STP JUnit – Le test

Initiation au framework JUnit

Modélisation et validation des logiciels – SIT211

Objectifs

Cette séance de travaux pratiques à deux objectifs :

- Découvrir un outil d'automatisation de l'exécution de tests unitaires.
- Découvrir le mode de développement logiciel dirigé par les tests (« *test-driven software development* »).

Préambule

Les détails pratiques fournis dans cet énoncé correspondent à l'environnement technique prévu pour vos TP, à savoir la machine virtuelle Ubuntu14_BD_Prog.

Exercice 1 (*Tester une méthode*)

Vous avez déjà une petite expérience de programmation en Java. Voyons en préambule comment réaliser le test « *boite noire* » d'une méthode, c'est-à-dire sans s'intéresser au code qui la compose.

▷ **Donnez un code Java qui affiche un message d'erreur si le résultat d'un appel à une méthode est différent de la valeur entière 4. Peut-on faire de même pour une méthode qui renvoie une valeur réelle et non pas entière ? Qu'en est-il des principaux types primitifs que vous connaissez ? Comment tester qu'une chaîne de caractères retournée par une méthode (au hasard : `toString`) « ressemble » à une chaîne connue ?**

▷ **Comment vérifier que la méthode gère correctement les différentes exceptions prévues ?**

Découverte de JUnit

JUnit [1] est un cadriciel (« *framework* »), c'est-à-dire un ensemble de classes, visant l'automatisation des tests unitaires en Java. Le principe de base de l'utilisation de cet outil est très simple : on définit un ensemble de tests, l'outil enchaîne automatiquement tous ces tests et met en évidence les tests qui ont échoué. Le test d'une méthode consiste typiquement à comparer son résultat à la valeur attendue (constante ou calculée).

L'usage veut que les tests ne soient pas mélangés avec le code et qu'ils soient définis dans des classes spécifiques. Des conventions de nommage peuvent être adoptées pour bien distinguer les tests du code. Par exemple, pour chaque classe à tester, on définit une classe de test correspondante, dont le nom est celui de la classe à tester suffixé par « `Test` » [2]. Ainsi, la classe de test d'une classe `A` se nommera `ATest`.

Cette classe de test pourra contenir autant de tests que nécessaire sous la forme de méthodes publiques, non statiques, sans paramètre et qui sont annotées par `@Test`. Dans le cas où la méthode de test est censée lever une exception, cette annotation est assortie de la déclaration de l'exception attendue (cf la méthode `testCreation` ci-après).

On donne, en exemple, les sources d'une classe `Compteur` et d'une classe `CompteurTest` correspondante. Le projet Eclipse qui contient ces sources est disponible sur Moodle (`https://formations.telecom-bretagne.eu/fad/course/view.php?id=23429`). Télécharger et décompresser ce fichier.

Importer ensuite le projet en utilisant le menu *File -> Import -> General -> Existing projets into workspace* et choisir tpJUnit.

```
package junit.compteur;

public class CompteurInvalide extends Exception {}
```

```
package junit.compteur;

public class Compteur {
  private int val;

  public Compteur(int a) throws CompteurInvalide {
    if (a < 0)
      throw new CompteurInvalide();
    else
      this.val = a;
  }

  public void ajouterVal(int a) {
    this.val = this.val + a;
  }

  public int getVal() {
    return this.val;
```

