

Les Formations au dojo 道場

Tests : Les fondamentaux

Durée : 2 jours

Sommaire :

Introduction

1 – Les types de test

2 – Junit & AssertJ (bases)

3 – Junit & AssertJ (intermédiaire)

4 – Bon test / Mauvais test

5 – TDD

Travaux Dirigés

Conclusion

Introduction :

Un test : C'est quoi ?

Dans la vie de tous les jours...

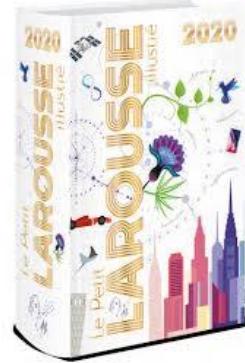


Avant de prendre la route, je vérifie systématiquement l'état de mon véhicule

Définition : [<https://www.larousse.fr>]

test n.m. :

- *épreuve ou expérience décisive*
- *opération témoin permettant de juger*



tester v.t. :

- *soumettre à une épreuve quelconque ; éprouver, expérimenter*
- *soumettre quelque chose à certaines expériences pour vérifier sa valeur, sa résistance ; mettre à l'épreuve*

Mais sinon...pourquoi on teste?





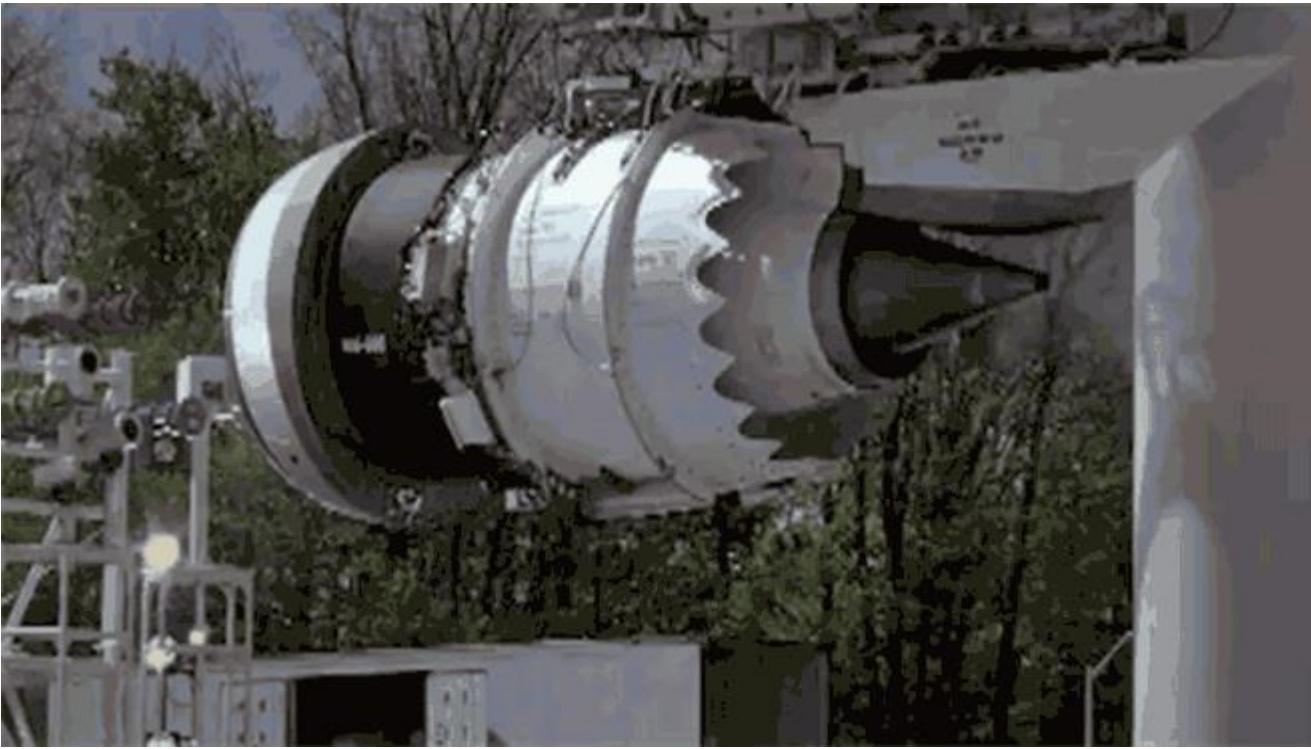




L'erreur informatique la plus couteuse de l'histoire



Pourquoi on teste ?

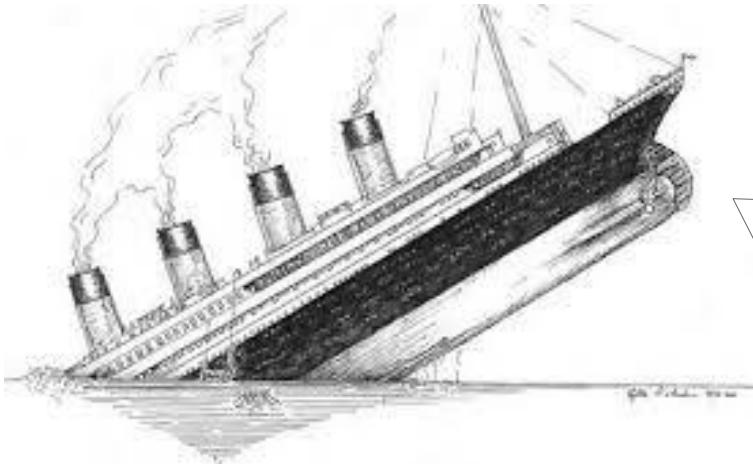




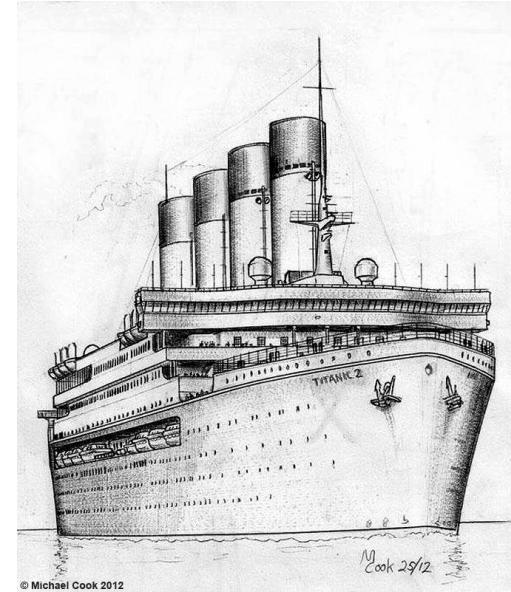
Pourquoi on teste ?

Pour s'assurer que
ça **fonctionne** !

Nous sommes trop
proche de l'eau !



Haaaa...beaucoup
mieux !



```

Double lProvSpecNotUsedAmt = provisionFacMap.values().stream()
    .filter(e -> facilityIdl.equals(e.getKey().getFacilityId()))
    .map(ProvisionFacRecord::getProvSpecNotUsedAmt)
    .filter(Objects::nonNull).mapToDouble(Double::doubleValue).sum();

Double lProvSpecUsedAmt = provisionFacMap.values().stream()
    .filter(e -> facilityIdl.equals(e.getKey().getFacilityId()))
    .map(ProvisionFacRecord::getProvSpecUsedAmt)
    .filter(Objects::nonNull).mapToDouble(Double::doubleValue).sum();

Double lProvSpecRnfIntsAmt = provisionFacMap.values().stream()
    .filter(e -> facilityIdl.equals(e.getKey().getFacilityId()))
    .map(ProvisionFacRecord::getProvSpecRnfIntsAmt)
    .filter(Objects::nonNull).mapToDouble(Double::doubleValue).sum();

Double lProvSpecUnpaidIntsAmt = provisionFacMap.values().stream()
    .filter(e -> facilityIdl.equals(e.getKey().getFacilityId()))
    .map(ProvisionFacRecord::getProvSpecUnpaidIntsAmt)
    .filter(Objects::nonNull).mapToDouble(Double::doubleValue).sum();

Double lProvSpecUnpkAmt = provisionFacMap.values().stream()
    .filter(e -> facilityIdl.equals(e.getKey().getFacilityId()))
    .map(ProvisionFacRecord::getProvSpecUnpkAmt)
    .filter(Objects::nonNull).mapToDouble(Double::doubleValue).sum();

return provSpec = lProvSpecUsedAmt + lProvSpecRnfIntsAmt +
    lProvSpecUnpaidIntsAmt + lProvSpecUnpkAmt
    + (confFlg * lProvSpecNotUsedAmt );

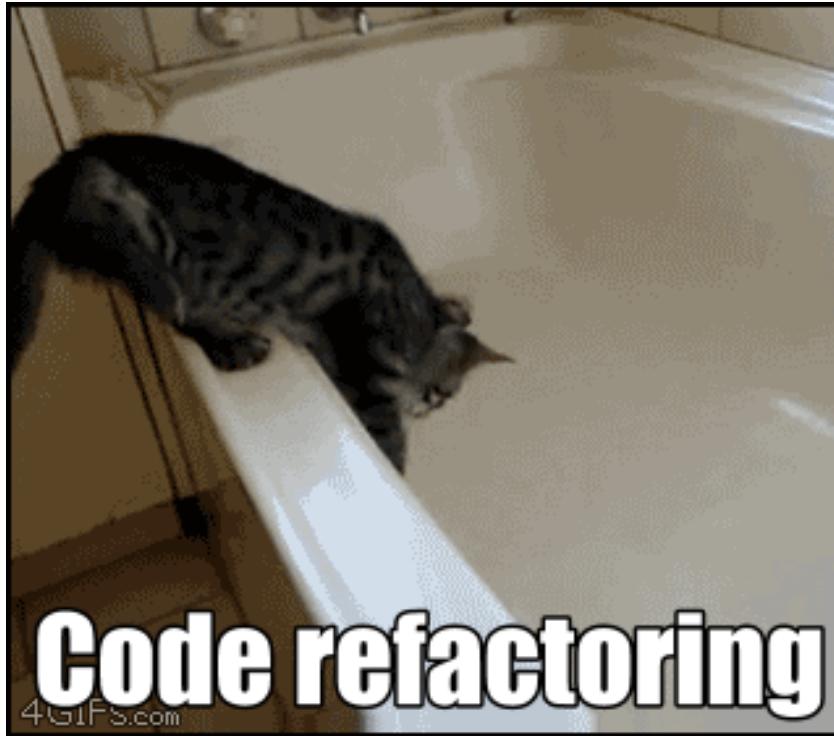
```



```

return provisionFacMap.values().stream()
    .filter(e ->
facilityIdl.equals(e.getKey().getFacilityId()))
    .flatMap(provision -> Stream.of(
        provision.getProvSpecNotUsedAmt(),
        provision.getProvSpecRnfIntsAmt(),
        provision.getProvSpecUnpkAmt(),
        provision.getProvSpecUsedAmt() * confFlg,
        provision.getProvSpecUnpaidIntsAmt()
    )
    .filter(Objects::nonNull)
    .mapToDouble(Double::new)
    .sum();

```



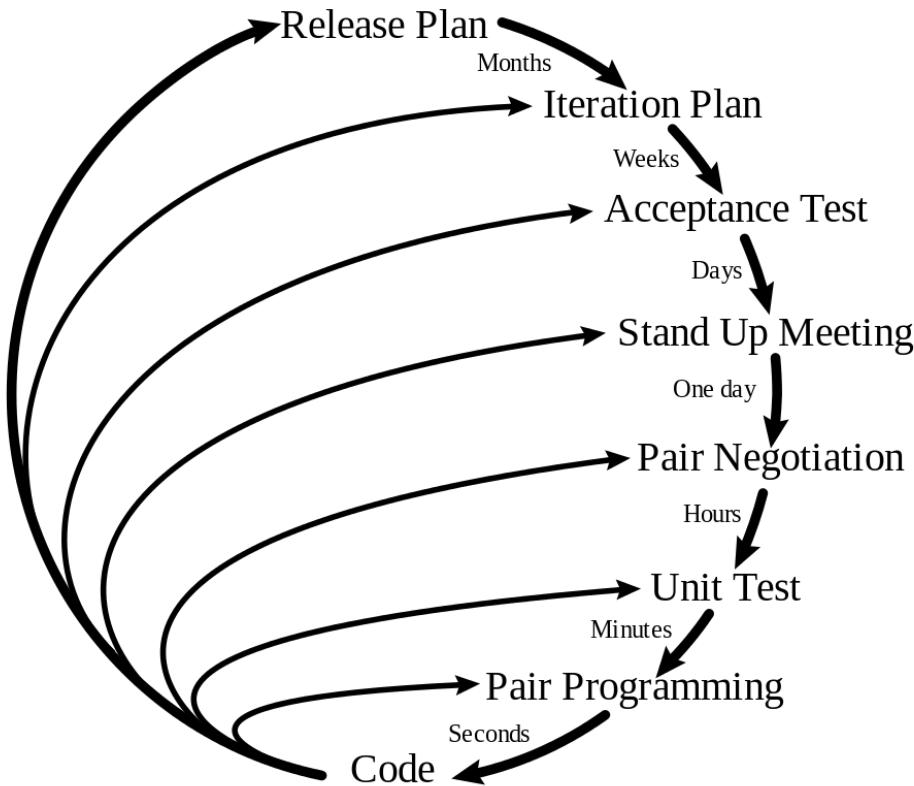




Pourquoi on teste ?

Pour s'assurer que
ça **fonctionne toujours !**

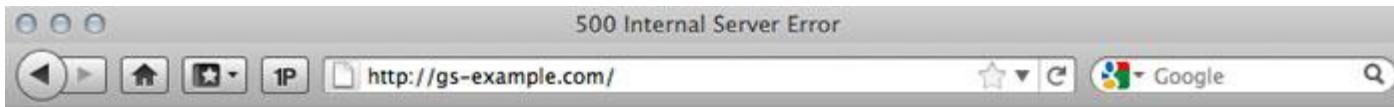






Pourquoi on teste ?

Pour s'assurer rapidement que
ça fonctionne !



Internal Server Error

The server encountered an internal error or misconfiguration and was unable to complete your request.

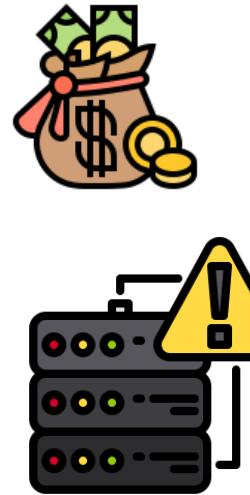
Please contact the server administrator, webmaster@gs-example.com and inform them of the time the error occurred, and anything you might have done that may have caused the error.

More information about this error may be available in the server error log.

Apache/2.0.54 Server at gs-example.com Port 80



Introduction : Pourquoi on teste ?



