R4.02 - Qualité logicielle - CM n°1

Pierre Nerzic

février-mars 2024

C'est un cours sur la *qualité logicielle*.

Qualité logicielle ?

C'est en rapport avec ce qu'on attend d'un logiciel, un travail sans erreur, efficace, et aussi *prouvable*. C'est à dire qu'on peut s'assurer par des mécanismes fiables que le fonctionnement est entièrement conforme au cahier des charges.

Comme c'est très vaste et qu'on a très peu de temps, on va se consacrer à ce qui est appelé les « tests », c'est à dire les vérifications de la conformité aux spécifications.

Dans cet enseignement, nous verrons beaucoup d'outils et de concepts pour développer un logiciel en assurant sa qualité.

Introduction

Pierre Nerzic

Extrait du PPN BUT Info semestre 4 Ressource R4.02

« L'objectif de cette ressource est d'approfondir la production de tests, mais également d'identifier les critères de faisabilité d'un projet informatique. »

- Savoirs de référence étudiés
  - Problématique de la non-régression
  - Tests d'intégration
- Prolongements suggérés
  - Tests d'utilisabilité
  - Tests fonctionnels
  - Continuous Integration / Continuous Delivery
  - Test UI
  - Couvertures de tests

Plan simplifié de cet enseignement

- **Tests unitaires** : chaque méthode est vérifiée isolément
  - Outils : Java, Maven, JUnit5, AssertJ, et Mockoon (plus tard)
- **Tests d'intégration** : des groupes de classes et méthodes sont vérifiées ensemble
  - Outils : Mockito
- **Couverture des tests** : est-ce que dans chaque méthode, chaque instruction programmée a été testée ?
  - Outils : JaCoCo
- **Tests fonctionnels** : ici, il s'agira des vérifications du bon fonctionnement d'une interface utilisateur
  - Outils : Robot Framework, Selenium
- **Intégration continue** sur GitLab : tous les tests sont effectués à chaque *commit*
  - Outils : GitLab runners

Pierre Nerzic

Bibliographie

- OpenClassRooms - Testez votre code Java pour réaliser des applications de qualité par Geoffrey Arthaud, excellent cours, à connaître.
- Documentation AssertJ, complète et bien écrite, avec des exemples utiles.

Déroulement des enseignements

Types de cours

L'enseignement consiste en :

- 3 CM comme celui-ci : connaissances générales pour comprendre où on va
- 8 TD et TP : indifférenciés, tous consistent en travaux à faire sur machine, avec des apports théoriques fournis par le sujet et de la documentation externe à consulter

Évaluation

- Travaux pratiques : le principe, c'est que le travail effectué en séance devra être déposé sur Moodle. Ce travail sera noté. Chaque séance rapportera ainsi quelques points dont la somme formera la note pratique.
  - avantages : vrai contrôle continu, pas de pression, pas de gros TP noté tout à la fin
  - mise en garde : interdiction de vous transmettre un travail fait par quelqu'un d'autre.

Mon but est que vous appreniez quelque chose, pas forcément que vous ayez une bonne note. J'ai parfois l'impression que votre but est exactement l'inverse.

Tests logiciels

De quoi s'agit-il ?

Le concept de test logiciel est simple :

- On dispose d'une classe à tester `Classe1` ayant différentes méthodes.
- On programme une autre classe `TestsClasse1` dont les méthodes vont essayer celles de `Classe1`, et vérifier qu'elles retournent les bonnes valeurs en fonction des paramètres fournis.
- Si les résultats sont ceux attendus, les tests réussissent. Sinon, `Classe1` est mauvaise... ou `TestsClasse1`, ou les deux...

