

# Tests en intégration continue

J.-M. Bruel

Version 1.0 2019-01-12

# Table des matières

1. Pourquoi tester ?	1
1.1. Pour livrer le <b>bon produit</b>	1
1.2. Ce qui marche pour 1 ne marche pas nécessairement pour 100	2
1.3. La loi de <b>Murphy</b>	2
1.4. Différents <b>OS</b> ou différents <b>terminaux</b>	3
1.5. Pour donner le <b>meilleur</b>	3
2. Un exemple concret de test <b>obligatoire</b>	3
3. Un exemple concret de <b>documentation</b> obligatoire	4
4. Typologie des tests	5
5. JUnit etc.	5
5.1. Quoi tester ?	5
5.2. Assertions	5
5.3. Stratégie de tests	5
5.4. L'ordre des tests	6
5.5. Sous Eclipse	6
5.6. Et pour les interfaces graphiques?	6
5.7. Couverture des tests	7
6. Application concrète pour ce cours	8
6.1. De <i>To Be Done</i> à <i>On going</i>	8
6.2. Créer une branche spécifique (si nouvelle <i>feature</i> )	8
6.3. Ecrire un test qui échoue	8
6.4. Essai de merge pour voir si tout le reste marche encore	12
6.5. Commit & Push dans devs	12
6.6. De <i>On going</i> à <i>Review</i>	13
Pour aller plus loin...	13

# 1. Pourquoi tester ?

A majority of the production failures (77%) can be reproduced by a unit test.

— Yuan et al. OSDI 2014



Figure 1. Un tweet récent!



Pour lire l'article en question : <https://blog.acolyer.org/2016/10/06/simple-testing-can-prevent-most-critical-failures/amp/>

## 1.1. Pour livrer le bon produit

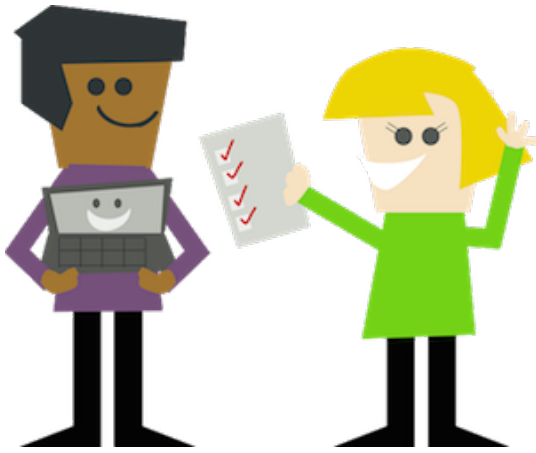


Figure 2. Un produit qui fait ce qu'il est censé faire (crédit photo <http://www.te52.com/testtalk/2014/08/07/5-reasons-we-need-software-testing/>)

## 1.2. Ce qui marche pour 1 ne marche pas nécessairement pour 100

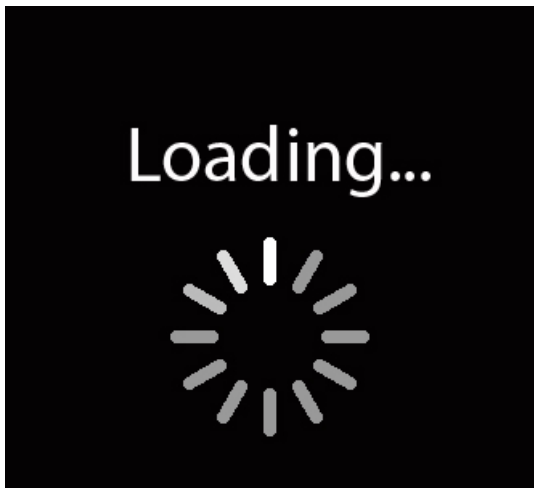


Figure 3. Passage à l'échelle (crédit photo <http://www.te52.com/testtalk/2014/08/07/5-reasons-we-need-software-testing/>)

## 1.3. La loi de Murphy

Tout ce qui est susceptible de mal tourner tournera nécessairement mal.

— Edward A. Murphy Jr.