Partie 1 :

Types de test





Types de tests

Fonctionnels

Cherche à tester les fonctionnalités qui étaient exprimées dans le cahier des charges du projet

Non-fonctionnels

Cherche à tester les aspects non-fonctionnels d'une application

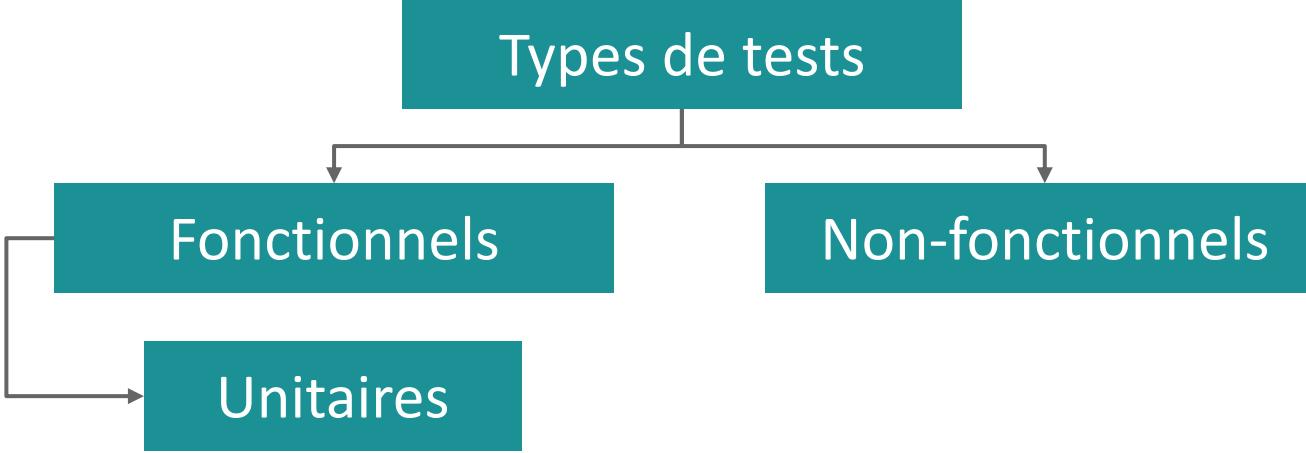


Types de tests

Fonctionnels

Non-fonctionnels

Unitaires



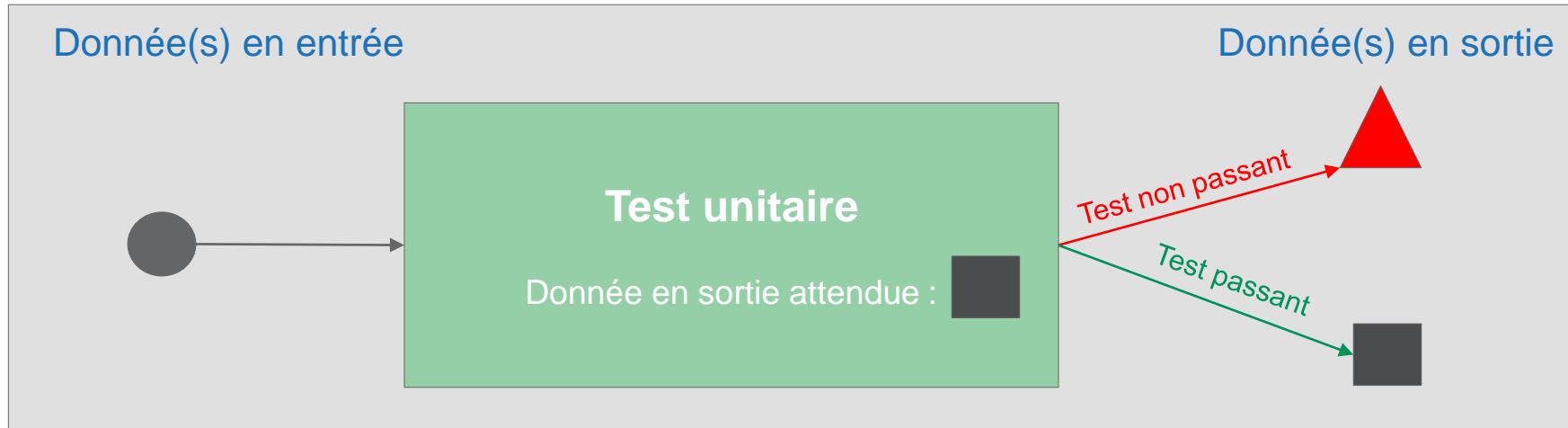
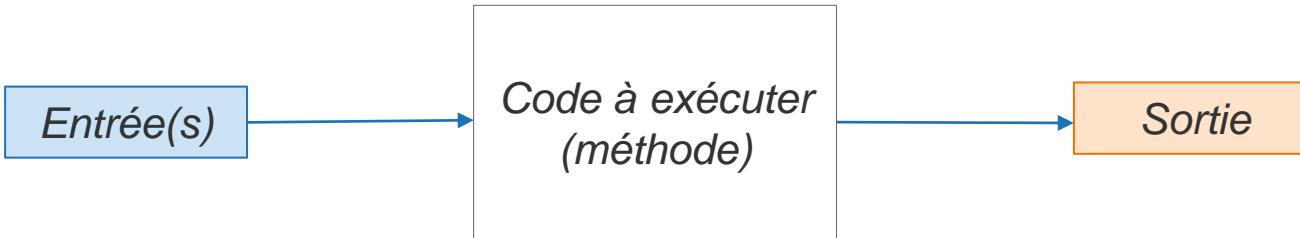


Test unitaire

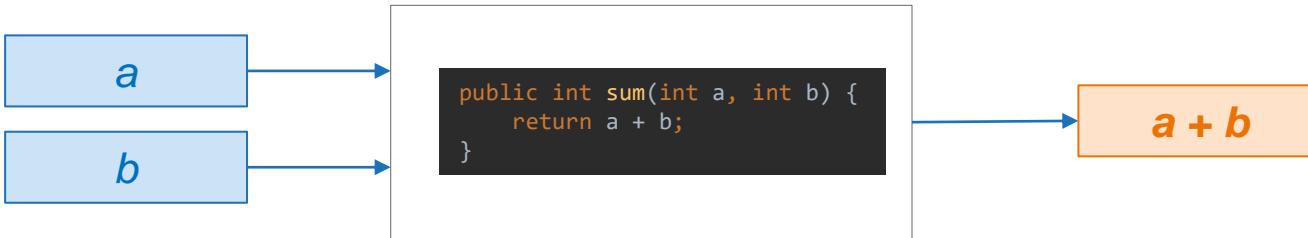
Définition :

Procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module »).

Test unitaire



Test unitaire



```
//given  
int a = 1;  
int b = 2;  
//when  
int c = sum(a,b);  
//then  
if(c == 3) {  
    // OK  
} else {  
    // ERROR  
}
```

Test unitaire

JUnit 5

KARMA

Jasmine



AssertJ

Fluent assertions for java



Hamcrest

jsunit
Bulletproof JavaScript



Types de tests

Fonctionnels

Non-fonctionnels

Unitaires

Intégration



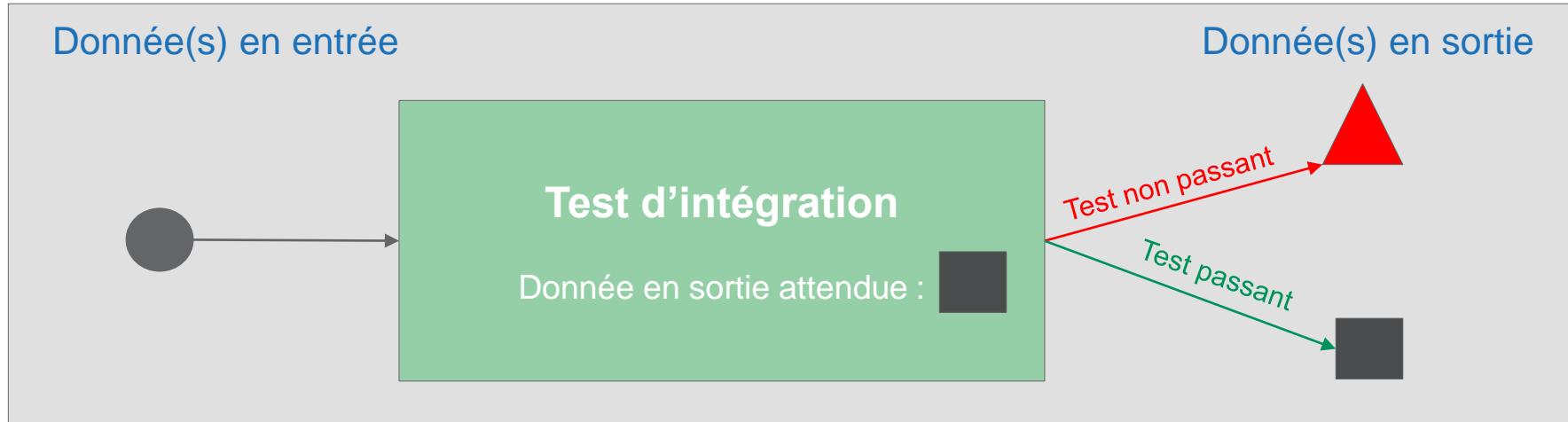
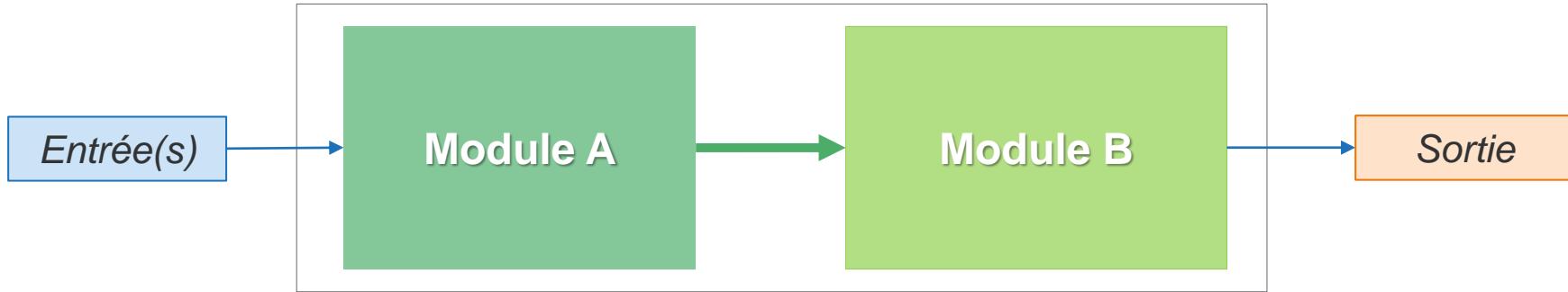


Test d'intégration

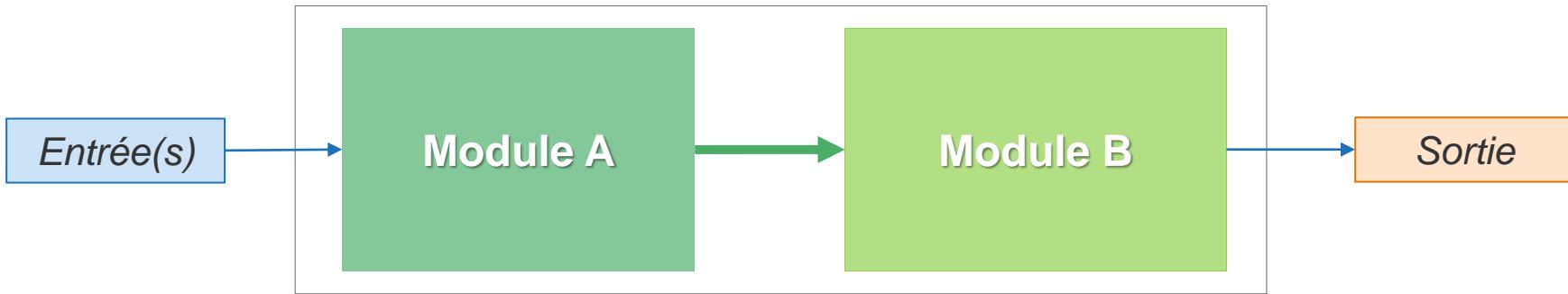
Définition :

Procédure permettant de vérifier le bon fonctionnement de l'action combiné de plusieurs modules.

Test d'intégration



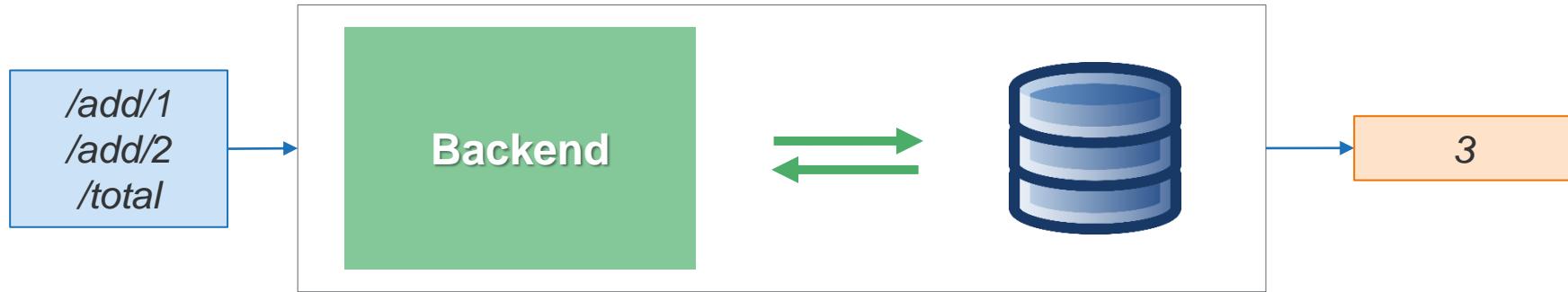
Test d'intégration



UNIT TESTS PASSING



Test d'intégration



Test d'intégration

JUnit 5



AssertJ

Fluent assertions for java

KARMA

Jasmine



Hamcrest

cucumber

Types de tests

Fonctionnels

Non-fonctionnels

Unitaires

Intégration

End-to-end





Test de end-to-end

Définition :

*Procédure permettant de vérifier le bon fonctionnement du système complet, traversant ainsi toutes les couches. Ces tests sont de type **boîte noire**.*



Boîte noire

Les tests de type boîte noire s'effectuent sans aucune connaissance technique. Ils se basent uniquement sur des attentes métier.

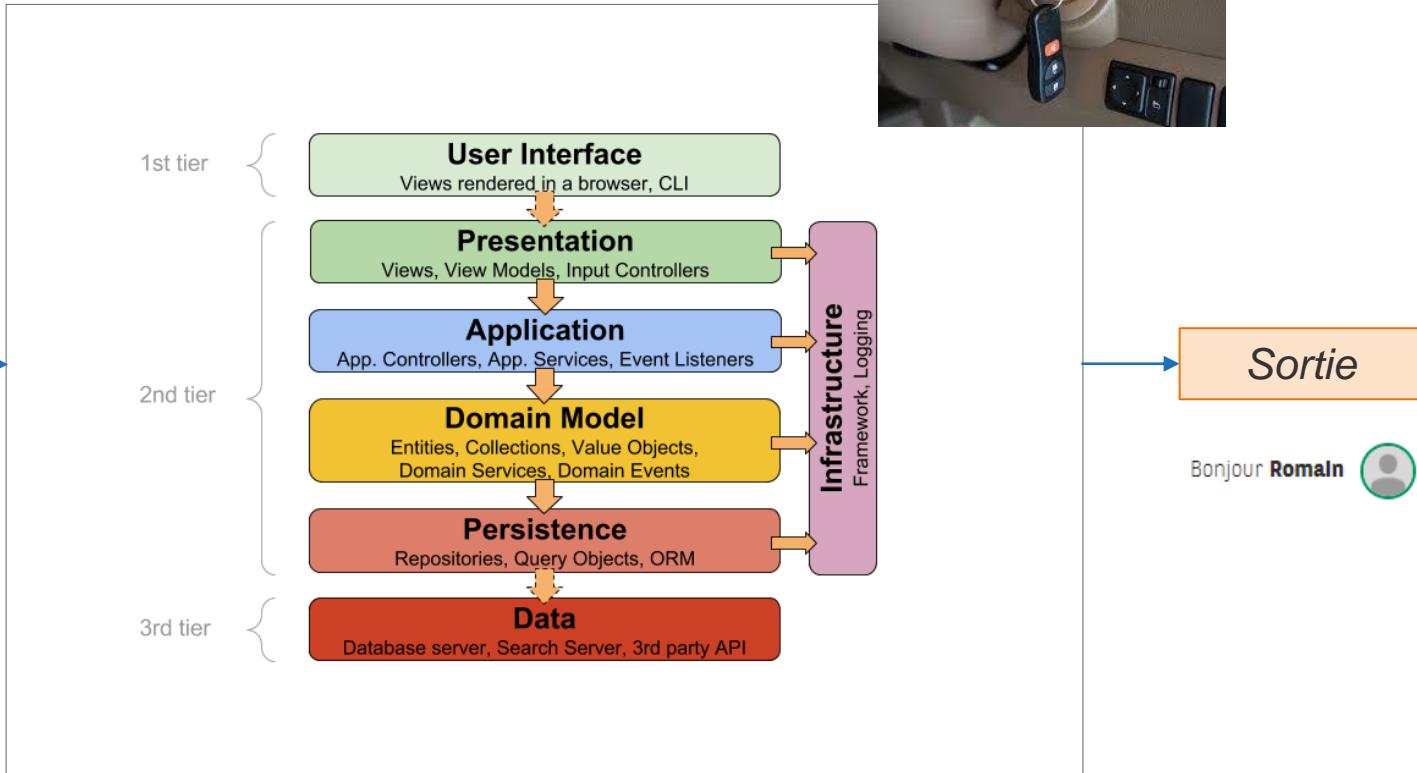
Boîte blanche

Les tests de type boîte blanche s'effectuent en pleine connaissance du code. Ils permettent de tester à un niveau de granularité plus fin mais nécessitent une connaissance de la structure interne de l'application