1. Le code de JUnit est téléchargeable sur le site `www.junit.org` sous forme d'une archive zip qui contient entre autre `junit.jar`. Ce TP a été testé avec la version 4.8.1 de JUnit.
2. Dans les versions 3.x de JUnit cette convention était obligatoire mais elle ne l'est plus. De plus, les méthodes de test devaient avoir un nom qui commençait par test. Bien que ce ne soit plus nécessaire, ces coutumes restent assez largement respectées, notamment pour améliorer la lisibilité.

```
  }
}
```

```
package junit.compteur;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

/**
 * @author Fabien Dagnat <fabien.dagnat@telecom-bretagne.eu>
 */
public class CompteurTest {

  /**
   * Test method for {@link junit.compteur.Compteur#ajouterVal(int)}.
   *
   * @throws CompteurInvalide
   */
  @Test
  public final void testAjouterVal() throws CompteurInvalide {
    Compteur c1;
    c1 = new Compteur(5);
    c1.ajouterVal(4);
    // premiere assertion de test :
    // si la valeur du compteur est differente de 9, on provoque un echec du test :
    assertEquals(9, c1.getVal());
    //
    c1.ajouterVal(3);
    // deuxieme assertion de test
    // on utilise ici la possibilite d'adjoindre un message d'erreur :
    assertEquals("Le compteur devrait valoir 12", 12, c1.getVal());
  }

  /**
   * Test method for {@link junit.compteur.Compteur#Compteur(int)}. Le test, ici, attend que le
   * constructeur leve l'exception precisee dans l'annotation
   *
   * @throws CompteurInvalide
   */
  @Test(expected = CompteurInvalide.class)
  public final void testCreation() throws CompteurInvalide {
    new Compteur(-2);
  }
}
```

Exercice 2 (*Exécution d'une classe de test*)

JUnit est intégré dans Eclipse. Son principe d'utilisation est le suivant :

— Comme vous le verrez tout à l'heure, l'intégration de JUnit dans Eclipse facilite l'élaboration

de la classe de test.
- Pour ce qui est de son exécution, il suffit de sélectionner la classe de test et de lancer (par le menu contextuel) l'exécution (menu *Run as -> JUnit test*). Une fenêtre s'affiche alors vous présentant les résultats de votre test (voir figure 1).
- Pour relancer un test, il suffit d'utiliser le bouton de lancement (rond vert) de cette fenêtre de résultat.

▷ **Exécutez le test de la classe `Compteur`. Modifiez les méthodes ou les valeurs attendues pour voir les différents comportements.**

Remarque :

Il est évidemment possible de lancer l'exécution des tests JUnit indépendamment d'Eclipse. Pour cela, il faut que les classes de la bibliothèque JUnit soient accessibles au compilateur et à la machine virtuelle (par le `CLASSPATH`). Une fois compilée, une classe de test peut être exécutée en version non graphique. Il faut utiliser la commande suivante :

`java org.junit.runner.JUnitCore` *ClasseTest*

N'hésitez pas à visiter le site `www.junit.org` qui contient de nombreux documents pédagogiques sur le test en général et sur l'utilisation de JUnit en particulier.

Exercice 3 (*Le test des triangles*)

Le projet importé précédemment contient également une classe `Triangle`, que vous allez tester en utilisant JUnit. L'API de cette classe vous est fournie en annexe.

Voici la signature des méthodes publiques de cette classe :

```
public Triangle(int cote1, int cote2, int cote3) throws TriangleInvalide;
public boolean estIsocele();
public boolean estEquilateral();
public boolean estRectangle();
public String toString();
public static boolean estUnTriangle(int cote1,int cote2,int cote3);
```

Sous Eclipse, la création des classes de test est facilitée car on peut en générer automatiquement la structure (squelette). Pour cela, on sélectionne la classe à tester et, à l'aide du menu contextuel (bouton droite de la souris), on sélectionne *New -> JUnit Test Case*. Il faut alors choisir la version de JUnit souhaitée - nous utiliserons la v4 - et ensuite sélectionner les méthodes à tester [3]. Eclipse génère alors la classe de test avec toutes les méthodes de test nécessaires. Bien évidemment, ces méthodes sont « vides » et c'est à vous de les compléter avec vos cas de test.

▷ **Élaborez la version « vide » de votre classe de test `TriangleTest` et rajoutez-y 2 ou 3 cas de test. Lancez le test. Quel est le verdict ?**

Votre objectif est de tester la classe `Triangle` jusqu'à atteindre un degré de confiance suffisant pour envisager de la réutiliser dans vos futurs programmes. Il vous faut pour cela trouver des cas de test pertinents, c'est-à-dire qui permettront de détecter le plus d'erreurs potentielles.

