Institut Universitaire de Technologie
Aix*Marseille Université

Module M6501

Techniques de programmation avancée

Chapitre 3: Tests unitaires

Hana ALOUAOUI

Hana.ALOUAOUI@univ-amu.fr

Introduction

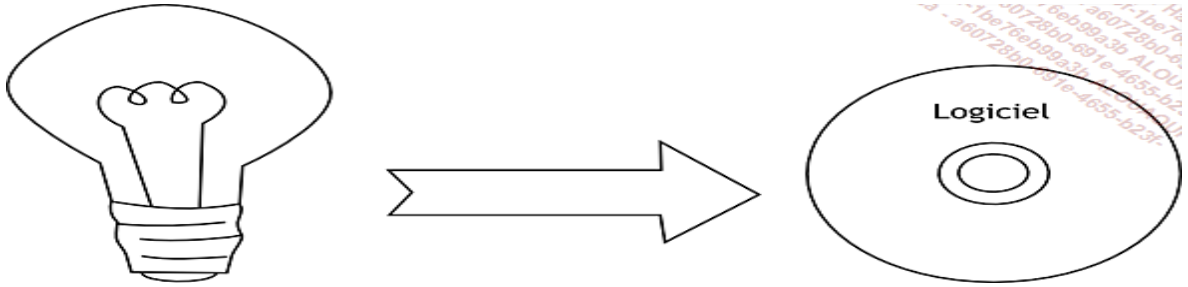
À quoi sert le test ?

- L’objet du Test : le logiciel.

- Le client a besoin de faire quelque chose et le logiciel l’aide à atteindre son but.

→ Le point de départ est donc le besoin client et le point d’arrivée est le logiciel produit.

Processus de fabrication d'un logiciel



Dans un monde parfait → tester est une perte de temps.

Le processus de fabrication d'un logiciel pourrait être décrit ainsi :

Recueillir les besoins → Imaginer → Écrire → Compiler → Exécuter

Dans un monde parfait, l'erreur n'existe pas. Il est donc inutile de tester.

→Nous ne vivons pas dans un monde parfait
→L'utilité du test est alors de vérifier que ce qui a été fabriqué est conforme à ce qui a été demandé.

Processus de fabrication d'un logiciel

Recueillir les besoins : le client exprime ses besoins, ses objectifs.

→ Le premier risque est que le client se trompe dans ce qu'il veut.

Imaginer : De toutes les étapes mentionnées, c'est la moins automatisable.

Les risques d'erreur:

- Mauvaise compréhension du besoin.
- Élaboration d'une solution valide sur le papier mais impossible à réaliser.

Processus de fabrication d'un logiciel

Écrire : la structuration de l'idée sous une forme standardisée.

→Le logiciel peut être écrit directement à l'aide du code.

Les erreurs possibles:

- Mauvaise compréhension de l'idée d'un autre.
- Difficultés à formaliser ses propres idées.
- Non respect du standard.

Processus de fabrication d'un logiciel

Compiler : transformer le code compréhensible par un humain en un langage compréhensible par une machine.

→ Le risque devient de plus en plus faible, mais il existe toujours.

- Différentes plates-formes d'exécution.
- Différents outils de compilation.
- Diverses bibliothèques de base.

