



**ORSYS**  
formation



## Test Driven Development en Java

**Dr. Ing. Chiheb Ameur ABID**

*Contact: [chiheb.abid@gmail.com](mailto:chiheb.abid@gmail.com)*

Janvier 2021

Version 1.1

# Plan

- 1 Définition et principes du TDD
- 2 Tests automatisés avec le framework JUnit
- 3 Tests automatisés avec le framework TestNG
- 4 Tests paramétrisés
- 5 Bonnes pratiques

# 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

## 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 ?

- 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.

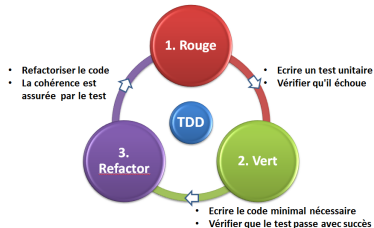
## TDD : les trois bonnes règles

- ➡ Il y a trois règles à respecter, selon Robert Martin (un leader dans le monde de TDD)
  - ❶ Écrire un test qui échoue avant d'écrire votre code lui-même.
  - ❷ Ne pas écrire un test compliqué.
  - ❸ 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

### Advantages of Test-Driven Development



Modular code



Effortless code maintenance



Tight team collaboration



Bugs prevention



Quick changes are possible

# Plan

- 1 Définition et principes du TDD
- 2 Tests automatisés avec le framework JUnit
  - Le framework JUnit 4
  - Le framework JUnit 5
- 3 Tests automatisés avec le framework TestNG
- 4 Tests paramétrisés
- 5 Bonnes pratiques



# 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



- └ Tests automatisés avec le framework JUnit
- └ Le framework JUnit 4

# 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
- └ Le framework JUnit 4

# 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
- └ Le framework JUnit 4

# 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
- └ Le framework JUnit 4

# 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
- └ Le framework JUnit 4

# 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 <i>avant chaque test</i>
@After	méthode exécutée <i>après chaque test</i>
@BeforeClass	méthode exécutée <i>avant le premier test</i>
@AfterClass	méthode exécutée <i>après le dernier test</i>
@Ignore	méthode qui ne sera pas lancée comme test

- └ Tests automatisés avec le framework JUnit
- └ Le framework JUnit 4

# Tests automatisés avec le framework JUnit

## Instructions de test

Instruction	Description
<code>fail(String)</code>	fait échouer la méthode de test
<code>assertTrue(true)</code>	toujours vrai
<code>assertEquals(expected, actual)</code>	teste si les valeurs sont les mêmes
<code>assertEquals(expected, actual, tolerance)</code>	teste de proximité avec tolérance
<code>assertNull(object)</code>	vérifie si l'objet est null
<code>assertNotNull(object)</code>	vérifie si l'objet n'est pas null
<code>assertSame(expected, actual)</code>	vérifie si les variables référencent le même objet
<code>assertNotSame(expected, actual)</code>	vérifie que les variables ne référencent pas le même objet
<code>assertTrue(boolean condition)</code>	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
- └ Le framework JUnit 4

# 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
- └ Le framework JUnit 4

# 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
- └ Le framework JUnit 4

# 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
- └ Le framework JUnit 4

# 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
- └ Le framework JUnit 4

# 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
- └ Le framework JUnit 4

# 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

## 1 Écrire un test

### ➤ On commence par écrire un test

```
package jlexemple1;
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



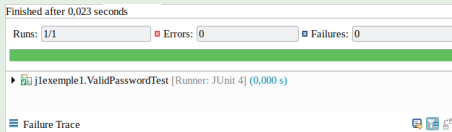
## ② Écrire le code applicatif

- ➡ 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



### ③ 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

Finished after 0,259 seconds

Runs: 1/1   Errors: 1   Failures: 0

ValidPasswordTest [Runner: JUnit 5] (0,008 s)

test() (0,008 s)

Failure Trace

java.lang.Error: Unresolved compilation problem:  
Cannot make a static reference to the non-static method isValid(String) from the type PasswordV;

at validpassword.ValidPasswordTest.test(ValidPasswordTest.java:10)  
at java.base/java.util.ArrayList.forEach(ArrayList.java:1541)  
at java.base/java.util.ArrayList.forEach(ArrayList.java:1541)



## ④ 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

Finished after 0,015 seconds

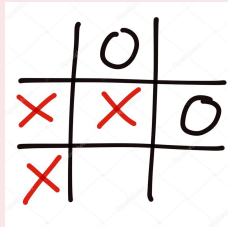
Runs:  Errors:  Failures:

 jlexemple1.ValidPasswordTest [Runner: JUnit 4] (0,000 s)

- └ Tests automatisés avec le framework JUnit
- └ Le framework JUnit 4

## 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`.

- └ Tests automatisés avec le framework JUnit
- └ Le framework JUnit 4

### 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
- └ Le framework JUnit 5

# Tests automatisés avec le framework JUnit 5

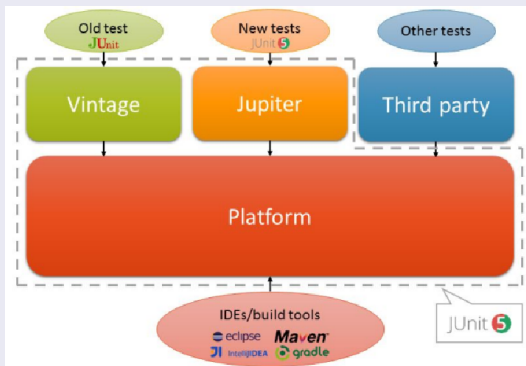
## JUnit5 : la nouvelle version

- 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
- └ Le framework JUnit 5

# Tests automatisés avec le framework JUnit 5

## Architecture de JUnit 5



- └ Tests automatisés avec le framework JUnit
- └ Le framework JUnit 5

# Tests automatisés avec le framework JUnit 5

## JUnit5 vs JUnit4 : les annotations

JUnit 4 (org.junit)	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
<code>assert*(message, expected, actual)</code>	<code>assert*(expected, actual, message)</code>

- └ Tests automatisés avec le framework JUnit
- └ Le framework JUnit 5

# 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
- └ Le framework JUnit 5

# 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 chaîne de caractères, i.e. le type `String`.



- Tests automatisés avec le framework JUnit

- Le framework JUnit 5

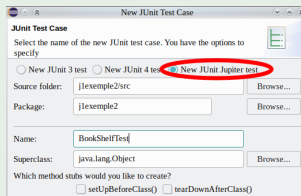
# Tests automatisés avec le framework JUnit 5

## 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
- └ Le framework JUnit 5

# 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
- └ Le framework JUnit 5

# 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
- └ Le framework JUnit 5

# 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
- └ Le framework JUnit 5

# 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()` ?
  - 🚫 Violier 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
- └ Le framework JUnit 5

# 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
- └ Le framework JUnit 5

# 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
- └ Le framework JUnit 5

# 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

# Plan

- 1 Définition et principes du TDD
- 2 Tests automatisés avec le framework JUnit
- 3 Tests automatisés avec le framework TestNG**
- 4 Tests paramétrisés
- 5 Bonnes pratiques

# Présentation du framework TestNG

## Le framework de test TestNG

- TestNG : Testing New Generation
- C'est un framework de tests unitaires open source pour JAVA. Il est inspiré par JUnit et NUnit
- L'objectif de TestNG étant de couvrir un large spectre de catégories de tests unitaires, d'intégration, système et d'acceptation

The logo for TestNG, featuring the word "Test" in black, "NG" in red, and a large yellow "G" to the right.

# Présentation du framework TestNG

## Avantages de TestNG

- Offre un rapport bien détaillé qui explique les tests effectués (succès ou erronés)
- Le regroupement des tests cases sont plus facilement regroupés grâce à un fichier au format XML en précisant la priorité de quel test case en 1er lieu , etc.
- Il est plus facile d'utiliser la priorité des tests à réaliser
- Il est possible d'effectuer des tests en parallèles sur différents navigateurs

# Présentation du framework TestNG

## JUnit vs TestNG

Feature	JUnit	TestNG
test annotation	@Test	@Test
run before all tests in this suite have run	—	@BeforeSuite
run after all tests in this suite have run	—	@AfterSuite
run before the test	—	@BeforeTest
run after the test	—	@AfterTest
run before the first test method that belongs to any of these groups is invoked	—	@BeforeGroups
run after the last test method that belongs to any of these groups is invoked	—	@AfterGroups
run before the first test method in the current class is invoked	@BeforeClass	@BeforeClass
run after all the test methods in the current class have been run	@AfterClass	@AfterClass
run before each test method	@Before	@BeforeMethod
run after each test method	@After	@AfterMethod
ignore test	@Ignore	@Test(enable=false)
expected exception	@Test(expected = ArithmeticException.class)	@Test(expectedExceptions = ArithmeticException.class)
timeout	@Test(timeout = 1000)	@Test(timeout = 1000)

# Présentation du framework TestNG

## Le fichier de configuration `testng.xml`

- C'est le fichier de configuration du lancement des tests pour un projet
- Il permet de configurer les tests à lancer en fonction de leurs packages, classes, groupes ou encore par groupes de groupes !
- Un fichier de configuration s'articule autour de deux éléments
  - ❶ La suite qui représente l'ensemble des tests à lancer,
  - ❷ Les tests devant être lancés. Un test peut contenir plusieurs classes ou groupes de tests.
- 📖 Chacun de ces éléments peuvent avoir un nom. La suite comporte un attribut permettant de spécifier le niveau de log à afficher.

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="Suite" verbose="5" >
  <test name="TestNum1" >
    ...
  </test>
  <test name="TestNum2" >
    ...
  </test>
  .
  .
  <test name="TestNumN" >
    ...
  </test>
</suite>
```