Test de end-to-end

Entrée(s) →

CONNEXION



Test de end-to-end



Selenium WebDriver





Types de tests

Fonctionnels

Non-fonctionnels

Unitaires

Intégration

End-to-end

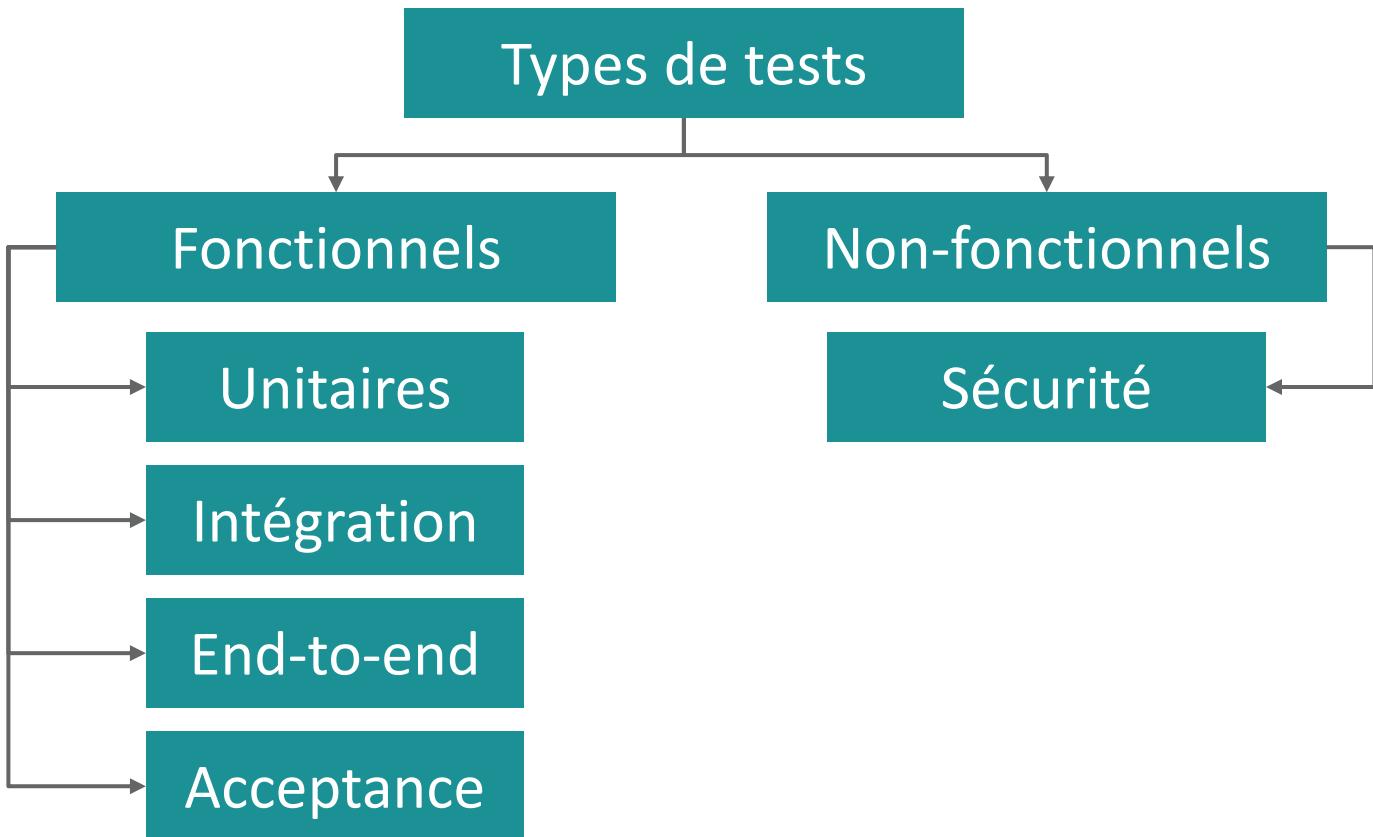
Acceptance



Test d'acceptance

Définition :

Tests manuels effectués par les utilisateurs finaux afin de valider (ou non) le respect des exigences annoncées. Ils nécessitent une expertise métier pour analyser les résultats.



Test de sécurité (intrusion, pénétration...)

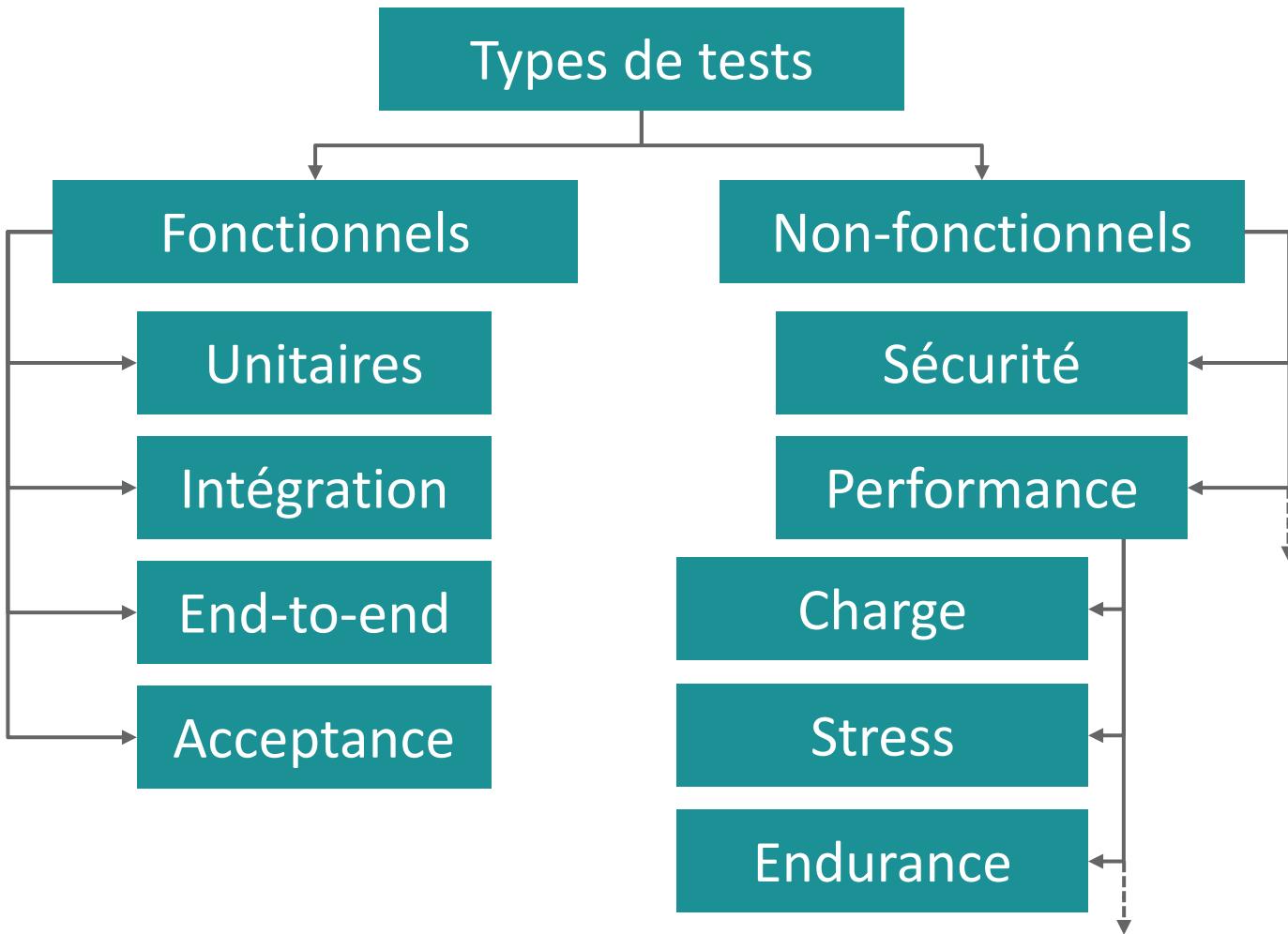


```
INSERT INTO TABLE plates VALUES ('ZU 0666',96,50);
```

```
INSERT INTO TABLE plates VALUES ('ZU 0666',0,0); DROP DATABASE TABLICE;-- ',96,50);
```

Test de sécurité (intrusion, pénétration...)





Test de performance



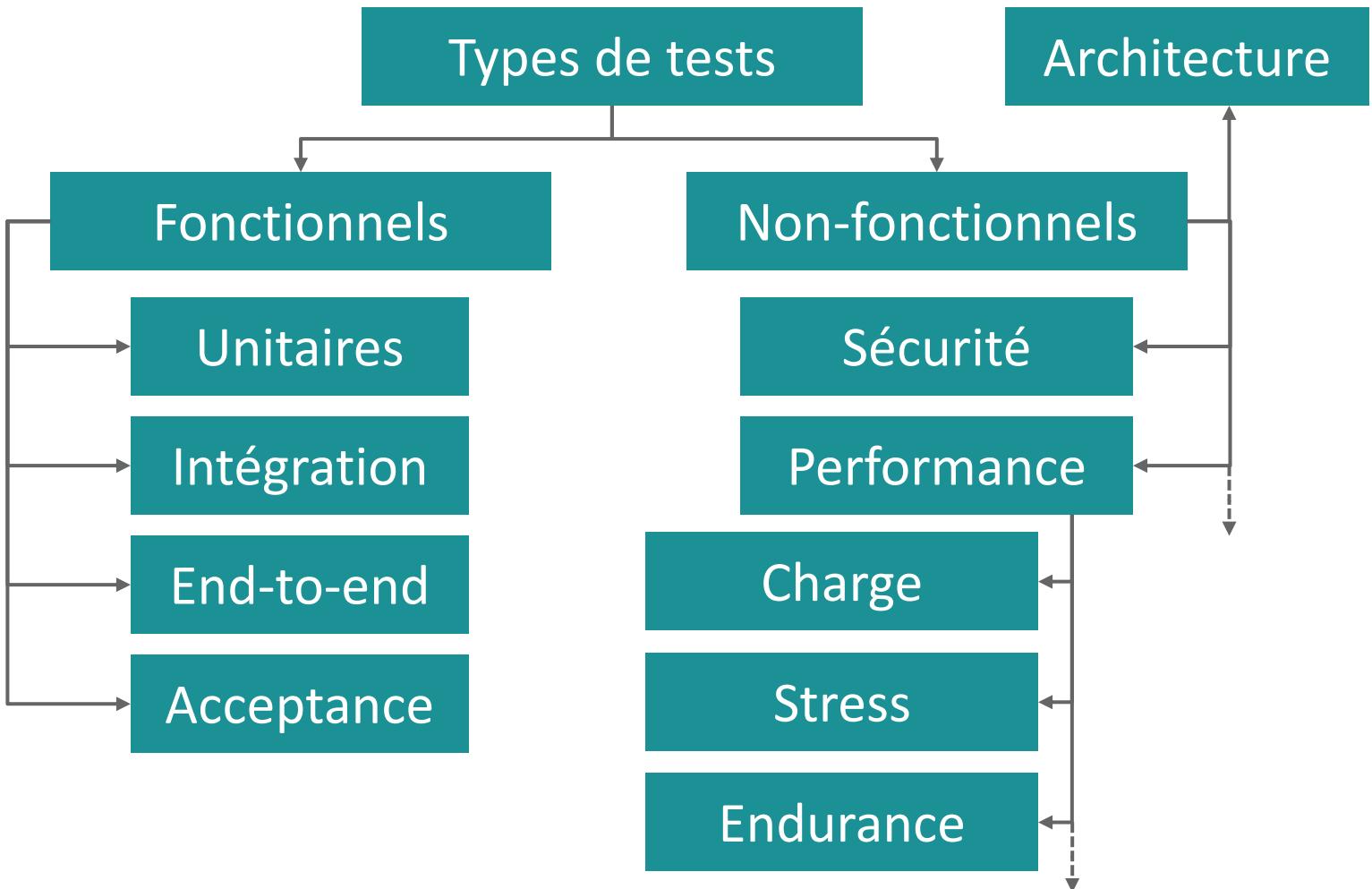


Test de performance

- Combien d'utilisateurs en simultané mon service peut-il supporter ?
- Pendant combien de temps ?
- Que se passe-t-il au-delà de cette charge ?
- En combien de temps mon service répond-t-il ?

Test de performance







Test d'architecture

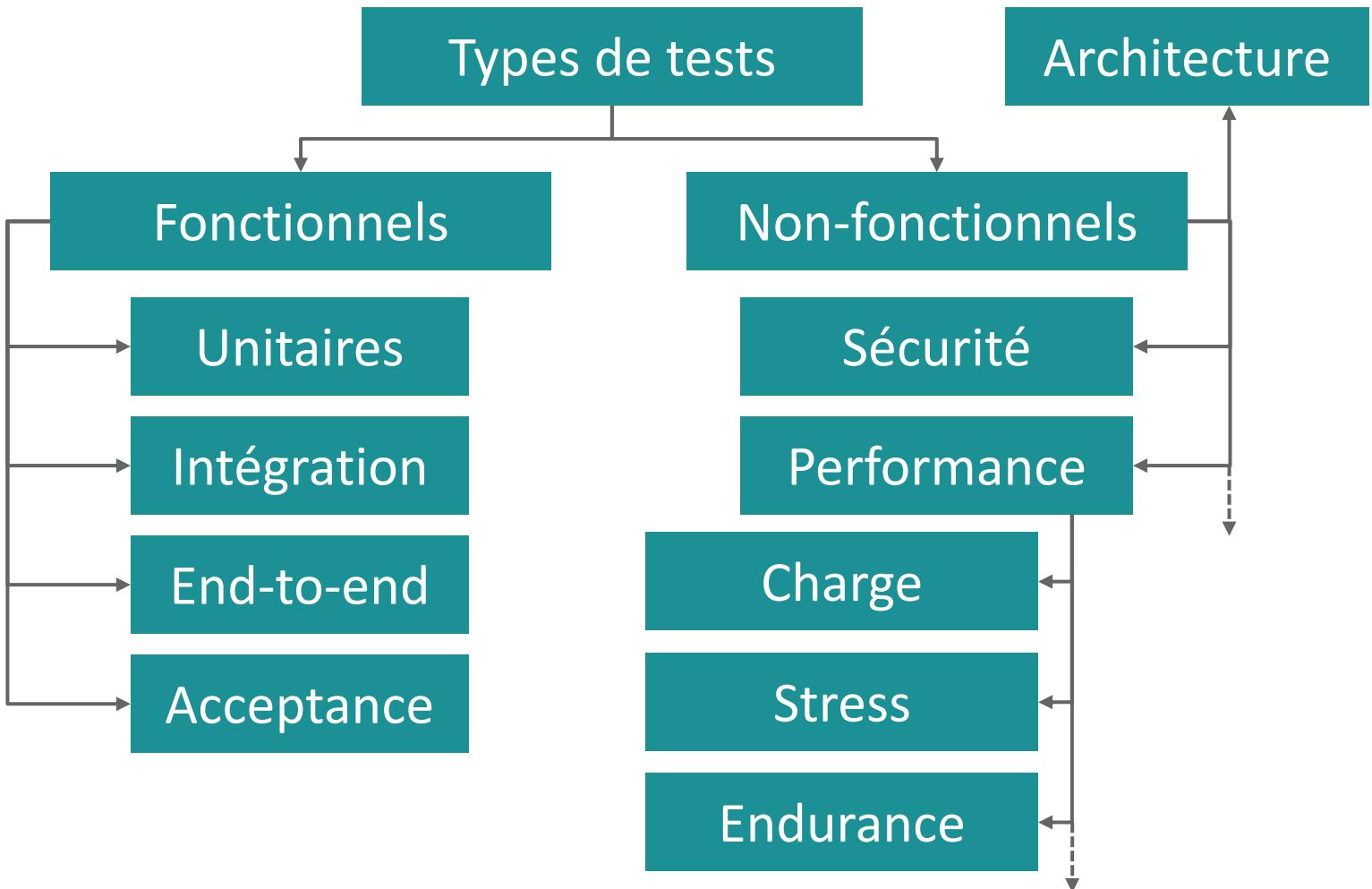
- Les classes sont-elles rangées au bon endroit ?
- Existe-t-il des attributs publics ? Des setters publiques ?
- Les packages sont-ils nommée correctement ?
- ...



Test d'architecture



JUnit The JUnit 5 logo features the word "JUnit" in a large, dark gray sans-serif font. To the right of the text is a circular icon divided diagonally, with "5" written in white on the red half and "5" in green on the green half.