▷ **En considérant l'API de la classe `Triangle`, trouvez 5 cas de test judicieux et rajoutez-les à votre programme de test. Quel est le verdict ?**

3. Selon votre configuration Eclipse, une fenêtre pourrait vous proposer d'ajouter la bibliothèque JUnit v4 dans votre projet. Répondez oui à la question si c'est le cas.

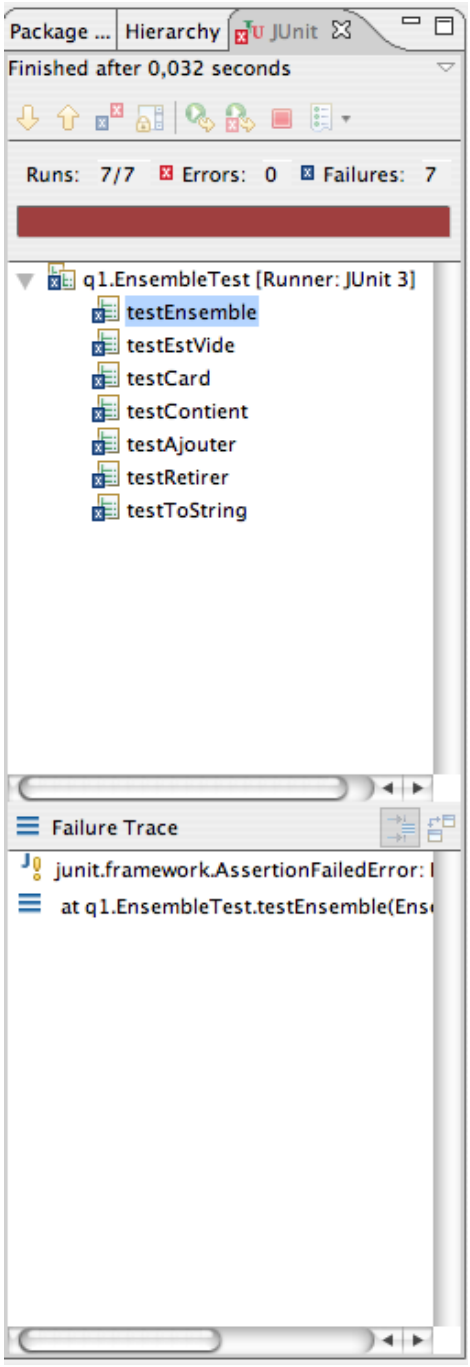

FIGURE 1 – La fenêtre de résultat d'Eclipse

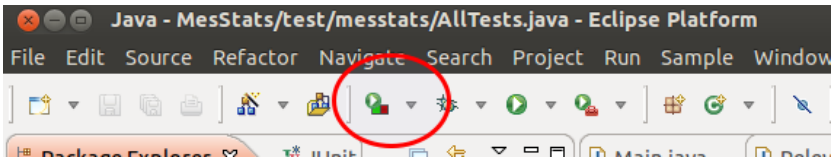

FIGURE 2 – Le bouton d'utilisation d'Emma dans Eclipse

Exercice 4 (*Test boîte blanche et couverture de test*)

Les cas de test que vous avez choisis dépendent de votre perception de l'API de la classe `Triangle` [4]. On est typiquement dans une approche de « *test boîte noire* ». Dans le cas où on a accès à l'intérieur de la boîte à tester (typiquement ici, le code source du composant à tester), on peut choisir des cas de test sur la base de la connaissance de la structure et du fonctionnement de cette boîte : c'est du « *test boîte blanche* ». C'est ce que vous allez faire maintenant, en vous appuyant sur une information propre au logiciel, la « *couverture de test* ». Des outils permettent en effet d'identifier les portions de code non mobilisé lors de l'exécution des tests, et donc des parties non testées du code. Ceci aide à trouver de nouveaux cas de test pertinents, qui vont améliorer la couverture de test. C'est dans cette démarche que vous allez maintenant travailler.

C'est le plugin Emma qui va calculer les couvertures de test. Sa présence dans Eclipse est matérialisée - dans la « *perspective d'affichage* » Java - par la présence à côté du bouton de debug d'un bouton spécifique de lancement d'exécution, avec un rectangle vert et rouge (cf figure 2). Si c'est le cas de votre configuration, il est évidemment inutile de procéder à l'installation détaillée ci-après.

