Test Unitaires avec JUnit

1 Introduction

1.1 Les tests - Pourquoi, Comment, Quand

Tous les programmeurs savent qu'ils doivent écrire des tests, mais peu le font, ce qui donne un code moins stable. Pour tester un programme, il est possible d'utiliser des affichages à l'écran, d'utiliser un debugger, etc. mais ces solutions nécessitent un jugement humain et sont limitées.

Le test d'un programme sert principalement à gagner du temps car il limite la présence d'erreurs. Les tests de non-régression par exemple permettent de vérifier que les méthodes qui étaient fonctionnelles avant une modification le sont toujours après.

Pour cela on automatise les tests et on systématise leur exécution même après une modification mineure. Les tests sont primordiaux dans les méthodes AGILE (par exemple eXtreme Programming) où en TDD (Test Driven Development).

Le développement piloté par les tests est une technique de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel. Il permet d'affiner l'analyse, les tests sont des use cases au sens d'UML. Ils vont permettre de modéliser les pré et les post conditions des algorithmes, d'éviter d'écrire du code inutile et sont une forme de documentation technique. De plus le test est un exemple d'utilisation de la classe qu'on vient d'écrire.

1.2 Les tests en Java : JUnit

JUnit est un outil Open Source servant à écrire des suites de tests de façon simple et systématique et d'en piloter la mise en œuvre. Pour plus de détails sur JUnit, vous pouvez consulter `www.junit.org`. La javadoc se trouve à `http://junit.org/javadoc/latest/`. Les outils de test comme JUnit sont mis en œuvre pour tester des modules de programme de façon répétée.

Au départ, JUnit n'était qu'une série de classes mais a ensuite été intégré complètement divers environnements de développement. Les classes de test JUnit suivent le principe suivant :

- Une classe de test hérite de junit.framework.Assert.*
- Dedans sont écrites des méthodes de test sur une autre classe
  - Elles portent le nom testXXXX()
  - Ou utilisent l'annotation @Test (jUnit 4.x)
  - Elle mettent en œuvre les assertions

Ensuite l'ensemble des tests de la classe est lancé en utilisant une suite de test.

2 Utilisation de JUnit

JUnit permet d'effectuer des tests automatisés sans intervention humaine pour les interpréter.

Terminologie

- **Test unitaire (Unit test)** : test d'une classe,
- **Cas de test (Test Case)** : teste les réponses d'une méthode à un ensemble particulier d'entrées,
- **Suite de tests (Test suite)** : collection de cas de tests,
- **Testeur (Test Runner)** : programme qui exécute des tests et rapporte les résultats.

2.1 Création des premiers tests

Télécharger les code à tester sur le serveur pédagogique (classes `Vectors.java` et `Utils.java`) et mettez-les dans un package `sample` d'un nouveau projet.
Pour créer une classe de test, utilisez `Tools>Create/Update Test` sur la classe `Vectors.java`. le framework `JUnit` dans la liste et décochez les cases `Test Initializer` et `Test Finalizer` puis OK. Ensuite choisissez la version 4 de JUnit.
Par défaut, JUnit 4 génère les méthodes "initializer" et "finalizer" avec les annotation @BeforeClass et @AfterClass. Cela signifie que ce seront des méthodes lancées avant et après toutes les méthodes de tests de la classe. Elles peuvent être utiles pour mettre en place un environnement de test spécifique et revenir à un état initial. Par exemple, au lieu de créer une connexion de base de données dans un initialiseur de test et de créer une nouvelle connexion avant chaque méthode de test, vous pouvez utiliser un initialiseur de classe de test pour ouvrir une connexion avant d'exécuter les tests. Vous pouvez ensuite fermer la connexion avec le finaliseur de classe de test.
Lancez le test par le menu contextuel `Run File`.
Modifiez la méthode `testScalarMultiplication()` par :

```
@Test
public void testScalarMultiplication() {
    System.out.println("scalarMultiplication");
    assertEquals(  0, Vectors.scalarMultiplication(new int[] { 0, 0}, new int[] { 0, 0}));
    assertEquals( 39, Vectors.scalarMultiplication(new int[] { 3, 4}, new int[] { 5, 6}));
    assertEquals(-39, Vectors.scalarMultiplication(new int[] {-3, 4}, new int[] { 5,-6}));
    assertEquals(  0, Vectors.scalarMultiplication(new int[] { 5, 9}, new int[] {-9, 5}));
    assertEquals(100, Vectors.scalarMultiplication(new int[] { 6, 8}, new int[] { 6, 8}));
}
```

puis la méthode `testEquals()` par :

```
@Test
public void testEquals() {
    System.out.println("equal");
    assertTrue(Vectors.equal(new int[] {}, new int[] {}));
    assertTrue(Vectors.equal(new int[] {0}, new int[] {0}));
    assertTrue(Vectors.equal(new int[] {0, 0}, new int[] {0, 0}));
    assertTrue(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 0}));
    assertTrue(Vectors.equal(new int[] {5, 6, 7}, new int[] {5, 6, 7}));
```

```
    assertFalse(Vectors.equal(new int[] {}, new int[] {0}));
    assertFalse(Vectors.equal(new int[] {0}, new int[] {0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0, 0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0}));
    assertFalse(Vectors.equal(new int[] {0}, new int[] {}));

    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 1}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 1, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {1, 0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0, 1}, new int[] {0, 0, 3}));
}
```

Exécutez ces tests.
Créez maintenant les tests pour la classe `Utils.java`.