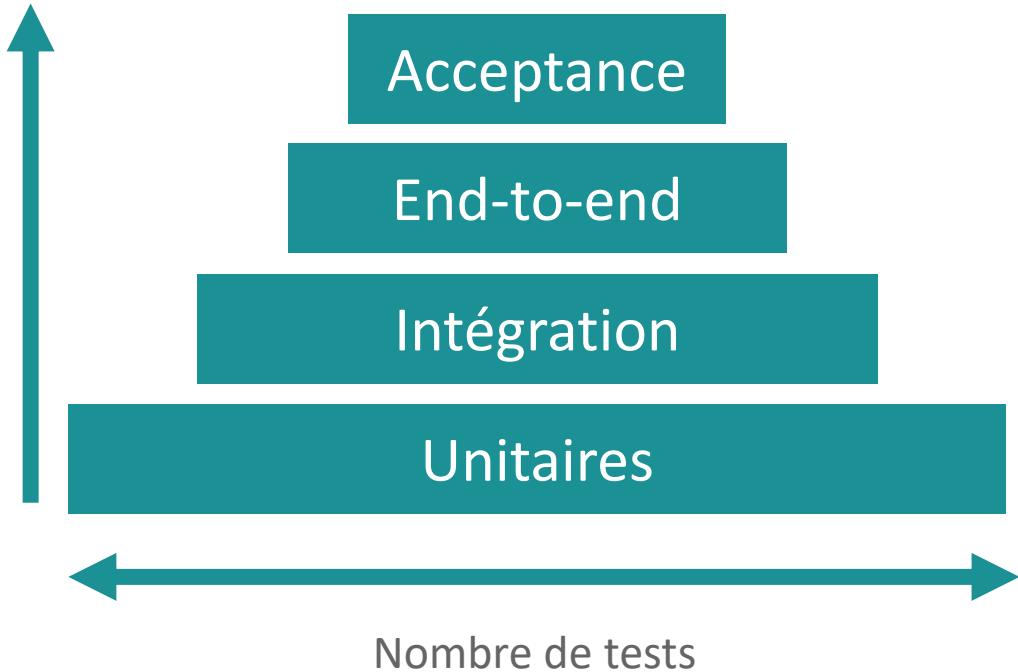
- 
- Fragilité
 - Coût
 - Maintenance
 - Durée
 - Couverture



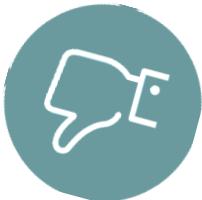
La « pyramide » de test



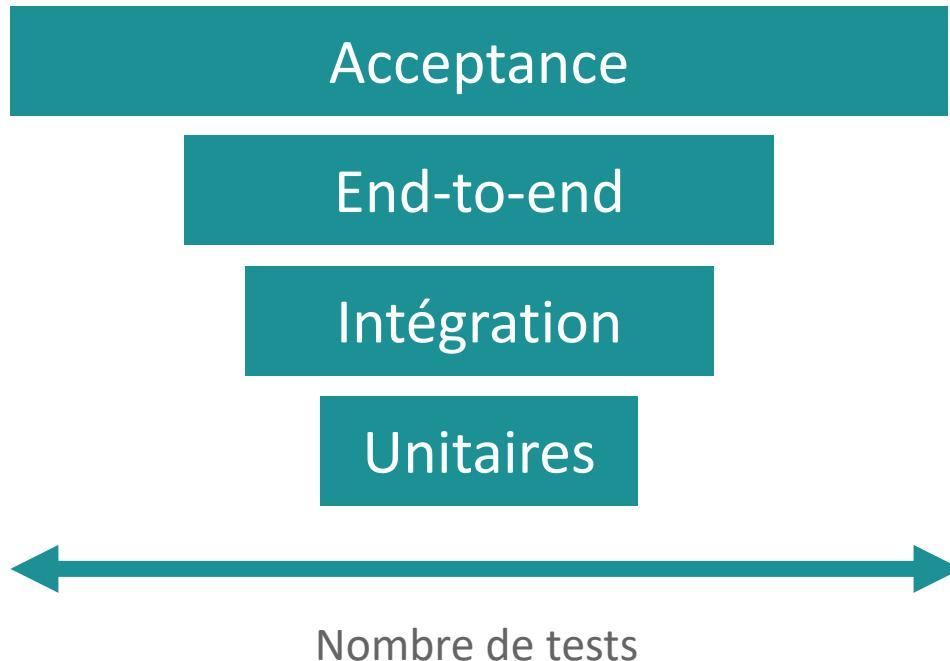
- Fragilité
- Coût
- Maintenance
- Durée
- Couverture



Le « ice cream cone » de test



- On découvre un bug
- On descend
- On cible le code à tester





Types de tests

<https://hackr.io/blog/types-of-software-testing>

Fonctionnels:

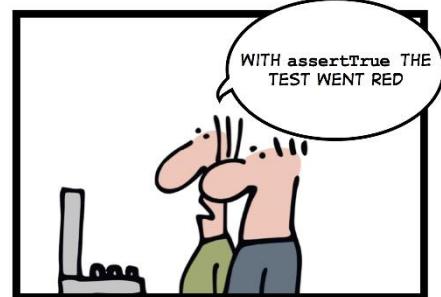
- Unit testing
- Integration testing
- End-to-end testing
- Smoke testing
- Sanity testing
- Regression testing
- Acceptance testing
- White box testing
- Black box testing
- Interface testing

Non-fonctionnels:

- Performance testing
- Security testing
- Load testing
- Failover testing
- Compatibility testing
- Usability testing
- Scalability testing
- Volume testing
- Stress testing
- Maintainability testing
- Compliance testing
- Efficiency testing
- Reliability testing
- Endurance testing
- Disaster recovery testing
- Localization testing
- Internationalization testing

Partie 2 :

Comment est-ce qu'on teste ?



DEVELOPMENT DRIVEN TESTS



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

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.6.2</version>
  <scope>test</scope>
</dependency>
```

<https://assertj.github.io/doc/>

```
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-core</artifactId>
  <version>3.16.1</version>
  <scope>test</scope>
</dependency>
```

AssertJ
Fluent assertions for java



Comment écrire un test ?

- JUnit est un framework de tests unitaires en Java (JavaUnit)
On utilisera la version 5 du framework (en parlant un peu de la 4)
- Il va permettre d'écrire des Tests qui pourront être joués facilement (et rapidement)
 - En ligne de commande
 - Depuis un IDE
 - Avec Maven (depuis une PIC)

Comment écrire un test ?

- Un test est un méthode annotée `@Test`
- Elle de devrait pas avoir de paramètres (pour l'instant)
- La visibilité et le type de retour n'a pas d'importance
- Par convention `maven`, les classes de test se trouvent dans le dossier `src/test/java`

Déclaration d'une méthode de test

```
@Test  
public void should_return3_when_adding1And2() {  
    //...  
}
```



Comment écrire un test ?

- En général, une classe de test, c'est une classe « normale »
- Une classe peut contenir autant de tests que l'on souhaite
- Pour les classes de test on utilisera un nommage :
[ChoseATester]Test
 - MySuperListTest (= test d'une classe)
 - RegistrationProcessTest (= test d'une fonctionnalité)
- Cette convention vient de Maven (et de son plugin Surefire), qui identifie les classes de tests selon le pattern suivant :
 - **/Test*.java (toute classe commençant par Test)
 - **/*Test.java (toute classe terminant par Test)
 - **/*TestCase.java (toute classe terminant par TestCase)



Comment écrire un test ?

- Il existe plusieurs conventions de nommage pour les méthodes de test

Voilà celle que je conseille pour la formation :

should_expectedBehavior_when_stateUnderTest

should_throwException_when_ageLessThan18
should_failToWithdrawMoney_forInvalidAccount
should_failToAdmit_ifMandatoryFieldsAreMissing

- En Java, on commence toujours par une minuscule
- Le but est d'être très expressif ! On parle de **documentation** par le tests

NB : JUnit5 permet également de mettre du texte en plus du nom de la méthode

Comment lancer un test ?

Junit5 utilise sa propre librairie de lancement que l'on peut appeler en ligne de commande :

```
$ java -jar junit-platform-console-standalone-X.X.X.jar <Options>
```



<https://junit.org/junit5/docs/current/user-guide/#running-tests-console-launcher-options>

ou sinon...



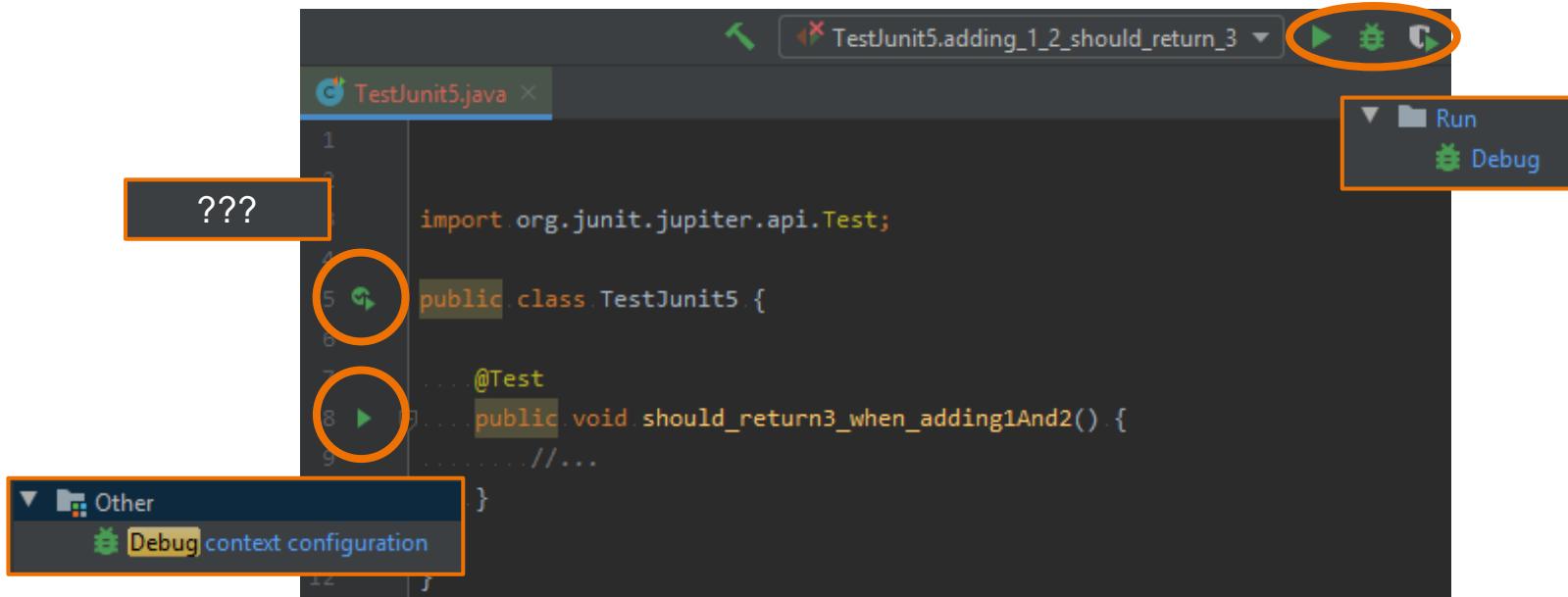
Comment lancer un test ?

Maven permet également de lancer les tests :

```
# Run all the unit test classes.  
$ mvn test  
  
# Run a single test class.  
$ mvn -Dtest=TestApp1 test  
  
# Run multiple test classes.  
$ mvn -Dtest=TestApp1,TestApp2 test  
  
# Run a single test method from a test class.  
$ mvn -Dtest=TestApp1#methodname test  
  
# Run all test methods that match pattern 'testHello**' from a test class.  
$ mvn -Dtest=TestApp1#testHello** test  
  
# Run all test methods match pattern 'testHello**' and 'testMagic**' from a test class.  
$ mvn -Dtest=TestApp1#testHello** +testMagic** test
```

ou sinon...

Comment lancer un test ?



Bon...pour l'instant, le test ne fait rien !

Anatomie d'un test



Comment écrire un test ?

A.A.A

Arrange

Given

En tant que

setup

Act

When

je veux que

execution

Assert

Then

Afin de

assertion



Comment écrire un test ?

Mise en place

Mise en place du contexte du test, création des objets, des mocks...

Exécution

Exécution du code qu'on veut tester

Vérification

Vérification (assertion) du résultat



Comment écrire un test ?

Mise en place

J'ai deux variables : $a=1$ et
 $b=2$

Exécution

J'appelle la méthode `add(a,b)`

Vérification

Elle me retourne 3

Comment écrire un test ?

Mise en place

Exécution

Vérification

```
@Test  
public void should_return3_when_adding1And2() {  
    int a = 1;  
    int b = 2;  
    int sum = add(1,2);  
    assertThat(sum).isEqualTo(3);  
}
```



Comment écrire un test ?

Définition :

Assertion n.f. : Proposition que l'on avance et que l'on soutient comme vraie.

e.g : La somme de 1 et 2 fait 3



Comment écrire un test ?

- Si l'assertion est vérifiée, alors le test est passant (vert)
- Sinon, le test est non-passant (rouge)

Il n'existe pas d'autre possibilité que
vert/rouge
(du moins dans un test unitaire)



Comment écrire un test ?

- JUnit dispose de sa propre API pour faire des assertions mais nous utiliserons AssertJ pour la suite de la formation
- Les deux frameworks couvrent un même périmètre fonctionnel
- AssertJ propose un langage plus « humain »
- Junit est un peu plus performant (surtout pour les soft assertions)



AssertJ

vs.

JUnit

```
Assertions.assertThat(sum)  
    .isEqualTo(3);
```

```
Assertions.assertEquals(sum, 3);
```

```
Assertions.assertThat(arrayList)  
    .hasSize(3)  
    .startsWith("Hello")  
    .containsOnlyOnce("world");
```

```
Assertions.assertEquals(arrayList.size(), 3);  
Assertions.assertTrue(arrayList.get(0).startsWith("Hello"));  
Assertions.assertTrue(/*code compliqué*/);
```

Comment écrire un test ?

En AssertJ, une assertion s'écrit comme suit :

```
Assert.assertThat(actual)  
    .method(arguments)
```

Type d'assertion
(ici assertion simple)

Les méthodes
disponibles dépendent
du type de l'objet à
tester

Objet ou type primitif
à tester

Comment écrire un test ?

Il existe beaucoup de méthodes permettant de faire des assertions. Le plus simple est de s'aider de l'auto-complétion de son IDE pour visualiser l'ensemble de méthodes disponibles.

The screenshot shows a code editor with Java code. A tooltip is open over the method call `Assertions.assertThat(anInt)`. The tooltip lists many static methods available on the `Assertions` class, each with a brief description and a 'capture of ?' placeholder. The methods listed include `isEqualTo`, `isBetween`, `info`, `closeTo`, `greaterThan`, `lessThan`, `negative`, `notCloseTo`, `not`, `notBetween`, `notEqual`, `notNegative`, `notPositive`, `notZero`, `one`, `positive`, and `strictlyBetween`. The `info` method is highlighted with a blue background.

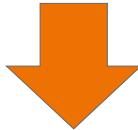
```
Assertions.assertThat(anInt).  
    m isEqualTo(int expected) capture of ?  
    m is(Condition<? super Integer> condition) capture of ?  
    m isEqualTo(long expected) capture of ?  
    m isEqualTo(Object expected) capture of ?  
    m isBetween(Integer start, Integer end) capture of ?  
    f info WritableAssertionInfo  
    m isCloseTo(int expected, Percentage percentage) capture of ?  
    m isCloseTo(int expected, Offset<Integer> offset) capture of ?  
    m isCloseTo(Integer expected, Percentage percentage) capture of ?  
    m isCloseTo(Integer expected, Offset<Integer> offset) capture of ?  
    m isGreaterThanOrEqual(int other) capture of ?  
    m isGreaterThanOrEqual(int other) capture of ?  
    m isGreaterThanOrEqual(Integer other) capture of ?  
    m isLessThan(int other) capture of ?  
    m isLessThanOrEqual(int other) capture of ?  
    m isLessThanOrEqual(Integer other) capture of ?  
    m isLessThan(Integer other) capture of ?  
    m isNegative() capture of ?  
    m isNotCloseTo(int expected, Percentage percentage) capture of ?  
    m isNot(Condition<? super Integer> condition) capture of ?  
    m isNotCloseTo(int expected, Offset<Integer> offset) capture of ?  
    m isNotCloseTo(Integer expected, Percentage percentage) capture of ?  
    m isNotCloseTo(Integer expected, Offset<Integer> offset) capture of ?  
    m isNotEqual(int other) capture of ?  
    m isNotNegative() capture of ?  
    m isNotPositive() capture of ?  
    m isNotZero() capture of ?  
    m isOne() capture of ?  
    m isPositive() capture of ?  
    m isStrictlyBetween(Integer start, Integer end) capture of ?
```

Ctrl+Bas and Ctrl+Haut will move caret down and up in the editor [Next Tip](#)

Comment écrire un test ?

Astuce : On peut faire un import statique de la méthode
Assertions.assertThat()

```
import static org.assertj.core.api.Assertions.assertThat;
```



```
Assertions.assertThat("chat").hasSameSizeAs("toto");
```

Les tests au service des Avengers



```
public class Humanoid {  
  
    private String name;  
    private int age;  
  
    public Humanoid(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public String getName() {return name;}  
    public int getAge() {return age;}  
}
```