## Le fichier de configuration testng.xml

### ➤ Spécifier les packages de tests

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="SuitePackage" verbose="5" >
  <test name="FirstTest" >
    <packages>
      <package name="test.testPkg" />
    </packages>
  </test>
</suite>
```

### ➤ Spécifier les classes de tests

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="SuiteClass" verbose="5" >
  <test name="FirstTest" >
    <classes >
      <class name="test.testClass.TestSample1" />
      <class name="test.testClass.TestSample2" />
    </classes >
  </test>
</suite>
```

### ➤ Sélectionner les méthodes de tests

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="SuiteClass" verbose="5" >
  <test name="FirstTest">
    <classes>
      <class name="test.testClass.TestSample1">
        <methods>
          <include name=".*enabledTestMethod.*"/>
          <exclude name=".*brokenTestMethod.*"/>
        </methods>
      </class>
    </classes>
  </test>
</suite>
```

# Le framework TestNG

## Le fichier de configuration testng.xml

### ➡ Spécifier les groupes de tests

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="SuiteClass" verbose="5" >
  <test name="FirstTest">
    <groups>
      <run>
        <include name="groupOk.*"/>
        <exclude name="groupNonOk.*"/>
      </run>
    </groups>

    <classes>
      <class name="test.testClass.TestSample1"/>
    </classes>
  </test>
</suite>
```

- 📖 L'appartenance d'une méthode à un groupe s'effectue à travers l'attribut groups de l'annotation @Test

```
@Test(groups="groupOk")
void someTestingMethod() {
    [...]
}
```



# Installation de TestNG

## Installation de TestNG avec Eclipse

Help

- Welcome
- Help Contents
- Search
- Show Contextual Help
- Show Active Keybindings... Ctrl+Shift+L
- Tip of the Day
- Tips and Tricks...
- Report Bug or Enhancement...
- Cheat Sheets...
- Eclipse User Storage >
- Perform Setup Tasks...
- Check for Updates
- Install New Software...**
- Eclipse Marketplace...
- About Eclipse IDE
- Contribute

➡ Ajouter le dépôt  
`http://dl.bintray.com/testng-team/testng-eclipse-release/`

Install

Available Software

Check the items that you wish to install.

Work with: `http://dl.bintray.com/testng-team/testng-eclipse-release/` Add... Manage...

Type filter text

Add Repository

Name: TestNG Local...

Location: `http://dl.bintray.com/testng-team/testng-eclipse-release/` Archive...

?

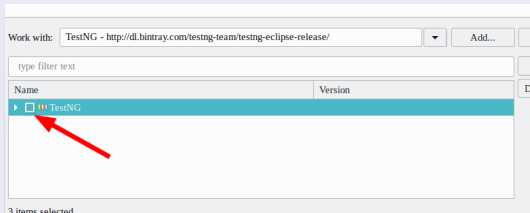
Add Cancel

Select All Deselect All

# Installation de TestNG

## Installation de TestNG avec Eclipse

- ➡ Sélectionner TestNG. Puis, suivre les instructions indiquées par Eclipse.



- ➡ Après l'installation de TestNG, il est nécessaire de redémarrer Eclipse

## Avertissement

- ➡ Pour chaque nouveau projet utilisant TestNG
  - 🔧 Intégrer la bibliothèque TestNG dans le Classpath (Project->Properties->Java Build Path)
  - 🔧 Télécharger le fichier guice-4.2.2.jar à partir du site <https://github.com/google/guice/wiki/Guice422> et l'intégrer dans le Classpath

# Tests automatisés avec le framework TestNG

## Exemple du TDD avec TestNG

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

# Tests automatisés avec le framework TestNG

## Exemple du TDD avec TestNG

- ➡ On commence par écrire la classe de test TestNG (Sélectionner File->New->Other->TestNG->TestNG class)

```
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 TestNG

## Exemple du TDD avec TestNG

- 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 TestNG

## Exemple du TDD avec TestNG

- 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 TestNG

## Exemple du TDD avec TestNG

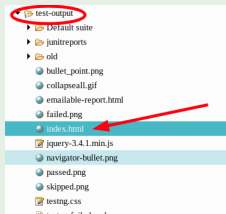
- Le test de la 6ème condition provoque une erreur!!!
- Modifions le code applicatif pour corriger l'erreur