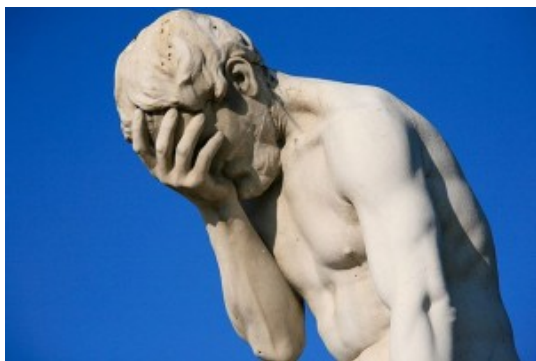


Figure 4. Murphy's law (crédit photo <http://www.te52.com/testtalk/2014/08/07/5-reasons-we-need-software-testing/>)

## 1.4. Différents OS ou différents terminaux

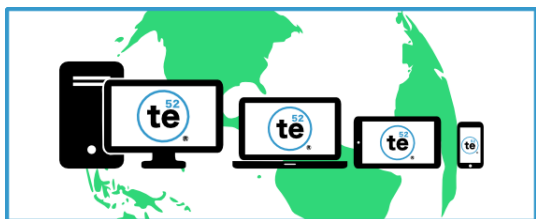


Figure 5. Diversité (crédit photo <http://www.te52.com/testtalk/2014/08/07/5-reasons-we-need-software-testing/>)

## 1.5. Pour donner le meilleur



Figure 6. Faire de son mieux (crédit photo <http://www.te52.com/testtalk/2014/08/07/5-reasons-we-need-software-testing/>)

## 2. Un exemple concret de test obligatoire

[Asciidoctor Contribution](#)



Figure 7. Autour d'une bière avec Dan Allen, à Denver, Colorado #ILoveMyJob

1. Fork the repository.
2. Run `bundle` to install development dependencies.
3. Create a topic branch
4. Add tests for your unimplemented feature or bug fix. (See [\[writing-and-executing-tests\]](#))
5. Run `bundle exec rake` to run the tests. If your tests pass, return to step 4.
6. Implement your feature or bug fix.
7. Run `bundle exec rake` to run the tests. If your tests fail, return to step 6.
8. Add documentation for your feature or bug fix.
9. If your changes are not 100% documented, go back to step 8.
10. Add, commit, and push your changes.
11. Submit a pull request.

### 3. Un exemple concret de documentation obligatoire



Figure 8. Après un footing avec Gaël Blondel, à Saint-Malo #ILoveMyJob

[...] an Eclipse project is providing extensible frameworks and applications accessible via documented APIs.

— Eclipse Development Process

## 4. Typologie des tests

Table 1. Différences entre Vérification et Validation (source

[https://www.tutorialspoint.com/software\\_testing/software\\_testing\\_quick\\_guide.htm](https://www.tutorialspoint.com/software_testing/software_testing_quick_guide.htm))

Vérification	Validation
Le produit est-il <b>bon</b> ?	Le produit est-il <b>le bon</b> ?
<i>Are you building it right?</i>	<i>Are you building the right thing?</i>
Réalisée par le <b>développeur</b>	Réalisée par le <b>testeur</b>
En premier	Après la vérification

## 5. JUnit etc.

### 5.1. Quoi tester ?

Les exceptions	<code>@Test (expected = Exception.class)</code>
Le temps d'exécution	<code>@Test(timeout=100)</code>
Uniquement certains environnements	<code>System.getProperty("os.name").contains("Linux");</code> Attention cette instruction n'est pas une annotation.
S'exécute avant les autres tests (e.g., accès à une base)	<code>@BeforeClass public static void method()</code>

### 5.2. Assertions

<code>fail([message])</code>	On force le test à échouer
<code>assertTrue([message,] condition)</code>	La condition est vraie
<code>assertFalse([message,] condition)</code>	La condition est fausse
<code>assertEquals([message,] attendu, actuel)</code>	Les deux valeurs sont égales
<code>assertNull([message,] object)</code>	Objet nul
<code>assertSame([message,] expected, actual)</code>	Objets identiques (même réf.)