```
public class SuperHero extends Humanoid {  
  
    private List<String> powers;  
  
    public SuperHero(Humanoid humanoid,  
                     ArrayList<String> powers) {  
        super(humanoid.getName(), humanoid.getAge());  
        this.powers = powers;  
    }  
  
    public List<String> getPowers() {  
        return Collections.unmodifiableList(powers);  
    }  
}
```

Les tests au service des Avengers



```
private Humanoid peterParker = new Humanoid("Peter Parker", 15);
private Humanoid tonyStark = new Humanoid("Tony Stark", 48);
private Humanoid bruceBanner = new Humanoid("Bruce Banner", 48);
private Humanoid thorOfAsgard = new Humanoid("Thor of Asgard", 1500);
private Humanoid natashaRomanoff = new Humanoid("Natasha Romanoff", 33);
private Humanoid steveRogers = new Humanoid("Steve Rogers", 106);
private Humanoid clintonBarton = new Humanoid("Clinton Barton", 35);
private Humanoid thanosTheTitan = new Humanoid("Thanos the Titan", 750000);

private SuperHero spiderman = new SuperHero(peterParker, "Spider-man", Arrays.asList("Web Shooting", "Agility"));
private SuperHero ironman = new SuperHero(tonyStark, "Iron Man", Arrays.asList("High-tech armor", "Rich"));
private SuperHero hulk = new SuperHero(bruceBanner, "Hulk", Arrays.asList("Strong"));
private SuperHero thor = new SuperHero(thorOfAsgard, null, Arrays.asList("Mjöllnir", "Immortal"));
private SuperHero blackWidow = new SuperHero(natashaRomanoff, "Black Widow", Arrays.asList("Close Combat"));
private SuperHero captainAmerica = new SuperHero(steveRogers, "Captain America", Arrays.asList("Vibranium Shield",
"Super-soldier"));
private SuperHero hawkEye = new SuperHero(clintonBarton, "Hawk Eye", Arrays.asList("Bow and arrows"));
private SuperHero thanos = new SuperHero(thanosTheTitan, null, Arrays.asList("Strong", "Infinity Gauntlet"));

private List<SuperHero> avengers = Arrays.asList(ironman, hulk, thor, blackWidow, captainAmerica, hawkEye);
```

Comment écrire un test ?



Exercice 1 : Ecrire un test qui vérifie que Spiderman est mineur (<=17ans)

Exercice 2 : Ecrire un test qui vérifie que Black Widow fait partie des Avengers

Exercice 3 : Ecrire un test qui vérifie qu'il n'y a pas deux fois le même super-héros dans les Avengers

Exercice 4 : Ecrire un test qui vérifie que les Avengers ne contiennent pas Thanos

Comment écrire un test ?



Exercice 5 : Tester que Thor et Thanos n'ont pas de noms de super-héros

Exercice 6 : Tester que Thanos possède au moins les mêmes super-pouvoirs que Hulk

Exercice 7 : Tester que Hawk Eye est de type SuperHero et Humanoid, mais que Clinton Barton n'est pas du type SuperHero ou String

Exercice 8 : Tester le fait que si l'on ne regarde que l'âge, Tony Stark et Bruce Banner sont égaux

Comment écrire un test ?



Exercice 9 : S'assurer que le nom de héros « Iron Man » contient bien un espace mais que « Spider-man » n'en contient pas

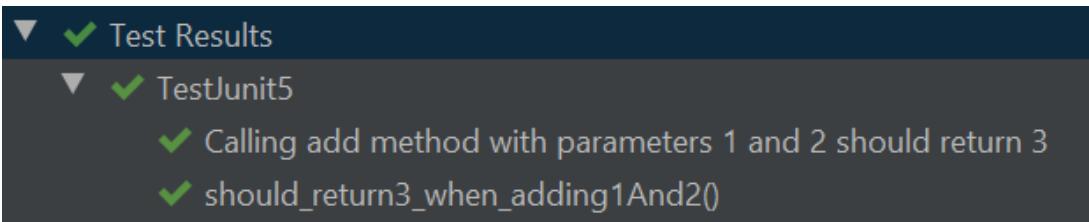


Partie 3 :

Fonctionnalités des frameworks

Annotations JUnit5 : @DisplayName

Pour éviter les noms de test à rallonge, Junit5 a mis en place l'annotation `@DisplayName(String name)` permettant de donner un nom plus lisible à la méthode de test



Avec `@DisplayName`
Sans `@DisplayName`

Annotations JUnit5 : @Disabled

Il est également possible d'ignorer un test grâce à l'annotation **@Disabled**

IMPORTANT : L'annotation accepte un String en paramètre. Pensez à toujours expliquer pourquoi le test est ignoré.

```
@Test  
@Disabled("Functionality not done yet")  
public void make_a_lot_of_money() {  
    //...  
}
```

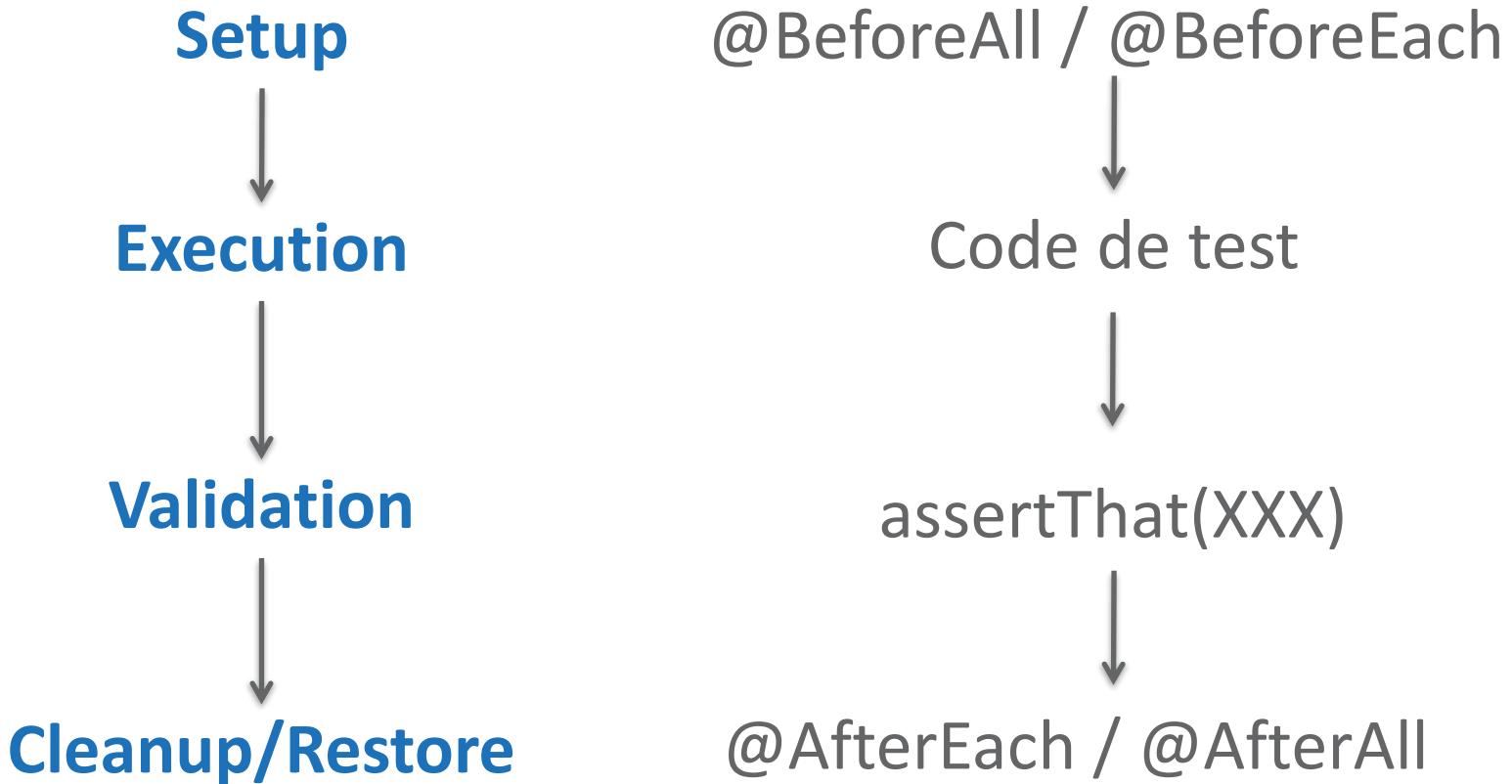
Il faut toujours une bonne raison pour désactiver un test!

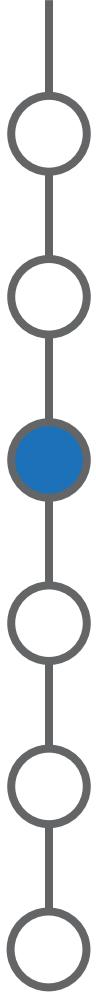
Annotations JUnit5 : : @Before/@After

@BeforeEach, @AfterEach, @BeforeAll et @AfterAll se mettent sur des méthodes qui vont respectivement se déclencher avant ou après chaque ou tout test de la classe.

```
@BeforeAll
public void beforeAll(){
    System.out.println("Starting test session");
}
@AfterAll
public void afterAll(){
    System.out.println("Ending test session");
    database.clean();
}
@BeforeEach
public void beforeEach(){
    database.setToNominalTestState();
}
```

Annotations JUnit5





Annotations JUnit5 : @Order

Les tests sont lancés par défaut dans un **ordre chaotique**. C'est en général une bonne chose car ils devraient être indépendants (cf : F.I.R.S.T.).

Cependant, il existe certains cas où il est nécessaire de définir un ordre. Pour cela, Junit fournit l'annotation **@Order(int i)**



Annotations JUnit5

Junit5 apporte également un lot de fonctionnalités supplémentaires pour des usages poussés :

- Tests imbriqués
- Tests dynamiques
- Catégorisation
- Extensions de types de tests

Ces usages ne seront pas vus à travers ce cours.

Annotations JUnit5

Junit5 étant assez récent, il y a de grande chances que vous travaillez sur des tests en JUnit4.

FEATURE	JUNIT 4	JUNIT 5
Declare a test method	@Test	@Test
Execute before all test methods in the current class	@BeforeClass	@BeforeEach
Execute after all test methods in the current class	@AfterClass	@AfterAll
Execute before each test method	@Before	@BeforeEach
Execute after each test method	@After	@AfterEach
Disable a test method / class	@Ignore	@Disabled



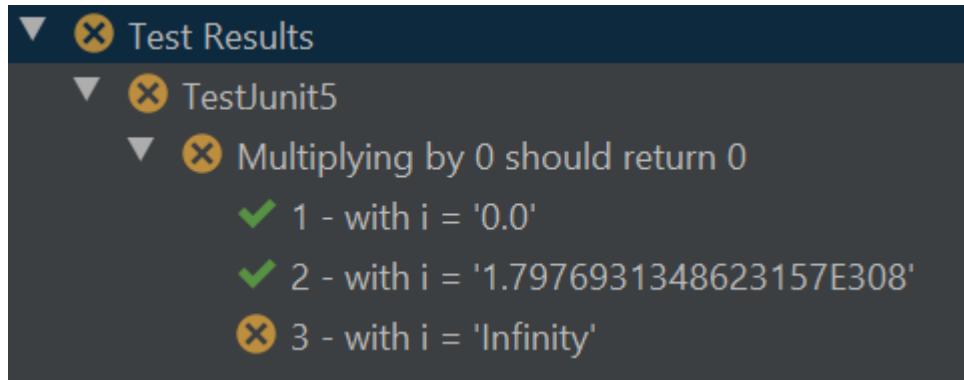
Annotations JUnit5 : @ParametrizedTest

Définition : Un **test paramétrisé** est un test que l'on exécute plusieurs fois avec des paramètres différents.

- On utilise alors l'annotation **@ParametrizedTest** plutôt que l'annotation **@Test**
- On utilise l'annotation **@XXXSource** pour définir les paramètres
 - **@ValueSource** pour des types primitifs et des Strings
 - **@EnumSource** pour des énumérations
 - **@MethodSource** pour des objets
 - **@NullSource** pour ajouter une étape où le paramètre est null

Annotations JUnit5 : @ParametrizedTest

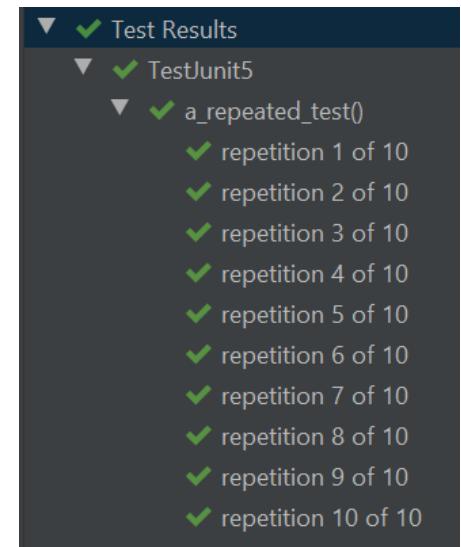
```
@DisplayName("Multiplying by 0 should return 0")
@ParameterizedTest(name = "{index} - with i = '{0}'")
@ValueSource(doubles = {0.0, Double.MAX_VALUE, Double.POSITIVE_INFINITY})
public void some_test(double i) {
    assertThat(i*0).isEqualTo(0.0);
}
```



Annotations JUnit5 : @RepeatedTest

Lors de comportements aléatoires, il peut être intéressant d'essayer d'exécuter une méthode plusieurs fois pour maximiser l'effet du test

Pour cela, il suffit de remplacer `@Test` par `@RepeatedTest(int n)` afin d'exécuter le test N fois



Gestion des exceptions

Il se peut que l'on cherche à tester la levée d'une exception.
Dans ce cas, le bloc d'exécution s'arrête à la seconde étape du test et il devient difficile compliqué d'effectuer une assertion.

```
@Test
public void testException() {
    int i = 1;
    int j = 0;
    int res = i/j; //Exception !
    assertThat(res) //On teste quoi du coup ici ?
}
```

Gestion des exceptions

Initialement, cela se faisait avec des annotation en JUnit4

Code concis !

Mais ne permet pas de savoir d'où provient l'exception dans le cas ou plusieurs instruction pouvant la lancer
Et il n'est pas non plus possible de faire des assertions sur le détail des exceptions (message par exemple)

Utile lorsqu'on à qu'une instruction problématique et qu'on à juste besoin de vérifier qu'une exception est lancée



```
@Test(expected = IllegalArgumentException.class)
public void sum_with_one_negative_number_throw_expected_exception() {
    String numbers = "//; \n1;2;-3;4";
    stringCalculator.add(numbers);
}
```



JUnit 4 et inférieur
N'existe plus en JUnit 5

Gestion des exceptions

En JUnit5, tout se passe à travers le mécanisme d'assertion (et donc, pour nous, grâce à AssertJ).

On utilise pour cela l'assertion `assertThatCode()` ou `assertThatThrownBy()`

```
@Test
public void testException() {
    int i = 1;
    int j = 0;
    assertThatThrownBy(() -> {
        int result = i / j;
    }).isInstanceOf(ArithmeticException.class)
        .hasMessageContaining("by zero");
}
```

Gestion des exceptions

On peut également utiliser l'assertion `assertThatExceptionOfType()` pour commencer par le type de l'erreur :

```
assertThatExceptionOfType(ArithmeticException.class)
    .isThrownBy(() -> {int result = i / j;})
    .withMessageContaining("/ by zero");
```

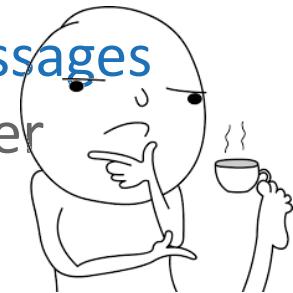
Enfin, pour les plus rigoureux, on peut récupérer l'exception dans un objet et effectuer l'assertion ensuite :

```
Throwable thrown = catchThrowable(() -> {int result = i / j;});

assertThat(thrown).hasMessage("/ by zero")
    .isInstanceOf(ArithmeticException.class);
```

Customisation des messages d'erreur

AssertJ (comme JUnit) permet de personnaliser les messages d'erreur. Cela permet de faire ressortir l'intention métier derrière le test.



