Tests unitaires xUnit

Contenu

Tests unitaires xUnit

Objectifs :

- S'assurer que chaque module fonctionne selon les spécifications du système.
- Vérifier que l'unité (fonction, méthode, classe, module, interface, code assigné à un individu) fonctionne correctement et répond aux spécifications.
- Les tests unitaires vérifient chaque donnée du jeu d'essai.

Quand débuter les jeux d'essais?

- Durant le développement

Quand exécuter les essais ?

- Plan d'essais est terminé
- L'unité est développée

Qui ?

- Développeur

Comment ?

- Outils? – XUnit

Critères de fin ?

- Jeux d'essais ont passé avec succès.
- Bonne couverture de code.

Quoi vérifier

- Valeurs basées sur le typage
- Valeurs extrêmes (limite & limite ±1)
- Valeurs spéciales (espace, rien, caractères spéciaux, etc.)
- Mélange de données valides et non-valides
- Couverture de code - Essais Structurels
- Fonctionnalité vs Spécifications
- Interface usager (unitaire)
- Fiabilité du code.

Les outils xUnit

Tout au long du processus de développement, il faut veiller à une chose : parvenir à finaliser le programme, en répondant au cahier des charges fixé par le client. JUnit fait partie de ces outils qui aident le programmeur dans ce sens.

Cet outil est un ensemble de classes (framework) développées pour écrire des tests unitaires. Il a été écrit par Érich Gamma et Kent Beck (il a défini XP et écrit le premier livre à ce sujet) et adapté pour la presque totalité des langages. Cet outil est traditionnellement nommée selon le schéma ***x*Unit** où *x* est une lettre ou une syllabe rappelant le nom du langage. On connaît donc des implémentations en Java (*JUnit*), en C++ (*CppUnit*) en Delphi (*Dunit*) en VB (*VBUnit*) en SmallTalk (*SUnit*), .NET (*NUnit*), etc.

Un principe fondamental des tests unitaires est que les tests doivent être écrits dans le même langage que celui utilisé pour l'application qu'on est en train de programmer. C'est cette

exigence, en particulier, qui a conduit à l'émergence de *x*Unit comme un produit *open source*, facile à adapter vers d'autres langages.

JUnit se compose de quelques classes qui encapsulent les concepts fondamentaux : TestCase qui représente un scénario de tests et TestSuite qui permet l'agrégation de ces scénarios en méta-scénarios. Ces deux interfaces sont les plus visibles auxquelles s'ajoutent TestResult et TestRunner qui facilitent notamment l'ajout d'extensions telles qu'une interface graphique. Avec la version 4, on utilise la librairie « org » au lieu de « junit ».

JUnit est disponible à l'adresse « **www.junit.org** », à partir des environnements de développement Eclipse, JUnit est automatiquement intégré. La dernière version stable est 4.12, celle installée en laboratoire est la 4.xx. On peut effectuer des tests individuellement ou en suite. Un test *JUnit* passe ou échoue. Pour effectuer un test, il suffit de créer une sous-classe de *junit.framework.TestCase*. Chaque test unitaire est représenté par une méthode **testXXX( ).**

Structure des tests unitaires

Les tests unitaires répondent en général à une structure des « **3A** » : Acteurs, Action, Assertion.

La **première partie** consiste à instancier un ou plusieurs objets, éventuellement à les initialiser d'une façon bien précise, comme si on plaçait des acteurs sur une scène.

Dans la **partie suivante**, on appelle en général une seule méthode d'un de ces objets, ce qui déclenche une ou plusieurs actions bien précises.

Enfin, presque tous les tests se terminent par une série d'assertions, qui vérifie que, compte tenu des conditions initiales, l'action demandée aux objets a bien eu l'effet escompté.

```
public void testFib1(){
  FibTest test = new FibTest();      //Acteur
  int result = test.fib(0);          //Action
  assertEquals(0,result);            //Assertion
}
```

Les méthodes assertXXX()

Méthode	Description

assertThat([value], [matcher statement]);	Meilleure lisibilité. Version 4
	Le test passe si l'assertion est vraie.
	
	assertThat(x, is(3));
	
	assertThat(x, is(not(4)));
	
	assertThat(responseString, either(containsString("color")).or(containsString("colour")));
	
	assertThat(myList, hasItem("3"));

assertEquals( )	Compare l'égalité de deux valeurs. Le test passe si les deux sont égales. assertEquals(a,b)

assertFalse( )	Le résultat de l'expression doit être faux pour que le test passe. assertFalse(maPile.isEmpty())
assertTrue( )	Évalue l'expression booléenne, Le test passe si l'expression est vraie
assertNotNull( )	Vérifie que la référence d'un objet n'est pas « null ». assertNotNull(obj)
assertNull( )	Vérifie que la référence d'un objet est null.
assertArrayEquals()	Vérifie que deux vecteurs sont égaux.
fail( )	Fait échouer le test, utile avec les exceptions ou pour des tests non terminés pour le test.