### 5.3. Stratégie de tests

Considérons une fonction `int add(int,int);` d'une classe `myClass`.

Définir le comportement normal de la fonction (sortie normale pour des paramètres corrects).

(source : <http://stackoverflow.com/questions/8751553/how-to-write-a-unit-test>)

```
//for normal addition
@Test
public void testAdd1Plus1() {
    int x = 1 ; int y = 1;
    assertEquals(2, myClass.add(x,y));
}
```

Ajouter des tests pour les cas particuliers :

- aucune exception non capturée en cas d'*overflow*
- les paramètres `null` sont gérés, e.g., :

(source : <http://stackoverflow.com/questions/8751553/how-to-write-a-unit-test>)

```
//if you are using 0 as default for null, make sure your class works in that case.
@Test
public void testAdd1Plus1() {
    int y = 1;
    assertEquals(1, myClass.add(null,y));
}
```

## 5.4. L'ordre des tests

Surtout aucun!!

JUnit assumes that all test methods can be executed in an arbitrary order. Well-written test code should not assume any order, i.e., tests should not depend on other tests.

— JUnit manual

## 5.5. Sous Eclipse

- Pour une classe existante : [ **Right-click** ] (dans le **Package Explorer** et **New > JUnit Test Case**).
- Utiliser le **JUnit wizards** (**File > New > Other... > Java > JUnit**).
- Il n'y a plus qu'à faire **Run-as > JUnit Test**.

Pensez à utiliser le plug-in [infinittest](#).

## 5.6. Et pour les interfaces graphiques?

Exemple de la librairie **Robot** :



(source : <http://stackoverflow.com/questions/16411823/junit-tests-for-gui-in-java>)

```
Robot bot = new Robot();
bot.mouseMove(10,10);
bot.mousePress(InputEvent.BUTTON1_MASK);
//add time between press and release or the input event system may
//not think it is a click
try{Thread.sleep(250);}catch(InterruptedException e){}
bot.mouseRelease(InputEvent.BUTTON1_MASK);
```

Exemple du plugin **Eclipse swingcoder** :

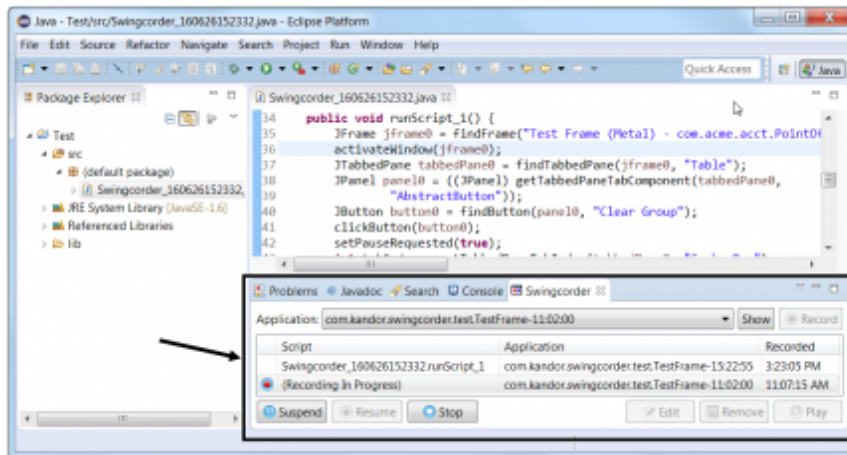


Figure 9. Simulation d'utilisation d'interface (source <https://marketplace.eclipse.org/content/swingcoder>)

## 5.7. Couverture des tests

Il existe des outils pour aller plus loin :

