Test Driven Development en Java

Dr. Ing. Chiheb Ameur ABID

Contact: chiheb.abid@gmail.com

Janvier 2023

Version 1.1

Plan

Les tests

Qu'est ce le test ?

- Test d'un système = processus d'essai des exécutions d'un système selon un certain critère. L'observation de chaque exécution est comparée avec la spécification du système sous test :
  - Si conforme : test passé (ACCEPT)
  - Sinon : test échoué (FAIL)

Les tests

C'est quoi un test ?

- Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus]. IEEE (Standard Glossary of Software Engineering Terminology)
- Tester c'est réaliser l'exécution du programme
- Oracle : résultats attendus d'une exécution du logiciel
- Coût du test : 30
- Deux grandes familles de tests
  - Test fonctionnel (ou test boîte noire)
  - Test structurel (ou test boîte blanche)
- Cas de test (TC) : une exécution du programme déclenchée par des données de test (DT).
- Suite de tests (TS) : un ensemble de DT.
- Objectif de test (TO) : comportement de la spéc à tester.
- Système sous test (SUT ou IUT) : implémentation du système à tester.

Les tests

Difficultés du test

- Le test exhaustif est en général impossible à réaliser :
  - L'ensemble des données d'entrée est en général infini ou de très grande taille
  - La qualité du test dépend de la pertinence du choix des données de test
- Difficultés d'ordre psychologique ou culturel
  - Le test est un processus destructif : un bon test est un test qui trouve une erreur. Alors que l'activité de programmation est un processus constructif.

Les tests

Niveaux et phases

- Niveaux de tests
  - Test unitaire = test des (petites) parties du code, séparément.
  - Test d'intégration = test d'un ensemble de parties du code qui coopèrent.
  - Test du système = test du système entier, en inspectant sa fonctionnalité.
  - Test d'acceptation = effectué par le client pour s'assurer de la conformité au besoin.
- Phases de tests
  - Test de régression = test réalisé pendant la maintenance aprés un changement, afin de s'assurer que les système continue de fonctionner correctement.
  - Test de robustesse = tester des entrées non-prévues.
  - Test sous stress = tester en conditions de surcharge.

Définition et principes du TDD

Approche classique du test

- L'approche classique du test consiste à coder puis tester à la fin
  - ✖ Il ne reste plus de temps pour tester !
  - ✖ Plus le bug est détecté tard, plus il remet de choix en cause, et plus sa correction est coûteuse
  - ✖ Comme le code est déjà écrit, on risque d'écrire — le nez sur le code — un test faux qui valide un code faux
- ☞ Un code non ou mal testé n'a aucune valeur

Définition et principes du TDD

TDD : développement piloté par le test

- On écrit le test avant d'écrire le code ☞ Réfléchir à ce que fait le code avant de coder
- Une ligne de code n'est écrite que si un test la rend nécessaire ☞ Impossible de livrer un code non testé
- Dans la mesure du possible on laisse l'architecture émerger au fil du développement (ce qui n'interdit pas une phase légère de conception à base d'UML), ce qui garantit une architecture testable et faiblement couplée ☞ le TDD est plus une approche de conception objet de qualité que de recherche de bugs
- On considère les tests comme une documentation exécutable de l'API du système, une spécification par l'exemple, et on les bichonne autant que son code.

Définition et principes du TDD

Comment un TDD peut améliorer la qualité d'un programme ?

- Élaborer les tests avant chaque partie développée
  - Ne pas livrer un code non testé
  - Détecter les erreur le plus tôt possible
- En découpant le problème en petites parties et les tester chacune à part.
  - Ceci réduit le taux d'erreur et augmente le temps de développement des applications.
  - Il réduit aussi le temps de maintenance.
- Séparation entre le code de tests et le code applicatif

Le test dans le processus de développement

Principe du TDD

- En TDD, vous allez écrire les solutions les plus basiques possibles pour faire passer vos tests.
- Une fois que vous avez écrit un bon test avec le code le plus simple possible, vous avez fini – et vous pouvez avancer au prochain test.

