Tests en intégration continue

Table of Contents

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

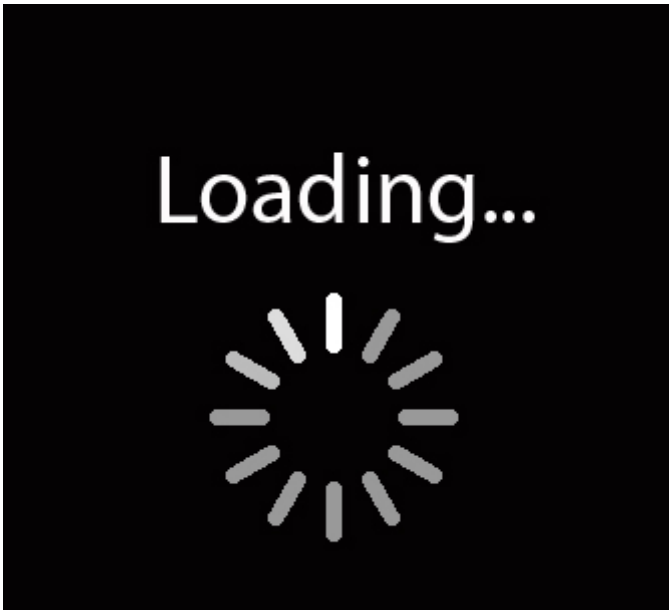

Figure 3. Passage à l'échelle (crédit photo http://www.te52.com/testtalk/2014/08/07/5-reasons-we-need-software-testing/)

1.3. La loi de Murphy

> Tout ce qui est susceptible de mal tourner tournera nécessairement mal.
>
> — Edward A. Murphy Jr.

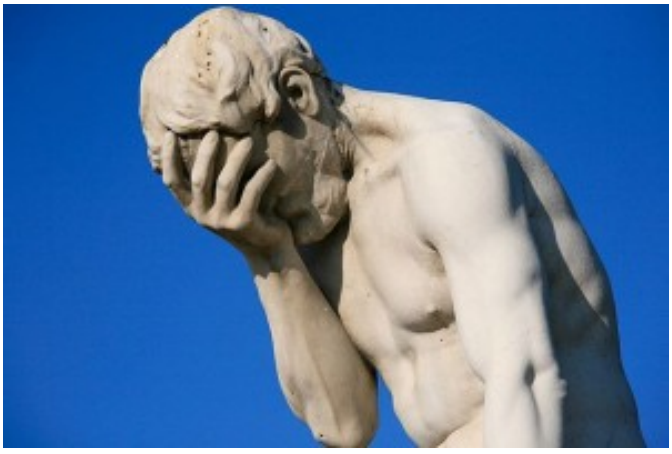

Figure 4. Murphy's law (crédit photo http://www.te52.com/testtalk/2014/08/07/5-reasons-we-need-software-testing/)

1.4. Différents OS ou différents terminaux

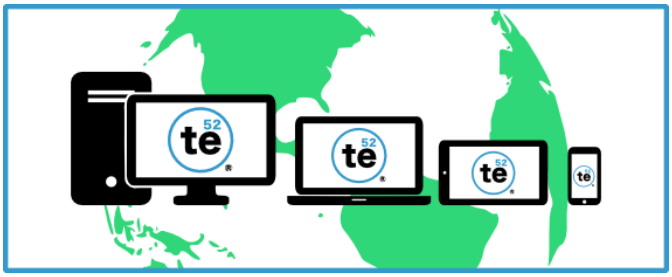

Figure 5. Diversité (crédit photo http://www.te52.com/testtalk/2014/08/07/5-reasons-we-need-software-testing/)

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)

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

5. JUnit etc.

5.1. Quoi tester ?

Les exceptions	`@Test (expected = Exception.class)`
Le temps d'exécution	`@Test(timeout=100)`

Uniquement certains environnement	`System.getProperty("os.name").contains("Linux"));` Attention cette instruction n'est pas une annotation.
S'exécute avant les autres tests (e.g., accès à une base)	`@BeforeClass public static void method()`