```
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;  
}
```

# Tests automatisés avec le framework TestNG

## Exemple du TDD avec TestNG

- ➡ À tout moment, il est possible de consulter le rapport des tests réalisés dans le format HTML



- ➡ La modification du niveau du log (de 0 à 10) est effectuée dans le fichier `testng.xml`  
`<suite verbose="3"...`



# Plan

- 1 Définition et principes du TDD
- 2 Tests automatisés avec le framework JUnit
- 3 Tests automatisés avec le framework TestNG
- 4 Tests paramétrisés**
- 5 Bonnes pratiques

# 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 effectuée 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

- 1 Définition et principes du TDD
- 2 Tests automatisés avec le framework JUnit
- 3 Tests automatisés avec le framework TestNG
- 4 Tests paramétrisés
- 5 Bonnes pratiques**

# 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 jouer 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 chaîne 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 :
    - ❶ Tous les pions sont placés sur la même ligne horizontale. Après avoir proposé une solution simple, ne pas oublier de réusiner le code
    - ❷ Tous les pions sont placés sur la même ligne verticale
    - ❸ Tous les pions sont placés sur la première diagonale
    - ❹ 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 ?



**ORSYS**  
formation



## Test Driven Development en Java

**Dr. Ing. Chiheb Ameur ABID**

*Contact: [chiheb.abid@gmail.com](mailto:chiheb.abid@gmail.com)*

Janvier 2021

Version 1.1

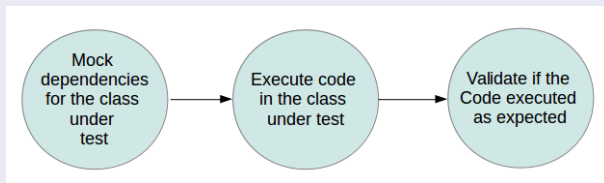
# Plan

- 1 Les objets Mock et Stub
- 2 Les techniques d'écriture de tests

# Présentation de Mockito

## Pourquoi le mocking ?

- Certains objets "réels" dans les tests sont difficiles à instancier et à les configurer
- Parfois, seulement les interfaces existent 🚫 Pas d'implémentation prête



# Présentation de Mockito

## Exemples d'utilisation

- Comportement non déterministe (heure courante, nombre généré aléatoirement, température ambiante, etc.)
- Initialisation longue (BD)
- Classe pas encore implémentée ou implémentation non stable
- États complexes difficiles à reproduire dans les tests (erreur réseau, exception sur fichiers)
- Pour tester, il faudrait parfois ajouter des attributs ou des méthodes aux classes applicatives (parasiter une classe applicative)



# Présentation de Mockito

## Définition

- Mock = Objet factice, doublure, bouchon
  - ☞ Les mocks (ou Mock object) sont des objets simulés qui reproduisent le comportement d'objets réels de manière contrôlée
- Un mock a la même interface que l'objet qu'il simule
- L'objet client ignore s'il interagit avec un objet réel ou un objet simulé

## Principe

- Avec les mocks, on teste le comportement d'autres objets, réels, mais liés à un objet inaccessible ou non implémenté
- L'utilisation des Mocks dans les tests consiste à spécifier
  - ➊ Quelles méthodes vont être appelées, avec quels paramètres et dans quel ordre
  - ➋ Les valeurs retournées par le mock

# Présentation de Mockito

## Concepts

- **Dummy** (panin, factice) : objets vides qui n'ont pas de fonctionnalités implémentées. Ils sont transmis mais ne sont jamais réellement utilisés. Habituellement, ils sont juste utilisés pour remplir des listes de paramètres selon les paramètres
- **Stub** (bouchon) : classes qui renvoient en dur une valeur pour une méthode invoquée. Ils ne répondent généralement pas du tout à ce qui est en dehors de ce qui est programmé pour le test.
- **Mock** (factice) : des objets préprogrammés avec des attentes qui forment une spécification des appels qu'ils sont censés recevoir.
- **Fake** (subsButut, simulateur) : implémentation partielle qui renvoie toujours les mêmes réponses
- **Spy** (espion) : les objets sont des répliques partielles d'objets réels : certaines méthodes sont moquées

# Présentation de Mockito

## Présentation de Mockito

- Mockito est un framework de simulation basé sur java, utilisé en conjonction avec d'autres frameworks de test tels que JUnit et TestNG, etc.
- Mockito est un générateur automatique de doublures
- Un Mock renvoie des données factices et évite les dépendances externes



# Génération des doublures avec Mockito

## Cycle de vie d'un mock dans un cas de test

- ➡ La procédure d'utilisation de Mockito est très simple
- ❶ Création d'un objet mock pour une classe ou une interface ;
  - ❷ Description du comportement qu'il est censé imiter ;
  - ❸ Utilisation du mock dans le code de test ;
  - ❹ Si nécessaire, interrogation du mock pour savoir comment il a été utilisé durant le test.

# Intégration de Mockito

## Intégration de Mockito avec Maven

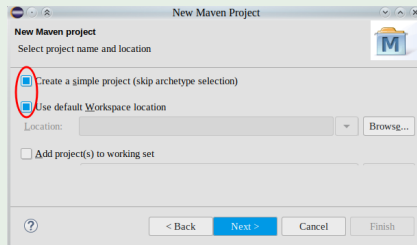
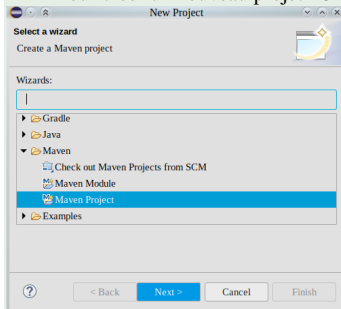
- Il est possible d'ajouter les fichiers `.jar` (Mockito et Junit) dans le projet
- Une manière plus simple consiste à utiliser Maven
- Maven est un outil créé par Apache, il permet de faciliter la gestion d'un projet Java
  - 📖 C'est un Outil de Gestion de Dépendance (Dependency Management Tool)
  - 📖 Il permet de mieux structurer un projet : séparer la partie liée au code du projet, le code des tests unitaires et autres fichiers statiques (images, PDF etc.)
  - 📖 L'ensemble du projet est géré à partir d'un seul fichier : `pom.xml`



# Utilisation de Mockito

## Exemple illustratif : Intégration de Mockito dans un projet Java

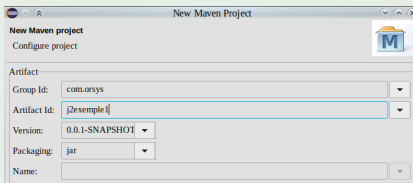
➡ Pour créer un nouveau projet Maven, aller au menu `File->New->Project`



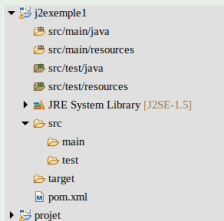
# Utilisation de Mockito

## Exemple illustratif : Intégration de Mockito dans un projet Java

- En général, le `Group Id` correspond au nom de l'organisation, et l'`Artifact Id` correspond au nom du projet



- Maven prend en charge la structuration du projet



# Utilisation de Mockito

## Exemple 1 : Intégration de Mockito dans un projet Java

- ➡ Aller au site [www.mvnrepository.com](http://www.mvnrepository.com) et récupérer les méta-données relatives au framework Mockito

```
Maven  Gradle  SBT  Ivy  Grape  Leinington

<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-all</artifactId>
  <version>1.9.5</version>
  <scope>test</scope>
</dependency>
```

- ➡ On peut ajouter les dépendances Mockito de deux manières
  - ① En Insérant directement les méta-données dans le fichier pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.9.5</version>
  </dependency>
</dependencies>
```



# Utilisation de Mockito

## Exemple illustratif : Intégration de Mockito dans un projet Java

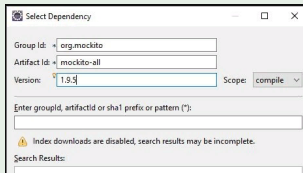
➡ On peut ajouter les dépendances Mockito de deux manières

- 1 En Insérant directement les méta-données dans le fichier `pom.xml`

```
<dependencies>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.9.5</version>
  </dependency>
</dependencies>
```



- 2 Après avoir ouvert le fichier `pom.xml`, choisir l'onglet Dependencies. Puis, cliquer sur le bouton Add : une boîte de dialogue s'affiche pour saisir les informations de dépendances



# Génération des doublures avec Mockito

## Mocking : création de doublure

- ➡ L'utilisation de Mockito nécessite au préalable l'importation statique du paquet Mockito  

```
import static org.mockito.Mockito.*;
```
- ➡ Deux manières de créer des doublures, soit avec la méthode `mock()` ou bien avec l'annotation `@Mock`.

## Avertissement

Mockito ne peut pas mocker ou espionner : les classes déclarées `final`, les méthodes déclarées `final`, les énumérations `enums`, les méthodes statiques, les méthodes privées (déclarées `private`), les méthode `hashCode()` et `equals()`, les classes anonymes, et les types primitifs.