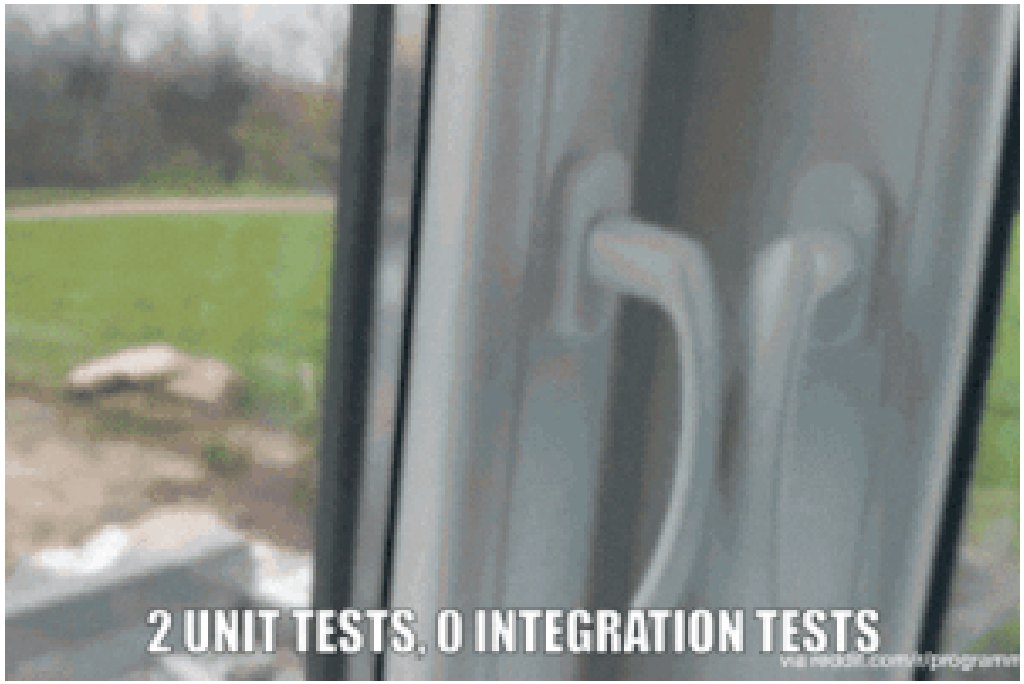
Tester, c'est douter.

Tests unitaires et autres

Le concept précédent est assez vague et recouvre plusieurs étendues de tests.

- Les **tests unitaires** (*unit tests*) font en sorte de ne tester qu'une seule méthode chacun, avec une seule combinaison de valeurs pour ses paramètres. Cette méthode doit retourner un résultat spécifié. Il y a donc une multitude de tests unitaires, pour chaque classe, pour chaque méthode, pour chaque combinaison de valeurs qu'on peut passer à ces méthodes.
- Les **tests d'intégration** vérifient les interactions entre deux classes ou seulement deux méthodes, que l'une a appelé l'autre dans des conditions précises.
- Les **tests fonctionnels** vérifient les fonctionnalités visibles par l'utilisateur du logiciel, par exemple l'interface graphique.

Tests d'intégration



Consulter une page où cette image est présente ainsi que de nombreux exemples.

