Université Abou Bekr Belkaid

Master 2 Génie Logiciel (2020/2021)

TP N^0 1

—

Validation et vérification.
Introduction à Junit

—

Introduction

JUnit est un framework de tests pour Java. Il permet de simplifier considérablement l'écriture des tests en offrant une panoplie d'outils. Il permet notamment d'écrire des jeux de tests facilement réutilisables, et de regrouper les tests en fonction des besoins, des objets, etc. JUnit est aussi un framework extensible : par exemple, DBUnit propose des outils supplémentaires pour les tests spécifiques aux bases de données.
Utiliser des tests unitaires permet non seulement de tester un programme et donc d'en assurer la qualité, mais également d'éviter les régressions dans le code. Avec JUnit, on définit des tests réutilisables que l'on peut exécuter à volonté. Lorsque le code a progressé, on peut relancer une série de tests. Si les tests se passent toujours bien, alors on peut considérer que le code n'a pas régressé.
Un des intérêts de JUnit est qu'il permet d'écrire les tests avant le code. En procédant ainsi, au début, le programme n'est pas livrable car il ne passe aucun test. Plus le code progresse et plus le nombre de tests passés augmente.
JUnit est un outil de gestion des tests unitaires pour les programmes Java, JUnit fait partie d'un cadre plus général pour le test unitaire des programmes, le modèle de conception (pattern) XUnit (CUnit, ObjcUnit, etc.).
JUnit offre :

- des primitives pour créer un test (assertions).
- des primitives pour gérer des suites de tests.
- des facilites pour l'exécution des tests.
- statistiques sur l'exécution des tests.
- interface graphique pour la couverture des tests.

Il existe un plug-in Eclipse / Netbeans pour JUnit. http://junit.org.

Principes de testes avec Junit

Dans les méthodes de test unitaire, les méthodes testées sont appelées et leur résultat est testé à l'aide des assertions :

- assertEquals(a,b) : Teste si a est égal à b (a et b sont soit des valeurs primitives, soit des objets possédant une méthode equals).
- assertTrue(a) et assertFalse(a) : Testent si a est vraie resp. fausse, avec a une expression booléenne.
- assertSame(a,b) et assertNotSame(a,b) : Testent si a et b réfèrent au même objet ou non.
- assertNull(a) et assertNotNull(a) : Testent si a est null ou non, avec a un objet.
- fail (message) : Si le test doit échouer (levée d'exception).

Exemple :

```
import junit.framework.*;
public class SimpleTest extends TestCase {
        public SimpleTest(String name) {
                super(name);
        }

        public void testSimpleTest() {
                int answer = 2;
                assertEquals((1+1), answer);
        }
}
```

Introduisez une erreur dans votre classe (par exemple, en changeant la valeur de la variable answer) et relancez le test. Que se passe-t-il ?

Quelques règles de bonne conduite avec JUnit

- Ecrire les tests en même temps que le code.
- Exécuter ses tests aussi souvent que possible, idéalement après chaque changement de code.
- Ecrire un test pour toute erreur signalée (même si elle est corrigée).
- Ne pas tester plusieurs méthodes dans un même test : JUnit s'arrête à la première erreur.
- Attention, les méthodes privées ne peuvent pas être testées.

Exerice 1 :

1 Écrire la classe **SommeArgent** dans le package **junit.monprojet** :

```
public class SommeArgent {
 private int quantite;
 private String unite;
```

```
public SommeArgent(int amount, String currency) {
 quantite = amount;
 unite = currency;
 }
public int getQuantite() {
return quantite;
}
public String getUnite() {
return unite;
}
public SommeArgent add(SommeArgent m) {
 return new SommeArgent(getQuantite()+m.getQuantite(), getUnite
      ());
}
}
```

Les objets de cette classe sont des quantités d'argent dans une certaine unité (ou monnaie).

2 Créer un package **junit.monprojet.test** et dans ce package une classe de tests JUnit 4.

3 On veut écrire une méthode de test qui construit deux sommes d'argent, en fait la somme et vérifie qu'elle est correcte. Pour cela, il faut enrichir la classe **SommeArgent** de la méthode **equals()** qui définit l'égalité (ou l'équivalence) de deux objets Deux sommes d'argent sont égales si elles sont de même unité et de même quantité. Écrire cette méthode **equals()** de la classe **SommeArgent**. Elle doit avoir pour signature :
public boolean equals(Object anObject).