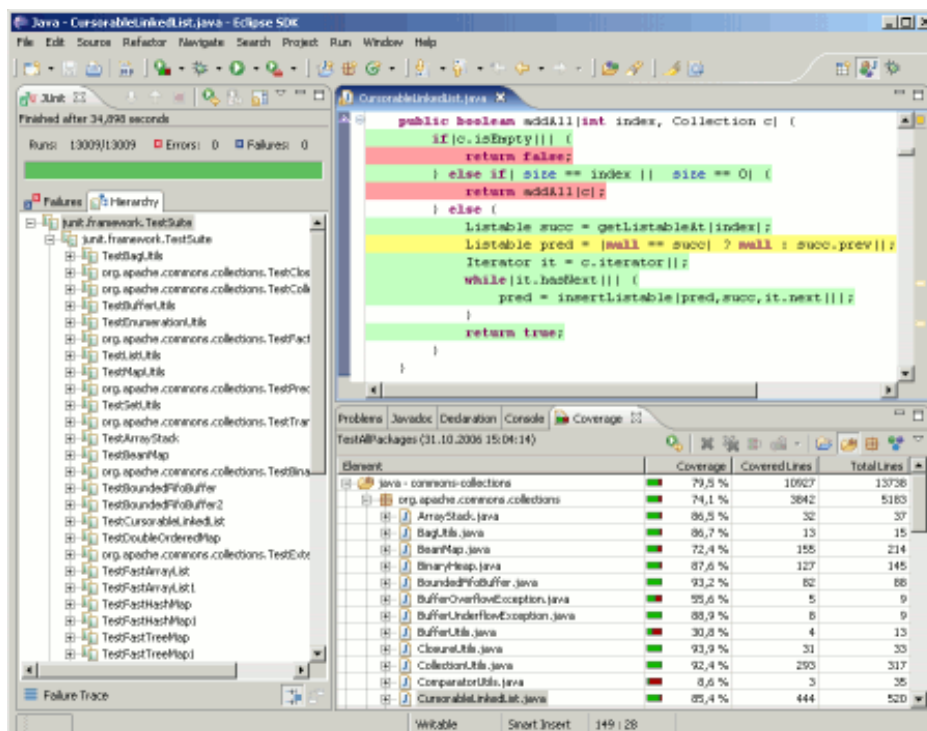
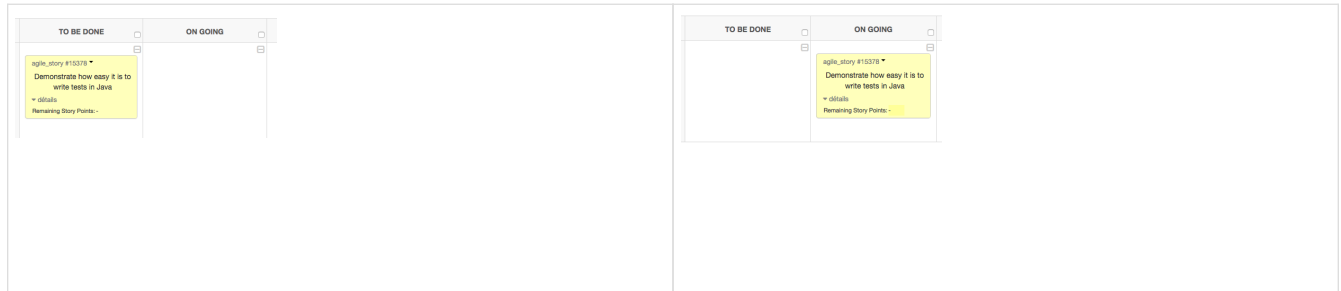


Figure 10. Couverture des tests (source <http://www.eclemma.org/>)

## 6. Application concrète pour ce cours

### 6.1. De *To Be Done* à *On going*

Table 2. Mettre à jour votre gestionnaire de tâche



MPA2016 » User Stories » User Stories #15378

#### Demonstrate how easy it is to write tests in Java

##### Dernières modifications



Jean-Michel Bruel (jmbuel)

05/10/2

- Status modifié de To be done à On going

Figure 11. Confirmation par email

### 6.2. Créer une branche spécifique (si nouvelle *feature*)

```
bruel (master) $ git checkout -b US-15378
Switched to a new branch 'US-15378'
bruel (US-15378) $
```

### 6.3. Ecrire un test qui échoue

#### 6.3.1. Etape 0 : Bien comprendre ce qu'on doit faire

Exemmples : Objectif de la tâche = créer une classe **Pile**.