Dans le cas contraire, il faut au préalable activer l'accès Internet de votre machine virtuelle pour pouvoir récupérer le plugin en ligne. Pour cela, le plus simple est de lancer le navigateur Web et d'accéder à n'importe quelle page Web : vos identifiants école vous seront alors demandés pour lancer une session d'accès Internet sous votre nom [5] . Dès lors, vous pouvez effectuer la procédure d'installation du plugin Emma :

- Help - Install New Software - Add
- Donnez un nom évocateur (par ex : EclEmma) et l'URL où récupérer le plugin : `http://update.eclemma.org`
- Faîtes OK et installez le plugin
- Acceptez la licence et redémarrez Eclipse
- Vérifiez que le bouton de lancement Emma apparaît bien dans la barre d'outils (cf figure 2)

Vous pouvez maintenant demander à Emma de générer toutes les informations concernant la couverture de votre test. Pour cela, vous lancez l'exécution de votre classe de test avec le bouton Emma. Observez la coloration syntaxique de votre code : en vert les lignes de code qui ont été exécutées, en rouge celles qui n'ont pas été exécutées, et en jaune celles qui ont été partiellement exécutées.

Emma vous donne également un récapitulatif (nombre et pourcentage d'instructions couvertes) dans la fenêtre Coverage (cf figure 3).

▷ **Quelle est le taux de couverture annoncé globalement pour votre projet ?**

4. D'ailleurs vos voisins n'ont probablement pas choisis les mêmes cas de test que vous.
5. Evidemment, vous n'autoriserez pas le navigateur à enregistrer ces informations puisque n'importe qui pourra utiliser cette machine virtuelle !

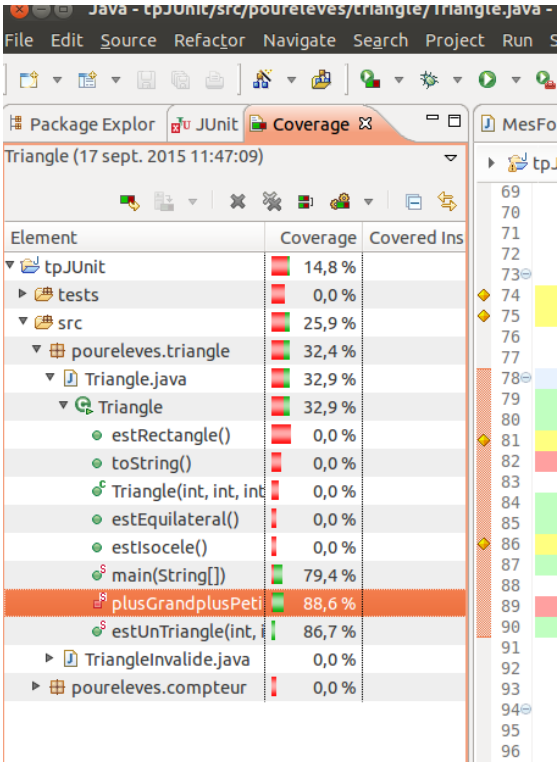

FIGURE 3 – Le bilan de couverture affiché par Emma

Notez comme, par défaut, toutes les lignes de code exécutables sont prises en considération dans les calculs, alors que vous pouvez préférer vous focaliser plus spécialement sur certaines parties du code [6].

▷ **Via l'*Explorer*, affichez les propriétés du fichier `Triangle.java`, et choisissez le détail de couverture (Coverage). Quelles sont les informations disponibles ?**

▷ **Quelles sont les lignes de code de `Triangle` non couvertes par votre test ? Rajoutez progressivement des cas à votre code de test jusqu'à obtenir une couverture de 100% sur les parties que vous jugez importantes. Cela vous permet-il de détecter de nouvelles erreurs dans le code de `Triangle` ? Une couverture de test de 100% garantit-elle un code sans erreur ?**

Plus loin avec JUnit

Nous venons de découvrir et d'utiliser les fonctionnalités de base de JUnit. Il en existe de nombreuses autres. Nous vous proposons d'en découvrir quelques unes.