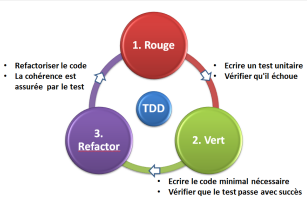
TDD : les trois bonnes règles

- Il y a trois règles à respecter, selon Robert Martin (un leader dans le monde de TDD)
  1. Écrire un test qui échoue avant d'écrire votre code lui-même.
  2. Ne pas écrire un test compliqué.
  3. Ne pas écrire plus de code que nécessaire, juste assez pour faire passer le test qui échoue.

Le test dans le processus de développement

Cycle de TDD

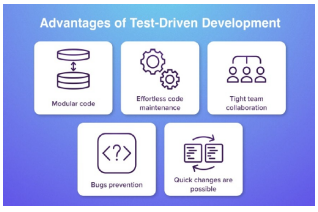
- Écrire un test
- Exécuter le test et constater qu'il échoue (barre rouge) ; si le verdict est en fait une erreur due au fait que le code ne compile pas (en Java) car le code applicatif n'a pas encore été écrit, alors écrire le code applicatif minimal du point de vue du langage, et vérifier cette fois que le test échoue à cause de l'oracle
- Écrire le code applicatif le plus simple qui permet de faire passer le test, et seulement ce code
- Lancer le test et vérifie qu'il passe (barre verte)
- Réusiner les tests et le code



Le test dans le processus de développement

Les gains du TDD

- Le test unitaire fournit un retour constant sur les fonctions
- La qualité de la conception augmente, ce qui contribue davantage à un bon entretien
- Le développement piloté par les tests agit comme un filet de sécurité contre les bogues
- TDD garantit que votre application répond réellement aux exigences définies pour elle
- TDD a un cycle de vie de développement très court



Plan

Présentation du framework JUnit

Le besoin d'un framework de test ?

- L'utilisation d'un framework des tests automatisés augmentera la vitesse et l'efficacité des tests d'une équipe,
- Améliorer la précision des tests et réduire les coûts de maintenance des tests ainsi que les risques.

Le framework JUnit

- JUnit est un framework de tests unitaires open source pour JAVA. Il est utile pour les développeurs Java d'écrire et d'exécuter des tests reproductibles
- JUnit est intégré à Eclipse

Le framework JUnit 4

Les annotations de JUnit

- Des annotations ont été introduites dans JUnit 4 rendant le code Java plus lisible et simple.
  - JUnit 4 est basée sur les annotations
  - Nécessite Java 5 ou supérieur
- Il n'est plus nécessaire d'imposer un nom pour les méthodes de test
- `static imports` pour les assertions

Tests automatisés avec le framework JUnit

Principe

- Une classe de tests unitaires est associée à une autre classe
- Une classe de tests unitaires hérite de la classe `junit.framework.TestCase` pour bénéficier de ses méthodes de tests
- Les méthodes de tests sont identifiées par des *annotations* `Java`

Méthodes de tests

- Nom quelconque
- Visibilité public, type de retour `void`
- Pas de paramètre, peut lever une exception
- Annotée `@Test`
- Utilise des instructions de test

Tests automatisés avec le framework JUnit

Exécution d'un test

1. Initialisation (Setup)
   - ☞ Définir le contexte et l'environnement du test
2. Exercice (Trigger)
   - ☞ Appel de l'unité à tester
3. Vérification (Verify)
   - ☞ Vérification du résultat ou état produit
4. Désactivation (Teardown)
   - ☞ Nettoyage, remettre le système dans son état initial

Tests automatisés avec le framework JUnit

Anatomie d'un test unitaire

```
@Test // Annotation designant la methode comme un test
public void testXXX() {
        //Define
                Instructions de mise en contexte
        //When
                Instruction sous test
        //Then
                Observation et verification de l'oracle
}
```


Tests automatisés avec le framework JUnit

Les annotations de JUnit 4

- Une annotation est désignée par un nom précédé du caractère `@`
- Une annotation précède l'entité qu'elle concerne
- Toutes les annotations sont définies dans le package `org.junit`

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

Tests automatisés avec le framework JUnit

Instructions de test

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

☞ L'instruction la plus importante est `fail()` : les autres ne sont que des raccourcis d'écriture !