5.2. Assertions

`fail([message])`	On force le test à échouer
`assertTrue([message,] condition)`	La condition est vraie
`assertFalse([message,] condition)`	La condition est fausse
`assertEquals([message,] attendu, actuel)`	Les deux valeurs sont égales
`assertNull([message,] object)`	Objet nul
`assertSame([message,] expected, actual)`	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(0, myClass.add(null,y));
}
```

- ça marche avec les paramètres négatifs, etc.

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 infinitest.

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` :

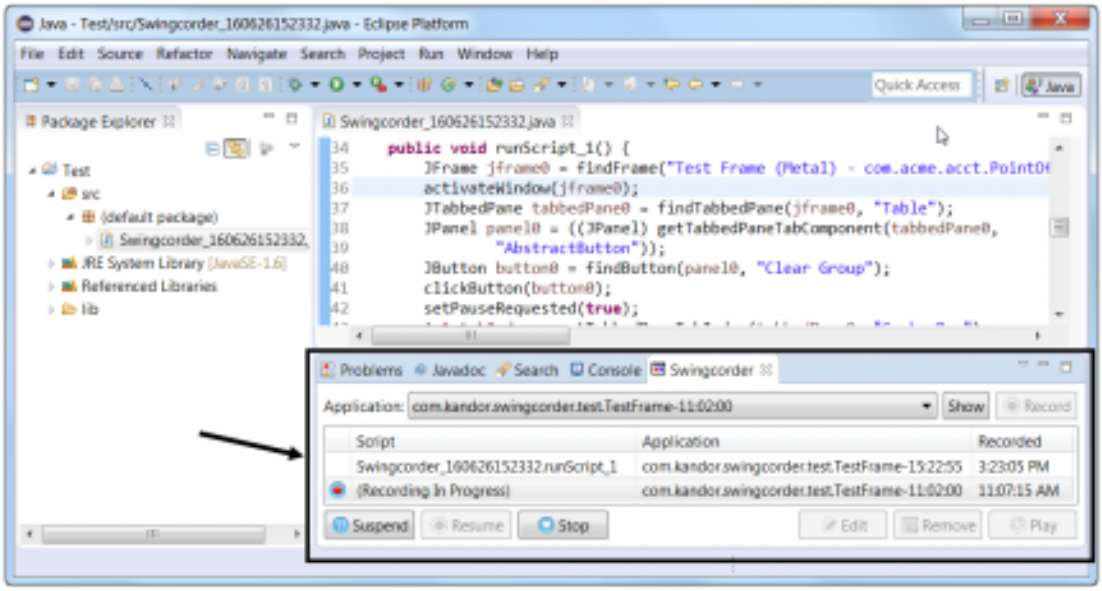

Figure 9. Simulation d'utilisation d'interface (source https://marketplace.eclipse.org/content/swingcorder)

5.7. Couverture des tests

Il existe des outils pour aller plus loin :

[coverage] | *coverage.gif*

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

6. Application concrète pour vos projets

6.1. De `To Be Done` à `On going`

Table 2. Mettre à jour le kanban

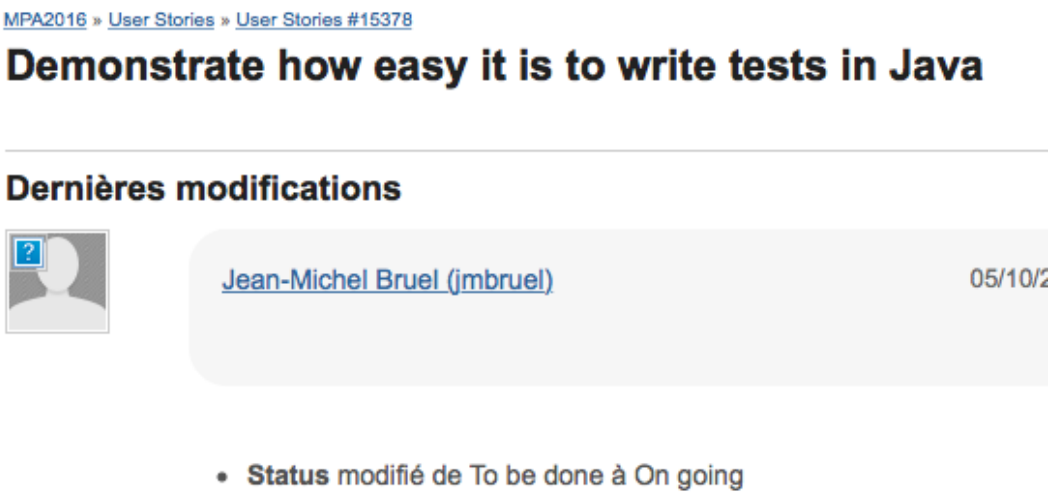

Figure 11. Confirmation par email

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

Objectif de la tâche : créer une classe `Pile`.

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

Opérations

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

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

Préconditions