Convention des noms de classes de tests

En préfixe ou en suffixe. Le suffixe est intéressant pour le repérage rapide. Par exemple : la classe Vecteur et sa classe de test VecteurTest. Le préfixe permet de regrouper tous les tests ensemble, c'est le choix d'Eclipse.

Les exceptions

Pour tester les exceptions dans les tests unitaires. On veut vérifier que la méthode lève bel et bien une exception. On utilise le **try/catch** pour capturer Dans l'exemple ci-dessous le constructeur de l'objet Person doit lancer une exception *IllegalArgumentException* si l'un des deux paramètres est null. Le test échoue si l'exception n'est pas lancée.

```
@Test
public void testPassNullsToConstructor( ) {
  try {
    Person p = new Person(null, null);
    fail("Expected IllegalArgumentException when both args are null");
  }
  catch (IllegalArgumentException expected) {
     // ignore this because it means the test passed!
  }
}
```

À utiliser uniquement lorsque vous attendez une exception, mais pas pour propager une exception.

```
//A NE PAS FAIRE!
public void testBadStyle( ) {
  try {
   SomeClass c = new SomeClass( );
   c.doSomething( );
   ...
  } catch (IOException ioe) {
      fail("Caught an IOException");
  } catch (NullPointerException npe) {
      fail("Caught a NullPointerException");
  }
}
```

Si vous voulez propager l'exception, avisez que la méthode risque de lever une exception comme ceci:

```
public void testGoodStyle( ) throws IOException {
  SomeClass c = new SomeClass( );
  c.doSomething( );

  ...

}
```

JUnit en pratique

@Test

Débuter vos méthodes par l'instruction @Test, vous n'aurez pas besoin de nommer vos méthodes en débutant par test, de plus votre classe n'aura pas besoin d'hériter de la classe TestCase.

```
@Test
public void addition() {
    assertEquals(12, simpleMath.add(7, 5));
}


@Test
public void subtraction() {
    assertEquals(9, simpleMath.substract(12, 3));
}
```

@Before and @After

Utiliser @Before et @After pour créer, initialiser et pour libérer les ressources. Ces méthodes s'exécutent au début et à la fin de chaque méthode de test. CE SONT VOS ACTEURS, ils permettent de replacer le système dans son état initial.

```
@Before
public void runBeforeEveryTest() {
    simpleMath = new SimpleMath();
}

@After
public void runAfterEveryTest() {
    simpleMath = null;
}
```

@BeforeClass and @AfterClass

Ces deux instructions servent pour placer le système dans un état initial et à le libérer à la fin. Ces méthodes s'exécutent au début et à la fin de la classe de test. On s'en sert pour, par exemple, établir une connexion avec une base de données, avec un serveur, etc.

```
@BeforeClass
public static void runBeforeClass() {
    // run for one time before all test cases
}

@AfterClass
public static void runAfterClass() {
    // run for one time after all test cases
```

```
}
```

Attente d'une exception

Utilisez le paramètre "expected" avec l'instruction @Test pour les tests où vous attendez une exception et inscrivez le type d'exception attendu. UNE AUTRE FAÇON DE TESTER LES EXCEPTIONS, CELA NÉCESSITE D'ÉCRIRE UN CAS DE TEST POUR VÉRIFIER L'EXCEPTION ET UN CAS DE TEST SANS L'EXCEPTION. C'EST UNE SOLUTION INTÉRESSANTE.

```
@Test(expected = ArithmeticException.class)
public void divisionWithException() {
    // divide by zero
    simpleMath.divide(1, 0);
}
```

@Ignore

Cette instruction ignore le test qui suit. AU LIEU DU FAIL, INTÉRESSANT ON VOIT QUE LE TEST N'A PAS ÉTÉ EXÉCUTÉ.

```
@Ignore("Not Ready to Run")
@Test
public void multiplication() {
    assertEquals(15, simpleMath.multiply(3, 5));
}
```