Tests automatisés avec le framework JUnit 4

Exemple simple

- Créer un nouveau projet Java
- On désire tester la méthode ci-après où on a introduit volontairement une erreur (classe Example)

```
public static int somme(int a,int b,int c) {
    return a+b-c;
}
```

- Créer un Junit Test Case implémentant le test suivant :

```
@Test
public void test() {
    int resultat= Example.somme(5,4,2);
    assertEquals(11,resultat);
}
```

- Tester le cas de test à partir du menu contextuel : choisir `Run As -> JUnit Test`

Tests automatisés avec le framework JUnit 4

Méthodes de test

- Une méthode de test est une méthode qui exécute un test unitaire
- Les méthodes de test sont annotées avec `@Test`
- Convention de nommage d'une méthode de test `test[méthode à tester]()`
  - Mais aucune obligation (nom quelconque possible)
- Publique, sans paramètre, type de retour `void`
- Les principaux paramètres de l'annotation `@Test`
  - `expected` : le test échoue si une exception n'est pas levée
  - `timeout` : durée maximale spécifiée en millisecondes
- L'annotation `@Ignore` permet d'ignorer un test

Tests automatisés avec le framework JUnit 4

Méthodes de test : vérification de levée d'une exception

- On peut utiliser l'attribut `expected` de l'annotation `@Test`

```
@Test(expected = NullPointerException.class)
public void whenExceptionThrown_thenExpectationSatisfied() {
  String test = null;
  test.length();
}
```

- En utilisant l'annotation `@Rule`
  - Elle permet de vérifier certaines propriétés de l'exception levée

```
@Rule
public ExpectedException exceptionRule = ExpectedException.none();

@Test
public void whenExceptionThrown_thenRuleIsApplied() {
  exceptionRule.expect(NumberFormatException.class);
  exceptionRule.expectMessage("For input string");
  Integer.parseInt("1a");
}
```


Tests automatisés avec le framework JUnit 4

Méthodes d'initialisation et de finalisation

- Méthodes d'initialisation utilisée avant chaque test
  - `setUp()` est une convention de nommage
  - L'annotation `@Before` est utilisée
- Méthodes de finalisation utilisée après chaque test
  - `tearDown()` est une convention de nommage
  - L'annotation `@After` est utilisée

Tests automatisés avec le framework JUnit

Méthodes d'initialisation et de finalisation

- Méthodes d'initialisation globale
  - Annotée `@BeforeClass`
  - Publique et statique
  - Exécutée une seule fois avant la première méthode de test
- Méthode de finalisation globale
  - Annotée `@AfterClass`
  - Publique et statique
  - Exécutée une seule fois après la dernière méthode de test
- Dans les 2 cas, une seule méthode par annotation

Tests automatisés avec le framework JUnit

Exemple du TDD

- En appliquant les principes de TDD, notre objectif est de développer une fonction permettant valider un mot de passe
  - Le mot de passe doit comprendre entre 5 et 10 caractères

Tests automatisés avec le framework JUnit

❶ Écrire un test

- On commence par écrire un test

```
package j1exemple1;
import static org.junit.Assert.*;
import org.junit.Assert;
import org.junit.Test;

class ValidPasswordTest {
  @Test
  void test() {
    PasswordValidator pv=new PasswordValidator();
    Assert.assertEquals(true,pv.isValid("123456"));
  }
}
```