```
Sommet(p) valide Si et Seulement Si estVide(p) == FAUX
Dépiler(p) valide Si et Seulement Si estVide(p) == FAUX
```

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

Axiomes

```
(1) estVide(CréerPile())
(2) estVide(Empiler(p,e)) == FAUX
(3) estVide(Dépiler(Empiler(p,e))) Si et Seulement Si estVide(p)
(4) Sommet(Empiler(p,e)) == e
(5) !estVide(p) => Sommet(Dépiler(Empiler(p,e))) == Sommet(p)
```

6.3.2. Etape 1 : Ecrire un test simple

```
import junit.textui.TestRunner;
import junit.framework.TestSuite;
import junit.framework.TestCase;

public class PileTest extends TestCase {

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

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

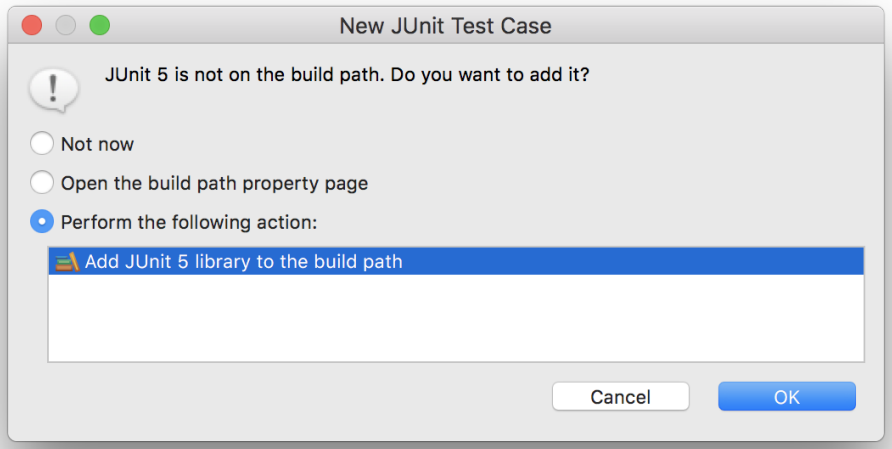

Figure 12. Oups, JUnit n'est pas dans le path...