```
@Test  
public void test72() {  
    String text = getLoggedDataInfo();  
    assertThat(text)  
        .matches("(?:[a-zA-Z0-9!#$%^&*+=?^`{|}|~-]+(?:\\.[a-zA-Z0-9!#$%^&*+=?^`{|}|~-]+)*|\\\"(?:[\\\\x01-\\\\x08\\\\x0b\\\\x0c\\\\x0e-\\\\x1f\\\\x21\\\\x23-\\\\x5b\\\\x5d-\\\\x7f]|\\\\\\\\\\\\\\\\x01-\\\\x09\\\\x0b\\\\x0c\\\\x0e-\\\\x7f)*\\\\\")@(?:(?:[a-zA-Z0-9](?:[a-zA-Z0-9-]*[a-zA-Z0-9])?|\\\\\\\\((?:(?:[0-5]|2[0-4][0-9]|\\\\[01]?[0-9][0-9]?)\\\\.){3}(?:[25[0-5]|2[0-4][0-9]|\\\\[01]?[0-9][0-9]?)|[a-zA-Z0-9-]*[a-zA-Z0-9]:(?:[\\\\x01-\\\\x08\\\\x0b\\\\x0c\\\\x0e-\\\\x1f\\\\x21-\\\\x5a\\\\x53-\\\\x7f]|\\\\\\\\\\\\\\\\x01-\\\\x09\\\\x0b\\\\x0c\\\\x0e-\\\\x7f]))+\\\\])");  
}
```

```
java.lang.AssertionError:  
Expecting:  
 "zeazae"  
to match pattern:  
 "(?:[a-zA-Z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-zA-Z0-  
9!#$%&'*+/=?^_`{|}~-]+)*"(?:\[x01-\x08\x0b\x0c\x0e-  
\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-  
\x7f])*")@(?:(:?(?:[a-zA-Z0-9](?:[a-zA-Z0-9-]*[a-zA-Z0-9])?\.\.)+[a-  
z0-9](?:[a-zA-Z0-9-]*[a-zA-Z0-9])?|[\(?(?:(:?25[0-5]|2[0-4][0-  
9]| [01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-  
9]| [01]?[0-9][0-9]?|[a-zA-Z0-9-]*[a-zA-Z0-9]:(:?[\x01-  
\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-  
\x09\x0b\x0c\x0e-\x7f])+)\])«
```

Exemple de test difficile à comprendre...à ne pas reproduire chez vous !

Customisation des messages d'erreur

Pour personnaliser le message, il faut utiliser la méthode **withFailMessage()** AVANT l'opération terminale d'assertion.

```
@Test
public void sould_match_email_regexp_when_get_from_logged_user() {
    String email = getLoggedDataInfo();
    assertThat(email)
        .withFailMessage("Email from logged user does not match email regexp")
        .matches("(?:[a-z0-9!#$%&'*+=?^`{|}~-]+(?:\\.[a-z0-9!#$%&'*+=?^`{|}~-]+)*|\\\"(?:(?:[\\x01-\\x08]\\x0b\\x0c\\x0e-\\x1f\\x21\\x23-\\x5b\\x5d-\\x7f]|\\\\\\\\\\\\\\\\[\\x01-\\x09]\\x0b\\x0c\\x0e-\\x7f))*\\\")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.|)[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\\\\[(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?]{3}(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\\\\x01-\\x08]\\x0b\\x0c\\x0e-\\x1f\\x21-\\x5a\\x53-\\x7f]|\\\\\\\\\\\\\\\\[\\x01-\\x09\\x0b\\x0c\\x0e-\\x7f])+\\\\])");
}
```

```
java.lang.AssertionError: Email from logged user does not match email regexp
```

Haaa...bien mieux !

Customisation des messages d'erreur

On peut également donner un nom et/ou un **description** à la variable afin qu'elle apparaisse dans les logs avec la méthode `as()`

```
@Test
public void should_match_email_regex_when_get_from_logged_user() {
    String email = getLoggedDataInfo();
    assertThat(email)
        .as("Email from logged user : '" + email + "'")
        .withFailMessage("does not match email regexp")
        .matches("(?:(a-z0-9!#$%&'*+/=?^`{|}~-]+(?:(\\.[a-z0-9!#$%&'*+/=?^`{|}~-]+)|\\\"(?:(\\x01-\\x09|\\x0b|\\x0c|\\x0e-\\x7f)*\\\")|@(?:(?:(a-z0-9)(?:(a-z0-9-)*(a-z0-9))|\\.).+|\\[\\[?:(?:(?:25[0-5]|2[0-4][0-9]|\\{01}\\{0-9})|\\{3}(?:25[0-5]|2[0-4][0-9]|\\{01}\\{0-9})|\\{2}[a-z0-9-]*[a-z0-9]:(?:(\\x01-\\x08\\x0b\\x0c\\x0e-\\x7f))+)\\])|\\\\\\\\[\\x01-\\x09\\x0b\\x0c\\x0e-\\x7f]))+)");
}
```

```
java.lang.AssertionError: [Email from logged user : 'zeazae'] does not match email regexp
```

Encore mieux ! Le must serait également d'expliquer ce que fait la regexp !

Soft Assertions

Entrez votre nouveau mot de passe : toto

Au moins 8 caractères : totototo

Au moins un chiffre : toto28121988

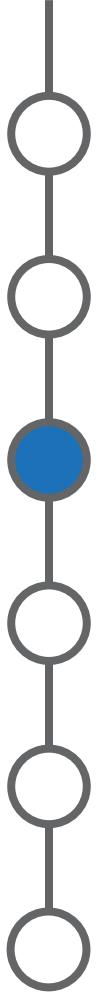
Ne doit pas contenir de date de naissance : totototo2020

Au moins une majuscule : Totototo2020

Au moins un caractère spécial : Toto&toto2020

Pas plus de 12 caractères : ...





Soft Assertions

Une assertion fausse (**rouge**) arrête le test en cours. Les autres assertions (s'il y en a) ne sont alors pas évaluées.

Pour évaluer toutes les assertions sans tenir compte de leur état, il faut utiliser le mécanisme de **soft assertion**.

On effectue alors plusieurs assertions qui seront toutes évaluées **en une fois**.

Soft Assertions

```
@Test
public void testPassword() {
    String newPassword = passwordProducer();
    SoftAssertions softly = new SoftAssertions();
    softly.assertThat(newPassword)
        .withFailMessage("Should have more than 8 characters")
        .hasSizeGreaterThanOrEqualTo(8);
    softly.assertThat(newPassword)
        .withFailMessage("Should have less than 12 characters")
        .hasSizeLessThanOrEqualTo(12);
    softly.assertThat(newPassword)
        .withFailMessage("Should contain at least one number")
        .containsPattern("[0-9]");
    softly.assertThat(newPassword)
        .withFailMessage("Should contain at least one uppercase")
        .containsPattern("[A-Z]");
    softly.assertThat(newPassword)
        .withFailMessage("Should contain at least one special character")
        .containsPattern("[\$&+,;=?@#|'<>.^*()%!-]");
    softly.assertThat(newPassword)
        .withFailMessage("Should not contain a binary character")
        .doesNotContainPattern("[0-9]{6}");
    softly.assertAll();
}
```

Ici, toutes les assertions concernent la même variable.
On peut donc simplifier !

```
Multiple Failures (4 failures)
-- failure 1 --Should have more than 8 characters
at TestJunit5.testPassword(TestJunit5.java:51)
-- failure 2 --Should contain at least one number
at TestJunit5.testPassword(TestJunit5.java:57)
-- failure 3 --Should contain at least one uppercase
at TestJunit5.testPassword(TestJunit5.java:60)
-- failure 4 --Should contain at least one special character
at TestJunit5.testPassword(TestJunit5.java:63)
```

Soft Assertions

```
@Test
public void testPassword() {
    String newPassword = passwordProducer();
    SoftAssertions softly = new SoftAssertions();
    softly.assertThat(newPassword)
        .withFailMessage("Should have more than 8 characters")
        .hasSizeGreaterThanOrEqualTo(8)
        .withFailMessage("Should have less than 12 characters")
        .hasSizeLessThanOrEqualTo(12)
        .withFailMessage("Should contain at least one number")
        .containsPattern("[0-9]")
        .withFailMessage("Should contain at least one uppercase")
        .containsPattern("[A-Z]")
        .withFailMessage("Should contain at least one special character")
        .containsPattern("[\$&+,;:=?@#|'<>.^*()%!-]")
        .withFailMessage("Should not contain a birthdate")
        .doesNotContainPattern("[0-9]{6}");
    softly.assertAll();
}
```

Multiple Failures (4 failures)

```
-- failure 1 --Should have more than 8 characters
at TestJunit5.testPassword(TestJunit5.java:51)
-- failure 2 --Should contain at least one number
at TestJunit5.testPassword(TestJunit5.java:57)
-- failure 3 --Should contain at least one uppercase
at TestJunit5.testPassword(TestJunit5.java:60)
-- failure 4 --Should contain at least one special
character
at TestJunit5.testPassword(TestJunit5.java:63)
```

Soft Assertions

Junit5 propose également cette fonctionnalité avec [assertAll\(\)](#).

La syntaxe est moins facile à appréhender, mais elle est beaucoup plus [performante](#) !

```
assertAll(  
    () -> assertThat(newPassword)  
        .withFailMessage("Should have more than 8 characters")  
        .hasSizeGreaterThanOrEqualTo(8),  
    () -> assertThat(newPassword)  
        .withFailMessage("Should have less than 12 characters")  
        .hasSizeLessThanOrEqualTo(12),  
    () -> assertThat(newPassword)  
        .withFailMessage("Should contain at least one number")  
        .containsPattern("[0-9]"),  
    () -> assertThat(newPassword)  
        .withFailMessage("Should contain at least one uppercase")  
        .containsPattern("[A-Z]"),  
    () -> assertThat(newPassword)  
        .withFailMessage("Should contain at least one special character")  
        .containsPattern("[\$&+,:=?@#|'<>.^*()%!-]"),  
    () -> assertThat(newPassword)  
        .withFailMessage("Should not contain a birthdate")  
        .doesNotContainPattern("[0-9]{6}"));
```

Annotations JUnit5



```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalves = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y;                                // evil floating point bit level
hacking
    i = 0x5f3759df - ( i >> 1 );                      // what the fuck?
    y = * ( float * ) &i;
    y = y * ( threehalves - ( x2 * y * y ) );          // 1st iteration
// y = y * ( threehalves - ( x2 * y * y ) );          // 2nd iteration, this can be
removed

    return y;
}
```

Annotations JUnit5



Complétez le jeu de test de la section précédente

Exercice 1 : Faire en sorte que chaque test imprime son temps d'exécution (en ms) sur la sortie standard

Exercice 2 : Lancez les tests dans un ordre précis

Exercice 3 : Désactivez un de vos tests

Annotations JUnit5



Exercice 4 : Soit [ce code java](#), réfléchissez à la meilleure manière de tester ce code et implémentez-la

Exercice 5 : Testez également la vitesse d'exécution par rapport la la méthode Math.sqrt native de java

```
public static float invSqrt(float x) {  
    float xhalf = 0.5f * x;  
    int i = Float.floatToIntBits(x);  
    i = 0x5f3759df - (i >> 1);  
    x = Float.intBitsToFloat(i);  
    x *= (1.5f - xhalf * x * x);  
    return x;  
}
```



Fonctionnalités des frameworks

Exercice 6 : Testez que l'on ne peut pas ajouter ni supprimer de pouvoirs à un super-héro

Exercice 7 : Testez que chaque super-héro possède un nom avec un espace (doit afficher que ce n'est pas le cas pour Spider-man et Hulk)

Exercice 8 : Afficher des messages d'erreur personnalisé pour chaque test de l'exercice 7

Partie 4 :

C'est quoi un bon test ?

Alors y'a le bon test et
le mauvais test...





Le bon test unitaire

F.I.R.S.T



Le bon test unitaire

- F_{ast}
- I_{ndependant / I_{solated}}
- R_{epeatable}
- S_{elf-validating}
- T_{horough / T_{imely}}



Le bon test unitaire

Fast :

- Le développeur ne devrait pas hésiter à lancer ses tests unitaires
- La durée du test devrait être de l'ordre de la milliseconde (<100ms) afin d'éviter toute contrainte de temps



Le bon test unitaire

Indépendant / Isolated :

- Ces tests doivent pouvoir se lancer individuellement ou en série de la même façon. Un test ne doit pas être dépendant des autres tests
- Un test de devrait pas modifier l'état du système sans le remettre dans son état initial par la suite



Le bon test unitaire

R_epeatable:

- Les tests devraient être idempotents (le nombre d'exécution n'influe pas sur le résultat du test) afin de s'assurer de la fiabilité des retours de celui-ci
- Les tests ne devraient pas dépendre des autres tests ou d'éléments extérieurs (réseau, BDD, services...) afin de pouvoir être réexécuté dans n'importe quel environnement.
- Les données devront donc être préparées par le test lui même pour s'assurer d'être en capacité de répéter le test régulièrement sans avoir de mauvaises surprises (*utilisation de mocks*).



Le bon test unitaire

Self-validating :

- Le résultat de chaque test devrait être booléen (vert/rouge)
- Le résultat devrait être donné de manière automatisée, sans action manuelle



Le bon test unitaire

T_{horough} (consciencieux) / T_{imely} :

- Le test devrait chercher à couvrir l'ensemble des cas de test et non l'ensemble du code et des entrées possibles. Voici ce qu'on pourrait définir comme générique :
 - Tester avec un volume de données conséquent et représentatif
 - Tester les cas nominaux
 - Tester les cas aux limites
 - Tester avec des entrées erronées / les exceptions
- Les tests devraient être écrits avant le code (TDD), ou en tous cas, avant la mise en production (test first)



Le bon test unitaire

Bonus : Les tests représentent les spécifications les plus précises possibles de votre application. Veuillez donc à ce qu'ils soient :

- Lisibles / Auto-documentés : rien que le nom de la méthode de test doit permettre de connaître ses entrants et son comportement attendu
- Compréhensibles : Le test doit être simple (**KISS**: Keep It Simple & Stupid). Ne testez qu'une seule chose à la fois.



Le mauvais test unitaire

Quelques **smells** quand on écrit/utilise des tests unitaires :

- Il existe un **ordre** d'exécution de test
- Des tests basés sur la **performance**
- Les tests sont **lents**
- Les tests testent le détails d'implémentation (**COMMENT**) plutôt que le comportement métier (**QUOI**)



Couverture de code

La **couverture de code** (ou couverture de test) est une mesure pour indiquer le taux de code source d'un programme qui est exécuté quand on lance les tests.

Il est exprimé en **%pourcentage%**

Couverture de code

Il existe plusieurs types de taux de couverture :

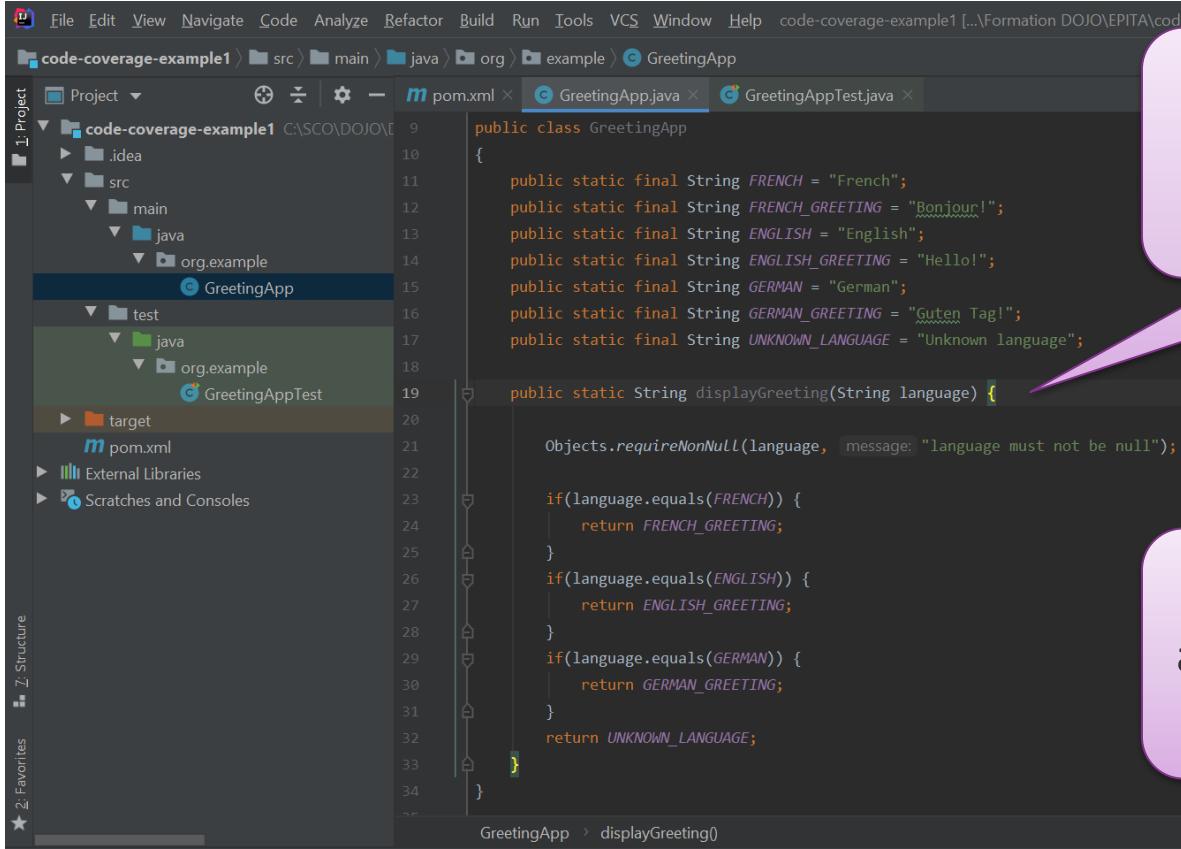
- Couverture de **ligne** : chaque ligne exécutée est considérée comme couverte
- Couverture de **branche** : Chaque condition est considérée comme une branche