Complexité des tests

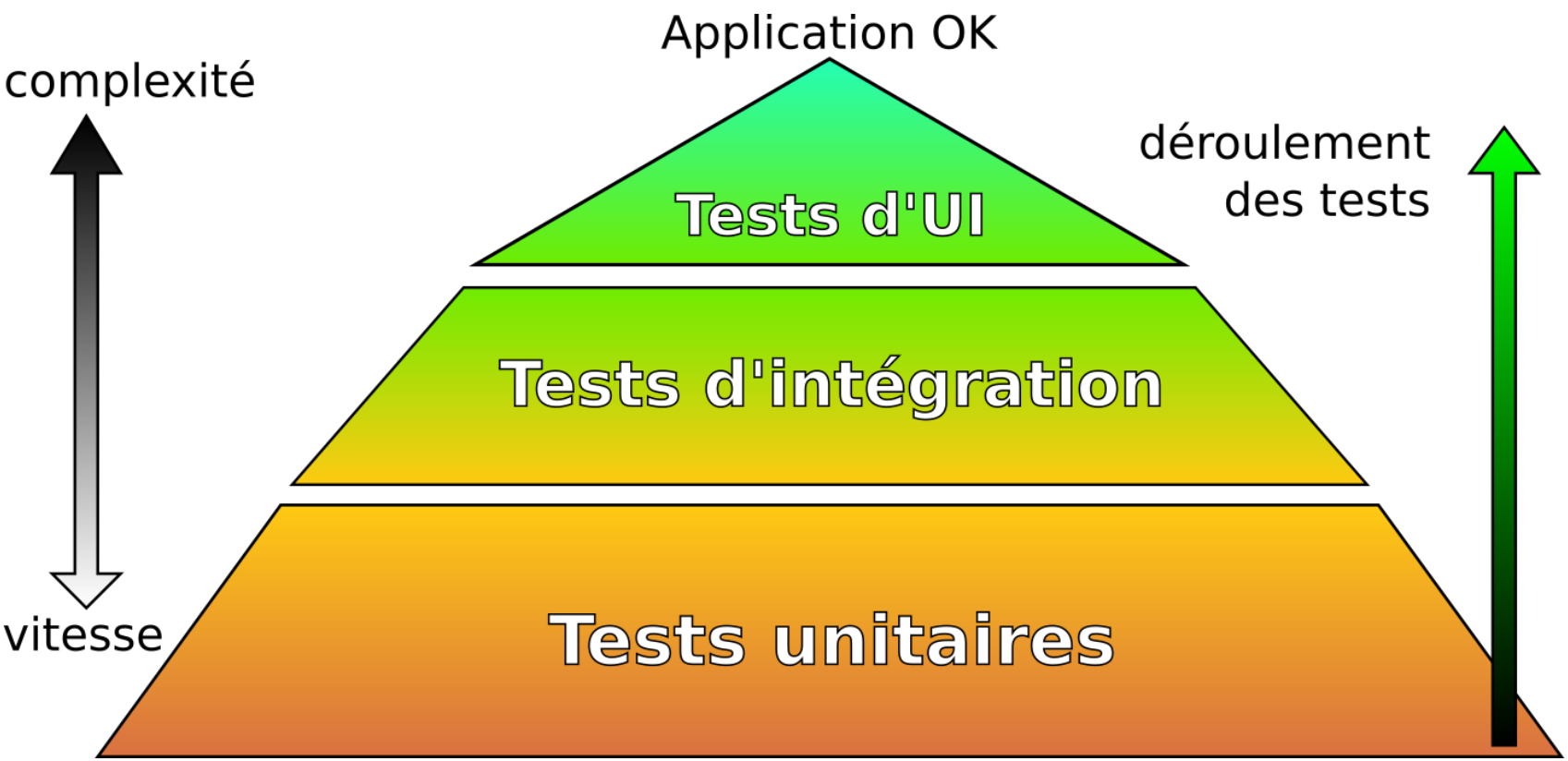
Ces catégories de tests n'ont pas tous la même complexité.

- Les tests unitaires sont rapides à vérifier, extrêmement nombreux, relativement simples à programmer, mais par définition, ils sont indépendants entre eux et travaillent sur des données figées, éloignées de ce que les utilisateurs pourront faire.
- Les tests d'intégration sont intermédiaires, moyennement nombreux, pas très faciles à programmer, pas très rapides.
- Les tests fonctionnels sont de loin les plus lents et les plus complexes à programmer. Ils correspondent à des scénarios d'usages réalistes par les utilisateurs (*use cases*).

On ne doit faire passer les suivants que si les précédents ont réussi.

Pyramide des tests

En résumé :



Pierre Nerzic

Couverture des tests

On doit, en principe, tester tout le logiciel. C'est ce qu'on appelle la **couverture des tests** (*test coverage*).

L'idée est de vérifier si toutes les lignes de programme font l'objet d'un test unitaire, ce qui témoigne de la qualité du logiciel :

- toutes les affectations, tous les calculs, tous les appels de méthodes...
- toutes les alternatives (telle condition vraie, telle condition fausse)...
- toutes les itérations (aucune boucle, au moins une boucle)...
- les interceptions d'exceptions (pas d'exception, une exception)...

Les grandes entreprises exigent de leurs sous-traitants un taux de couverture approchant 100%.