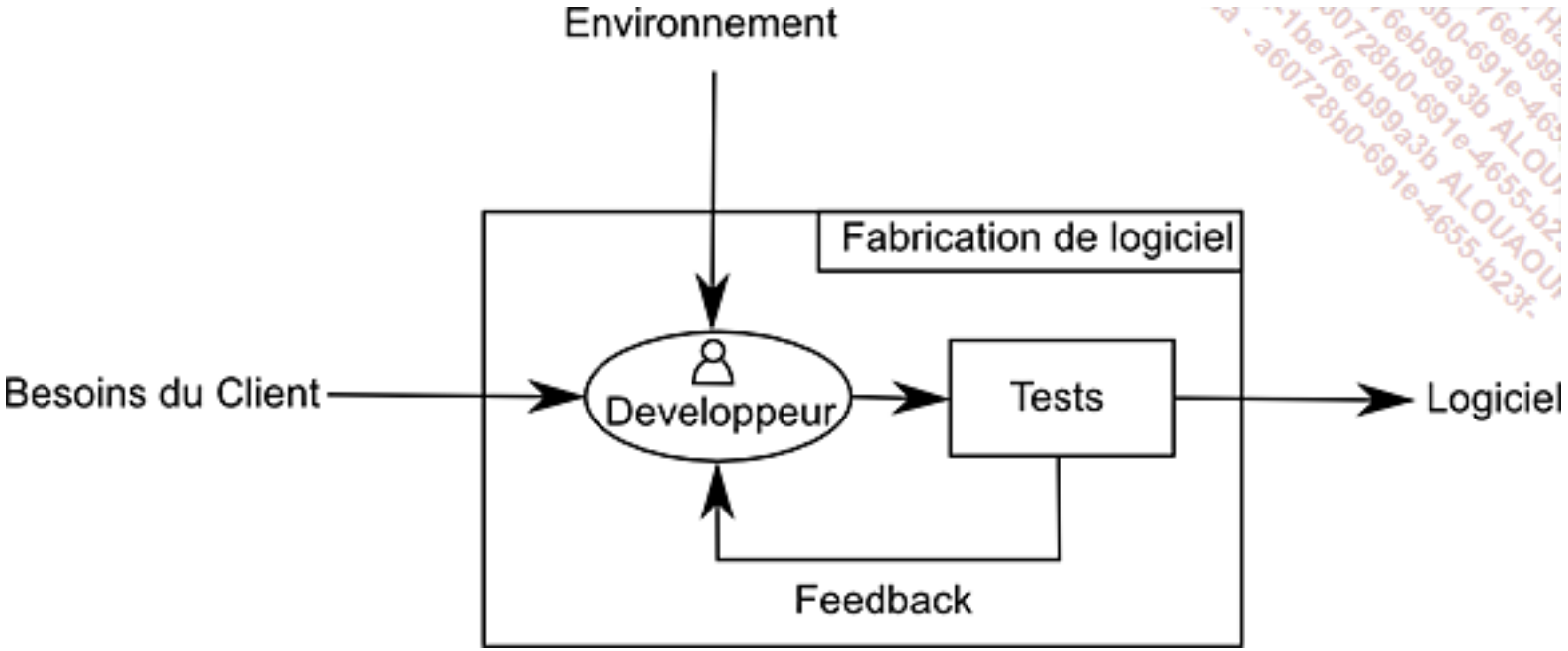
Processus de fabrication d'un logiciel

Exécuter : les risques d'erreurs propres à cette phase deviennent de plus en plus faibles.

Durant cette phase, toutes les erreurs non corrigées dans les phases précédentes deviendront visibles :

- Un besoin client mal implémenté.
- Un défaut de conformité aboutissant à un plantage.
- Tout autre comportement anormal ayant des conséquences plus ou moins graves ou visibles.