# Génération des doublures avec Mockito

## Création d'une doublure avec la méthode `mock()`

### ➤ Création d'un Mock sans nom

```
MaClasse monMock = mock(MaClasse.class);
```

### ➤ Création d'un Mock avec attribution de nom

```
MaClasse mockAvecNom = mock(MaClasse.class, "Mon mock");
```

## Exemple

```
public interface Calcul {  
    int Somme(int x,int y);  
}  
/*****/  
import org.junit.Test;  
import static org.mockito.Mockito.*;  
  
public class ExempleTest {  
    @Test  
    public void testMock() {  
        Calcul monMock=mock(Calcul.class);  
        assertTrue(true);  
    }  
}
```

# Génération des doublures avec Mockito

## Création d'une doublure avec l'annotation `@Mock`

- ➡ L'annotation se met au-dessus de la déclaration de l'attribut du type du mock. Le nom du mock est automatiquement celui de l'attribut.

```
@Mock
```

```
private MaClasse monMock;
```

- ➡ Il ne faut pas oublier d'initialiser l'interpréteur des annotations de Mockito

- ➊ Initialisation par l'appel de méthode `initMocks()` annotés par `@Mock`, qui se trouvent dans la classe passée en paramètre.
- ➋ Initialisation par le moteur d'exécution `MockitoJUnitRunner`
- ➌ Initialisation par la règle `MockitoRule`

# Création d'une doublure avec l'annotation @Mock

## Initialisation par l'appel de méthode `initMocks()`

- ➡ C'est la méthode la plus classique
- ➡ La méthode `MockitoAnnotations.initMocks(this)` initialise tous les attributs annotés par `@Mocks`, qui se trouvent dans la classe passée en paramètre.

```
public void FooTest {  
    private Foo foo;  
    @Mock  
    private MaClasse monMock;  
    @Before  
    public void setUp() {  
        MockitoAnnotations.initMocks(this);  
        foo = new Foo(monMock);  
        ... // Testing  
    }  
}
```

# Création d'une doublure avec l'annotation @Mock

## Initialisation par le moteur d'exécution MockitoJUnitRunner

- ➡ Cette option n'est utilisable que si Mockito est le seul moteur utilisé. Ceci exclut par exemple l'utilisation des paramètres de test JUnit

```
@RunWith(MockitoJUnitRunner.class)
public void FooTest {
    private Foo _foo;
    @Mock
    private MaClasse _monMock;
    @Before
    public void setUp() {
        _foo = new Foo(_monMock);
        ... // Testing
    }
    ...
}
```

# Création d'une doublure avec l'annotation @Mock

## Initialisation par la règle MockitoRule

➡ La règle JUnit invoque implicitement la méthode `initMocks(this)`

```
public void FooTest {  
    private Foo _foo;  
    @Mock  
    private MaClasse _monMock;  
    @Rule  
    public MockitoRule mockitoRule = MockitoJUnit.rule();  
    @Before  
    public void setUp() {  
        _foo = new Foo(_monMock);  
    }  
    ...  
}
```

# Génération des doublures avec Mockito

## Stubbing

- Le stubbing consiste à définir le comportement des méthodes d'un mock.
- Appel d'une méthode avec une valeur de retour unique  
`when(monMock.retourneUnEntier()).thenReturn(3);`
- Appel d'une méthode avec des valeurs de retour consécutives  
`when(monMock.retourneUnEntier()).thenReturn(3, 4);`  
`when(monMock.retourneUnEntier()).thenReturn(3).thenReturn(4);`

## Exemple

```
public class ExempleTest {  
    @Test  
    public void testMock() {  
        Calcul monMock=mock(Calcul.class);  
        when(monMock.Somme(4,3)).thenReturn(7);  
        when(monMock.Incrementer()).thenReturn(0).thenReturn(1);  
  
        System.out.println("Somme(4,3) retourne: "+monMock.Somme(4,3));  
        System.out.println("Incrementer: "+monMock.Incrementer());  
        System.out.println("Incrementer: "+monMock.Incrementer());  
        assertTrue(true);  
    }  
}
```



# Génération des doublures avec Mockito

## Stubbing : Appel d'une méthode avec n'importe quel paramètre

- ➡ Il est possible de spécifier un appel sans que les valeurs des paramètres aient vraiment d'importance. On utilise pour cela des **Matchers**

- ✎ `any()`, `anyObject()`
- ✎ `anyBoolean()`, `anyDouble()`, `anyFloat()`, `anyInt()`, `anyString()`, etc.
- ✎ `anyList()`, `anyMap()`, `anyCollection()`, `anyCollectionOf()`, `anySet()`, `anySetOf()`

## Avertissement

Si on utilise des Matchers, tous les arguments doivent être des Matchers

# Génération des doublures avec Mockito

## Stubbing : Levée d'exception

- Il est possible de lever une exception lorsqu'une méthode est appelée

```
when(monMock.methode()).thenThrow(new BidonException);
```

```
doThrow(new BidonException()).when(monMock).methode();
```
- Il est possible de cumuler le retour d'une valeur donnée puis la levée d'une exception

```
when(monMock.methode()).thenReturn(3).thenThrow(new BidonException());
```
- Un test JUnit permettant de vérifier si l'appelle d'une méthode a levé une exception

```
@Test(expected = BidonException.class)
public void should_throw_exception() {
    monMock.methode();
}
```

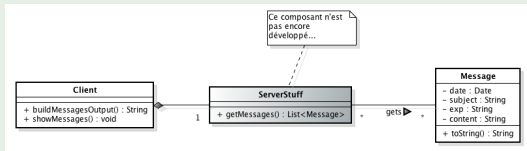
## Exemple

```
@Test(expected=IllegalArgumentException.class)
public void testMockException() {
    Calcul monMock=mock(Calcul.class);
    when(monMock.Diviser(5,0)).thenThrow(new IllegalArgumentException("Argument_nul"));
    when(monMock.Diviser(5,2)).thenReturn(2);
    System.out.println("Diviser_ (5,2) _="+monMock.Diviser(5,2));
    System.out.println("Diviser_ (5,0) _="+monMock.Diviser(5,0));
}
```

## Exemple 1 : utilisation de Mockito

### Exemple 1 : utilisation de Mockito

On considère la structure de classes ci-après décrivant un système de messagerie simplifié



- La classe `Client` décrit un utilisateur de la messagerie
- La classe `ServerStuff` représente le serveur de messagerie associé à un client. Cependant, cette classe n'est pas encore implémentée. On sait seulement qu'elle dispose d'une méthode qui permet de récupérer les nouveaux messages du client associé.
- Un message est décrit par une instance de la classe `Message`
- 🔴 Notre objectif est de tester la classe `Client`

## Exemple 1 : utilisation de Mockito

### Code de la classe Message

```
public class Message {  
    Date date;  
    String subject;  
    String exp;  
    String content;  
  
    public Message(Date date, String subject, String exp, String content) {  
        this.date = date;  
        this.subject = subject;  
        this.exp = exp;  
        this.content = content;  
    }  
  
    public String toString() {  
        return subject + ",_recu_le_" + date + ",_de_" + exp + "\n" + content;  
    }  
}
```

### Code de l'interface ServerStuff!!

```
import java.util.List;  
  
public interface ServerStuff {  
    List<Message> getMessages();  
}
```

## Exemple 1 : utilisation de Mockito

### Code de la classe Client

```
public class Client {  
  
    ServerStuff serverStuff;  
  
    public Client(ServerStuff serverStuff) {  
        this.serverStuff = serverStuff;  
    }  
  
    public String buildMessagesOutput() {  
        List messages = this.serverStuff.getMessages();  
  
        String output = "";  
  
        output += "=====\n";  
        output += "Vos_nouveaux_messages\n";  
        output += "=====\n";  
        for (Message message : messages) {  
            output += message + "\n";  
        }  
        output += "=====";  
  
        return output;  
    }  
}
```

# Exemple 1 : utilisation de Mockito

## Code de la classe de test

```
package j2exemple1;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
import java.util.*;
class ClientTest {
    private Client client;
    @BeforeEach
    void setUp() throws Exception {
        // Mocking
        ServerStuff serverStuff = mock(ServerStuff.class);
        // Stubbing
        when(serverStuff.getMessage()).thenReturn((List<Message>) Arrays.asList(
            new Message(new Date(2010, 11, 7), "Bonjour!", "bob@u-picardie.fr", "Comment_
                ca_va?"),
            new Message(new Date(2010, 11, 8), "Un_test", "testeur@u-picardie.fr", "
                Bonjour_le_test!")
        ));
        // Instancier le client
        this.client = new Client(serverStuff);
    }

    @Test
    void test() {
        // Checking
        String output = this.client.buildMessagesOutput();
        assertTrue(output.equals("=====\nVos_nouveaux_messages\n=====\n
            nBonjour!,_recu_le_Wed_Dec_07_00:00:00_CET_3910,_de:_bob@u-picardie.fr\nComment_ca_va
            _\nUnUn_test,_recu_le_Thu_Dec_08_00:00:00_CET_3910,_de:_testeur@u-picardie.fr\nBonjour
            le_test!\n====="));
    }
}
```