Intérêt des tests

Les classes de tests accompagnent le logiciel durant toute sa vie. Le moindre changement est confronté aux tests :

- Les tests doivent continuer à réussir. Si ce n'est pas le cas, alors le changement a causé une **régression**, c'est à dire un retour à un moins bon fonctionnement, le retour d'un bug, etc.
- Il est important de découvrir des problèmes le plus tôt possible dans la vie d'un logiciel. Les coûts de correction d'une erreur augmentent très fortement en fonction du retard dans sa découverte, car de plus en plus de lignes de code sont à revoir.
- Il se peut aussi que les tests doivent évoluer, en particulier en développement Agile. Il faut maintenir les tests à chaque évolution du cahier des charges.
- On peut même se servir des tests pour guider le développement (**TDD** = *Test Driven Development*).

Test Driven Development

Le principe, voir wikipedia, est d'organiser le travail en cycles très courts :

1. Écriture de tests destinés à vérifier une nouvelle fonctionnalité ; en principe, ces tests échouent car la fonctionnalité n'est pas encore programmée,
2. Programmation de la fonctionnalité pour faire réussir les tests,
3. Nettoyage, simplification du code source, en préservant la réussite de tous les tests.

L'inconvénient est qu'il est préférable de confier la programmation des tests et des méthodes testées à des personnes différentes, afin de ne pas commettre les mêmes erreurs de conception (oublier de définir ce qui arrive aux valeurs `null`, par exemple).

Tests unitaires en Java

JUnit5

En TP, nous allons utiliser un outil appelé *JUnit5*. C'est un ensemble de classes permettant de programmer des tests unitaires, et de les exécuter.

Les tests unitaires sont chacun dans une méthode annotée par `@Test`. Cette méthode est dans une classe qui a le même *package* que la classe qu'elle teste.

Et également, cette classe est dans un autre dossier que celui des sources du projet (voir l'organisation Maven en TP).

Pour finir, la classe des tests ne contient pas de méthode `main`.

Exemple de classe et de test

Classe testée :

```
package fr.iutlan.ql.persons;


class Person {

    public Person(...) {
        ...
    }

    public String getInfo() {
        ...
    }
}
```

Classe de test :

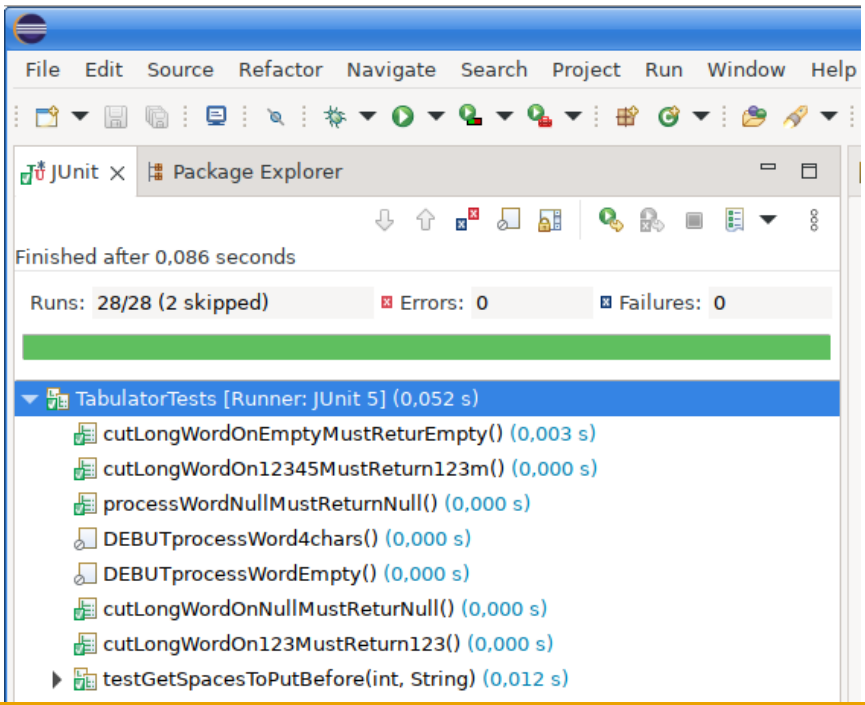
```
package fr.iutlan.ql.persons;

import ...(JUnit5)...

class TestsPerson {

    @Test
    void personMust...() {
        ...getInfo()...
    }

    ...
}
```


Pierre Nerzic

Lancement des tests

L'exécution des tests est entièrement automatisée. Dans Eclipse, on a un menu `Run as... JUnit Test` et dans Maven, les tests sont faits obligatoirement avant la construction du *jar* (voir en TP).