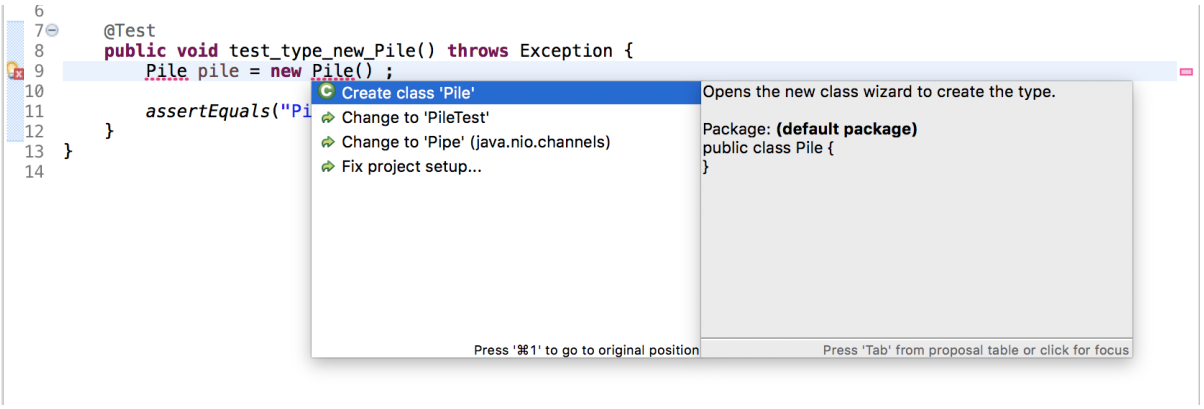

Figure 13. Création rapide de la classe `Pile`

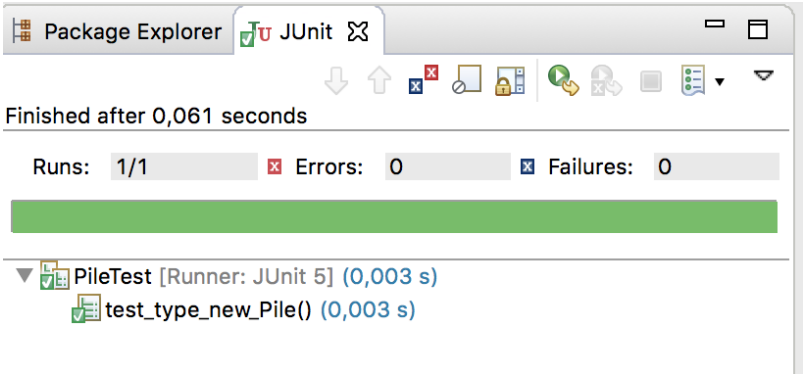

Figure 14. Run as JUnit Tests

6.3.3. Etape 1' : améliorer avec un `main`

Pour ceux qui veulent vraiment un `main` :

```
public class PileTest extends TestCase {
        static int totalAssertions = 0;
        static int bilanAssertions = 0;

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

                totalAssertions++ ;
                assertEquals("new Pile() retourne une Pile", "Pile",
pile.getClass().getName());
                bilanAssertions++ ;
        }

        public static void main(String[] args) {
                junit.textui.TestRunner.run(new TestSuite(PileTest.class));
                if (bilanAssertions == totalAssertions) { System.out.print("Bravo !");
}
                System.out.println(" "+bilanAssertions+"/"+totalAssertions+"
assertions vérifiées");
        } // fin main

} // fin PileTest
```

En exécutant le test comme un programme Java on obtient :

```
...
Time: 0,005

OK (3 tests)

Bravo ! 3/3 assertions vérifiées
```

Si vous utilisez d'autres environnement qu'Eclipse, come SciTE, pour compiler le programme de Test n'oubliez pas de placer les fichiers `SciTE.properties` et `junit.jar` dans le répertoire de vos sources (avant d'ouvrir SciTE) ou bien exécutez ceci :

```
javac -cp .;junit.jar PileTest.java
```