# Génération des doublures avec Mockito

## Exercice

- On veut modéliser un jeu de casino : le jeu de la boule
- Le jeu de la boule est un jeu de casino simplifié du jeu de la roulette. Il utilise les chiffres de 1 à 9. Le joueur qui joue fait un pari en misant une somme.
- Il est possible de miser sur rouge, noir, manque, passe, pair ou impair.
- Les chiffres 1, 3, 7, 9 sont impairs. Les chiffres 2, 4, 6, 8 sont pairs.
- Les chiffres 1, 3, 6, 8 sont noirs. Les chiffres 2, 4, 7, 9 sont rouges.
- Les chiffres 1, 2, 3, 4 sont "manque" ("on a manqué de dépasser 5"). Les chiffres 6, 7, 8, 9 sont "passe" ("on a dépassé 5").
- Ces chances sont des chances simples : si la chance simple mise sort, le joueur gagne une fois la mise (qui lui est restituée), sinon la mise est perdue.
- Le chiffre 5 n'est ni pair, ni impair, ni manque, ni passe, ni rouge, ni noir. Si le 5 sort, la mise jouée sur une chance simple est perdue.
- Le joueur peut miser sur un numéro. Si celui-ci sort, le joueur gagne 7 fois la mise qui lui est restituée sinon la mise est perdue.
- Un joueur peut évidemment miser sur plusieurs cases (et même sur rouge et noir !). Il ne peut miser que des quantités entières (des jetons de valeur entière en euro).

# Génération des doublures avec Mockito

## Exercice

- ➔ Pour modéliser ce jeu on utilise aux moins deux classes : la classe `Joueur` qui modélise un joueur et `CroupierBoule` qui modélise le gestionnaire du jeu de la boule : le croupier.
- ➔ La classe `CroupierBoule` possède la méthode `public int getNumSorti()` qui retourne le numéro sorti
- ❶ Un joueur est lié au casino et c'est le casino qui lui indique combien il a gagné ou perdu. Il peut savoir quel numéro est sorti en le demandant au casino (ou au représentant du casino). Donner le code de la classe `Joueur`. C'est la classe à tester et on veut l'isoler de la classe qui modélise le casino (son mock).
- ❷ Écrire les tests pour les cas suivants :
  - 🔴 Le joueur gagne : Le joueur n'a joué que sur le 8 avec 3 jetons et le 8 est sorti
  - 🔴 Le joueur perd : Le joueur n'a joué que sur le 8 avec 3 jetons et le 9 est sorti



# Génération des doublures avec Mockito

## Espionnage

- Mockito garde trace de tous les appels de méthode. Vous pouvez utiliser la méthode `verify()` sur un objet mock pour vérifier qu'une méthode a été appelée et avec certaines valeurs de paramètre.
- Ce type de test correspond à un test de comportement
- Il est possible d'espionner un objet classique à l'aide de la méthode `spy()` ou l'annotation `@Spy`

# Fonctionnement de la méthode `verify()`

## Fonctionnement de la méthode `verify()`

- ➡ Elle permet de vérifier :
  - 📖 Quelles méthodes ont été appelées sur un mock,
  - 📖 Combien de fois,
  - 📖 Avec quels paramètres,
  - 📖 Dans quel ordre.
- ➡ Si la vérification échoue, il y a une levée d'exception et le test unitaire échoue

# Fonctionnement de la méthode `verify()`

## Vérification du nombre d'appels

- Vérifier qu'une méthode est appelée exactement une fois  
`verify(monMock).retourneUnBooleen();`
- Vérifier qu'une méthode est appelée exactement `n` fois  
`verify(monMock, times(n)).retourneUnBooleen();`
- Vérifier qu'une méthode est appelée au moins une fois  
`verify(monMock, atLeastOnce()).retourneUnBooleen();`
- Vérifier qu'une méthode est appelée au plus `n` fois  
`verify(monMock, atMost(n)).retourneUnBooleen();`
- Vérifier qu'une méthode n'est jamais appelée  
`verify(monMock, never()).retourneUnBooleen();`

## Fonctionnement de la méthode `verify()`

### Vérification de l'ordre des appels

- Cette vérification nécessite d'importer la classe `InOrder`  
`import org.mockito.InOrder;`
- Vérifier qu'un appel est effectué avant un autre  
`InOrder ordre = inOrder(monMock);`  
`ordre.verify(monMock).retourneUnEntierBis(4, 2);`  
`ordre.verify(monMock).retourneUnEntierBis(5, 3);`
- On peut utiliser Vérifier qu'une méthode est appelée au moins une fois  
`verify(monMock, atLeastOnce()).retourneUnBooleen();`

### Vérification des appels avec n'importe quel paramètre

- On utilise pour cela des `Matchers`  
`verify(mockedList).someMethod(anyInt());`
- Si on utilise des `Matchers`, tous les arguments doivent être des `Matchers`  
✗ `verify(mock).someMethod(anyInt(), anyString(), "un argument effectif");`

# Injection des Mocks et des Spy

## Injection des Mocks et des Spy

- L'annotation `InjectMock` permet l'injection des Mocks et des Spy
- L'injection des Mocks est utile lorsqu'on souhaite tester une classe qui dépend d'une autre où la dernière doit être mockée
- Mockito tentera d'injecter des Mocks uniquement par injection de constructeur, injection de setter ou injection de propriété
  - ❗ Si l'une des stratégies suivantes échoue, Mockito ne signalera pas d'échec ; c'est-à-dire que vous devrez fournir les dépendances vous-même

# Injection des Mocks et des Spy

## Exemple : Injection des Mocks et des Spy

```
public class Two {
    public void doSomething() {
        System.out.println("Un_autre_service_fonctionne...");
    }
}

public class One {
    private Two _two;
    public One( Two two ) {
        _two = two;
    }
    public void work() {
        System.out.println("Un_premier_service_fonctionne");
        _two.doSomething();
    }
}

////////////////////////////////
public class OneTest {
    @Mock
    private Two _two;
    @InjectMocks
    private One _one;
    @Before
    public void setup() {
        MockitoAnnotations.initMocks(this);
    }
    @Test
    public void oneCanWork() throws Exception {
        one.work();
        Mockito.verify(_another).doSomething();
    }
}
```

# Injection des Mocks et des Spy

## Exemple : Injection des Mocks et des Spy

➡ On crée une doublure pour la classe `Two`, puis on l'injecte dans `One`

```
public final class OneTest {  
    @Mock  
    private Two _two;  
    @InjectMocks  
    private One _one;  
    @Before  
    public void setup() {  
        MockitoAnnotations.initMocks(this);  
    }  
    @Test  
    public void oneCanWork() throws Exception {  
        _one.work();  
        Mockito.verify(_one).doSomething();  
    }  
}
```

## Utilisation de Mockito : Connexion à une BD avec Mockito

### Exemple 2 : Objectif

- ➡ On se propose de simuler la connexion à une BD avec Mockito
  - 🔴 On simule la création d'une connexion à une BD et l'exécution d'une requête

### Exemple 2 : Création du projet

- ➡ On crée un projet Maven en incluant les dépendances relatives à Mockito et les pilotes pour la connexion à une BD Mysql
  - 🔴 Récupérer les méta-données relatives aux dépendances de Mysql Connector

```
Maven  Gradle  SBT  Ivy  Grape  Leiningen  Bu
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.21</version>
</dependency>
```



