Test Unitaires

Introduction

- Méthode pour tester individuellement les unités d'un logiciel
  - Plus petit élément pouvant être isolé
  - Module, classe, méthode…
- Technique de test *white box*
  - Le testeur connaît l'implémentation de ce qu'il teste
- En général premier niveau de test
- Utilisé en *Test Driven Development*

Intérêts

- Vérifier qu’une unité répond aux spécifications
  - Confiance dans le code
  - Détecter les problèmes en amont
- Forcer la modularité
  - Le code doit être découpé en unités
- Faciliter le debuggage
  - Erreur localisable plus facilement

Comment

- Utiliser un framework propre au langage
- Identifier les morceaux à tester
  - Idéalement tout le logiciel doit être couvert (*code coverage)*
- Automatiser l’exécution des tests
  - Tout test devant être lancé à la main ne le sera pas

JUnit

Principes

JUnit

- Framework pour l'écriture et l'exécution de tests unitaires en Java
  - Alternative : TestNG
- Projet OpenSource (https://junit.org/junit5/)
- Utilisation d'annotations Java
  - @Test
- Version 5 un peu différente
  - **JUnit 5 = *JUnit Platform* + *JUnit Jupiter* + *JUnit Vintage***
  - Abstraction du moteur de test
  - Nécessite Java >= 8

Vocabulaire

- *Test Class :* une classe contenant au moins un test
  - pas abstract et avec un unique constructeur
- *Test Method* : méthode d'instance annotée avec *@Test*, *@RepeatedTest*, *@ParameterizedTest*, *@TestFactory*, ou *@TestTemplate*.
- *Lifecycle Method* : méthode annotée avec @BeforeAll, @AfterAll, @BeforeEach, or @AfterEach.

- Les méthodes ne doivent rien retourner et ne pas être *private*

Exemple simple

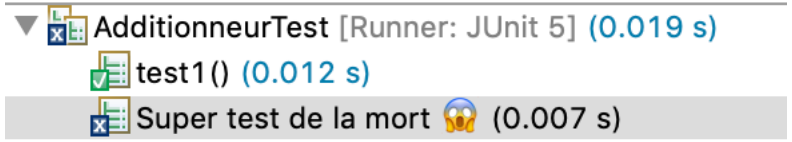
- Un test ne doit pas forcément tester…

```java
package fr.unice.miage;

import org.junit.jupiter.api.Test;

public class SimpleTest {

    public SimpleTest() {}

    @Test
    public void test1() {

    }

}
```

Nommage

- Par défaut un test a le nom de la méthode

- On peut changer son nom avec @DisplayName(" ....")



Assertions simples

- Méthodes statiques de Assertions
- Permettent de tester des égalités, non égalités…
- 2 ou 3 paramètres
  - Ce qui est attendu
  - Ce que le test vient de produire
  - Un message en cas d'échec
- Exemple :
  - *assertEquals(42, 1, « je suis tristesse »)*

```
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.015 s <<< FAILURE! - in fr.miage.tests.FirstJUnit5Tests
[ERROR] assertions  Time elapsed: 0.009 s  <<< FAILURE!
org.opentest4j.AssertionFailedError: je suis tristesse  ==> expected: <42> but was: <1>
        at fr.miage.tests.FirstJUnit5Tests.assertions(FirstJunit5Tests.java:16)
```

Assertions timeout

- Permet de donner une durée limite d'exécution
- Nécessite une référence vers une fonction (lambda)

```
public void test3() {
    Assertions.assertTimeout(Duration.ofMillis(5000), () -> {
        Thread.sleep(6000);});
    }
}
```

Assertions Exceptions

- Assertion AssertThrows
- Possible de tester la levée d'exception
  - Utilisation d'une lambda (fonction anonyme)

```
public void testLeveeException() {
    Assertions.assertThrows(NullPointerException.class, () -> {
        String s = null;
        s.charAt(1);
    });

}
```

Tests multiples

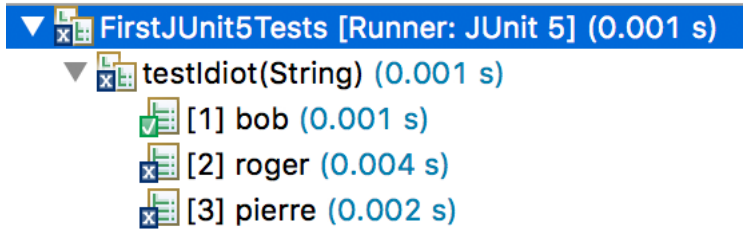
- Tester plusieurs assertions à la suite

```
@Test
public void test4() {
    Additionneur adder = new Additionneur();
    assertAll("additions",
            () -> assertEquals(6, adder.add(1, 5)),
            () -> assertEquals(11, adder.add(10, 1))
        );
}
```

Tests paramétrés

- Comment tester plusieurs paramètres pour un test ?
  - Écrire plusieurs tests
  - Faire une boucle for et plein d'asserts (moins lourd mais quand même)
  - Utiliser assertAll
- Tests parametrés
  - Tests exécutés sur une liste de paramètres

```
@ParameterizedTest
@ValueSource(strings = { "bob", "roger", "pierre" })
void testIdiot(String candidate) {
    assertTrue(candidate.equals("bob"));
}
```



Ordre d’exécution des tests

- Ordre des tests déterministe mais pas évident
  - Deux exécutions successives vont donner le même ordre
  - Mais on ne sait pas lequel !
- Possible de spécifier un ordre avec @TestMethodOrder(OrderAnnotation.class) et @Order(int)

```
@TestMethodOrder(OrderAnnotation.class)
public class AdditionneurTest {
```

```
@Test
@Order(1)
public void test1() {
```

Préparation des tests

- Un test peut nécessiter de la préparation
  - Ouverture/création de fichiers
- Plusieurs annotations possibles
  - @BeforeEach, @BeforeAll
- Possibilité de nettoyer le code après
  - @AfterEach, @AfterAll
- @BeforeAll et @AfterAll sur méthodes *static*

Intégration à Maven

- Placer les tests au bon endroit
  - src/test/java
- Règles :
  - Utiliser le même package pour les tests que le code testé
  - Regrouper les tests dans une classe post-fixée par Test
- Utilisation du plugin surfire de Maven
  - Plugin exécuté dans phase test
- Pensez à indiquer les dépendances

```xml
<dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>${junit.jupiter.version}</version>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-params</artifactId>
        <version>${junit.jupiter.version}</version>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>${junit.jupiter.version}</version>
        <scope>test</scope>
</dependency>
```

https://github.com/junit-team/junit5-samples/blob/r5.3.1/junit5-jupiter-starter-maven/pom.xml