Rappels sur les propriétés d'une **Pile** (opérations)

##### Opérations

```
CréerPile : -> Pile
estVide   : Pile -> Booléen
Empiler   : Pile * Élément -> Pile
Dépiler   : Pile -> Pile
Sommet    : Pile -> Élément
```

Rappels sur les propriétés d'une **Pile** (préconditions)

### Préconditions



Sommet(p) valide Si et Seulement Si  $\text{estVide}(p) == \text{FAUX}$   
Dépiler(p) valide Si et Seulement Si  $\text{estVide}(p) == \text{FAUX}$

Rappels sur les propriétés d'une **Pile** (axiomes)

### Axiomes



- (1)  $\text{estVide}(\text{CréerPile}())$
- (2)  $\text{estVide}(\text{Empiler}(p,e)) == \text{FAUX}$
- (3)  $\text{estVide}(\text{Dépiler}(\text{Empiler}(p,e)))$  Si et Seulement Si  $\text{estVide}(p)$
- (4)  $\text{Sommet}(\text{Empiler}(p,e)) == e$
- (5)  $\text{!estVide}(p) \Rightarrow \text{Sommet}(\text{Dépiler}(\text{Empiler}(p,e))) == \text{Sommet}(p)$

## 6.3.2. Etape 1 : Ecrire un test simple

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class PileTest {

    @Test
    public void test_type_new_Pile() throws Exception {
        Pile pile = new Pile() ;

        assertEquals("Pile", pile.getClass().getName(),"new Pile() retourne
une Pile");
    }
}
```

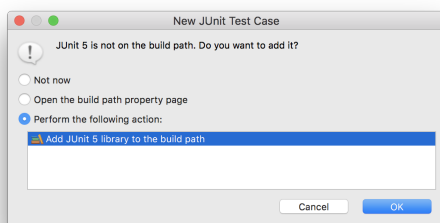


Figure 12. Oups, JUnit n'est pas dans le path...



Figure 13. Création rapide de la classe `Pile`

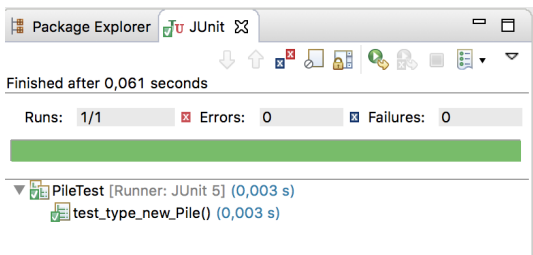


Figure 14. Run as JUnit Tests

### 6.3.3. Etape 2 : écrire un test qui passe

```
@Test
public void test_type_empiler() throws Exception {
    Pile pile = new Pile() ;

    assertEquals("Pile", pile.empiler("XXX").getClass().getName(), "empiler(pile, 'XXX')
retourne une Pile");
}
```

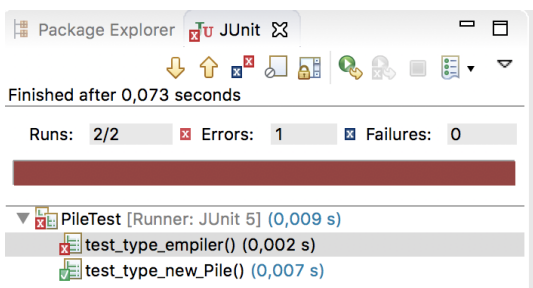


Figure 15. Erreur de syntaxe

```
public class Pile {

    public Object empiler(String string) {
        // TODO Auto-generated method stub
        return this;
    }
}
```



La méthode générée par défaut retourne **null** ce qui provoque une **NullPointerException**. Nous avons modifié la méthode en conséquence.