- Pour exécuter le test, à partir du menu contextuel, choisir `Run As -> JUnit Test`
- Évidemment, on est en rouge, puisque le code applicatif n'existe pas

Tests automatisés avec le framework JUnit

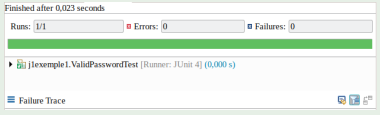
❷ Écrire le code applicatif

- On écrit le code applicatif pour faire passer le test

```
package j1exemple1;

public class PasswordValidator {
  public boolean isValid(String pw) {
    if (pw.length()>=5 && pw.length()<=10)
          return true;
        else
          return false;
  }
}
```

- Résultat de l'exécution du test : succès



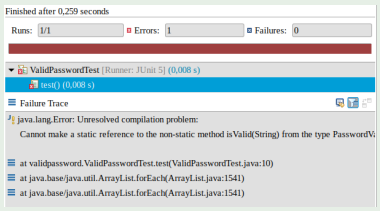
❸ Reusiner

- On optimise notre code de test
  - Il n'est pas nécessaire de créer une instance de la classe `PasswordValidator`
  - Associer un message personnalisé à l'assertion

```
package jlexemple1;
import static org.junit.Assert.*;
import org.junit.Assert;
import org.junit.Test;

class ValidPasswordTest {
  @Test
  void test() {
    Assert.assertEquals("Verifier longueur mot de passe",true,PasswordValidator.isValid("123456
        "));
  }
}
```

- Résultat de l'exécution de test : échec



❹ Reusiner le test

- On corrige le code applicatif pour passer le test

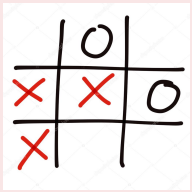
```
package jlexemple1;

public class PasswordValidator {
  static public boolean isValid(String pw) {
      if (pw.length()>=5 && pw.length()<=10)
    return true;
      else
        return false;
  }
}
```

- Résultat de l'exécution de test : succès

Exercice : jeu de Tic-Tac-Toe

- On se propose de développer certaines fonctionnalités du célèbre jeu de Tic-Tac-Toe. Ce jeu se déroule sur une grille 3x3 où deux joueurs posent leurs pions dans le but d'être le premier à en aligner trois.



- On désire commencer par implémenter la méthode `play (int x,int y)` permettant de vérifier que les coordonnées `x` et `y` de placement d'un pion sur la grille sont valides. Dans le cas contraire, la méthode `play` doit générer une exception de type `RuntimeException`
  - D'abord, effectuer un test de dépassement selon l'axe des `X`.
  - Puis, effectuer un test de dépassement selon l'axe des `Y`.

Exercice : jeu de Tic-Tac-Toe

- Ajouter la fonctionnalité permettant de garantir que le placement d'un pion ne s'effectue que dans une case vide. En cas d'échec, une exception de type `RuntimeException` est générée indiquant que la case est déjà occupée
- Réusiner le code
  - Prévoir une fonction permettant de vérifier s'il y a un dépassement sur un axe
  - Prévoir une fonction pour effectuer le placement d'un pion

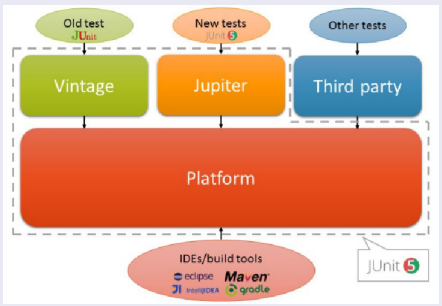
Tests automatisés avec le framework JUnit 5

JUnit5 : la nouvelle version

- JUnit 5 est une réécriture complète de l'API de JUnit 4 en Java 8
  - Supporte les nouveautés de Java 8
- Un framework modulaire
  - JUnit 4 est livré sous forme d'un seul fichier `jar`
  - JUnit 5 est composé de trois projets : `Platform`, `Jupiter`, et `Vintage`
- JUnit 5 est extensible
- Les classes de tests JUnit 5 sont similaires à celles de JUnit 4 : basiquement, il suffit d'écrire une classe contenant des méthodes annotées avec @Test.