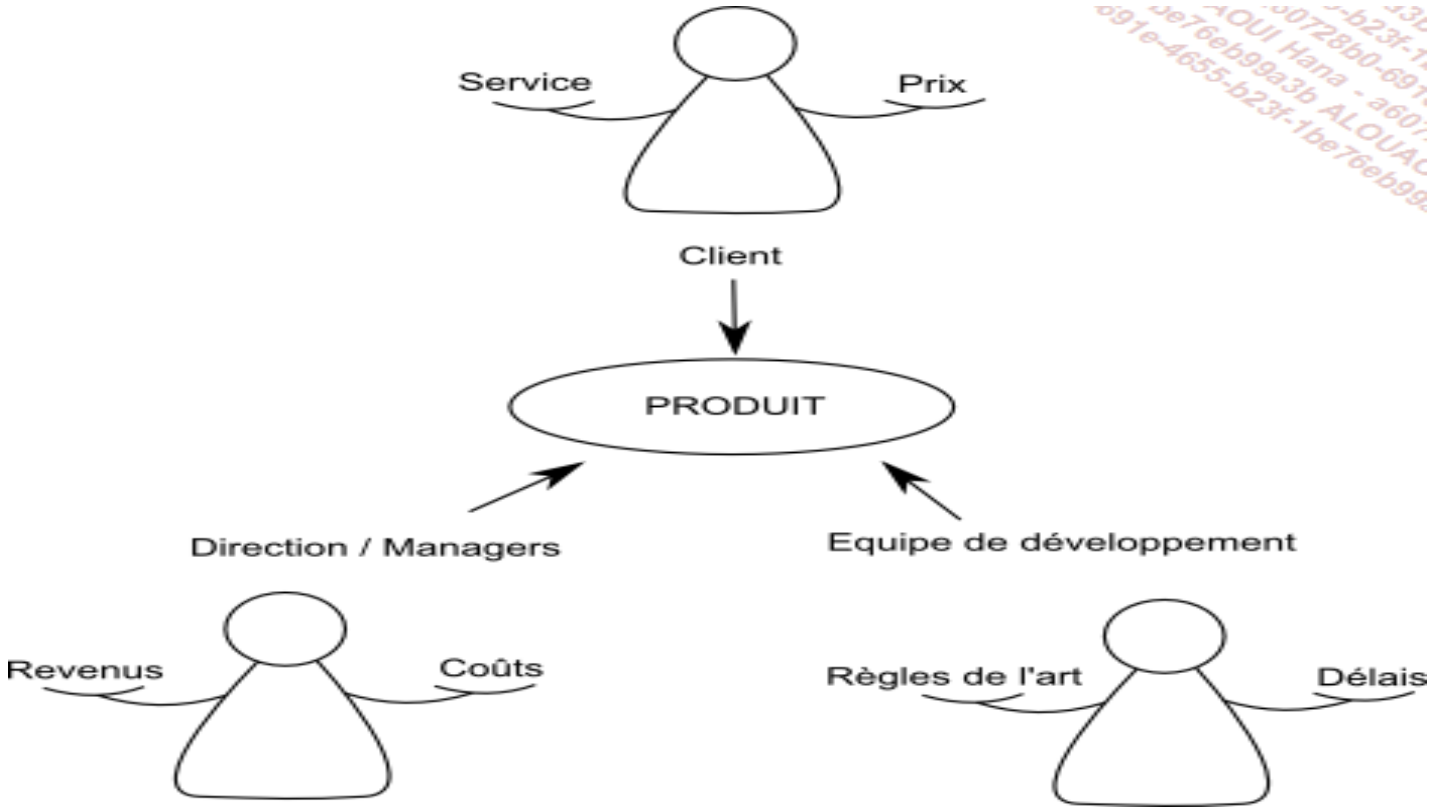
Processus de fabrication d'un logiciel



Le test permet une boucle de rétro-actions et donne du feedback sur la qualité de ce qui est produit.

Concept de qualité

Différents points de vue



Concept de qualité

Le client

- Le client est le point de départ qui définit le besoin.
- Sa perception de la qualité se fera à travers la recherche d'un point d'équilibre entre le service rendu et le prix du logiciel.
- Plus le service sera adapté et de bonne qualité, plus le prix pourra être élevé.

Concept de qualité

La direction / Les managers

- Ils ont la responsabilité de la pérennité de l'entreprise.
- Ils doivent prendre des décisions basées sur les coûts et revenus de chaque projet avec un objectif vital de rentabilité.
- Leur perception de la qualité se fera au travers du prisme financier : que coûte-t-elle/que rapporte-t-elle.

Concept de qualité

L’équipe de développement

- Tous ceux qui contribuent à l’élaboration du produit.
- La qualité du produit est en liaison directe avec la qualité de leur travail.
- Le développeur devra trouver un point d’équilibre entre le respect des règles de l’art qui sont son référentiel de « bonne qualité » et le respect des délais du projet.