2.2 "Initialiseur" et "Finaliseur" de test

"Initialiseur" de test L'annotation `@Before` marque une méthode comme une méthode d'initialisation de test. L'"initialiseur" de test est exécuté avant chaque cas de test dans la classe de test. Une méthode d'initialisation de test n'est pas forcément nécessaire pour exécuter des tests, mais si vous avez besoin d'initialiser certaines variables avant d'exécuter un test, ce type de méthode est approprié.

"Finaliseur" de test L'annotation `@After` marque une méthode comme une méthode de finalisation de test. Une méthode de finalisation de test est exécuté après chaque cas de test dans la classe de test. Une méthode de finalisation de test n'est pas nécessaire pour des tests, mais vous pouvez avoir besoin d'un "finaliseur" pour nettoyer toutes les données qui ont été nécessaire lors de l'exécution des cas de test.
Modifiez et écrivez les méthodes suivantes :

```
@BeforeClass
public static void setUpClass() throws Exception {
    System.out.println("* UtilsJUnit4Test: @BeforeClass method");
}

@AfterClass
public static void tearDownClass() throws Exception {
    System.out.println("* UtilsJUnit4Test: @AfterClass method");
}

@Before
public void setUp() {
    System.out.println("* UtilsJUnit4Test: @Before method");
}

@After
public void tearDown() {
    System.out.println("* UtilsJUnit4Test: @After method");
}
```

2.3 Test sur des assertions simples

Modifiez la méthode `testConcatWords` par :

```
@Test
public void testConcatWords() {
System.out.println("concatWords");
    assertEquals("Hello, world!", Utils.concatWords(new String[]{"Hello", ", ", "world", "!"}));
}
```

2.4 Test sur un compteur de temps

L'annotation `@Test(timeout=1000)` permet de tester que la méthode met moins de 1000 millisecondes à s'exécuter.

```
  @Test(timeout=1000)
    public void testComputeFactorial() {
        System.out.println("computeFactorial");
        final int factorialOf = 1 + (int) (30000 * Math.random());
        System.out.println("computing " + factorialOf + "!");
        System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));
    }
```

2.5 Test de levée d'exception

L'annotation `@Test(expected=IllegalArgumentException.class)` permet de vérifier que la méthode de test renvoie bien une instance de la classe `IllegalArgumentException`. Testez que la méthode suivante renvoie bien une exception :

```
@Test (expected=IllegalArgumentException.class)
public void checkExpectedException() {
    System.out.println("checkExpectedException");
    final int factorialOf = -5;
    System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf));
}
```

2.6 Désactiver un Test

Pour désactiver un test, il suffit d'ajouter l'annotation `@Ignore` au dessus de l'annotation `@Test` du test a désactiver.

2.7 Simuler une entrée utilisateur

Simuler une entrée utilisateur peut être utile pour créer des test et les exécuter. la méthode `java.lang.System.setIn()` réassigne le flux d'entrée standard.

```
/**
*  source : https://www.tutorialspoint.com/java/lang/system_setin.htm
*/
public class UserInput {
```

```
public static void main(String[] args) throws Exception {

// existing file
System.setIn(new FileInputStream("file.txt"));

// read the first character in the file
char ret = (char)System.in.read();

// returns the first character
System.out.println(ret);
}
}
```

Si le fichier contient "test de user input" la sortie sera simplement "t".

2.8 Création d'une suite de tests

Pour créer une suite de test avec JUnit 4, il faut générer un nouveau fichier de type `Unit Test` et choisir `Test Suite` a générer dans le même package `sample` que les fichiers de test précédents.

1. Clic droite sur le projet et choisir `New > Other`
2. Sélectionner `Test Suite` dans la catégorie `Unit Tests`. Ensuite clic sur `Next`.
3. Écrire le nom du fichier
4. Sélectionner le package `sample` pour créer la suite de tests dans le dossier `sample` du dossier des packages de test
5. Ne pas sélectionner "Test Initializer" et "Test Finalizer". Clic "Finish".

Le code

```
@RunWith(Suite.class)
@Suite.SuiteClasses({sample.UtilsTest.class, sample.VectorsTest.class})
```

indique quelles classes de test seront lancées.

3 Mise en œuvre

3.1 Traitement d'images

Terminez et tester vos classes du TP de traitement d'images.

3.2 Jeu de dames

Implémentez avec une **interface texte** (aucune interface graphique n'est demandée) un jeu de dame à deux joueurs. Les règles du jeu de dame sont disponibles ici : `https://fr.wikipedia.org/wiki/Dames`.

Vous devrez en particulier :

— Permettre à chaque joueur de jouer alternativement

— Déplacer un pion

— Transformer un pion en dame
— Capturer un pion ou une dame
— Permettre la prise multiple pour une dame
— Enregistrer une partie en cours et ouvrir une partie enregistrée

Toutes les fonctions doivent être testées.