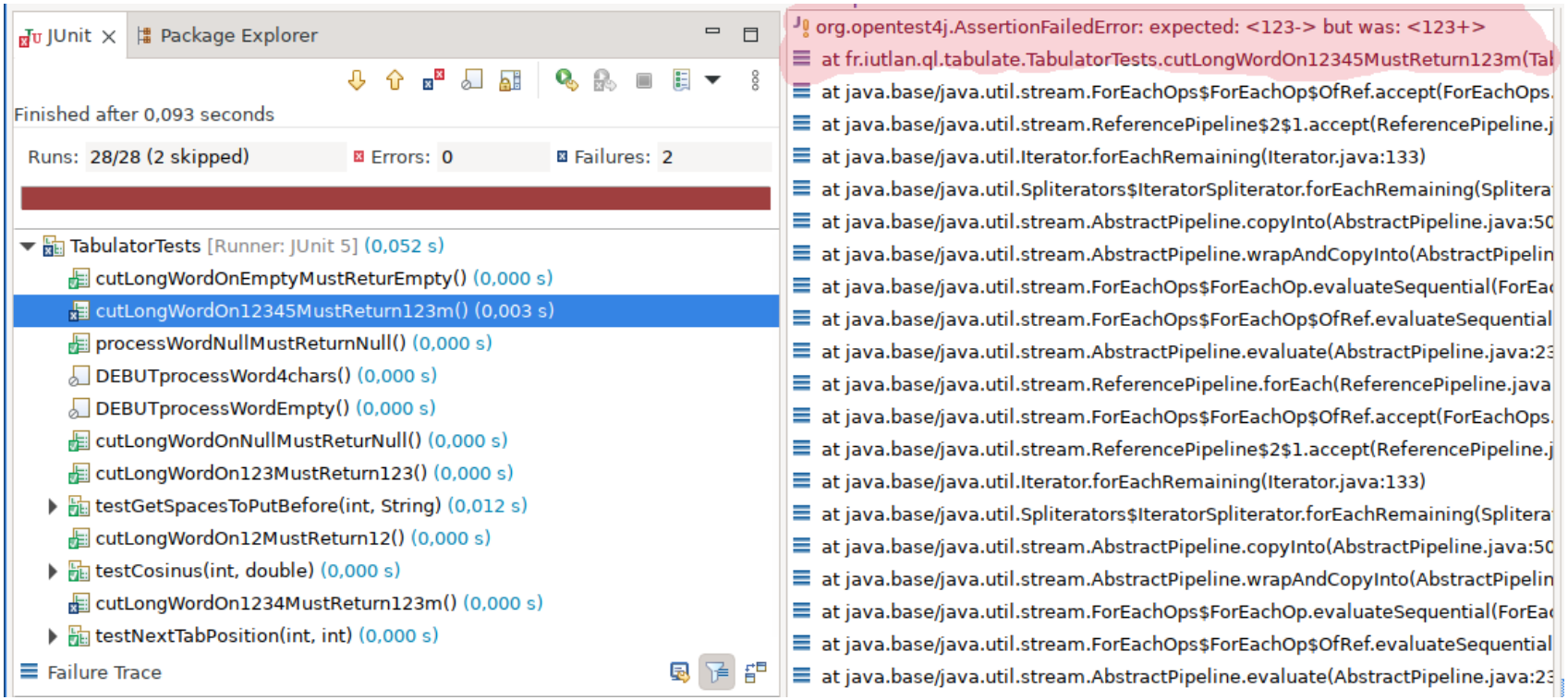
Voici ce qu'on voit dans Eclipse quand les tests ont réussi :



Pierre Nerzic

Analyse d'un test échoué

Par contre, en cas d'échec, les résultats sont assez horribles :



Analyse d'un test échoué, suite

Il faut trouver `expected: <chose> but was: <chauze>`, puis aller voir ce qu'il en est à la fois dans le test lui-même et dans la méthode qui est testée.