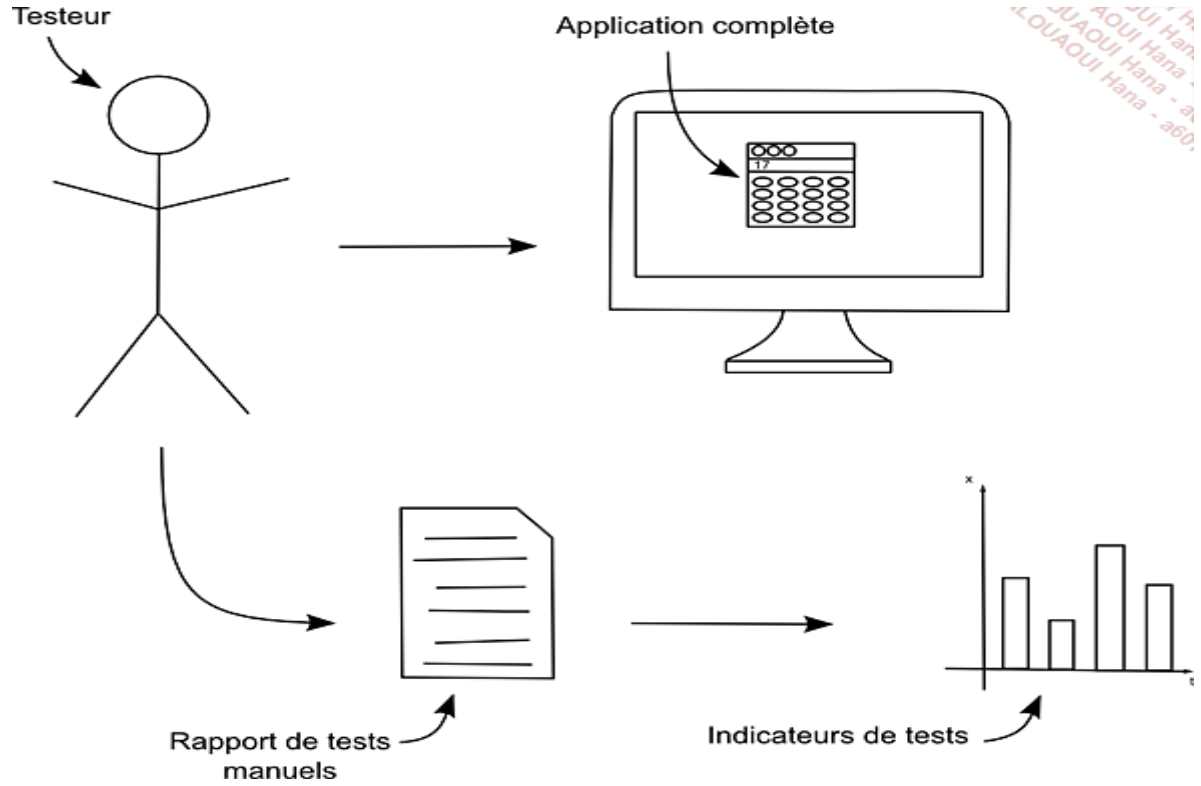
Concept de logiciel assez bon

- La qualité d'un logiciel = compromis entre les différentes parties prenantes d'un projet.
- Risque d'impact de chaque défaut de conformité.
- À moins de travailler sur un système dans lequel la vie humaine est impliquée, comme dans l'aéronautique ou le nucléaire, la règle du logiciel juste assez bon est la généralité
- Un logiciel à zéro défaut est difficile à fabriquer et à tester.

Les tests manuels et automatiques
Les tests manuels

- Les tests manuels nécessitent une personne pour être exécutés.
- Le test consiste à exécuter l'application, à interagir avec et à vérifier le comportement attendu via un contrôle humain.
- Le résultat est consigné dans une fiche de tests.
- Les différentes fiches de tests constituent le plan de tests.
- Une exécution de ce plan de tests donne un rapport de tests.

Les tests manuels et automatiques
Les tests manuels



→ Ces tests sont plus ou moins longs à exécuter.
→ Le temps total dépend bien sûr du test voulu, mais l'unité de temps est en général la minute.

Les tests manuels et automatiques
Les tests manuels

Exemple: Calculatrice:

- Dans l'exemple de la calculatrice, la fiche de test suivante permet de vérifier l'opération d'addition.

Titre: Test d'addition

1. Lancer l'application
2. Vérifier que l'afficheur indique '0'
3. Cliquer sur la touche 1
4. Vérifier que l'afficheur indique '1'
5. Cliquer sur la touche '+'
6. Vérifier que l'afficheur indique '+'
7. Cliquer sur la touche 3
8. Vérifier que l'afficheur indique '3'
9. Cliquer sur la touche '='
10. Vérifier que l'afficheur indique '4'

Les tests manuels et automatiques
Les tests automatiques

- Par opposition aux tests manuels, les tests automatiques sont exécutés par un automate.
- Cela prend le plus souvent la forme d'un programme dont la mission est d'exécuter la batterie de tests.
- Elle est en général programmée par les développeurs.
- Le résultat de la batterie de tests est ensuite consigné dans un rapport de tests automatiques.

