Faculté des Sciences
Aix*Marseille Université

Département Informatique et Interactions
Année 2017-2018

Master 2 FSIL
Fiabilité 1

TP Tests Unitaires : PARTIE I

Les environnements de tests unitaires existent pour la plupart des langages et plusieurs peuvent exister pour un même langage (CPPUnit, google test pour C et C++ par exemple). Dans ce TP vous allez mettre en oeuvre 3 environnements de tests pour 3 langages : Java, python et C.

La première partie concerne Java et JUnit et doit être rendue à la fin de la séance.

A savoir : un étudiant diplomé d'un master en informatique doit avoir des compétences sur la gestion, l'installation et le fonctionnement des systèmes d'exploitation. Il peut ne pas tout savoir, mais il doit pouvoir acquérir et maitriser l'information utile. Des sites classiques style Stackoverfow et le net en général contiennent la plupart des réponses aux problèmes courants. Acquérez cette compétence avant la fin de l'année si vous ne l'avez pas.

1 Principes généraux

Les environnements de tests unitaires permettent d'écrire des suites de tests selon le schéma présenté en cours.

1. partie initialisation et cloture de la suite de test (test fixture). Par exemple connection à une base de donnée, puis deconnection de celle-ci.
2. mise en place de la suite de test
3. pour chaque test
   (a) Initialisation et cloture pour chaque test,
   (b) calcul des données de test,
   (c) appel de la fonction testée sur les données,
   (d) utilisation d'une assertion pour vérifier que le résultat (value) est égal à la valeur attendue (expected).

L'exécution de la suite de test renvoie un rapport de test indicant les échecs (fail), succès (pass) des tests ou erreurs (error) déclenchées.

Un principe du test unitaire est d'écrire une suite de test pour chaque unité du programme (une classe ou une fonction). Un principe est de ne tester qu'une seule

propriété par test : si ce n'est pas le cas le test doit être réécrit en plusieurs tests, donc un seul assert par test et pas de condition booléenne compliqué.

2 JUnit : environnement de tests pour Java

Junit est un environnement de tests unitaires pour Java et l'IDE Eclipse contient usuellement un plugin JUnit dans sa distribution (sinon charger le plugin). La dernière version JUnit est la 5 mais la version d'Eclipse installée fonctionne avec JUnit4. Tout les TPs se feront avec JUnit4 (mais vous pouvez installer Eclipse Equinoxe et JUnit5 pour votre compte). Le site pour JUnit `http://junit.org` (suivre le lien indiqué pour JUnit 4).

2.1 Ecriture des tests

Une fois créé le squelette de la classe de test (via l'onglet du menu), il faut écrire les tests. Un test est une méthode précédée de l'annotation JUnit *@Test*. Un test utilisera les assertions de la classe *Assert* qui seront utilisées pour vérifier des égalités (de valeurs, objets, ...) et feront échouer le test si elles ne sont pas vraies.

```
@Test
public void testAdditionZero(){
double expected = 1.0;
Essai e = Essai(1.0);
e.ajouter(0.0);
double val = e.getVal();
Assert.assertTrue("Test 0 neutre", expected == val);
}
```

D'autres annotations JUnit permettent de gérer les initialisations et cas particuliers. Elles sont de la forme *@mot-clé*.

- *@BeforeClass* et *@AfterClass* permettent d'exécuter des instructions avant et après l'exécution de la suite de tests (test fixture).
- *@Before* permet de définir des initialisations à faire avant chaque test (typiquement définir un objet qui sera utilisé par tous les tests) et *@After* est similaire mais est effectué après chaque test.
- *@Ignore* permet de ne pas effectuer le test qui suit.
- *@Test(expected=MonException.class)* teste si la méthode déclenche bien une exception de la classe $MonException$.

— *@Test(timeout=val)* fera échouer le test quand le temps d'exécution dépasse la valeur *val* de timeout (donnée en millisecondes)

Ajouter des instructions d'impressions dans les annotations `Before` et `After` pour visuaiser l'exécution de ces parties de code dans la console. Voir la documentation en ligne pour un panorama complet des annotations.

2.2 Partie JUnit du TP

Lire la documentation !

2.2.1 Prise en main

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

1. Créer un projet *tptestUnitaires* et le paramétrer pour pouvoir utiliser *junit4*.
2. Dans un package *essai*, écrire une classe *Essai* ayant un attribut `val` de type `double`, les méthodes `getVal()` et `setVal(int )` et `void ajouter(double v)` qui ajoute v à *val*, et le constructeur *Essai(double)*.
3. Ecrire la classe de test *EssaiTest* avec une initialisation pour des objets *essai1, essai2* et des tests pour les méthodes *getVal()* et *setVal(int)* et le constructeur. Utiliser notamment *assertEquals* et *assertNotNull*.
4. Lancer JUnit et vérifier que les tests fonctionnent. Lire et comprendre les résultats dans la fenêtre JUnit. Ajouter un test qui échoue afin de voir la différence avec une suite de test dont tous les tests réussissent.
5. En rajoutant des instructions d'impression, vérifier que *@Before* et *@After* sont effectuées avant et après chaque test. Idem pour *@BeforeClass* et *@AfterClass*.
6. Ajouter dans *Essai* une méthode `double inverserVal()` qui renvoie *1/val* et lève une exception de type *IllegalArgumentException* pour $val == 0.0$. Ecrire les tests pour cette nouvelel méthode.

2.2.2 Suite de tests pour la fonction typeTriangle

Ecrire une suite de test pour l'application vue en cours pour le triangle. L'application sera une classe Java `Triangle` avec 3 attributs privés pour les cotés et deux méthodes `double [] readData(String)` et `int typeTriangle(double, double, double)` (notez qu'il serait plus classique d'avoir une méthode qui ne prend pas d'arguments explicites mais utilise les attributs privés).

Ecrire la classe `Triangle` avec la méthode `typeTriangle` et la suite de test JUnit `testTypeTriangle`. Notez que le cas où les trois valeurs sont positives mais

ne définissent pas un triangle doit se raffiner en plusieurs cas. La méthode `readData` ne sera pas encore écrite.

La spécification de `readData` est précisée ainsi : si le fichier n'existe pas, une exception est renvoyée et on pourra supposer que seuls les fichiers textes sont traités. Le fichier ne doit contenir qu'une ligne format cvs correspondant à trois valeurs de type double. Ecrire la suite de tests `testReadData` pour la méthode `readData`.

Utiles : méthode `split` de la classe `String`. Pour les entrées/sorties en Java, de nombreux sites expliquent avec des exemples, au hasard `http://thecodersbreakfast.net/index.php`

3 Rendu des TPs

3.1 Modalités

`A respecter absolument`

— Chaque groupe de TP rendra une archive au format zip appelée `TPTEST-PARTIEI-TPi.zip` (avec i le numéro du groupe) à déposer dans l'activité `Devoir Rendu TP TEST` du site AMETICE **à la fin du TP**.

— La décompression de l'archive créera un répertoire TPTEST-PARTIEI-TPi

— Chaque fichier source contiendra une `en-tete` qui est un commentaire avec les noms des membres du groupe et tout autre information jugée utile.
Exemple minimaliste d'en-tête :

```
/**
*@author : Etu1
*@author : Etu2
*
*/
```

ATTENTION : rendre uniquement une partie bien faite est bien mieux évalué que tout le TP baclé et mal fait.

3.2 Travail demandé

L'archive zip contiendra la classe `Triangle` et les classes JUnit pour `TypeTriangleTest` et `ReadDataTest` (chaque classe contiendra des commentaires permettant de comprendre ce qui est fait ou testé).