# Utilisation de Mockito :Connexion à une BD avec Mockito

## Exemple 2 : Code applicatif

- On crée la classe à tester DatabaseService
- Cette classe contient deux méthodes
  - 🔴 La première méthode sera chargée de créer la session de base de données.
  - 🔴 La deuxième méthode sera responsable de l'exécution de la requête.

```
package j2exemple2;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseService {
    private Connection dbConnection;
    public void getDbConnection() throws ClassNotFoundException, SQLException {
        dbConnection = DriverManager.getConnection("http://127.0.0.1:3306/mockito_db", "
            root", "passwd");
    }
    public int executeQuery(String query) throws SQLException {
        return dbConnection.createStatement().executeUpdate(query);
    }
}
```

# Utilisation de Mockito :Connexion à une BD avec Mockito

## Exemple 2 : Classe de test

➡ On crée la classe de test DatabaseServiceTest

🔗 Aller au menu File -> New -> JUnit Test Case

```
package j2exemple2;
import static org.junit.jupiter.api.Assertions.*;
import java.sql.*;
import org.junit.jupiter.api.*;
import org.mockito.*;
class DatabaseServiceTest {
    @InjectMocks
    private DatabaseService dbConnection;
    @Mock
    private Connection mockConnection;
    @Mock
    private Statement mockStatement;
    @BeforeEach
    public void setUp() {
        MockitoAnnotations.initMocks(this);
    }
    @Test
    void test() throws Exception {
        Mockito.when(mockConnection.createStatement()).thenReturn(mockStatement);
        Mockito.when(mockConnection.createStatement().executeUpdate(Mockito.anyString()))
            .thenReturn(1);
        int value=dbConnection.executeQuery("");
        assertEquals(1, value);
    }
}
```

# Injection des mocks

## Exercice

- ➡ On s'intéresse à un jeu de hasard et d'argent
- ➡ Le joueur possède une somme d'argent qui lui permet de jouer à un jeu (la somme étant physiquement étalée devant lui, on suppose que le jeu n'a aucun contrôle à effectuer sur le montant de la mise).
- ➡ Le jeu qui nous intéresse se joue avec 2 dés et une banque qui gère les pertes et les gains. Les dés sont du modèle classique à tirage aléatoire entre 1 et 6.
- ➡ La banque est censée être toujours solvable. Néanmoins, il arrive que le casino soit débordé par un joueur chanceux et n'arrive plus à suivre : la banque "saute". Le gain est quand même donné au joueur, mais le jeu ferme immédiatement.
- ➡ La règle du jeu est la suivante. On ne peut jouer qu'à un jeu ouvert. Le joueur qui joue fait un pari en misant une somme. Il est débité du montant de son pari qui est encaissé par la banque. Ensuite les 2 dés sont lancés. Si la somme des lancers vaut 7, alors le joueur gagne : la banque paye deux fois la mise, somme créditée au joueur. Si le pari a fait sauter la banque, le jeu ferme immédiatement. Si la somme des lancers ne vaut pas 7, le joueur a perdu sa mise

# Injection des mocks

## Exercice

- ➡ Le sujet consiste à tester en isolation la méthode `public void jouer() throws . . .` de la classe qui représente le jeu. On ne demande pas d'écrire ni de tester les autres classes : limitez-vous à des interfaces et utilisez des doublures
- ➡ Exemples de scénarios à tester
  - ❏ Le plus simple : la banque est fermée
  - ❏ Le joueur perd : vérifier notamment que le joueur a été débité de sa mise, laquelle a été créditée à la banque, et que le jeu reste ouvert.

# Plan

- 1 Les objets Mock et Stub
- 2 Les techniques d'écriture de tests

# Organisation des tests

## Organisation des tests

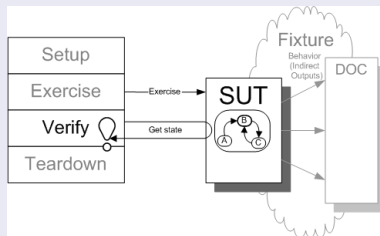
➡ Les tests peuvent être organisés de différentes façons

- ❶ Classe de test par classe : on met toutes les méthodes de test pour une classe de système sous test (SUT) sur une seule Classe de test
- ❷ Classe de test par fonctionnalité : on regroupe les méthodes de test sur les classes de test en fonction de la fonction testable de le SUT qu'ils exercent.
- ❸ Classe de test par fixature : on organise les méthodes de test dans des classes de test basées sur la similitude du test fixation.

# Principales stratégies pour vérifier les résultats d'un test

## Vérification de l'état

- ➡ On détermine si la méthode exercée a fonctionné correctement en examinant l'état du SUT et de ses collaborateurs après la a été exercée
  - 📖 L'état d'un objet est défini par les valeurs de ses attributs
- ➡ Approche de test classique



# Principales stratégies pour vérifier les résultats d'un test

## Vérification de l'état : principe de fonctionnement

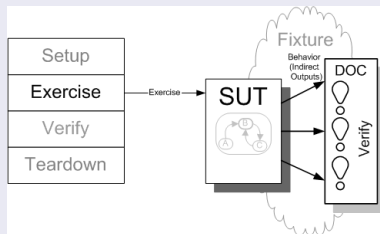
- ➊ Initialisation : Instancier le SUT et ses collaborateurs
- ➋ L'exécution : l'étape où la méthode à vérifier est exécutée
- ➌ La vérification : l'exécution **des assertions** qui permettent de vérifier l'**état** du SUT et de ses collaborateurs.
- ➍ Le nettoyage (teardown en anglais) : l'étape où les données de test sont réinitialisées, supprimées ou autre.



# Principales stratégies pour vérifier les résultats d'un test

## Vérification comportementale

- Elle consiste à vérifier le comportement du SUT et de sa collaboration avec les autres objets.
  - 📖 Elle garantit que le SUT se comporte vraiment comme spécifié plutôt que juste se retrouver dans le bon état post-exercice
- Elle est basée sur l'utilisation des Mocks



# Principales stratégies pour vérifier les résultats d'un test

## Vérification comportementale : principe de fonctionnement

### ❶ Initialisation

- Instancier le SUT et les **mocks** à partir des classes interfaces qui représentent les collaborateurs
- Initialiser les attentes (les "Expectations") : on écrit le comportement attendu par l'objet mocké

### ❷ L'exécution et vérification : la méthode à vérifier est exécutée ce qui induit que les appels des méthodes des Mocks sont surveillés par Mockito

### ❸ Le nettoyage (teardown en anglais) : l'étape où les données de test sont réinitialisées, supprimées ou autre.

## Exemple : vérification de l'état vs comportementale

### Exemple : énoncé

Il y a deux objets : un entrepôt (warehouse) et une commande (order). L'entrepôt contient différents types de produits dans des quantités limitées. Une commande concerne un produit pour une quantité donnée. S'il y'a suffisamment de stock dans l'entrepôt, la commande est **passée**; sinon, **rien ne se passe**.

### Principe

- ➡ L'objet à tester (SUT) : une commande
  - 📌 Vérifier que la commande change d'état lorsqu'il y a suffisamment de stock dans l'entrepôt
- ➡ L'objet collaborateur : un entrepôt

# Principales stratégies pour vérifier les résultats d'un test

## Exemple : Méthodes de la classe de tests

```
private static String GUINNESS = "Guinness";
private static String CHIMAY = "Chimay";
private Warehouse warehouse;

// Initialization, processed before each test thanks to the Before annotation
@BeforeEach
protected void setUp() throws Exception {
    // The fixtures or, collaborators
    warehouse = new Warehouse();
    warehouse.add(GUINNESS, 50);
    warehouse.add(CHIMAY, 25);
}

// Teardown, processed after each test thanks to the After annotation
@AfterEach
protected void tearDown() throws Exception {
    warehouse = null;
}
```

# Vérification de l'état

## Vérifier l'état d'une commande

- ➡ Vérifier qu'une commande a été effectuée lorsque le stock contient suffisamment de produits

```
public void testOrderIsFilledIfEnoughInWarehouse() {  
    // The System Under Testing  
    Order order = new Order(GUINNESS, 50);  
    // Exercise  
    order.fill(warehouse);  
  
    // Check  
    assertTrue(order.isFilled());  
    assertEquals(0, warehouse.getInventory(GUINNESS));  
}
```