Exercice 5 (*Les annotations `@Before` et `@After`*)

Avant chaque exécution d'une méthode de test, JUnit exécute les méthodes annotées par `@Before`. On peut donc utiliser ces méthodes pour (ré)initialiser des valeurs si cela s'avère nécessaire. La

6. *Emma - Coverage configurations...* permet notamment de sélectionner les packages pris en compte dans les calcul de couverture

classe de test est en effet une classe comme les autres et peut donc définir et utiliser des attributs. On pourra ainsi définir des attributs pour des données souvent utilisées dans les différentes méthodes de tests.

On peut, par exemple, ajouter à la classe `CompteurTest.java` un attribut `valeur` de type `Compteur` et définir une méthode `setUp()` :

```
@Before
protected void setUp() throws Exception {
  super.setUp();
  valeur = new Compteur(5);
}
```

Ainsi, le compteur `valeur` sera réinitialisé à 5 avant l'appel de chaque méthode de test, ce qui évite d'avoir à le faire au début de chacune d'elles.

De la même façon, les méthodes déclarées `@After` seront exécutées après chaque exécution d'une méthode de test. Ceci permet surtout de libérer des ressources.

▷ **Pouvez-vous alléger votre code de test avec une méthode `@Before` ?**

Exercice 6 (*Utilisation des Rules*)

Par défaut, JUnit comptabilise chaque méthode de test comme un seul et unique test. Si celle-ci enchaîne plusieurs `assert`, il suffit que l'un d'eux échoue pour JUnit considère que le test a échoué. L'éventuel message d'erreur associé est affiché. Mais les `assert` suivants ne sont même pas exécutés : cela empêche la mise en évidence rapide de tous les bugs détectables par le code de test.

On peut amener JUnit à exécuter tous les cas de test et à signaler toutes les erreurs détectées en une seule fois en utilisant les `ErrorCollector` *Rules*.

▷ **Consultez quelques ressources explicatives sur le Web et essayez de réécrire votre code de test en utilisant cette possibilité.**

Annexes

L'API de la classe `Triangle`

Class Triangle

La classe `Triangle` construit un objet triangle sur la base d'un triplet d'entiers censés représenter la longueur de ses côtés. Elle lève une exception si ce triplet est incompatible avec la définition d'un triangle.

NB : concernant les cas limites, la convention choisie est qu'un triangle peut se limiter à un simple point (3 côtés de longueur nulle).

La classe offre plusieurs méthodes destinées à caractériser le triangle : isocèle, équilatéral, rectangle.

Constructor Summary

`Triangle(int a, int b, int c)`
Le constructeur prend en argument les valeurs respectives des côtés du triangle.

Method Summary

`boolean`	`estEquilateral()`
`boolean`	`estIsocele()`
`boolean`	`estRectangle()`
`static boolean`	`estUnTriangle(int cote1, int cote2, int cote3)` Permet de tester un triplet d'entiers a priori
`String`	`toString()` Rend une chaîne de caractères décrivant le triangle : longueur des côtés + propriétés particulières

Constructor Detail

`public Triangle(int a, int b, int c) throws TriangleInvalide`

le constructeur prend en argument les valeurs respectives des côtés du triangle

Parameters :

`a` - la longueur d'un des 3 côtés du triangle

`b` - la longueur d'un des 3 côtés du triangle

`c` - la longueur d'un des 3 côtés du triangle

Throws :

`TriangleInvalide` - triplet d'entiers passé en argument incompatible avec le concept de triangle

Method Detail

`public boolean estIsocele()`

Returns : vrai si le triangle est isocèle

`public boolean estEquilateral()`

Returns : vrai si le triangle est équilatéral

`public boolean estRectangle()`

Returns : vrai si le triangle est rectangle

`public String toString()`

Rend une chaîne de caractères décrivant le triangle : longueur des côtés + propriétés particulières