6.3.4. Etape 2 : écrire un test qui passe

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

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

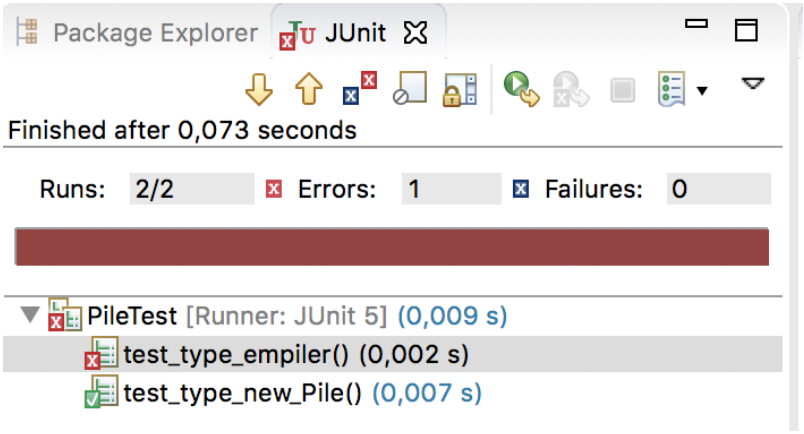

Figure 15. Erreur de syntaxe

```
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.

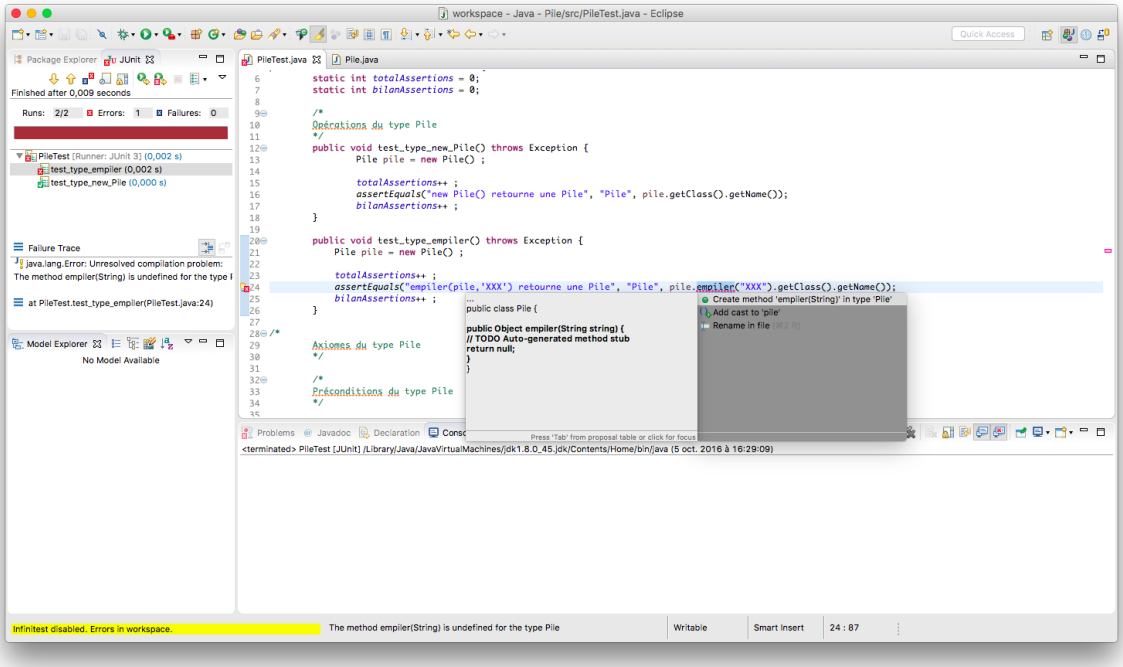

Figure 16. Ajout simple de la méthode manquante

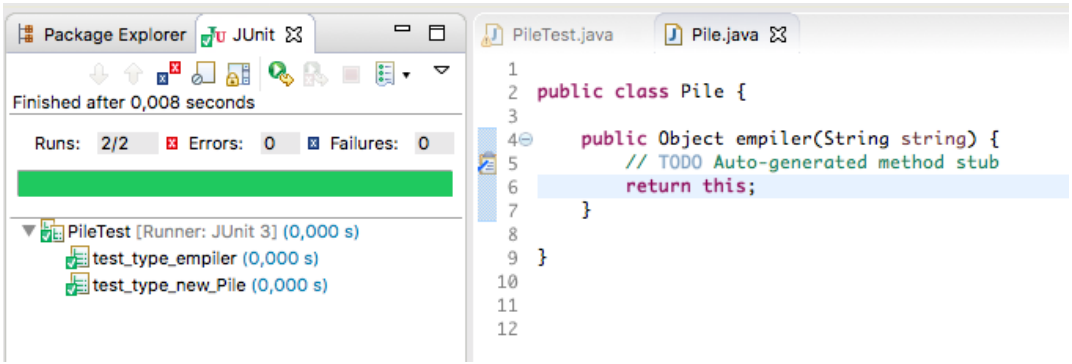

Figure 17. Passage du test

6.3.5. Etape 2 : écrire un test qui échoue

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

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

Méthode ajoutée par défaut

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

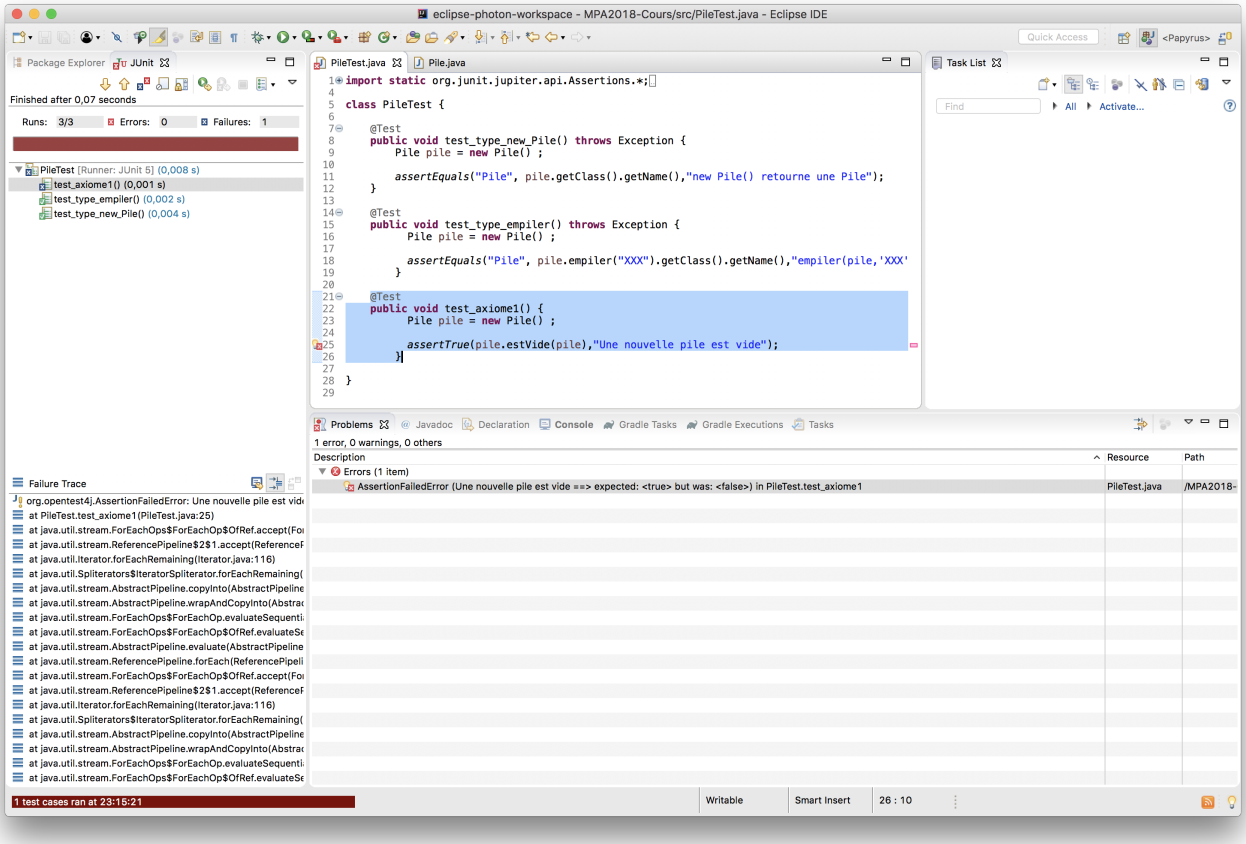

Figure 18. Passage du test

Junit utilise l'annotation `@Test` devant les tests, comme dans l'exemple ci-dessous. Il suffit de l'enlever pour ne pas exécuter ce test :

