers séparés par une virgule. Si les entrées sont ne sont pas des entiers, lancez une exception de type `InvalidNumberFormatException`. Les doubles délimiteurs (`1,,3`) doivent simplement être ignorés. La table 1 détaille quelques entrées et sorties attendues.

Entrée	Sortie attendue
`""`	0
`"1"`	1
`"1,2"`	3
`"1,"`	1
`"1,,3"`	4
`"4,a"`	`InvalidNumberFormatException`

Table 1 – Entrées et sorties attendues à l'étape 1

3.2 Étape 2

Vous devez modifier la méthode `add` afin de pouvoir additionner un nombre arbitraire d'entiers séparés par des virgules. La table 2 détaille quelques entrées et sorties attendues.

Entrée	Sortie attendue
`"1,2,3"`	6
`"1,2,3,4,5"`	15

Table 2 – Entrées et sorties attendues à l'étape 2

3.3 Étape 3

Modifiez la méthode `add` afin de permettre l'utilisation de `newlines` (`\n`) entre les nombres plutôt que des virgules. En bref, il doit maintenant être possible de séparer les nombres par des virgules ou des `\n`, ou les deux. La table 3 détaille quelques entrées et sorties attendues.

Entrée	Sortie attendue
`"1\n2,3"`	6
`"1,\n"`	1

Table 3 – Entrées et sorties attendues à l'étape 3

3.4 Étape 4

Vous devez modifier `add` afin de pouvoir changer, de façon optionnelle, le délimiteur entre les nombres selon la chaîne de caractères reçue. Voici le format attendu :

```
"//[delimiter]\n[numbers...]"
```

où `[delimiter]` est un caractère représentant le nouveau délimiteur et `[numbers...]` est une liste de nombres séparée par `[delimiter]`. Ainsi, si l'entrée commence par le format précédent, les entiers doivent être séparés par `[delimiter]` ou par des `\n`.

L'exemple suivant devrait retourner 6.

`"//;\n1;2\n3"`

Il est à noter que la première ligne (`//[delimiter]\n`) est optionnelle, les exemples des étapes précédentes doivent encore fonctionner.

3.5 Étape 5

Modifiez `add` afin que la méthode lance une exception de type `NegativeNumberException` si au moins un nombre négatif est passé à la méthode.

3.6 Étape 6

Modifiez la méthode `add` afin qu'elle ignore les nombres supérieurs à `1000`. Ainsi, `2 + 1001 = 2`, mais `2 + 1000 = 1002`.

3.7 Étape 7

Modifiez `add` afin de supporter des délimiteurs de longueur arbitraire (au moins un caractère) selon le format :

`"//[delimiter]\n[numbers...]"`

L'exemple suivant devrait retourner 6.

`"//[***]\n1***2***3"`

3.8 Étape 8

Modifiez `add` afin de permettre plusieurs délimiteurs selon le format suivant :

`"//[delim1][delim2]\n[numbers...]"`

L'exemple suivant devrait retourner 6.

`"//[*][%]\n1*2%3"`

3.9 Étape 9

Finalement, assurez-vous qu'il est possible d'utiliser plusieurs délimiteurs de longueur arbitraire.
L'exemple suivant devrait retourner 6.

`"//[***][%%%]\n1***2%%%3"`

4 Solution

La solution de cet exercice se trouve sur glo2003/Exercice-TDD-string-calculator. Il y a une branche par étape de la solution.

Références

[1] Software Crafters. Kata-log : String calculator kata. Available at https://kata-log.rocks/string-calculator-kata (2020/06/16), 2019.

[2] Robert C. Martin. *Clean Code : A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, 1 edition, 2008. ISBN 0132350882.