```
public class Pile {

    public Pile empiler(String string) {
        // TODO Auto-generated method stub
        return new Pile();
    }

}
```

### 6.3.4. Etape 2 : écrire un test qui échoue

```
@Test
public void test_axiome1() {
    Pile pile = new Pile() ;

    assertTrue(pile.estVide(pile),"Une nouvelle pile est vide");
}
```

Méthode ajoutée par défaut

```
public boolean estVide(Pile pile) {
    // TODO Auto-generated method stub
    return false;
}
```

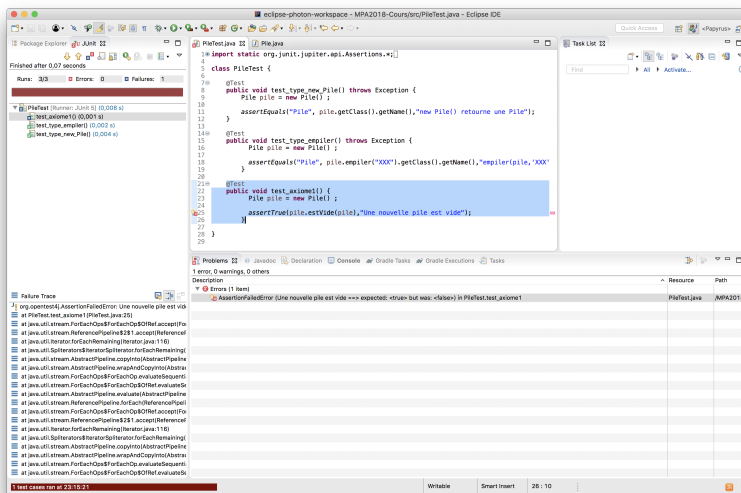


Figure 16. Passage du test



Junit n'exécute que les fonctions qui commencent pas **@Test**.

### 6.3.5. Etape 3 : On fait passer le test

```
public boolean estVide(Pile pile) {
    // Smartly modified by JMB to pass the test!
    return true;
}
```

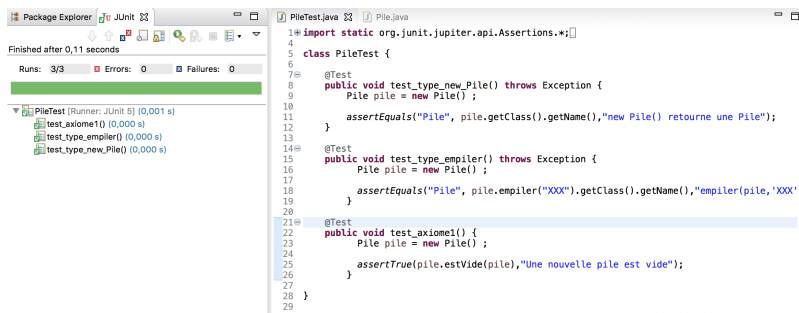


Figure 17. Passage du test

Bien sûr le code n'est pas correcte pour l'instant (on s'en rendra compte dès les tests suivants)! Une meilleure solution pourrait être :



```
public class Pile {
    int count;
    ...
    public boolean estVide(Pile pile) {
        return (count == 0);
    }
}
```

## 6.4. Essai de merge pour voir si tout le reste marche encore

```
bruel (US-15378) $ git commit -am "Adding push feature. Tests OK"
[US-15378 78f3242] Adding push feature. Tests OK
1 file changed, 2 insertions(+), 3 deletions(-)
bruel (US-15378) $ git checkout devs
Switched to branch 'devs'
bruel (devs) $ git merge US-15378
```

## 6.5. Commit & Push dans **devs**

```
bruel (devs) $ git commit -am "..."  
...  
bruel (devs) $ git push origin devs  
...  
bruel (devs) $ git branch -D US-15378  
Deleted branch US-15378 (was f392a73).
```

## 6.6. De On going à Review



Figure 18. Penser à mettre à jour le tableau de bord

## Pour aller plus loin...

- <http://rpouiller.developpez.com/tutoriels/java/tests-unitaires-junit4/>
- [https://jmbruel.github.io/teaching/topics/agile.html#\\_les\\_tests](https://jmbruel.github.io/teaching/topics/agile.html#_les_tests)
- <http://www.vogella.com/tutorials/JUnit/article.html>
- <http://junit.org>
- <http://stackoverflow.com/questions/8751553/how-to-write-a-unit-test>
- <https://www.quora.com/How-do-you-get-developers-to-love-testing-their-code>