Aix*Marseille université

Année 2021-22

Domaine Sciences et Technologies
MASTER INFORMATIQUE

Fiabilité Logicielle : TP 2
Code UE : SINBU22

Tests unitaires sous JUnit

Ce TP introduit à l'utilisation de l'environnement JUnit de tests unitaires. JUnit est un environnement de tests unitaires pour Java et nous allons l'utiliser au sein de l'IDE IntelliJ [1].

La version JUnit actuelle est la version 5. La version 4 est de même philosophie, s'appuyant également sur les annotations introduites depuis Java 5. La version 4 reste très utilisée mais elle n'est pas pleinement compatible avec la version 5 [2] .

La version 5 est néanmoins une évolution majeure qui offre de nouvelles constructions. Le site pour JUnit 5 se trouve à `https://junit.org/junit5` [3].

1 Ecriture de test

Comme évoqué nous allons utiliser IntelliJ et JUnit et cela en "natif". Il est bien évidemment possible d'utiliser ces deux outils dans le cadre d'automatisation de production logicielle via les outils, Gradle ou Maven, par exemple, mais cela sort du cadre de ce TP.

Sous IntelliJ avec JUnit, on peut créer une classe de test. Les cas de test s'y retrouvent sous la forme de méthodes identifiées (précédées) par l'annotation JUnit `@Test` [4]. . Un cas de test utilisera des assertions de la classe `Assertions` de JUnit, assertions qui seront utilisées pour vérifier notamment des égalités (de valeurs, objets, ...) et feront échouer le test si elles ne sont pas vraies.

Voici un exemple de (méthode de) test :

```
@Test
public void testSumReal(){
    Complex z1 = new Complex (1.0F, 2.0F);
    Complex z2 = new Complex (3.0F, 4.0F);

    float expected = 1.0F+3.0F;

    Complex z = z1.sum(z2);

    Assertions.assertTrue(z.getRealPart()==expected, "problem with Real part of Sum");
}
```

D'autres annotations JUnit permettent de gérer les initialisations et cas parti- culiers. Comme toutes les annotations Java, elles sont de la forme `@mot-clé`.

— `@BeforeAll` et `@AfterAll` permettent d'exécuter des instructions avant et après l'exécution de la suite de tests [5].
— `@BeforeEach` permet de définir des initialisations à faire avant chaque test (typiquement définir un objet qui sera utilisé par tous les tests) et `@AfterEach` est similaire mais est effectué après chaque test [6].
— `@Disabled` permet de ne pas effectuer le test qui suit.

Les annotations `@...` et assertions `assert...` sont à importer si nécessaire (voir les fichiers exemples et la documentation).

1. `https://www.jetbrains.com/fr-fr/idea/`
2. Les versions 3 et antérieurs de JUnit utilisaient d'autres techniques pour mettre en place les tests, ces versions ne sont plus vraiment utilisées
3. Un site qui pourrait également vous être utile `https://www.jmdoudoux.fr/java/dej/chap-junit5.htm`
4. Lors de l'apparition de cette annotation, IntelliJ vous proposera notamment d'importer les classes JUnit nécessaires aux tests. Attention à bien choisir la version 5 de JUnit.
5. Cela constitue ce qu'on appelle le " test fixture".
6. Ceci constitue les préambules et postambules des cas de tests. A noter qu'ils seront exécutés avant et après chaque (méthode de) test mais que le code est identique pour tous.

2 Utilisation de Junit

Cette première partie va vous faire voir ou revoir un usage basique de JUnit.