Les tests manuels et automatiques
Les tests automatiques

- L'ordre de grandeur du temps d'exécution d'un test est celui de la seconde.
- Une fois écrits, les tests peuvent être exécutés à l'infini pour un coût négligeable

→ l'exécution d'une série de tests ne prend que quelques secondes.

- Erreur détectée→ une modification est nécessaire, soit dans le code soit dans les tests.
- Les résultats sont binaires : un test automatique est réussi ou échoué.

Les tests FIRST

a. Fast : Rapides

Les tests doivent être rapides.

S’ils ne le sont pas, ils ne sont pas lancés fréquemment. Le risque est alors qu’un ou plusieurs d’entre eux finissent par ne plus passer.

Les tests FIRST

b. Independant : Indépendants

Les tests doivent être indépendants les uns des autres et pouvoir être lancés dans n'importe quel ordre.

Les tests FIRST

c. Repeatable : Répétables

Les tests doivent pouvoir tourner dans n'importe quel environnement.

Il faut donc être vigilant aux pré-requis et s'assurer que ces derniers sont documentés.

Les tests FIRST

d. Self validating : Auto-validant

Le résultat d'un test est binaire. Il passe ou ne passe pas.

Une batterie qui passe retourne un OK et le nombre de tests exécutés.

Les tests FIRST

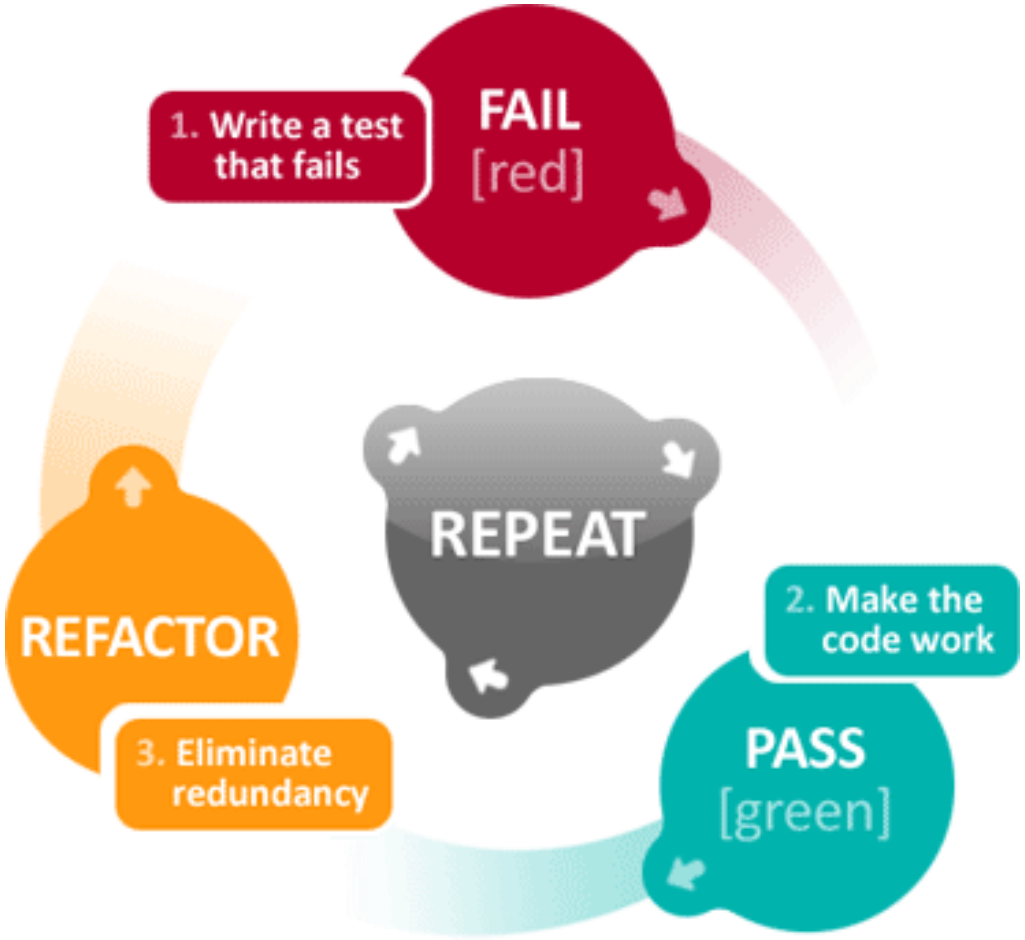
e. Timely : Au bon moment

Ecrire le Test Avant / Après le code..