## Vérifier l'état d'une commande

- ➡ Vérifier qu'une commande n'était pas effectuée lorsque le stock ne contient pas suffisamment de produits

```
public void testOrderDoesNotRemoveIfNotEnough() {  
    Order order = new Order(GUINNESS, 51);  
    order.fill(warehouse);  
    assertFalse(order.isFilled());  
    assertEquals(50, warehouse.getInventory(GUINNESS));  
}
```

# Vérification comportementale

## Principe

- L'objet à tester (SUT) : une commande
- L'objet collaborateur : un entrepôt
  - 🔴 Cet objet sera mocké et surveillé

# Vérification comportementale

## Vérifier une commande

- ➡ Vérifier qu'une commande a été effectuée lorsque le stock contient suffisamment de produits

```
public void testFillingRemovesInventoryIfInStock() {  
    // Setup  
    Order order = new Order(GUINNESS, 50);  
    Warehouse warehouseMock = mock(Warehouse.class);  
  
    // stubs  
    when(warehouseMock.hasInventory(GUINNESS, 50)).thenReturn(true);  
  
    // exercise  
    order.fill(warehouseMock);  
  
    // verify  
    InOrder inOrder = inOrder(warehouseMock);  
    inOrder.verify(warehouseMock).hasInventory(GUINNESS, 50);  
    inOrder.verify(warehouseMock).remove(GUINNESS, 50);  
    assertTrue(order.isFilled());  
}
```

# Vérification comportementale

## Vérifier une commande

- ➡ Vérifier qu'une commande n'était pas effectuée lorsque le stock ne contient pas suffisamment de produits

```
public void testFillingDoesNotRemoveIfNotEnoughInStock() {  
    Order order = new Order(GUINNESS, 51);  
    Warehouse warehouseMock = mock(Warehouse.class);  
  
    // stubs  
    when(warehouseMock.hasInventory(GUINNESS, 51)).thenReturn(false);  
  
    // exercise  
    order.fill(warehouseMock);  
  
    // verify  
    verify(warehouseMock).hasInventory(GUINNESS, 51);  
    verify(warehouseMock, never()).remove(anyString(), anyInt());  
}
```



**MERCI POUR VOTRE ATTENTION**



Questions ?



**ORSYS**  
formation



## Test Driven Development en Java

**Dr. Ing. Chiheb Ameur ABID**

*Contact: [chiheb.abid@gmail.com](mailto:chiheb.abid@gmail.com)*

Janvier 2021

Version 1.1

# Plan

- 1 Test du code hérité
- 2 Tests fonctionnels avec FitNesse
- 3 Tests unitaires et couverture du code
- 4 Selenium et JUnit

# Test du code hérité

## C'est quoi le code hérité (legacy code) ?

- ➡ Du code legacy, c'est du code :
  - 🚫 Sans tests unitaires (Déf. Michael Feathers)
  - 🚫 Plus vieux que ceux y travaillent
  - 🚫 Modification en mode patchage
  - 🚫 Nécessitant un guide de marais pour s'y retrouver

# Test du code hérité

## Problèmes avec le code hérité

- C'est un code qui ne possède pas un test suite alors il est difficile à maintenir.
- Il est parfois désordonné, plein d'erreurs
- Utilise des anciens API et Framework.
- Il utilise parfois des anciens systèmes qui ne sont plus utilisés

# Test du code hérité

## Exemple de test du code hérité

- Soit le code modélisant une machine à café qui fabrique différentes boissons à partir d'ingrédients définis.
- L'initialisation des recettes pour chaque boisson doit être codée, bien qu'il devrait être relativement facile d'ajouter de nouvelles boissons.
- La machine doit afficher le stock d'ingrédients (+ coût) et le menu au démarrage, et après chaque entrée utilisateur valide.
- Le coût de la boisson est déterminé par la combinaison des ingrédients. Par exemple, le café est 3 unités de café (75 cents par), 1 unité de sucre (25 cents par), 1 unité de crème (25 cents par).
- Les ingrédients et les éléments du menu doivent être imprimés par ordre alphabétique. Si la boisson est en rupture de stock, elle doit s'imprimer en conséquence.
- Si la boisson est en stock, elle doit afficher "Distribution :". Pour sélectionner une boisson, l'utilisateur doit saisir un numéro approprié. S'ils soumettent «r» ou «R», les ingrédients doivent se réapprovisionner, et «q» ou «Q» doivent cesser. Les lignes vides doivent être ignorées et une entrée non valide doit imprimer un message d'entrée non valide.

# Test du code hérité

## Code Java de la machine à café

- On a trois classes : Ingredient, Drink et DrinkMachine
- Extrait du code Java de la classe DrinkMachine

```
public class DriveMachine {  
    public static void startIO() {  
        BufferedReader reader=new BufferedReader(new InputStreamReader(System.in));  
        String input=""; int compteur=0;  
        while(true) {  
            try {  
                input=reader.readLine().toLowerCase();  
                if (input.equals(""))  
                    continue;  
                else if (input.equals("q"))  
                    System.exit(0);  
                else if (input.equals("r")) {  
                    compteur=0;  
                    System.out.println("Compteur_"+compteur);  
                }  
                else if (Integer.parseInt(input)>=1 && Integer.parseInt(input)<=10) {  
                    compteur++;  
                    System.out.println("Compteur_"+compteur);  
                }  
                else throw new IOException();  
            }  
            catch (Exception e) {  
                System.out.println("Invalid_selection_"+input+"\n");  
            }  
        }  
    }  
    public static void main(String[] args) {  
        System.out.println("Effectuer_un_choix:_q,r,1-10_");  
        startIO();  
    }  
}
```

# Test du code hérité

## Exemple de test du code hérité

- On doit d'abord examiner et observer le fonctionnement de programme en testant les différentes entrées
- Exécuter le programme et essayer différentes entrées

```
DrinkMachine [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (11 sept. 2020 à 00:30:09)
```

```
Inventory:
```

```
Cocoa,10
```

```
Coffee,10
```

```
Cream,10
```

```
Decaf Coffee,10
```

```
Espresso,10
```

```
Foamed Milk,10
```

```
Steamed Milk,10
```

```
Sugar,10
```

```
Whipped Cream,10
```

```
Menu:
```

```
1,Caffe Americano,€3,30,true
```

```
2,Caffe Latte,€2,55,true
```

```
3,Caffe Mocha,€3,35,true
```

```
4,Cappuccino,€2,90,true
```

```
5,Coffee,€2,75,true
```

```
6,Decaf Coffee,€2,75,true
```



# Test du code hérité

## Exemple de test du code hérité

- ➡ On doit ajouter un test case pour commencer les tests : File->New->JUnit Test Case

**JUnit Test Case**

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

☐ New JUnit 3 test ☐ New JUnit 4 test ☒ **New JUnit Jupiter test**

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()  
☐ setUp() ☐ tearDown()  
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Class under test:

⚠ JUnit 5 requires a Java 8 project. [Configure](#) project compliance and the project build path.

# Tester du code hérité

## Code Java de la machine à café

➡ On teste la méthode `main` de la classe `DrinkMachine`

```
package drinkmachine;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class DrinkMachinTest {
    @Test
    void testMain() {
        DrinkMachine.main(new String[] {});
    }
}
```

☹ Le teste ne cesse de tourner sans donner la main pour vérifier les résultats!!!