```
@Test
public void constructeur_correct() {
   ...
}
```

6.3.6. Etape 3 : On fait passer le test

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

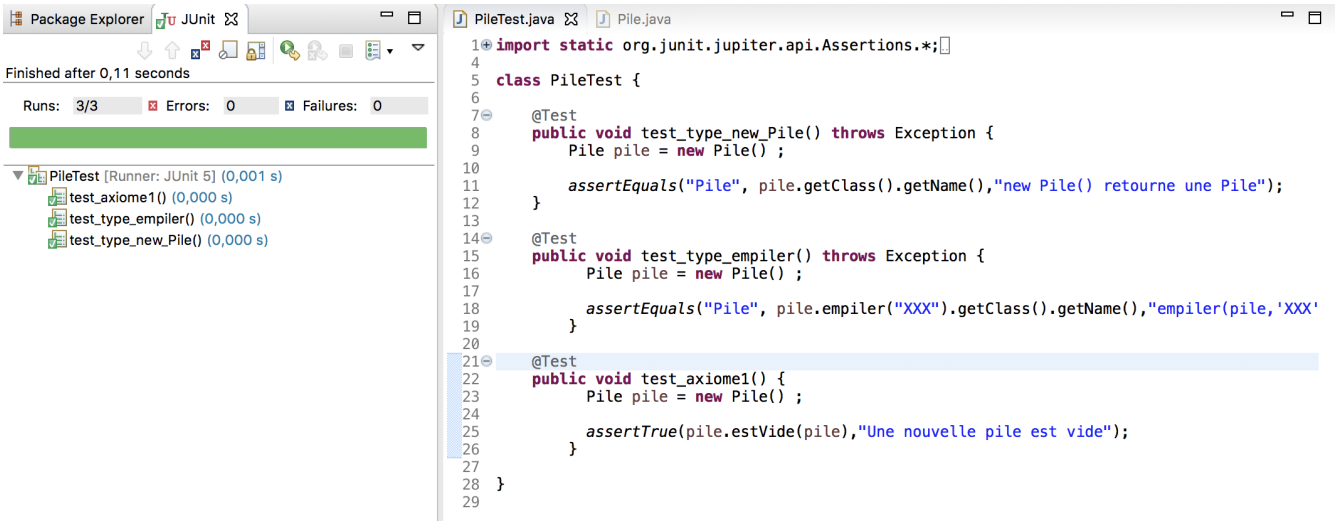

Figure 19. Passage du test

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 20. 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
- 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