Timeout

Le paramètre timeout avec l'instruction @Test, permet de faire échouer un test s'il prend plus que le temps prescrit.

```
@Test(timeout = 1000)
public void infinity() {
    while (true)
        ;
}
```

Exécuter un test

Pour exécuter un test et voir afficher les résultats, il faut utiliser la classe suivante :

```
org.junit.runner.JUnitCore.runClasses(ValidationTest.class);
```

On utilise dans le "main", cependant dans Eclipse, si on exécute le programme en tant que test unitaire JUnit, l'environnement effectue ce travail pour vous.

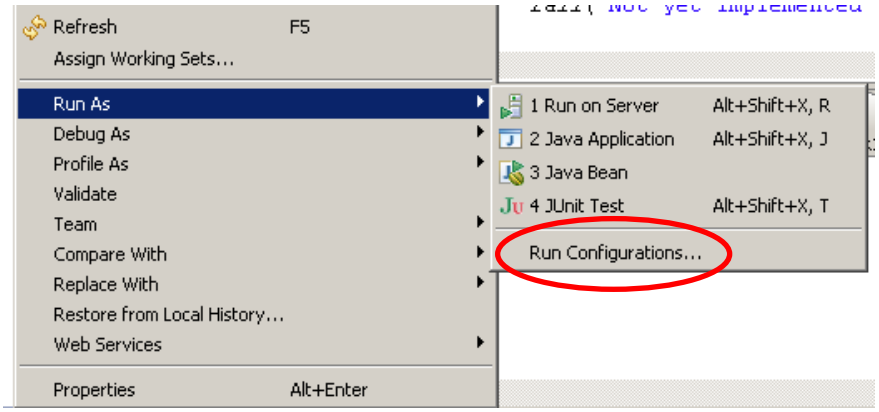
```
public static void main(String args[]) {
  org.junit.runner.JUnitCore.runClasses(ValidationTest.class);
}
```

Exécuter une série de tests

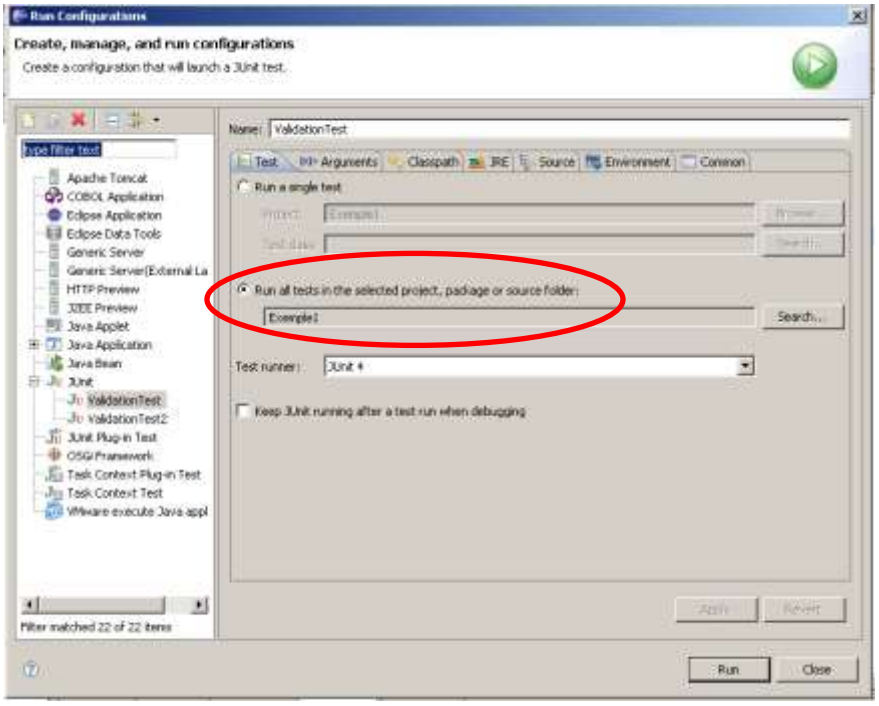
Pour exécuter une série de tests et voir afficher les résultats, il faut utiliser la classe suivante :

```
org.junit.runner.JUnitCore.runClasses(ValidationTest.class,
ValidationTest2.class);
```

On utilise dans le “main”, cependant dans Eclipse, on peut demander à l’environnement d’exécuter l’ensemble des tests du projet. On clique sur une classe de teste avec le bouton droit, et choisissez Run **As/Run Configurations**



Ensuite vous allez mentionner que vous désirez exécuter tous les tests du package.



