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

Principes

- Un test JUnit se trouve dans une méthode
  - *Test method*
- Plusieurs tests peuvent être regroupés dans une classe
  - *Test class*
  - Cette classe ne sert qu'aux tests
  - Convention : postfixée par *Test*
- JUnit fournit des outils pour vérifier les résultats (assertions)
  - *assert**

Exemple simple

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

import org.junit.jupiter.api.Test;

class FirstJUnit5Tests {

    @Test
    void myFirstTest() {
        assertEquals(2, 1 + 1);
    }

}
```

```
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running fr.miage.tests.FirstJUnit5Tests
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.009 s - in fr.miage.tests.FirstJUnit5Tests
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.793 s
[INFO] Finished at: 2018-09-30T15:09:14+02:00
[INFO] ------------------------------------------------------------------------
```

https://junit.org/junit5/docs/current/user-guide/

Assertions

- Méthodes statiques de org.junit.jupiter.api
- 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'erreur
- 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

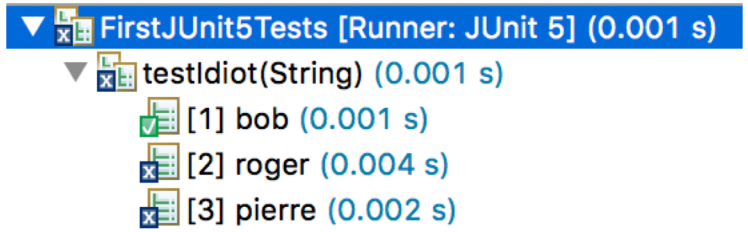
- Possible de tester la levée d'exception
  - Utilisation d'une lambda (fonction anonyme)

```
@Test
void assertThrow() {
    assertThrows(NullPointerException.class, () -> {
        String s = null;
        s.toString();
    });
}
```

Tests paramétrés

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

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



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

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