Tests automatisés avec le framework JUnit 5

Architecture de JUnit 5



Tests automatisés avec le framework JUnit 5

JUnit5 vs JUnit4 : les annotations

Junit 4 (org.juint)	Junit 5 (org.junit.jupiter.api)
Test	Test
Before	BeforeEach
BeforeClass	BeforeAll
After	AfterEach
AfterClass	AfterAll
Ignore	Disabled

JUnit5 vs JUnit4 : les assertions

JUnit 4	JUnit 5
assert*(message, expected, actual)	assert*(expected, actual, message)

Tests automatisés avec le framework JUnit 5

Méthodes de test : vérification de levée d'une exception

- L'attribut `expected` de l'annotation `@Test` n'est plus valide avec JUnit 5
- L'annotation `@Rule` est aussi enlevée de JUnit 5

```
@Test
public void whenExceptionThrown_thenAssertionSucceeds() {
    Exception exception = assertThrows(NumberFormatException.class, () -> {
        Integer.parseInt("1a");
    });

    String expectedMessage = "For input string";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}
```

  - Toute exception de type classe fille de l'exception attendue est acceptée

Tests automatisés avec le framework JUnit 5

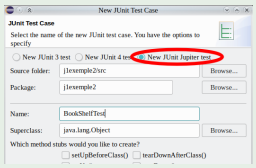
Exemple de TDD avec JUnit 5

On se propose d'organiser des livres dans une étagère. En suivant une démarche TDD, on se propose d'implémenter la fonctionnalité d'ajouter un livre. Pour simplifier, on commence par identifier un livre par une chaine de caractères, i.e. le type `String`.

Tests automatisés avec le framework JUnit 5

Exemple de TDD avec JUnit 5

- Commençons par écrire un premier test
  - Créer un nouveau projet `Java`
  - Choisir `File->New->JUnit Test Case` pour créer une classe de test. Vérifier bien l'utilisation de JUnit 5



- Nous écrivons un premier test permettant de vérifier que l'étagère est initialement vide

```
@Test
public void emptyBookShelfWhenNoBookAdded() {
  BookShelf shelf = new BookShelf();
  List<String> books = shelf.books();
  assertTrue(books.isEmpty(), () -> "BookShelf should be empty.");
}
```


Tests automatisés avec le framework JUnit 5

Exemple de TDD avec JUnit 5

- Pour passer au vert dans le cycle de TDD, on écrit une première version du code applicatif
  - Il s'agit de la classe qui représente une étagère

```
import java.util.Collections;
import java.util.List;

public class BookShelf {
  public List<String> books() {
    return Collections.emptyList();
  }
}
```


Tests automatisés avec le framework JUnit 5

Exemple de TDD avec JUnit 5

- On désire maintenant ajouter la fonctionnalité d'ajouter un livre dans une étagère.
- On propose d'écrire un test permettant de vérifier que si on effectue deux opérations d'ajout, alors on aurait bien deux livres dans l'étagère

```
@Test
void bookshelfContainsTwoBooksWhenTwoBooksAdded() {
  BookShelf shelf = new BookShelf();
  shelf.add("Programmer en Java");
  shelf.add("Tester avant");
  List<String> books = shelf.books();
  assertEquals(2, books.size(), () -> "BookShelf should have two books.");
}
```

Tests automatisés avec le framework JUnit 5

Exemple de TDD avec JUnit 5

- Modifier le code applicatif pour passer au vert

```
import java.util.ArrayList;
import java.util.List;
public class BookShelf {
  private final List<String> books = new ArrayList<>();
  public List<String> books() {
    return books;
  }
  public void add(String bookToAdd) {
    books.add(bookToAdd);
  }
}
```