Voilà, il ne reste qu’à exécuter comme programme JUnit pour exécuter l’ensemble des tests.

On peut aussi écrire une classe suite soi-même ainsi :

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses(value = {
          Classe1Test.class,
          Classe2Test.class,
          Classe3Test.class
          })

public class AllTests {
}
```

Source : http://www.junit.org/

Exemple d'utilisation JUnit

Classe à valider

```
/**
 * Classe Validation.java
 * But : Cette classe sert de premier exemple au cours Validation
          d'application
 *  La classe crée 2 entiers valides - valeur comprise entre -100 et 100 -
 *  effectue la somme entre les 2 nombres et détermine si le résultat est
          valide
 *  Un résultat est valide si la somme est une valeur positive.
 *
 * @author  Julie Frève
 * @version 1.0
 *
 * Date de création :  8 juin 2004
 * Révisée le       :  8 septembre 2008
 */

public class Validation {
/**
 *  Constantes pour les valeurs minimales et maximales
 */
    private static final int MAXVAL = 100;
    private static final int MINVAL  = -100;
/**
 *  Constantes pour le résultat texte
 */
    private static final String VALIDE = "Valide";
    private static final String INVALIDE = "pas Valide";

/**
 *  Attributs de la classe
 *  Les entiers pour les deux nombres valeurA et valeurB
 *  Chaîne de caractère pour explication du résultat texteResultat
 */
    private int valeurA;
    private int valeurB;
    private String texteResultat;


/**
 * Validation(int, int)
 * Constructeur qui crée un objet avec 2 valeurs valides
```

```
 * @param intA un entier entre -100 et 100 pour la valeurA
 * @param intB un entier entre -100 et 100 pour la valeurB
 * @see setValA
 * @see setValB
 * @throws <code>Exception</code> dans le cas de valeurs invalides
 *         (à l'extérieur de l'intervalle -100 et 100).
 */
    public Validation(int intA, int intB)throws Exception{
        try {
            setValA(intA);
            setValB(intB);
            texteResultat = "";
        }
        //L'exception est relancée à l'application appelante
        catch (Exception e){
            throw e;
        }
    }


/**
 * setValA(int)
 * Méthode publique pour l'assignation de l'attribut valeurA

 * @param a un entier  pour la valeurA
 * @see valideValeur
 * @throws <code>Exception</code> dans le cas d'une valeur invalide
 *         (à l'extérieur de l'intervalle -100 et 100).
 */
    public void setValA(int a)throws Exception {
       if(valideValeur(a))
            this.valeurA = a;
       else
           throw new Exception("La valeur du coté A est invalide, la valeur
                               doit être entre "+MINVAL+" et "+MAXVAL+".");
    }

/**
 * setValB(int)
 * Méthode publique pour l'assignation de l'attribut valeurB

 * @param b un entier  pour la valeurB
 * @see valideValeur
 * @throws <code>Exception</code> dans le cas d'une valeur invalide
 *         (à l'extérieur de l'intervalle -100 et 100).
 */
     public void setValB(int b)throws Exception{
       if(valideValeur(b))
            this.valeurB = b;
       else
           throw new Exception("La valeur du coté B est invalide, la valeur
                               doit être entre "+MINVAL+" et "+MAXVAL+".");

     }
/**
 * valideValeur(int)
 * Méthode privée pour valider un entier dans l'intervalle -100 et 100
 * Cette méthode est appelée par setValA ou setValB
 *
 * @param val un entier
 * @see setValB
 * @see setValA
```

```
 * @return boolean vrai si la valeur si situe entre -100 et 100
 */
     private boolean valideValeur(int val){
        return ((val >= MINVAL) && (val <= MAXVAL));
    }
/**
 * getCoteA()
 * Méthode publique qui retourne la valeur de l'attribut valeurA
 *
 * @return un entier entre -100 et 100 représentant l'attribut valeurA
 */
    public int getCoteA(){
        return valeurA;
    }

/**
 * getCoteB()
 * Méthode publique qui retourne la valeur de l'attribut valeurB
 *
 * @return un entier entre -100 et 100 représentant l'attribut valeurB
 */
    public int getCoteB() {
        return valeurB;
    }

