

# Mauvaises pratiques dans l'écriture de tests

Arnaud Blouin

Les exemples de ce document sont écrits en JUnit 5 et Mockito mais les mauvaises pratiques qu'ils illustrent s'appliquent à toutes les bibliothèques de test.

## Assertions

Parmi les mauvaises pratiques de test unitaire les plus courantes, une récurrente consiste à utiliser `assertTrue` et `assertFalse` à tout va.



```
@Test public void testSetStr() {  
    obj.setStr("foo");  
    assertTrue(pt.getStr().equals("foo"));  
}  
  
@Test public void testSetA() {  
    obj.setA(a);  
    assertTrue(obj.getA()==a);  
}
```

Ce cas simple n'est pas très grave en soi mais il existe des assertions dédiées à la comparaison d'objets. La bonne pratique est donc d'utiliser ces assertions :



```
@Test public void testSetStr() {  
    obj.setStr("foo");  
    assertEquals("foo", pt.getStr()); // La valeur attendue est à gauche en JUnit  
}  
  
@Test public void testSetA() {  
    obj.setA(a);  
    assertEquals(a, obj.getA()); // La valeur observée est à droite  
}
```

Attention au test de valeurs flottantes : la comparaison de valeurs flottantes (*double* en Java, mais pas de problème avec *float*) n'est pas exacte. Il faut utiliser un delta de tolérance. Donc à la place du code suivant :



```
@Test public void testSetX() {  
    pt.setX(10.0);  
    assertEquals(10.0, pt.getX());  
}
```

il faut écrire :



```
@Test public void testSetX() {  
    pt.setX(10.0);  
    assertTrue(10.0, pt.getX(), 0.00001); // Troisième argument : delta tolérance  
}
```

Il n'existe pas, et à raison, d'assertion "*success*" : si aucune assertion n'échoue ou si aucune exception n'est levée, alors le test passe automatiquement. Vous ne devez donc pas écrire quelque chose du style :



```
assertTrue(true);
```

ou :



```
assertEquals(true, true);
```

Il existe beaucoup d'assertions, cf : <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>.  
Ne pas utiliser la bonne assertion est une mauvaise pratique. Par exemple :



```
assertTrue(a!=b);
```

À la place :



```
assertNotEquals(a, b);
```

Un autre exemple, l'assertion :



```
assertTrue(a!=null);
```

devrait être écrite :



```
assertNotNull(a);
```

Les assertions `assertTrue` et `assertFalse` soit utilisées pour tester des expressions booléennes. L'utilisation d'opérateurs à l'intérieur d'une expression d'une assertion (exemple : `assertTrue(a==b)`) est une mauvaise pratique : vous n'utilisez pas la bonne assertion.

Autres mauvaises pratiques : **mauvaise utilisation du `try / catch` dans un test unitaire qui peut lever des exceptions.**



```
@Test
public void testHorizontalSymmetryException() {
    try{
        pt.horizontalSymmetry(null);
        fail();
    }catch(IllegalArgumentException e) {
        // OK
    }
}
```

Si le test ci-dessus est correct, JUnit fournit une fonctionnalité pour éviter cette lourdeur syntaxique : le paramètre *expected* de l'annotation `@Test` :



```
@Test
public void testHorizontalSymmetryException() {
    assertThrows(IllegalArgumentException.class, ()-> pt.horizontalSymmetry(null));
}
```

Autre mauvaise pratique concernant les exceptions :



```
@Test
public void testHorizontalSymmetry() {
    try{
        pt.horizontalSymmetry(new Point(1,2));
    }catch(IllegalArgumentException e) {
        fail();
    }
}
```

Le `try / catch` du test ci-dessus est inutile. Il suffit de déclarer l'exception au niveau du prototype du test. Ainsi, si une exception est levée le test JUnit sera marqué comme ayant échoué.



```
@Test
public void testHorizontalSymmetry() throws IllegalArgumentException {
    pt.horizontalSymmetry(new Point(1,2));
}
```

Il est également inutile de vérifier l'instanciation d'un objet. Ici, p est forcément non nul. Le test suivant peut donc est supprimé :



```
@Test
public void testPointNotNullWhenCreated() throws IllegalArgumentException {
    pt = new Point();
    assertNotNull(p);
}
```

## Test fixture

Au niveau de la classe de tests, une mauvaise pratique consiste à avoir du code dupliqué dans chaque test :



```
public class TestPoint {
    @Test
    public void testHorizontalSymmetry() throws IllegalArgumentException {
        Point pt = new Point();
        pt.horizontalSymmetry(new Point(1,2));
    }
    @Test public void testSetGetX() {
        Point pt = new Point();
        pt.setX(10.0);
        assertEquals(10.0, pt.getX(), 0.001);
    }
    @Test public void testSetGetY() {
        Point pt = new Point();
        pt.setY(10.0);
        assertEquals(10.0, pt.getY(), 0.001);
    }
}
```