Tests automatisés avec le framework JUnit 5

Exemple de TDD avec JUnit 5

- Est-il possible de modifier la liste retournée par la méthode `books()` ?
  - Violer le principe d'encapsulation ! ?
- Écrire un test pour vérifier cette contrainte

```
@Test
void booksReturnedFromBookShelfIsImmutableForClient() {
  BookShelf shelf = new BookShelf();
  shelf.add("Effective Java");
  shelf.add("Code Complete");
  List<String> books = shelf.books();
  try {
    books.add("The Mythical Man-Month");
    fail(() -> "Should not be able to add book to books");
  } catch (Exception e) {
    assertTrue(e instanceof UnsupportedOperationException, () -> "Should throw
        UnsupportedOperationException.");
  }
}
```


Tests automatisés avec le framework JUnit 5

Exemple de TDD avec JUnit 5

- Modifier le code applicatif pour passer au vert pour le dernier test
  - Il suffit de modifier le code de la méthode `books()` pour renvoyer une liste non modifiable

```
public List<String> books() {
  return Collections.unmodifiableList(books);
}
```

- Toujours, refaire les tests pour vérifier qu'ils sont au vert

Tests automatisés avec le framework JUnit 5

Exemple de TDD avec JUnit 5

- Peut-on améliorer la lisibilité du code de tests ? (Refactoring)
  - Utiliser l'annotation `@BeforeEach` pour effectuer l'initialisation une seule fois pour tous les tests

Tests automatisés avec le framework JUnit 5

Exemple de TDD avec JUnit 5

```
private BookShelf shelf;

@BeforeEach
void init() throws Exception {
  shelf = new BookShelf();
}

@Test
public void emptyBookShelfWhenNoBookAdded() {
  List<String> books = shelf.books();
  assertTrue(books.isEmpty(), () -> "BookShelf should be empty.");
}

[...]
```


Tests automatisés avec le framework JUnit 5

Exercice de TDD avec JUnit 5

1. En continuant l'exemple précédent, et en suivant une démarche de TDD, ajouter la fonctionnalité permettant de trier la liste des livres selon l'ordre lexicographique
2. On veut retourner une nouvelle liste triée en gardant la liste originale dans le même ordre

Tests automatisés avec le framework JUnit 5

Exemple illustratif du TDD avec JUnit 5

- En appliquant les principes de TDD, créer un projet avec `JUnit 5` qui permet de supprimer les lettres 'A' s'ils existent dans la première ou la deuxième position au début d'une chaine de caractères.
- Conditions à vérifier
  'A'—'', 'AB'—'B', 'AABC'—'BC', 'BACD'—'BCD', 'BBAA'—'BBAA', 'AABAA'—'BAA'

Tests automatisés avec le framework JUnit 5

Exemple du TDD avec JUnit 5

- On commence par écrire la classe de test

```
package testing;
import org.testng.Assert;
import org.testng.annotations.Test;
public class RemoveAFirstTest {
  @Test
  public void removeATest() {
        RemoveAfirst a=new RemoveAfirst();
        Assert.assertEquals("", a.removeLetterA("A"));
  }
}
```

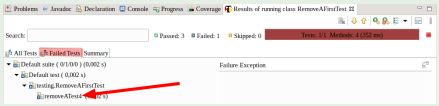
- On propose un première solution pour le code applicatif

```
package testing;
public class RemoveAFirst {
  public String removeLetterA(String chaine) {
    String nvChaine="";
        if (chaine.length()>2) {
          if (chaine.charAt(0)=='A' && chaine.charAt(1)=='A')
                nvChaine=""+chaine.substring(2);
    }
        else if (chaine.length()==2) {
          if (chaine.charAt(0)=='A')
        nvChaine=""+chaine.substring(1);
    }
        return nvChaine;
  }
}
```

Tests automatisés avec le framework JUnit 5

Exemple du TDD avec JUnit 5

- Continuer le cycle de développement TDD pour les autres conditions
- Le test de la 4ème condition échoue