 /**
 * sommeCotes()
 * Méthode publique qui effectue la somme des attributs vaeurA et valeurB
 * qui affecte l'attribut TexteResultat
 *
 * @return un entier représentant le résultat de la sommation
 */
   public void sommeCotes(){
   }

/**
 * getTexteResultat()
 * Méthode publique qui retourne la valeur de l'attribut texteResultat
 *
 * @return une chaîne indiquant si le résultat de la somme est valide ou non
 */
   public String getTexteResultat(){
     return this.texteResultat;
   }
}
```

Test de validation de la classe

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;

/**
 * Classe ValidationTest.java
 * But : Cette  classe sert à effectuer le test unitaire de la classe
 * Validation.java à l'aide de la librairie jUnit.
 *
 * @see Validation.java, la classe à tester
 * @see junit.framework
 * @author  Julie Frève
 * @version 1.0
 *
 * Date de création :  8 juin 2004
 * Révisée le       :  22 septembre 2008
 */

public class ValidationTest {

    //Les instances de la classe de test
    protected Validation t1,t2,t3,t4,t5;

    /**
     * Avant l'exécution de la classe de test
     * Pour les objets persistants
     * Ne s'applique pas ici
     *
     * @throws Exception
     */
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    /**
     * Après l'exécution de la classe de test
     * Pour fermer les objets persistants
     * Ne s'applique pas ici
     *
     * @throws Exception
     */
    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }


    /**
     * Appelée avant chaque test
     * Création et initialisation des acteurs
     * Les jeux d'essais sont ici
     * Si la construction est invalide on fait échouer le test
     *
     * @throws Exception
     */
    @Before
    public void setUp() {

        try{
```

```
         t1= new Validation(0,0);
         t2= new Validation(2,2);
         t3 = new Validation(-4,2);
         t4 = new Validation(100,100);

      //ou on place tous les jeux en échec ici ou on utilise
      //une méthode setUp différente pour les invalides
      //VOTRE SOLUTION ICI POUR LE TEST DES INVALIDES
         t5 = new Validation(-101,0);
         fail("la valeur 101 doit être invalide");
      }
      catch (Exception e)
      {}
 }


/**
 * Après chaque test
 * Libérer les ressources
 *
 * @throws java.lang.Exception
 */
@After
public void tearDown()  {
       t1=null;
       t2=null;
       t3=null;
       t4=null;
}

/**
 * Test method for {@link Validation#Validation(int, int)}.
 */
@Ignore("Testvalidation pas prêt")
@Test
public void testValidation() {
       fail("Not yet implemented");
}

/**
 * Test sans s'occuper de l'exception
 */
@Test
public void testSetValA()throws Exception {

         t1.setValA(-25);

         assertEquals(t1.getCoteA(),-25);
         t1.setValA(100);
         assertEquals(t1.getCoteA(),100);

         t1.setValA(-100);

         assertEquals(t1.getCoteA(),-100);
         t1.setValA(0);

         assertEquals(t1.getCoteA(),0);
         t1.setValA(55);

         assertEquals(t1.getCoteA(),55);
         t1.setValA(55);
```

```
            // t1.setValA(101);
            //assertEquals(t1.getCoteA(),101);
    }

    /**
     * Test en s'occupant de l'exception
     */
    @Test(expected = Exception.class)
    public void setValB()throws Exception {

           t1.setValB(101);
           assertEquals(t1.getCoteA(),101);
    }



    /**
     * Test method for {@link Validation#setValB(int)}.
     */
    @Test(timeout = 1000)
    public void testsetValB(){
           int i = 101;

           while (i > 100)
                  i++;

           fail("Not yet implemented");
    }

    /**
     * Test method for {@link Validation#getCoteA()}.
     */
    @Test
    public void testGetCoteA() {
           fail("Not yet implemented");
    }

    /**
     * Test method for {@link Validation#getCoteB()}.
     */
    @Test
    public void testGetCoteB() {
           fail("Not yet implemented");
    }

    /**
     * Test method for {@link Validation#sommeCotes()}.
     */
    @Test
    public void testSommeCotes() {
           fail("Not yet implemented");
    }

    /**
     * Test method for {@link Validation#getTexteResultat()}.
     */
    @Test
    public void testGetTexteResultat() {
           fail("Not yet implemented");
    }
}
```