Cela nous montre pourquoi les tests unitaires doivent être les plus élémentaires possible : pour isoler les problèmes et n'avoir aucune difficulté à comprendre ce qui n'a pas marché et quoi corriger.

Cela passe par une bonne présentation des tests : leur nom et la structuration des opérations de test à l'intérieur. En effet, ces tests ayant la durée de vie du logiciel et étant très nombreux, il n'est pas question de devoir se plonger dans du code compliqué six mois après.

Normes de présentation des tests unitaires

Les tests sont des méthodes dans une classe. La classe elle-même doit être nommée pour faire référence à celle qu'elle teste. Et chaque méthode doit porter un nom, en CamelCase, pouvant être très long, explicitant totalement ce qu'elle fait, de manière à ce qu'une seule lecture suffise.

Exemples :

- `setLastnameWithNullMustThrowNullPointerException`
- `getFirstnameOnUninitializedInstanceMustReturnNull`
- `setAddressEmptyMustThrowIllegalArgumentException`
- `getCarsCountOnEmptyCarsListMustReturnZero`

Il peut vraiment y avoir des centaines de méthodes comme ça, donc interdiction absolue de noms comme `test1`, `testOk`, etc.

Normes de présentation des tests unitaires, suite

Ensuite, le corps de chaque méthode de test doit être organisé de la manière suivante :

- Une partie appelée **Arrange** ou *Given* qui prépare des données en vue de mener un test, par exemple créer une instance de la classe testée et la paramétrer avec des méthodes testées par ailleurs elles-aussi.
- Une partie **Act** ou *When* qui effectue le test unitaire.
- Une partie **Assert** ou *Then* qui compare le résultat à ce qui est attendu.
- Ces trois parties sont à séparer par une ligne vide.

Cette norme s'appelle **AAA** et nous l'appliquerons en TP.

Exemple de méthode JUnit5 en AAA

```
@Test
void personToStringMustReturnFirstNameSpaceLastName() {
    // ARRANGE
    Person pers = new Person();
    pers.setLastName("Nerzic");
    pers.setFirstName("Pierre");

    // ACT
    String result = pers.toString();

    // ASSERT
    assertEquals("Pierre Nerzic", result);
}
```

Intérêt de ces normes

L'intérêt de ces normes est de permettre la compréhension des tests unitaires par n'importe qui dans l'équipe.

Le pire serait d'avoir des méthodes de test bizarres, voire boguées, difficile d'être certain de ce qu'elles testent.

La contrainte, c'est de perdre un peu de temps à faire une jolie présentation. En même temps, c'est un signe de qualité de votre travail.

Rq: dans quelques cas, le patron *AAA* doit être un peu modifié, dans le cas des exceptions ou de mises en place en cascade.

Assertions JUnit5

Les vérifications sont exprimées sous forme d'**assertions**. C'est à dire des propositions qui doivent être vraies, par exemple telle méthode retourne telle valeur quand on l'appelle avec tels paramètres.

JUnit5 n'a pas beaucoup d'assertions à proposer :

- `assertEquals(attendu, résultat)` : le résultat doit être égal à attendu
- `assertNotEquals(attendu, résultat)` : le résultat ne doit pas être égal à attendu
- `assertNull(résultat)` et `assertNotNull(résultat)` : le résultat est comparé à `null`
- `assertTrue(résultat)` et `assertFalse(résultat)`

Assertions pour les exceptions

Dans le cas des exceptions, il faut faire ceci :

```
@Test
void addCarNullMustThrowNullPointerException() {
    // ARRANGE
    Person pers = new Person();

    // ASSERT
    assertThrows(NullPointerException.class, () -> {

        // ACT
        pers.addCar(null);
    });
}
```

Le test réussit si la partie *ACT* déclenche l'exception attendue.

Initialisation des tests

Il est fréquent d'avoir besoin de la même initialisation pour chaque test, par exemple, la même création et paramétrage d'instance.