```
public static int badMethod(boolean isCoolUser){  
    String user = null;  
    if (isCoolUser){  
        user = "Romain";  
    }  
    return user.length();  
}
```

```
assertThat(SomeClass.badMethod(true)).isEqualTo(6);
```

100% line coverage
50% branch coverage

Couverture de code



```
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help code-coverage-example1 ...\\Formation DOJO\\EPITA\\code

code-coverage-example1 > src > main > java > org > example > GreetingApp

Project pom.xml GreetingApp.java GreetingAppTest.java

public class GreetingApp {
    public static final String FRENCH = "French";
    public static final String FRENCH_GREETING = "Bonjour!";
    public static final String ENGLISH = "English";
    public static final String ENGLISH_GREETING = "Hello!";
    public static final String GERMAN = "German";
    public static final String GERMAN_GREETING = "Guten Tag!";
    public static final String UNKNOWN_LANGUAGE = "Unknown language";

    public static String displayGreeting(String language) {
        Objects.requireNonNull(language, message: "language must not be null");

        if(language.equals(FRENCH)) {
            return FRENCH_GREETING;
        }
        if(language.equals(ENGLISH)) {
            return ENGLISH_GREETING;
        }
        if(language.equals(GERMAN)) {
            return GERMAN_GREETING;
        }
        return UNKNOWN_LANGUAGE;
    }
}

GreetingApp > displayGreeting()
```

Que fait la
méthode
DisplayGreeting?

La méthode
displayGreeting
affiche la salutation en
fonction de la langue
reçue en paramètre.

Couverture de code

Taux de couverture du code à 0%.

The screenshot shows the IntelliJ IDEA interface with a Java project named "code-coverage-example1". The "Coverage" tool window is open, displaying a table with one row for "GreetingApp". The table shows 0% coverage for classes, methods, and lines. A tooltip over the "Run 'GreetingAppTest' with Coverage" option in the context menu for the test class indicates the coverage status.

Element	Class, %	Method, %	Line, %
GreetingApp	0% (0/1)	0% (0/1)	0% (0/8)

Exécuter les tests et afficher le taux de couverture.

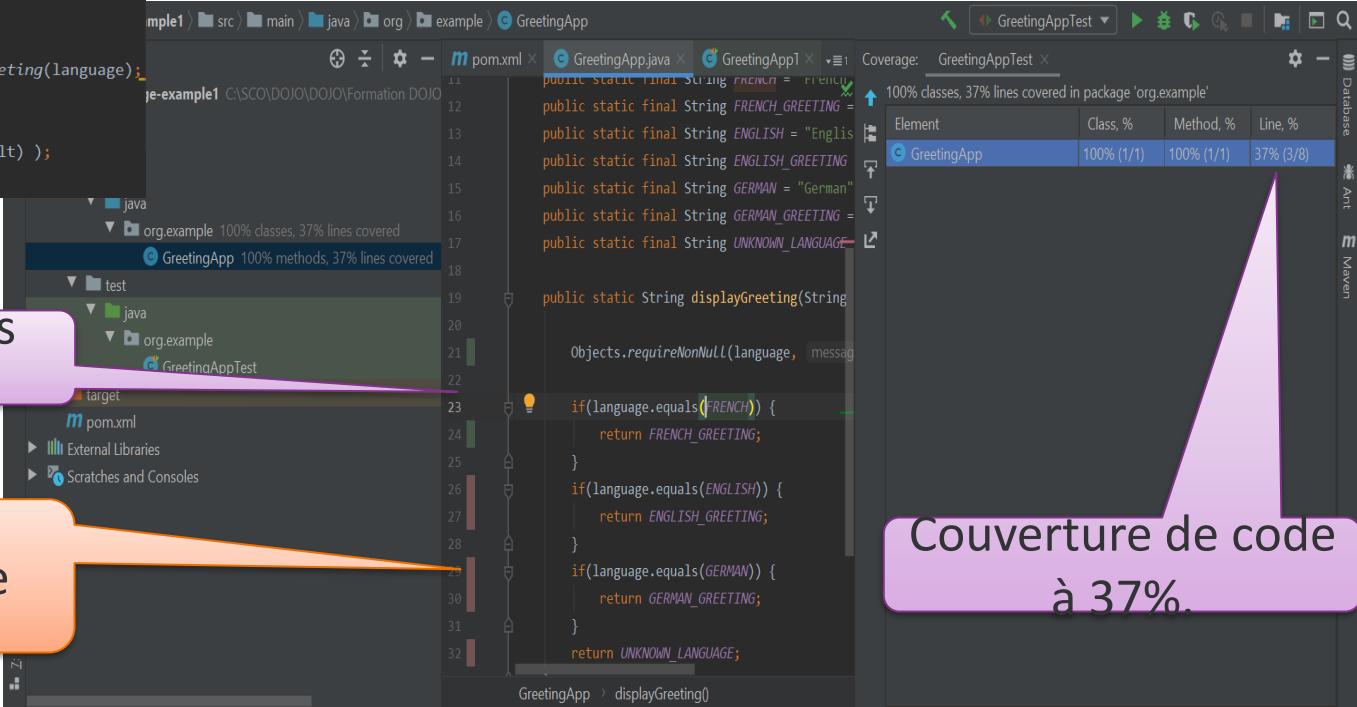
Couverture de code

```
@Test  
public void should_return_french_greeting()  
{  
    //given  
    String language = "French";  
    String expectedResult = "Bonjour!";  
  
    //when  
    String result = GreetingApp.displayGreeting(language);  
  
    //then  
    assertTrue( result.equals(expectedResult) );  
}
```

Ajout d'un test pour la langue Française.

Lignes couvertes par le test.

Lignes non-couvertes par le test.



Couverture de code

```
@Test  
public void should_return_french_greeting()  
{...}  
  
@Test  
public void should_return_english_greeting()  
{...}  
  
@Test  
public void should_return_german_greeting()  
{...}
```

Lignes couvertes par les tests

Lignes non-couvertes par les tests

Tests pour les 3 langues: Français, Anglais, Allemand

Ligne couverte même sans avoir déclenché d'exception

The screenshot shows an IDE interface with several windows:

- Pom.xml:** Shows the project's build configuration.
- GreetingApp.java:** The source code for the application. It contains static final strings for French, English, and German greetings, and a method to display a greeting based on a language parameter. A specific line of code, `Objects.requireNonNull(language, "language must not be null");`, is highlighted with a red box and a callout, indicating it is covered even though no exception was thrown.
- GreetingAppTest.java:** The test class for GreetingApp.
- Coverage Statistics:** A sidebar displays coverage details: 100% classes, 87% lines covered in package 'org.example'. Below this, a table shows coverage for the GreetingApp class: 100% methods, 87% lines covered.

A large orange diagonal shape sweeps across the interface, highlighting the coverage analysis and the specific test case being discussed.

```
public static final String FRENCH_GREETING = "Hello";  
public static final String ENGLISH_GREETING = "Hello";  
public static final String GERMAN_GREETING = "Guten Tag";  
public static final String UNKNOWN_LANGUAGE = "Unknown language";  
  
public static String displayGreeting(String language) {  
    Objects.requireNonNull(language, "language must not be null");  
    if(language.equals(FRENCH)) {  
        return FRENCH_GREETING;  
    }  
    if(language.equals(ENGLISH)) {  
        return ENGLISH_GREETING;  
    }  
    if(language.equals(GERMAN)) {  
        return GERMAN_GREETING;  
    }  
    return UNKNOWN_LANGUAGE;  
}
```

Couverture de code à 87%.



Couverture de code

C'est **combien** une bonne
couverture de code ?

10% ... 20% ... 50% ... 80% ... 100% ... 200% ... ?

Couverture de code





Couverture de code

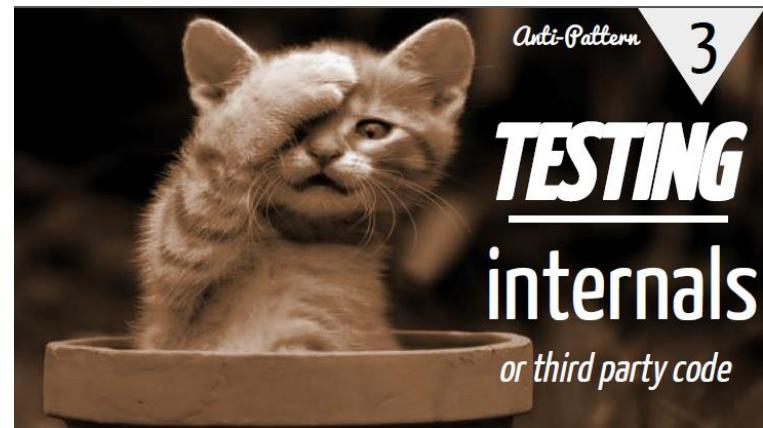
En général, le taux de couverture est décidé en fonction du **coût** de deux facteurs :

- Le coût de **mise en place** des tests :
 - Est-ce facile de tester?
 - Est-ce facile à maintenir?
- Le coût de **l'erreur** (en terme d'argent, d'image, de santé...)

Couverture de code

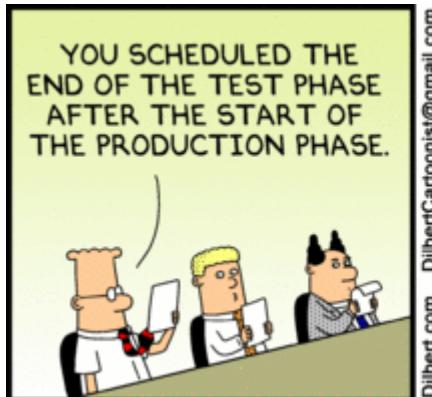
Ce qu'il ne faut pas tester :

- Les éléments d'un frameworks ou d'un langage
- Le code auto-généré
- Les méthodes privées, qui forment la mécanique interne (le comment)



Partie 5 :

Quand est-ce qu'on teste ? (TDD)



I WANT YOU



TO START TESTING !

...LATER



Quand est-ce qu'on teste ?

- Historiquement on développait une application, qu'on essayait de couvrir avec des tests (unitaires, d'intégration...), comme en général les tests arrivent en fin de projet, ils sont souvent « **oubliés** » pour gagner du temps, du budget etc...
- Sans couverture de tests suffisante, la moindre modification dans le code d'une application représente un risque de **régression**
- Sans pilotage précis, on écrit souvent « **un peu trop de code** », et on pourrait se retrouver à produire (et donc à tester) du code mort

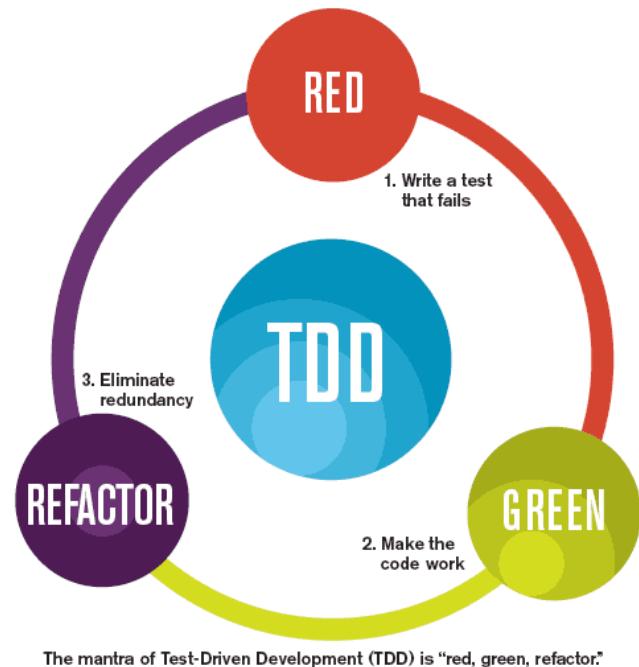
Quand est-ce qu'on teste ?



**ALL CODE IS GUILTY
UNTIL PROVEN INNOCENT**

Test-Driven Development

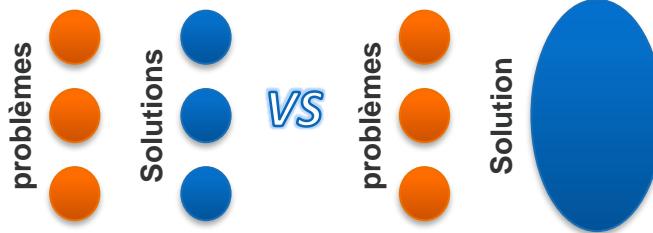
- Méthode mise au point par Kent Beck en 2003 (aussi papa de l'XP)
- Méthodes liée au « Tests first programming concept » de l'extreme programming
- C'est une discipline **de développement** qui demande de la rigueur ainsi qu'un changement de mindset
- Utilisée et reconnue dans le monde du développement



Test-Driven Development

Le TDD s'appuie tout d'abord sur une façon de raisonner :

- Piloter les développements par les tests (Tests first)
- Décrire chaque facette d'un problème, les résoudre une à une, plutôt que de chercher une solution à tous les problèmes en une fois



Test-Driven Development

Les 3 Lois du TDD (par Robert C. MARTIN) :



You are **not allowed** to write any production code unless it is to make a failing unit test pass.

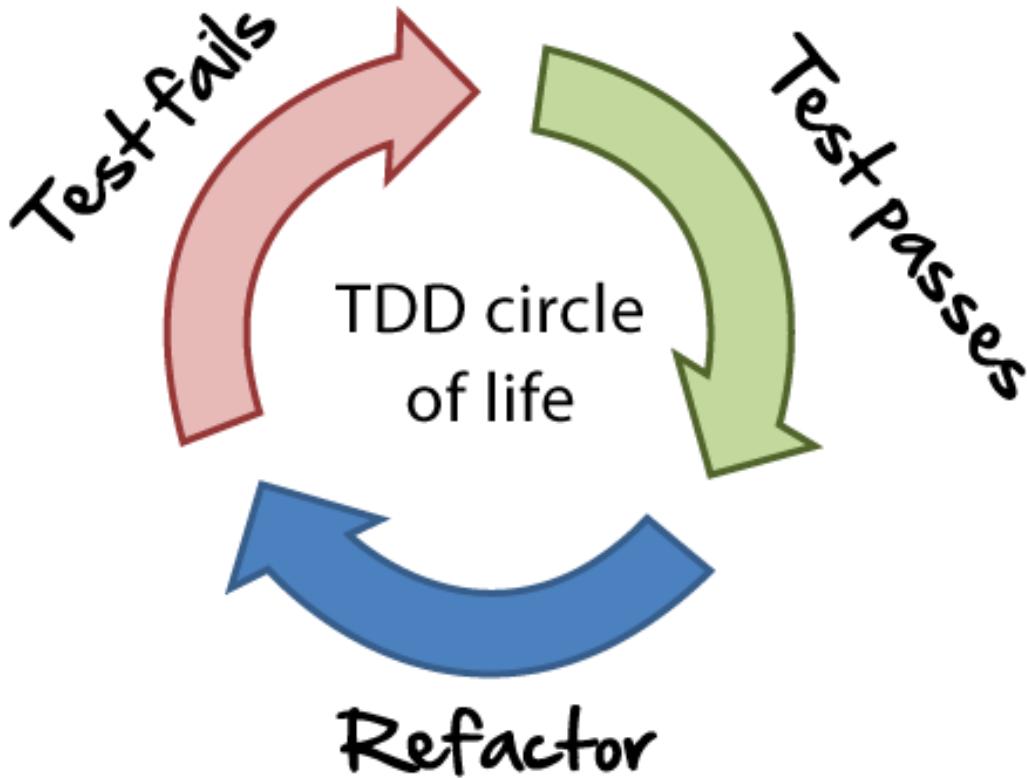


You are not allowed to write **any more** of a unit test **than is sufficient** to fail; and compilation failures are failures.



You are not allowed to write **any more** production code **than is sufficient** to pass the one failing unit test.

Test-Driven Development





Test-Driven Development

L'étape **RED** : L'écriture d'un cas non passant

- Au départ le test échoue car le code applicatif (code de production) n'existe pas (une erreur de compilation est bien un échec)
- Le test doit être court & simple (quelques lignes de code)
- Le nom du test doit être très explicite et bien représenter le use case ou la fonctionnalité qui est testé
- Seul le nouveau test doit échouer, les autres tests existant ne doivent pas avoir été affectés et doivent tous être au vert (cf : indépendance des tests)

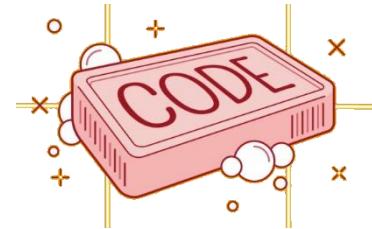
Test-Driven Development



L'étape **GREEN** : Rendre le test passant

- Il s'agit d'écrire le code applicatif **minimum** pour que notre test devienne passant
- Il n'est pas nécessaire d'écrire un code parfait, il faut juste passer le test au vert (c'est le boulot de la troisième étape)
- Attention ! A ce moment tous les tests doivent être au vert, hors de question qu'un nouveau test vert ait un effet de bord sur un autre test. Si ce n'est pas le cas, il faut retravailler le code que l'on vient d'écrire
- En général on traitera tous les cas « aux bornes » dans des tests séparés du cas nominal

Test-Driven Development



L'étape **REFACTOR** : Clarifier, simplifier généraliser

- Ici, la phase de refactoring embarque
 - Clarification (règles de noms...)
 - Factorisation (supprimer les duplications de code etc...)
 - Généralisation (le test est fait avec UN jeu de valeur, le code doit fonctionner pour TOUTES les valeurs légitimes)
 - La modélisation objet peut être revue
- On peut faire du refactoring sans craintes car la couverture de test va nous permettre de détecter les régressions de manière immédiate.
- Il faut garder en tête la règle du bon scout, il faut rendre le code dans un état plus propre que celui dans lequel on l'a trouvé
- Attention ! Le refactoring n'est pas réservé qu'au TDD !



Test-Driven Development



Cas concret :

Nous cherchons à développer une méthode qui additionne deux nombres.

Test-Driven Development



Etape préliminaire, l'étape « think » :

- L'élément à tester est-il suffisamment petit, indépendant et découpé des autres?
- Les attendus (conditions d'échec/succès) sont-ils clairs?
- A-t-on des exemples sur lesquels s'appuyer pour nos test?

$$0+1 = 1$$

$$0+2 = 2$$

$$1+2 = 3$$

Test-Driven Development



Etape RED :

Ecriture d'un test :