TDD-Développement dirigé par les tests
Cycle de TDD

1. Ecrire un premier test
2. Vérifier qu'il échoue (car le code qu'il teste n'existe pas)
3. Ecrire le code suffisant pour passer le test
4. Vérifier que le test passe
5. Réviser le code (refactoring), i.e. l'améliorer tout en gardant les mêmes fonctionnalités

TDD-Développement dirigé par les tests



JUnit

JUnit offre les caractéristiques indispensables d'une plate-forme de test automatique :

- Il intègre la notion de résultat binaire : un test passe ou ne passe pas.
- Lire le résultat du test ne nécessite pas de validation à posteriori puisqu'il suffit de le lire à l'écran.
- Les tests peuvent être regroupés en suites de tests permettant l'exécution de plusieurs tests.
- Le rapport de tests est présenté dans un format structuré. Chaque test qui échoue indique à quel endroit il a échoué.

Junit / JUnit 4

- intégré à Eclipse

Principe

- Une classe de tests unitaires est associée à une autre classe
- Une classe de tests unitaires hérite de la classe **junit.framework.TestCase**

  pour bénéficier de ses méthodes de tests
- Les méthodes de tests sont identifiées par des annotations Java

Définitions de tests en JUnit

Méthodes de tests

- Nom quelconque
- visibilité public, type de retour void
- pas de paramètre, peut lever une exception
- annotée @Test
- utilise des instructions de test

Annotations

Les annotations sont à placer avant les méthodes d'une classe de tests unitaires

Annotation	Description
`@Test`	méthode de test
`@Before`	méthode exécutée *avant chaque test*
`@After`	méthode exécutée *après chaque test*
`@BeforeClass`	méthode exécutée *avant le premier test*
`@AfterClass`	méthode exécutée *après le dernier test*
`@Ignore`	méthode qui ne sera pas lancée comme test

Instructions de Test

Instruction	Description
`fail(String)`	fait échouer la méthode de test
`assertTrue(true)`	toujours vrai
`assertsEquals(expected, actual)`	teste si les valeurs sont les mêmes
`assertsEquals(expected, actual, tolerance)`	teste de proximité avec tolérance
`assertNull(object)`	vérifie si l'objet est `null`
`assertNotNull(object)`	vérifie si l'objet n'est pas `null`
`assertSame(expected, actual)`	vérifie si les variables référencent le même objet
`assertNotSame(expected, actual)`	vérifie que les variables ne référencent pas le même objet
`assertTrue(boolean condition)`	vérifie que la condition booléenne est vraie

Exemple

- Dans l'exemple de la calculatrice, considérons la classe Addition. Celle-ci définit deux méthodes **calculer et lireSymbole** :

```
package math;

public class Addition {
    public Long calculer (Long a, Long b) {
        return a+b;
    }
    public Character lireSymbole() {
        return '+';
    }
}
```

Exemple

- Il est possible d'écrire une classe de tests dédiée à la classe Addition dont la mission est de valider son comportement.

- Celle-ci intègre deux tests qui utilisent JUnit :

- Le premier test, testCalculer, vérifie que le résultat de la méthode calculer ayant pour paramètre 1 et 3 retourne 4.
- Le second test, testLireSymbole, vérifie que son symbole est le caractère '+'.

Exemple

```
public class AdditionTest {
   protected Addition op;

   @Before
   public void setUp() {
      op = new Addition();
   }

   @Test
   public void testCalculer() throws Exception {
         assertEquals(new Long(4), op.calculer(new Long(1),new Long(3)));
   }

   @Test
   public void testLireSymbole() throws Exception {
         assertEquals((Character)'+', op.lireSymbole());
   }
}
```

Exécution des méthodes de tests

Appel des tests

En ligne de commande :

java org.junit.runner.JUnitCore TestClass1 [...other test classes...]

Depuis un code Java :

org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...) ;

Depuis Eclipse :

Run > Run As > JUnit test