Au lieu de répéter les mêmes instructions au début de chaque test, il y a la possibilité de définir :

- Une méthode d'initialisation de *chaque* test ; elle est annotée avec `@BeforeEach` au lieu de `@Test`,
- Une méthode d'initialisation de *tous* les tests de la classe, annotée `@BeforeAll`, **attention** la méthode doit être `static`.

Attention à la différence : la première est ré-exécutée pour chaque test, la seconde n'est exécutée qu'une seule fois en tout.

Exemple avec @BeforeEach

```
class TestPerson {

    Person persPN;

    @BeforeEach
    void initPersPN() {
        persPN = new Person();
        pers.setLastName("Nerzic");
        pers.setFirstName("Pierre");
    }

    @Test
    void personMust...() {
        ...
    }
```

Exemple avec @BeforeAll

```
class TestPerson {

    static Person persPN;

    @BeforeAll
    static void initPersPN() {
        persPN = new Person();
        pers.setLastName("Nerzic");
        pers.setFirstName("Pierre");
    }

    @Test
    void personMust...() {
        ...
    }
```


Pierre Nerzic

Différences entre ces deux exemples

Dans le second, `@BeforeAll`, la méthode `initPersPN` n'est appelée qu'une seule fois en tout. Donc l'instance de `Person` qui est employée par les tests est toujours la même. Si l'un des tests modifie cette instance, la modification sera transmise à tous les tests ultérieurs.

Dans le premier, `@BeforeEach` la méthode `initPersPN` est appelée autant de fois qu'il y a des tests, et avant chacun. Donc l'instance de `Person` est différente à chaque fois, mais initialisée de la même manière.

À quoi sert donc `@BeforeAll` ? À initialiser quelque chose qui doit persister, mais qui n'est pas affecté en soi par les tests. Par exemple à ouvrir une connexion avec une base de données.

Autres annotations

Pour libérer des ressources réservées par `@BeforeAll` et `@BeforeEach`, il y a les deux annotations inverses, `@AfterAll` et `@AfterEach`. Dans un environnement de tests, il est important de maîtriser les ressources utilisées. Tout ce qui a été alloué doit être libéré en miroir.

L'annotation `@Timeout(n)` permet de vérifier qu'une méthode ne dure pas plus de n secondes.

Si on veut désactiver temporairement un test, par exemple, pour qu'il ne gène pas le temps de le mettre au point, on peut lui ajouter l'annotation `@Disabled`.

Tests paramétrés

Mêmes tests avec des valeurs différentes

Soit une méthode à tester, qui demande des paramètres, par exemple numériques. Il est utile de vérifier le comportement face à la plus petite valeur possible, la plus grande possible, la valeur moyenne, des valeurs induisant des erreurs, etc.

Cela fait plusieurs tests très similaires, seules les valeurs fournies à la méthode changent. Faut-il créer autant de variantes du même test ?

Non : il est plus simple de définir un seul test, mais qui prend un paramètre, et fournir les valeurs qu'il devra passer à la méthode testée.

Mise en œuvre de tests paramétrés

Voici comment faire en *JUnit5* : on remplace `@Test` par `@ParameterizedTest` et on lui ajoute une autre annotation `@ValueSource` pour spécifier les valeurs à employer :

```
@ParameterizedTest
@ValueSource(ints = { 18, 19, 22, 34 })
void checkAgeIsAdult(int age) {
    // ARRANGE
    Person pers = new Person();
    pers.setAge(age);
    // ACT
    boolean result = pers.isAdult();
    // ASSERT
    assertTrue(result);
}
```

Types des valeurs

L'annotation `@ValueSource(TYPE = { VALEURS })` sert à spécifier les données à passer à la fonction de test.

Le *TYPE* est, entre autres, `ints`, `longs`, `floats`, `doubles`, `strings`, correspondant aux valeurs.

Pour fournir plusieurs paramètres ou des objets, c'est un peu plus compliqué.

Fournisseur de valeurs

Pour plusieurs paramètres ou de types objets, il est pratique d'utiliser un *générateur de valeurs*. C'est une méthode qui retourne une liste de n-uplets constituant les valeurs à fournir à la méthode de test.