- Modifier le code applicatif pour passer au vert : on introduit les variables `pos1` et `pos2` pour localiser les deux premières occurrences de `'A'`

```
public String removeLetterA(String chaine) {
  String nvChaine="";
  int pos1=-1,pos2=-1;
  if (chaine.length()>2) {
    pos1=chaine.indexOf('A');
        pos2=chaine.lastIndexOf('A');
        if (pos1==0 && pos2==1) nvChaine=""+chaine.substring(2);
    else if (pos1==1)  nvChaine=chaine.charAt(0)+chaine.substring(2);
  }
  else if (chaine.length()==2) {
    pos1=chaine.indexOf('A');
    if (pos1==0) nvChaine=""+chaine.substring(1);
    else if (pos1==1) nvChaine=""+chaine.charAt(0);
  }
  return nvChaine;
}
```


Tests automatisés avec le framework JUnit 5

Exemple du TDD avec JUnit 5

- En continuant les tests, cette fois c'est la 5ème condition qui provoque une erreur
- Modifier le code applicatif pour corriger l'erreur et passer au vert

```
public String removeLetterA(String chaine) {
  String nvChaine="";
  int pos1=-1,pos2=-1;
  if (chaine.length()>2) {
    pos1=chaine.indexOf('A');
        pos2=chaine.lastIndexOf('A');
        if (pos1==0 && pos2==1) nvChaine=""+chaine.substring(2);
    else if (pos1==1)  nvChaine=chaine.charAt(0)+chaine.substring(2);
    else nvChaine=chaine;
  }
  else if (chaine.length()==2) {
    pos1=chaine.indexOf('A');
    if (pos1==0) nvChaine=""+chaine.substring(1);
    else if (pos1==1) nvChaine=""+chaine.charAt(0);
  }
  return nvChaine;
}
```


Tests automatisés avec le framework JUnit 5

Exemple du TDD avec JUnit 5

- Le test de la 6ème condition provoque une erreur ! ! !
- Modifions le code applicatif pour corriger l'erreur

```
public String removeLetterA(String chaine) {
  String nvChaine="";
  int pos=-1;
  if (chaine.length()>2) {
  pos=chaine.indexOf('A');
  if (pos==0 && chaine.charAt(1)=='A')
    nvChaine=""+chaine.substring(2);
    else if (pos==1)  nvChaine=chaine.charAt(0)+chaine.substring(2);
    else nvChaine=chaine;
  }
  else if (chaine.length()==2) {
    pos=chaine.indexOf('A');
    if (pos==0) nvChaine=""+chaine.substring(1);
    else if (pos==1) nvChaine=""+chaine.charAt(0);
  }
  return nvChaine;
}
```


Plan

Tests paramétrisés

Tests paramétrisés

- `Junit 5` permet les tests paramétrisés en utilisant des différentes valeurs pour la même méthode de test
- Un test paramétrisé est exécuté autant de fois que les valeurs spécifiées sur ses entrées
- L'annotation `@ParameterizedTest`, du package `org.junit.jupiter.params`, est utilisée pour indiquer qu'un test est paramétrisé
- L'annotation `@ValueSource`, du package `org.junit.jupiter.params.provider`, permet de spécifier les valeurs sur les entrées
- Les types acceptés
  `short , byte ,int , long , float , double ,char , java.lang.String`

Tests paramétrisés

Exemple : un seul argument

- Soit à tester la méthode suivante

```
public static boolean isOdd(int number) {
    return number % 2 != 0;
}
```

- Code de la méthode de test

```
@ParameterizedTest
@ValueSource(ints = {1, 3, 5, -3, 15, Integer.MAX_VALUE})
void testOddShouldReturnTrueForOddNumbers(int number) {
    assertTrue(Operations.isOdd(number));
}
```