Dans le code ci-dessus, la déclaration et l'initialisation d'un objet est dupliquée dans chaque test. Dans ce cas il faut utiliser la méthode "*@BeforeEach*" comme le montre le code suivant :



```
public class TestPoint {
    Point pt;

    @BeforeEach
    public void setUp() {
        pt = new Point();
    }
    @Test
    public void testHorizontalSymmetry() throws IllegalArgumentException {
        pt.horizontalSymmetry(new Point(1,2));
    }
    @Test public void testSetGetX() {
        pt.setX(10.0);
        assertEquals(10.0, pt.getX(), 0.001);
    }
    @Test public void testSetGetY() {
        pt.setY(10.0);
        assertEquals(10.0, pt.getY(), 0.001);
    }
}
```

Concernant la méthode *@BeforeEach*, attention à ne pas écrire :



```
@BeforeEach
public void setUp() {
    Point pt = new Point();
}
```

La variable *pt* est dans ce cas local et donc pas accessible pour les tests (exemple vu de nombreuses fois dans les copies...). Il ne faut également pas écrire :



```
public class TestPoint {
    Point pt = new Point();

    @Test
    public void testHorizontalSymmetry() throws IllegalArgumentException {
        pt.horizontalSymmetry(new Point(1,2));
    }
    @Test public void testSetGetX() {
        pt.setX(10.0);
        assertEquals(10.0, pt.getX(), 0.001);
    }
    @Test public void testSetGetY() {
        pt.setY(10.0);
        assertEquals(10.0, pt.getY(), 0.001);
    }
}
```

En effet, cela peut éventuellement fonctionner mais il faut laisser le travail de l'initialisation des tests à JUnit via la méthode *@BeforeEach*.

## Structure des tests

Autre mauvaise pratique plus compliquée, écrire plusieurs tests dans une unique méthode de test :



```
@Test public void testSetGetXY() {  
    pt.setX(10.0);  
    assertEquals(10.0, pt.getX(), 0.001);  
    pt.setY(10.0);  
    assertEquals(10.0, pt.getY(), 0.001);  
}
```

Il faut ici écrire deux tests unitaires séparés :



```
@Test public void testSetGetX() {  
    pt.setX(10.0);  
    assertEquals(10.0, pt.getX(), 0.001);  
}  
  
@Test public void testSetGetY() {  
    pt.setY(10.0);  
    assertEquals(10.0, pt.getY(), 0.001);  
}
```

L'avantage serait d'améliorer la lecture des tests et surtout de faciliter le débogage : dans le cas du mauvais exemple, si l'assertion sur *setX* ne passe pas, le test sur *setY* n'est pas exécuté, ce qui n'est pas cohérent. Il en est de même avec l'exemple suivant :



```
@Test public void testSetGetValue() {  
    pt.setValue(10.0);  
    assertEquals(10.0, pt.getValue(), 0.001);  
    pt.setValue(-1.0); // value cannot be negative  
    assertEquals(10.0, pt.getValue(), 0.001);  
}
```

Deux tests sont présents dans le code ci-dessus : le test nominal (bon fonctionnement) ; le test du cas d'erreur de valeur négative (dans l'exemple *setValue* ne fait rien si la valeur est négative). Il faut séparer ces deux tests :



```
@Test public void testSetGetValueOK() {  
    pt.setValue(10.0);  
    assertEquals(10.0, pt.getValue(), 0.001);  
}  
  
@Test public void testSetGetValueKONegValue() {  
    pt.setValue(10.0);  
    pt.setValue(-1.0); // value cannot be negative  
    assertEquals(10.0, pt.getValue(), 0.001);  
}
```

## Mock

Concernant les mocks. Il n'y a aucun intérêt à simuler l'objet que l'on teste. Le but des mocks est de casser des dépendances à l'aide de marionnettes (les mocks donc) que vous contrôlez. Le code suivant n'a donc pas de sens :



```
public class TestPoint {
    Point pt;

    @BeforeEach
    public void setUp() {
        pt = Mockito.mock(Point.class);
        Mockito.when(pt.getX()).thenReturn(10.0);
    }
    @Test public void testSetGetX() {
        assertEquals(10.0, pt.getX(), 0.001);
    }
}
```

Autre erreur, moins dramatique que la précédente : créer une classe pour pouvoir mocker une interface. Cela n'est pas nécessaire car le mocking fonctionne également avec une interface sans implémentation (c'est un de ses buts).



```
interface Vector {
    double getTx();
}

public class TestPoint {
    Point pt;
    VectorImpl v;

    static class VectorImpl implements Vector {
        public double getTx() { return 0; }
    }

    @BeforeEach
    public void setUp() {
        v = Mockito.mock(VectorImpl.class);
        Mockito.when(v.getTx()).thenReturn(10.0);
        pt = new Point(v);
    }
}
```

Il faut donc simplifier le code précédent pour obtenir le code suivant. À noter que la classe `VectorImpl` est ce que l'on appelle un *stub* (une fausse classe codée par le testeur) et peut être remplacée par un mock : moins verbeux, plus flexible.



```
interface Vector {  
    double getTx();  
}  
  
public class TestPoint {  
    Point pt;  
    Vector v;  
  
    @BeforeEach  
    public void setUp() {  
        v = Mockito.mock(Vector.class);  
        // Même s'il s'agit d'une interface,  
        // il faut écrire Vector.class et non Vector.interface  
        Mockito.when(v.getTx()).thenReturn(10.0);  
        pt = new Point(v);  
    }  
}
```