```
@Test  
public void should_return_1_when_adding_0_and_1() {  
    int sum = add(0,1);  
    assertThat(sum).isEqualTo(1);  
}
```

- Le test est non passant
- Le test est court & simple
- Le nom du test est très explicite
- Le test est FIRST



Test-Driven Development

Etape GREEN :

Ecriture du minimum de code requis :

```
public int add(int a, int b){  
    return 1;  
}
```

- Le test devient passant
- Le minimum de code a été produit

Test-Driven Development



Etape REFACTOR :

Peut-on généraliser ou simplifier ce code?

```
public int add(int a, int b){  
    return 1;  
}
```

Le code est simple et lisible

Test-Driven Development



Etape RED :

Ecriture d'un test :

```
@Test  
public void should_return_2_when_adding_0_and_2() {  
    int sum = add(0,2);  
    assertThat(sum).isEqualTo(2);  
}
```

- Le test est non passant
- Le test est court & simple
- Le nom du test est très explicite
- Le test est FIRST



Test-Driven Development

Etape GREEN :

Ecriture du minimum de code requis :

```
public int add(int a, int b){  
    return b;  
}
```

- Le test devient passant
- Le minimum de code a été produit
- Les autres tests sont toujours passants

Test-Driven Development



Etape REFACTOR :

Peut-on généraliser ou simplifier ce code?

```
public int add(int a, int b){  
    return b;  
}
```

Le code est simple et lisible

Test-Driven Development



Etape RED :

Ecriture d'un test :

```
@Test  
public void should_return_3_when_adding_1_and_2() {  
    int sum = add(1,2);  
    assertThat(sum).isEqualTo(3);  
}
```

- Le test est non passant
- Le test est court & simple
- Le nom du test est très explicite
- Le test est FIRST

Test-Driven Development



Etape GREEN :

Ecriture du minimum de code requis :

```
public int add(int a, int b){  
    if(a == 0) {return b;}  
    return a+b;  
}
```

- Le test devient passant
- Le minimum de code a été produit
- Les autres tests sont toujours passants



Test-Driven Development



Etape REFACTOR :

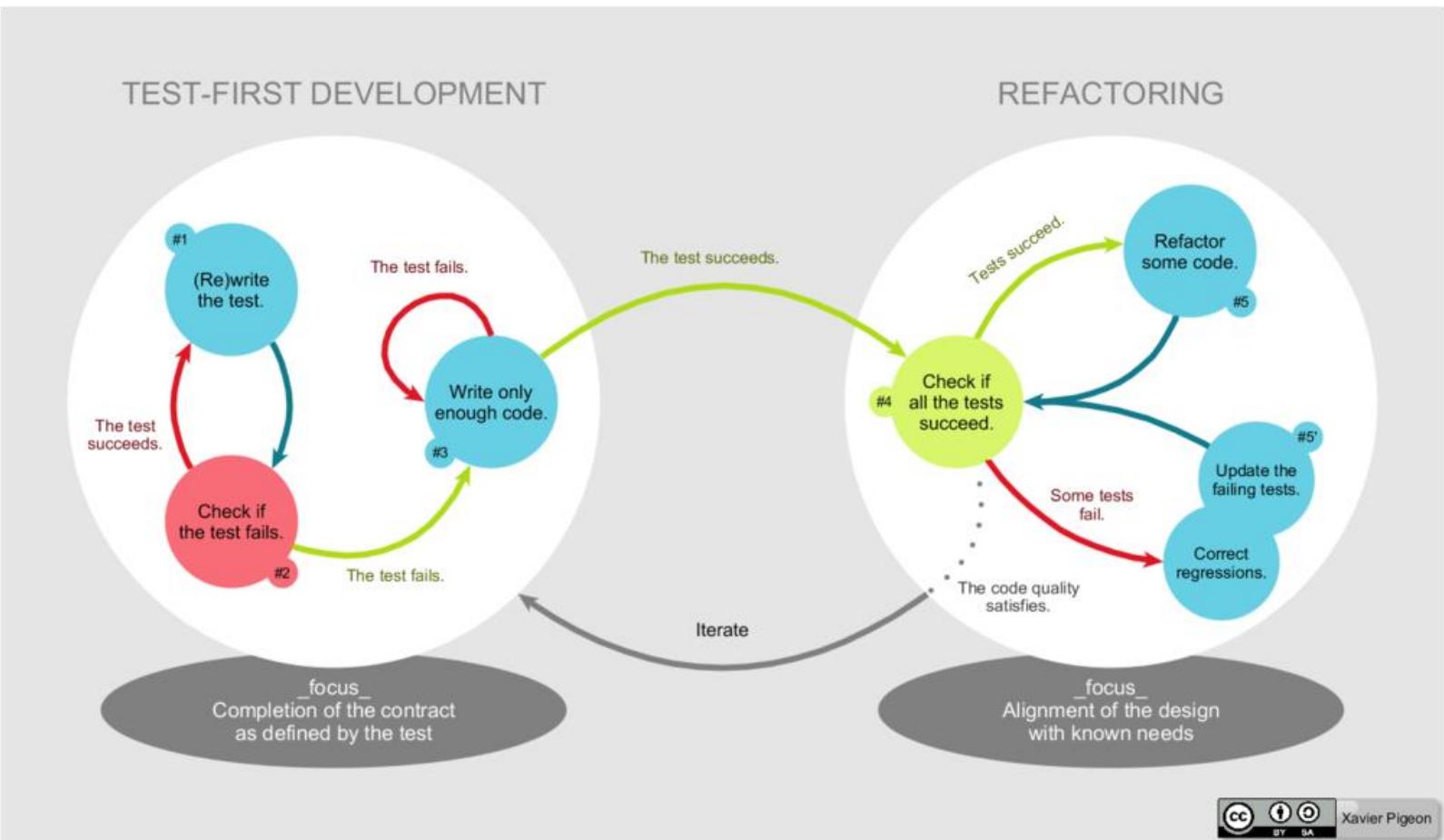
Peut-on généraliser ou simplifier ce code?

```
public int add(int a, int b){  
    if(a == 0) {return b;}  
    return a+b;  
}
```

```
public int add(int a, int b){  
    return a+b;  
}
```

Le code est simple et lisible

Test-Driven Development



Test-Driven Development



A-t-on réellement terminé ?

```
@Test  
public void should_be_positive_when_adding_MAX_and_1() {  
    int sum = add(Integer.MAX_VALUE,1);  
    assertThat(sum).isGreaterThan(0);  
}
```

Il est souvent important de tester les cas aux limites ainsi que les paramètres erronés (e.g : null)



Test-Driven Development

Les bienfaits du TDD (1/2) :

- Permet de n'écrire que le code nécessaire, pas de code superflu : YAGNI !
- Améliore la compréhension du besoin métier et force à se baser sur des exemples concrets
- Documente l'intention métier à travers les tests
- Facilite de débugging : Tout est designé pour être testé
- Tous les tests sont utiles



Test-Driven Development

Les bienfaits du TDD (2/2) :

- Permet de s'assurer que tout le code est couvert par au moins un test
- Va permettre de gagner en confiance et va permettre aux développeurs une plus grande part d'initiative sans risquer de « tout casser »
- Pilote le design d'une application et force à avoir des classes et interfaces plus précises, plus modulaires, ce qui engendre un design globalement plus propre

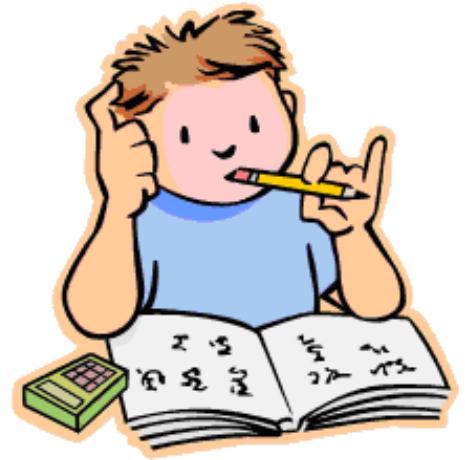


Test-Driven Development

Les limites du TDD :

- Le TDD n'est pas très adapté dans le cas d'application qui nécessite de nombreux tests fonctionnels pour déterminer le bon fonctionnement (le cas des interfaces graphiques par exemple)
- Le TDD (et le testing en général) est souvent vu par le management comme une perte sèche plutôt qu'un investissement
- Les tests et le code correspondant sont écrits par la même personne. Le code n'est alors pas exempts de faux positifs et peut mener à un faux sentiment de sécurité

Travaux dirigés





- Les exercices suivants sont à effectuer en suivant la méthode du TDD
- Chaque étape ajoute une fonctionnalité ou une subtilité qui ne doit pas impacter les étapes précédentes
- Le refactoring étant un élément important de ces exercices, essayez de ne pas lire les énoncés à l'avance afin de ne pas anticiper les modifications



Exercice 1 : Le FizzBuzz est un jeu pour enfant où le but est de compter à haute voix en remplaçant les multiples de 3 par « FIZZ » et les multiples des 5 par « BUZZ ».

Un, Deux, Fizz, Quatre, Buzz, Fizz, Sept...

**FIZZ
BUZZ
FIZZBUZZ**



Exercice 1.1 : Ecrire une méthode qui prend un entier en paramètre et qui renvoie la représentation de ce nombre en chaîne de caractère

`7 => "7"`

Exercice 1.2 : Si le nombre est divisible par 3, retourner FIZZ

`6 => "FIZZ"`

Exercice 1.3 : Si le nombre est divisible par 5, retourner BUZZ

`5 => "BUZZ"`



Exercice 1.4 : Si le nombre est divisible par 3 et 5, retourner « FIZZBUZZ »

15 => "FIZZBUZZ"

Exercice 1.5 : Les nombres négatifs doivent jeter une exception contenant le message « No negative numbers allowed »



Exercice 1.6 : De la même manière que les deux précédents, ajouter QIX lorsque le nombre est divisible par 7

14 => "QIX"

21 => "FIZZQIX"



Exercice 1.7 : Pour chaque 3, 5, ou 7, ajouter respectivement FIZZ, BUZZ ou QIX en fin de chaîne renversée.

15 => FIZZBUZZBUZZ

(divisible par 3, divisible par 5, contient 5)

33 => FIZZFIZZFIZZ

(divisible par 3, contient deux 3)



Exercice 1.8 : Le client ajoute une contrainte !
Remplacer chaque 0 par le caractère '*'.

$101 \Rightarrow 1*1$

$303 \Rightarrow \text{FIZZFIZZ}*\text{FIZZ}$

$105 \Rightarrow \text{FIZZBUZZQIX}*\text{FIZZ}$

$10101 \Rightarrow \text{FIZZQIX}**$



Exercice 2 : L'exercice « String Calculator » est un exercice de refactoring incrémental imaginé par Roy OSHEROVE (ce qui en fait un bon candidat pour le TDD).



<http://osherove.com/tdd-kata-1/>



Exercice 2.1 : Développer une méthode « add » qui :

- Prend en paramètre un String, représentant 0, 1 ou 2 nombres, séparés par une virgule
- Renvoie leur somme (sous forme de String)
- Un String vide renvoie 0

“1” => “1”

“1,2” => “3”

“” => “0”



Exercice 2.2 : Faire en sorte que le String en entrée puisse accepter un nombre inconnu d'éléments séparés par une virgule

“5,5,6,4” => “10”



Exercice 2.3 : Faire en sorte que le String en entrée accepte également le caractère de fin de ligne « \n » en tant que séparateur de nombres.

“5,5\n6,4” => “10”

Gérer également le cas des entrées invalides.

“175.2,\n35” => “Number expected but ‘\n’ found at position 6”



Exercice 2.4 : Ne pas accepter de séparateur en fin de chaîne.

“1,3,” => “Number expected but EOF found”



Exercice 2.5 : Il est maintenant possible de laisser l'utilisateur choisir son délimiteur. Cela se fait en début de chaîne grâce à la syntaxe suivante :

```
//[delimiter]\n[numbers]
```

" //;\n1;2 " => "3"

" //|\n1|2|3 " => "6"

" //sep\n2sep3 " => "5"

"//|\n1|2,3" => " '|' expected but ',' found at position 3. "

NB : Les cas précédents continuent à fonctionner



Exercice 2.6 : Les nombres négatifs ne sont pas autorisés. La méthode doit alors retourner la liste des nombres négatifs.

" -1,2 " => “Negative not allowed : -1”

" -1,2,-4 " => “Negative not allowed : -1, -4”



Exercice 2.8 : Ignorer les nombre supérieur à 1000.

" 1001;2 " => " 2 "



Exercice 2.9 : Il est maintenant possible d'accepter plusieurs délimiteurs si ces derniers sont entre crochets.

“//[*][%]\n1*2%3” => “6”

Conclusion





Les tester c'est...

S'assurer que le code fonctionne correctement

Documenter son code en explicitant le métier

Comprendre le métier et chercher des exemples

Spécifier

Eviter d'avoir à débugger

Ne pas régresser

Améliorer la maintenabilité

Une méthode de développement



Les tester c'est...

Arrêter de faire des main pour débugger !



Conclusion

Félicitations !

Vous avez survécu aux fondamentaux du test !

A ce stade, vous savez :

- Ce qu'est qu'un test et les différents types de test
- Développer des tests unitaires avec Junit et AssertJ
- Et même un peu plus !
- Les qualités d'un bon test
- Faire du TDD

Conclusion

Pour réviser (ou aller plus loin) :

Le blog de Roy Osherove:

- <http://osherove.com/>

Les cycles du TDD (Robert C. Martin):

- <http://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html>

A propos des conventions de nommage:

- <https://www.petrikainulainen.net/programming/testing/writing-clean-tests-naming-matters/>
- <http://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html>

Katas TDD :

- <https://codingdojo.org/kata/>



Conclusion

JUnit5:

- <https://junit.org/junit5/>
- Avec maven: <https://junit.org/junit5/docs/current/user-guide/#running-tests-build-maven>

JUnit System Rules: <http://stefanbirkner.github.io/system-rules/>

Mockito: <http://site.mockito.org/>

Hamcrest: <http://hamcrest.org/JavaHamcrest/>

JUnitParams: <https://github.com/Pragmatists/junitparams/wiki/Quickstart>

AssertJ: <http://joel-costigliola.github.io/assertj/>

Best practice Junit/TDD: <https://www.slideshare.net/NarendraPathai/test-driven-development-junit-basics-and-best-practices>

That's all Folks!