1. Récupérer sur AMETICE les fichiers `Complex.java` et `TestComplex.java`. Le premier contient une implémentation des nombres complexes et le second un exemple de fichiers pour tester cette implémentation.
2. Sous IntelliJ, vous avez la possibilité de lancer soit l'ensemble des tests de la classe `TestComplex`, soit individuellement chaque méthode de test (via le triangle vert d'exécution dans la colonne à gauche de la fenêtre du source).
   — Lancez la méthode de test `testGetterImaginary`. Que constatez-vous ?
   — Lancez l'ensemble des tests de la classe `TestComplex`. Que constatez vous ?
3. Modifiez à minima la classe de test (au regard de ce qui a été implémenté ou non) afin que la suite des tests réussisse.
4. En rajoutant des instructions d'impression, vérifier que `@BeforeAll` et `@AfterAll` sont effectuées avant et après chaque exécution de l'ensemble des tests. Idem pour `@BeforeEach` et @`AfterEach` avant chaque méthode de test.
5. Modifiez le code de `@BeforeEach` afin de mutualiser le code de création des objets `Complex` sur lequel va se faire chacun des tests.
6. Ajouter trois méthodes pour tester la méthode `inverse` de la classe `Complex` : la première et la seconde devront tester que les parties réelle et imaginaire du résultat sont correctes dans le cas d'un complexe non nul tandis que la troisième devra tester la levée d'une exception dans le cas contraire.
7. Complétez la méthode `product` de la classe `Complex` et tester votre implémentation.
8. Complétez le code de la méthode static `infinite` afin que l'exécution de cette méthode ne termine pas puis activer le test sur `infinite`. Que se passe-t-il ? Modifier le test pour qu'il échoue si `infinite` ne termine pas en 100ms (utiliser l'assertion `assertTimeoutPreemtively` (voir la partie `Timeout` de la documentation). Vérifier que l'exécution des tests termine.

Avertissement : JUnit 5 permet de programmer une gestion de test très évoluée (voir la documentation) ce qui signifie qu'il est possible d'introduire des erreurs dans les suites de test.

3 Travail à faire

Les deux applications à tester sont celles du premier TP

3.1 Application triangle

Vous allez transcrire en Java le code C de la fonction `typeTriangle`. Cette fonction va devenir une méthode sans argument (ceux-ci étant les attributs) de la classe `Triangle`. Cette classe sera dans un package triangle du répertoire `src`.

1. Ecrire la classe Java `Triangle` avec 3 attributs privés pour les cotés, les getters et setters des attributs et la méthode int `typeTriangle()` en reprenant votre code C du précédent TP.
2. Ecrire la suite de test JUnit `testTypeTriangle`. Consigne : un test ne vérifie qu'une seule assertion à la fois. Cette suite de test sera dans un package `testTriangle` du répertoire `test`.
3. Pour rajouter `readData` la spécification est précisée [7] : si le fichier n'existe pas, une exception est renvoyée et on pourra supposer que seuls les fichiers textes sont traités. Le fichier ne

7. Pour un rappel sur les entrées-sorties en Java, vous pourrez consulter par exemple `http://thecodersbreakfast.net/index.php?post/2012/01/15/java-io-explique-simplement`

doit contenir que 3 lignes correspondant chacune à une valeur de type `float`. Ecrire la suite de tests `testReadData` pour la méthode `readData`.

3.2 Application "recherche dans un tableau trié"

1. Récupérer sous AMETICE l'archive `search_Array.jar` contenant l'interface `search_Array_Interface`

```
package com.company;

public interface search_Array_Interface {

    // recherche l'élement elt dans le tableau trié par ordre croissant tab
    // retourne l'indice de la plus petite case contenant elt et -1 si elt
    // n'est pas présent dans le tableau

    public int search(int[] tab, int elt);

}
```

   et `search_Array_Class_1`, `search_Array_Class_2`, `search_Array_Class_3`, et `search_Array_Class_4`, 4 implémentations de cette interface.

2. Le tableau `tab` est supposé être trié et on ne vérifiera pas cette propriété. Ecrire la classe de test JUnit correspondante sous la forme d'un projet `test_search_Array`, qui inclura l'archive jar comme une librairie externe, et adapter cette classe pour tester chacune des implémentations.

3.3 Rendus demandés

1. Les fichiers `Complex.java` et `TestComplex.java` modifiés (en conservant en commentaire les modifications successives de ces fichiers, les détails étant à décrire dans le compte-rendu).
2. l'application et les tests pour le triangle avec les packages `triangle` et `testTriangle` avec la classe `Triangle` et les classes JUnit de tests pour `typeTriangle` et `ReadData` (chaque classe contiendra des commentaires permettant de com- prendre ce qui est fait ou testé).
3. l'application et les tests pour les différentes implémentation de l'interface `search_Array_Interface` sous la forme d'un projet `test_search_Array` (chaque classe contiendra des commentaires permettant de comprendre ce qui est fait ou testé).
4. Le compte rendu donnera le résultat de vos tests avec une analyse des résultats et des constructions JUnit utilisées.