On a deux choses. D'abord la méthode de test :

```
@ParameterizedTest
@MethodSource
void testFtOnAReturnsB(int paramA, string paramB) {
    ...
}
```

Il y a cette nouvelle annotation `@MethodSource` qui indique que les valeurs sont générées par une méthode portant exactement le même nom que le test, mais *statique* et surchargée différent.

Fournisseur de valeurs, suite

Cette seconde fonction doit retourner une liste de n-uplets contenant les valeurs. Un n-uplet en JUnit5 est une instance de la classe `Arguments`. Cette classe possède un constructeur de type *fabrique* avec la méthode `of`.

```
// méthode qui génère les arguments pour testFtOnAReturnsB
private static Stream<Arguments> testFtOnAReturnsB() {
    return Stream.of(
        Arguments.of( 10, "ten"),
        Arguments.of(-23, "minus twenty three"),
        Arguments.of(  7, "seven")
    );
}
```

La classe `Stream` est une sorte de liste améliorée. On la remplit avec des instances de `Arguments` destinées à la méthode de test.

AssertJ et Truth

Insuffisances de JUnit

Avec ses trop peu nombreuses assertions, JUnit5 rend difficile des vérifications comme : « *tous les éléments de la liste doivent être des entiers positifs* », ou « *au moins l'un des éléments de la collection doit être différent de* `null` ».

Comment feriez-vous sans écrire un algorithme potentiellement faux ? Le comble serait de faire une erreur algorithmique dans une assertion.

C'est pour cela que des librairies tierces ont été proposées, *Hamcrest*, *AssertJ* et *Truth* notamment. *Hamcrest* est plutôt difficile à utiliser à cause de la syntaxe des *matchers* utilisant la généricité, très complexe en Java.

Et pour choisir entre *AssertJ* et *Truth*, voir cette discussion. Elles se ressemblent énormément.

Présentation rapide de Truth et AssertJ

Le principe, c'est d'écrire des sortes de phrases, ressemblant à de l'anglais, exprimant ce qu'il faut vérifier.

Quelques exemples *AssertJ*, presque identiques en *Truth* :

```
assertThat(result).isNotNull();
assertThat(result).isEqualTo(555);
assertThat(result).isGreaterThan(0);

assertThat(message).contains("jour");
assertThat(message).startsWith("bon");
assertThat(message).hasSizeBetween(5, 10);
assertThat(message).matches("b[a-z]*r");
```

Il y a une multitude de possibilités qui seront détaillées en TP.

Pierre Nerzic

Comparaisons entre AssertJ et Truth

- Documentation : la documentation de AssertJ est bien meilleure (explications, exemples) que celle de Truth (uniquement une FAQ et la JavaDoc).
- Programmation fluide (*fluent*) en *AssertJ*, impossible en *Truth*[1]

```
assertThat(colors)
    .isNotEmpty()
    .hasSize(3)
    .doesNotHaveDuplicates()
    .contains("white", "red", "blue")
    .startsWith("white")
    .endsWith("red")
    .containsSequence("white", "blue")
    .doesNotContain("black", "orange", null);
```

[1] car les méthodes retournent `void`, malgré les prétentions de la doc.

Comparaisons entre AssertJ et Truth, suite

Certains tests pas possibles directement avec Truth :

- il n'y a aucun `null` dans la liste :
  Truth `assertThat(liste).doesNotContain(null);`
  AssertJ `assertThat(liste).doesNotContainNull();`
- il y a au moins un `null` dans la liste :
  Truth `assertThat(liste).contains(null);`
  AssertJ `assertThat(liste).containsNull();`
- il n'y a pas que des `null` dans la liste (pas avec Truth ⇒ JUnit5+Java) :
  J `assertTrue(liste.stream().anyMatch(n -> n != null));`
  A `assertThat(liste).anyMatch(n -> n != null);`
- il n'y a que des `null` dans la liste (pas avec Truth ⇒ JUnit5+Java) :
  J `assertTrue(liste.stream().allMatch(n -> n == null));`
  A `assertThat(liste).allMatch(n -> n == null);`

Mode d'emploi complet en TP...

C'est tout pour aujourd'hui

Cette présentation est finie. Rendez-vous en TD et TP pour mettre cela en pratique.

Pierre Nerzic