`public static boolean estUnTriangle(int cote1, int cote2, int cote3)`

Permet de tester un triplet d'entiers a priori

Returns : vrai si le triplet d'entiers passé en argument est compatible avec la définition d'un triangle

La classe `org.junit.Assert`

Class org.junit.Assert

A set of assertion methods useful for writing tests. Only failed assertions are recorded. These methods can be used directly : `Assert.assertEquals(...)`, however, they read better if they are referenced through static import :

```
import static org.junit.Assert.*;
...
assertEquals(...);
```

Method Summary	
`static void`	`assertArrayEquals(byte[] expecteds,byte[] actuals)` Asserts that two byte arrays are equal.
`static void`	`assertArrayEquals(char[] expecteds,char[] actuals)` Asserts that two char arrays are equal.
`static void`	`assertArrayEquals(int[] expecteds,int[] actuals)` Asserts that two int arrays are equal.
`static void`	`assertArrayEquals(long[] expecteds,long[] actuals)` Asserts that two long arrays are equal.
`static void`	`assertArrayEquals(Object[] expecteds,Object[] actuals)` Asserts that two object arrays are equal.
`static void`	`assertArrayEquals(short[] expecteds,short[] actuals)` Asserts that two short arrays are equal.
`static void`	`assertArrayEquals(String message,byte[] expecteds,byte[] actuals)` Asserts that two byte arrays are equal.
`static void`	`assertArrayEquals(String message,char[] expecteds,char[] actuals)` Asserts that two char arrays are equal.
`static void`	`assertArrayEquals(String message,int[] expecteds,int[] actuals)` Asserts that two int arrays are equal.
`static void`	`assertArrayEquals(String message,long[] expecteds,long[] actuals)` Asserts that two long arrays are equal.
`static void`	`assertArrayEquals(String message,Object[] expecteds,Object[] actuals)` Asserts that two object arrays are equal.
`static void`	`assertArrayEquals(String message,short[] expecteds,short[] actuals)` Asserts that two short arrays are equal.
`static void`	`assertEquals(double expected,double actual,double delta)` Asserts that two doubles or floats are equal to within a positive delta.
`static void`	`assertEquals(long expected,long actual)` Asserts that two longs are equal.
`static void`	`assertEquals(Object expected,Object actual)` Asserts that two objects are equal.

`static void`	`assertEquals(String message,double expected,double actual,double delta)` Asserts that two doubles or floats are equal to within a positive delta.
`static void`	`assertEquals(String message,long expected,long actual)` Asserts that two longs are equal.
`static void`	`assertEquals(String message,Object expected,Object actual)` Asserts that two objects are equal.
`static void`	`assertFalse(boolean condition)` Asserts that a condition is false.
`static void`	`assertFalse(String message,boolean condition)` Asserts that a condition is false.
`static void`	`assertNotNull(Object object)` Asserts that an object isn't null.
`static void`	`assertNotNull(String message,Object object)` Asserts that an object isn't null.
`static void`	`assertNotSame(Object unexpected,Object actual)` Asserts that two objects do not refer to the same object.
`static void`	`assertNotSame(String message,Object unexpected,Object actual)` Asserts that two objects do not refer to the same object.
`static void`	`assertNull(Object object)` Asserts that an object is null.
`static void`	`assertNull(String message,Object object)` Asserts that an object is null.
`static void`	`assertSame(Object expected,Object actual)` Asserts that two objects refer to the same object.
`static void`	`assertSame(String message,Object expected,Object actual)` Asserts that two objects refer to the same object.
`static <T> void`	`assertThat(String reason,T actual,org.hamcrest.Matcher<T> matcher)` Asserts that `actual` satisfies the condition specified by `matcher`.
`static <T> void`	`assertThat(T actual,org.hamcrest.Matcher<T> matcher)` Asserts that `actual` satisfies the condition specified by `matcher`.
`static void`	`assertTrue(boolean condition)` Asserts that a condition is true.
`static void`	`assertTrue(String message,boolean condition)` Asserts that a condition is true.
`static void`	`fail()` Fails a test with no message.
`static void`	`fail(String message)` Fails a test with the given message.