4 Écrire la méthode de test qui construit deux sommes d'argent, en fait la somme et vérifie qu'elle est correcte. Le corps de cette méthode est :

```
SommeArgent m1= new SommeArgent(12, "DINARS");
SommeArgent m2= new SommeArgent(14, "DINARS");
SommeArgent expected = new SommeArgent(26, "DINARS");
SommeArgent result = m1.add(m2); // (2)
Assert.assertTrue(expected.equals(result));
```

5 Lancer les testes.

6 On a oublié de tester la méthode **equals()** de la classe **SommeArgent**. Écrire une méthode de test pour cette méthode **equals()**. Le corps de la méthode de test peut être :

```
SommeArgent m1= new SommeArgent(12, "DINARS");
SommeArgent m2= new SommeArgent(14, "DINARS");
SommeArgent m3= new SommeArgent(14, "EURO");
Assert.assertTrue(!m1.equals(null));
Assert.assertEquals(m1, m1);
```

```
Assert.assertEquals(m1, new SommeArgent(12, "
    DINARS")); // (1)
Assert.assertTrue(!m1.equals(m2));
Assert.assertTrue(!m3.equals(m2));
```

7 Que teste-t-on dans la dernière ligne de ce code ?

8 On veut regrouper ces diverses sommes d'argent dans un porte-monnaie et, pour cela construire une nouvelle classe **PorteMonnaie**. Une première version de cette classe peut être :

```
public class PorteMonnaie {
        HashMap<String,Integer> contenu;
        public HashMap<String,Integer> getContenu(){
                return contenu;
        }
        public PorteMonnaie(){
                contenu = new HashMap<String,Integer>();
        }

        public void ajouterSomme(sommeArgent sa){
                // a definir dans la question suivante.
        }
}
```

9 On veut ajouter dans cette classe la possibilité d'ajouter des sommes d'argent. Les spécifications du porte-monnaie sont :

- Si on ajoute 12 Euro et qu'il n'y a pas déjà d'euro dans le porte monnaie, cette somme est simplement mise. Si il y avait déjà 10 Euro, il y aura désormais 22 Euro.
- Coder cette spécification dans la méthode **ajouteSomme()**.

10 Ecrire une méthode de public String toString() qui affiche le contenu du porte-monnaie.

11 Construire, dans le package de test, une classe de test pour la classe **PorteMonnaie**, contenant deux méthodes de tests (au moins) qui vérifient la bonne cohérence de la méthode **ajouteSomme()** (cf. les spécifications du porte monnaie à la question 9°). On pourra par exemple écrire la méthode **public boolean equals(Object obj)** dans la classe **PorteMonnaie** qui retourne true si l'instance et le porte monnaie passé en paramètres ont les mêmes devises en même quantité.

Exercice 2 :

- Écrire un programme permettant de calculer toutes les racines carrées des nombres compris entre A et B, A et B étant deux nombres entiers tels que $A \leqslant B$.

- Écrire un programme permettant d'afficher une matrice de taille M×N remplie par des nombres aléatoires compris entre A et B. Les valeurs M, N, A et B doivent être passées en paramètres.
- Écrivez l'intégralité de la classe test. Cette classe doit comprendre :
  - Des assertions comme vu précédemment.
  - Des tests vérifiant que les exceptions sont bien levées quand elles doivent l'être (par exemple : de mauvais paramètres sont passés dans l'appel).
  - Des tests vérifiant que les boucles s'effectuent dans des temps raisonnables.
  - Formatez correctement les sorties de vos tests en utilisant les annotations @Before et @After.

Annexes

Remarques sur Eclipse :

- s'il vous manque des déclarations import (ou si vous en avez trop !), vous pouvez, dans Eclipse, les mettre par CTRL+MAJ+O. Ce sera le cas, entre autres, lorsque vous allez utiliser les annotations utiles à JUnit 4.
- pour indenter correctement tout votre programme taper CTRL A suivi de CTRL I.