Tests paramétrisés

Plusieurs arguments

- L'annotation `@CsvSource` permet de spécifier des données au format CSV
- La conversion est effectué systématiquement à partir de type `String`

Exemple : plusieurs arguments

- La méthode à tester

```
public static int somme(int a, int b, int c) {
        return a+b+c;
}
```

- Code de la méthode de test

```
@ParameterizedTest
@CsvSource({"1,2,3","␣3,␣5,␣-3", "15,␣10,0"})
void testSommeShouldReturnSumofNumbers(int a,int b,int c) {
    assertEquals(a+b+c,Operations.somme(a,b,c));
}
```

Tests paramétrisés

Données à partir d'un fichier

- L'annotation `@CsvFileSource` permet de spécifier des données au format CSV à partir d'un fichier
- La première ligne du fichier csv doit contenir le nom des arguments

Exemple : données à partir d'un fichier

- Code de la méthode de test

```
@ParameterizedTest
@CsvFileSource(resources = "./test.csv", numLinesToSkip = 1)
void testSommeShouldReturnSumofNumbersfromFile(int a,int b,int c) {
    assertEquals(a+b+c,Operations.somme(a,b,c));
}
```

- Exemple de contenu du fichier `test.csv`

```
a,b,c
1,2,3
4,5,6
7,8,9
```

Tests paramétrisés

Exercice : données à partir d'un fichier

- Ré-écrire la classe de test de l'exemple 3 (test de la méthode `RemoveAFirst`) pour utiliser une seule méthode de test paramétrisée, dont les entrées sont spécifiées à partir d'un fichier csv.

Plan

Bonnes pratiques

Qualité d'un bon test

- Précis
  - Teste un comportement en particulier
  - Un test = une assertion
- Répétable et déterministe
  - S'attend toujours au même comportement
- Isolé et indépendant
  - Ne dépend de rien d'autre

Bonnes pratiques

Nommer les tests

- Pour les classes
  - Suffixer avec "Test"
- Pour les méthodes
  - Décrire le contexte et le résultat attendu
  - Approche Sujet_Scénario_Résultat

`ProductPurchaseAction_IfStockIsZero_RendersOutOfStockView()`

Bonnes pratiques

Tester les exceptions

- Lister les exceptions qui doivent être levées
- Écrire un test pour chaque exception
- Vérifier que l'exception est bien levée et que le message est clair, compréhensible

Ne jamais faire confiance au client de vos services

- Le client utilise mal vos services
  - Mauvais inputs, valeurs limites, arguments nuls, valeurs inattendues
  - Comportements limites, `pop()` sur une collection vide

Exercice : jeu de Tic-Tac-Toe (suite)

- En continuant l'exercice, on souhaite maintenant alterner le jeu entre les deux joueurs symbolisés par les types de leurs pions (les caractères 'X' et 'O'). On suppose toujours que le joueur X commence le jeu. En suivant une démarche de TDD, implémenter une fonction permettant de retourner le joueur qui va joueur à un moment donné
  - D'abord, écrire un test pour vérifier que le joueur X commence le jeu
  - Puis, écrire un deuxième test pour vérifier l'alternance entre les deux joueurs
- Développer la fonctionnalité pour déterminer le vainqueur
  - Définir un test qui vérifie que le jeu est en cours. La méthode `play` retourne l'état du jeu à travers une chaine de caractères
  - Définir les tests pour la vérification qu'un joueur gagne. On procède en écrivant un test pour chaque type de condition pour annoncer le vainqueur :
    1. Tous les pions sont placés sur la même ligne horizontale. Après avoir proposé une solution simple, ne pas oublier de réusiner le code
    2. Tous les pions sont placés sur la même ligne verticale
    3. Tous les pions sont placés sur la première diagonale
    4. Tous les pions sont placés sur la deuxième diagonale
- Développer la fonctionnalité qui annonce que le jeu se termine sans vainqueur. La grille doit être initialisée.

MERCI POUR VOTRE ATTENTION

Questions ?