# Tester du code hérité

## Code Java de la machine à café

- ➡ L'envoi des données au programme s'effectue à travers `System.setIn()`, et la récupération des résultats (affichage) s'effectue à travers la fonction `System.setOut()`

```
void testMain() {  
    byte[] bytes="q\n".getBytes();  
    System.setIn(new ByteArrayInputStream(bytes));  
    ByteArrayOutputStream output=new ByteArrayOutputStream();  
    System.setOut(output);  
    DrinkMachine.main(new String[] {});  
    assertEquals("", output);  
}
```

- ➡ Afin d'éviter que le programme se termine sans pouvoir récupérer les résultat, il faut remplacer dans la classe `DrinkMachine` l'appel à `System.exit(0)` par `break`

# Tester du code hérité

## Code Java de la machine à café

➡ La comparaison des résultats d’affichage est effectuée en utilisant `assertThat`

```
assertThat(output.toString()).isEqualToNormalizingNewLines("Inventory:\r\n"+
"\r\n"+
"Cocoa,10\r\n"+
"\r\n"+
"Coffee,10\r\n"+
"\r\n"+
"Cream,10\r\n"+
"\r\n"+
"Decaf_Coffee,10\r\n"+
"\r\n"+
"Espresso,10\r\n"+
"\r\n"+
"Foamed_Milk,10\r\n"+
"\r\n"+
"Steamed_Milk,10\r\n"+
"\r\n"+
"Sugar,10\r\n"+
"\r\n"+
"Whipped_Cream,10\r\n"+
...)
```

# Plan

- 1 Test du code hérité
- 2 Tests fonctionnels avec FitNesse
- 3 Tests unitaires et couverture du code
- 4 Selenium et JUnit

# Tests d'acceptation / tests unitaires

## Tests d'acceptation / tests unitaires

### ➡ Tests unitaires

- 📖 Orientés « petite unité de code »
- 📖 Compréhension réservée aux développeurs

Tests unitaires = Est-ce que j'ai écrit le code correctement ?

### ➡ Tests d'acceptation (Tests de recette)

- 📖 Orientés « fonctionnalité »
- 📖 Compréhension accessible au client

Tests d'acceptation = Est-ce que j'ai écrit le bon code comme demandé par le client ?

# FitNesse en bref

## Présentation générale



- Un outil type wiki permettant de spécifier des tests d'acceptation au milieu de texte informel
- Serveur autonome, donc facile à installer
- Il est basé sur le Framework de test intégré de Ward Cunningham et est conçu pour prendre en charge les tests d'acceptation plutôt que les tests unitaires
  - 📖 Il facilite la création d'une description lisible et détaillée de la fonction du système
- Développé en langage Java
  - 📖 Fourni et expédié sous la forme d'un fichier jar exécutable unique.
  - 📖 À télécharger depuis <http://www.fitnesse.org/>

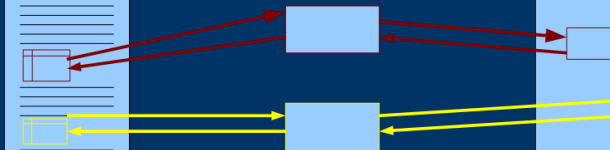
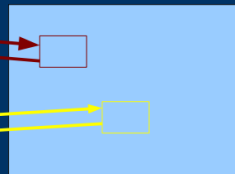
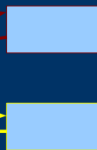
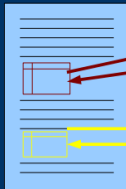
# FitNesse en bref

## Fonctionnement général

Document FitNesse

Fixtures (code java)

Application (java)





# FitNesse en bref

## Installation de FitNesse

- ❶ Récupérer l'archive (`fitnessse.jar`) et mettre le jar là où l'on veut installer le logiciel
  - 📄 `http://fitnessse.org/FitNesseDownload`
- ❷ Exécuter une première fois l'archive
  - 📄 `java -jar fitnessse.jar`
- ❸ Exécuter une deuxième fois l'archive
  - 📄 Soit : `java -jar fitnessse.jar` : dans ce cas le port standard 80 est choisi
  - 📄 Soit : `java -jar fitnessse.jar -pXXXX` : le port XXXX est choisi
- ❹ Lancer son navigateur et se connecter sur la bonne machine et le bon port
  - 📄 `http://localhost` ou `http://localhost:XXXX`

# Utilisation de FitNesse

## Exemple illustratif

On se propose de vérifier le bon fonctionnement d'une calculatrice permettant de réaliser les opérations de soustraction et d'addition.

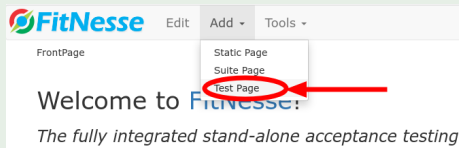
Nous supposons ainsi qu'on a défini la classe `Calculator` qui contient volontairement une erreur dans la méthode `minus`.

```
public class Calculator {  
    public int add(int a, int b) {  
        return a + b;  
    }  
    public int minus(int intToBeSubtracted, int minus) {  
        return intToBeSubtracted + minus;  
    }  
}
```

# Utilisation de FitNesse

## Exemple illustratif

- Accéder à FitNesse à travers le navigateur et créer une nouvelle page de test.



- Nommer cette page CalculatorTests et y ajouter le contenu suivant :  
!contents -R2 -g -p -f -h  
Définit que le test doit être exécuté avec SLIM  
!define TEST\_SYSTEM {slim}

# Utilisation de FitNesse

## Exemple illustratif

- ➡ Accéder à la page de test créée en spécifiant comme adresse `http://localhost/FrontPage.CalculatorTests`, et y ajouter une page fille nommée `CalculatorTest`. Cette dernière contient le code wiki pour les spécifications sous forme d'un tableau.

```
|Add two number |  
|first int|second int|result of addition?|  
|0 |1 |1 |  
|2 |0 |2 |  
|2 |3 |5 |  
|Subtract an integer to another |  
|int to be subtracted|minus|result of subtraction?|  
|0 |1 |-1 |  
|4 |2 |2 |  
|3 |1 |2 |
```

# Utilisation de FitNesse

## Exemple illustratif

- ➡ On crée les classes (custom fixtures) permettant d'effectuer la liaison entre le code wiki et la classe à tester
- ❗ Cette première classe est utilisée comme fixture pour la fonctionnalité d'addition

```
public class AddTwoNumber {  
    private int firstInt;  
    private int secondInt;  
    private Calculator calculator;  
    public AddTwoNumber() {  
        calculator = new Calculator();  
    }  
    public void setFirstInt(int firstInt) {  
        this.firstInt = firstInt;  
    }  
    public void setSecondInt(int secondInt) {  
        this.secondInt = secondInt;  
    }  
    public int resultOfAddition() {  
        return calculator.add(firstInt, secondInt);  
    }  
}
```

# Utilisation de FitNesse

## Exemple illustratif

- ② Cette deuxième classe est utilisée comme fixture pour la fonctionnalité de soustraction

```
public class SubtractAnIntegerToAnother {  
    private int intToBeSubtracted;  
    private int minus;  
    private Calculator calculator;  
    public SubtractAnIntegerToAnother() {  
        calculator = new Calculator();  
    }  
    public void setIntToBeSubtracted(int intToBeSubtracted) {  
        this.intToBeSubtracted = intToBeSubtracted;  
    }  
    public void setMinus(int minus) {  
        this.minus = minus;  
    }  
    public int resultOfSubtraction() {  
        return calculator.minus(intToBeSubtracted, minus);  
    }  
}
```

# Utilisation de FitNesse

## Exemple illustratif

- ➡ Avant d'effectuer les tests, il est nécessaire de spécifier dans la page de configuration `CalculatorTests` le chemin vers les fichiers binaires de notre système à tester en ajoutant le code wiki suivant :  
`!path <chemin des fichiers binaires>`
- ➡ Aller à la page `CalculatorTest` et exécuter les tests

# Utilisation de FitNesse

## Avantages

- ✓ Fitnessse est un outil collaboratif. D'un côté, les analystes et les clients peuvent écrire la spécification du logiciel dans Fitnessse à l'aide du code Wiki, de l'autre côté les développeurs et les testeurs peuvent écrire les Custom Fixtures pour faire la liaison entre les spécifications et le système à tester.
- ✓ Les tests sont lisibles par tous. Analystes, clients, développeurs, ...
- ✓ Il n'y a pas de redondance d'information, les spécifications sont les tests.
- ✓ Tout ceci étant basé sur un serveur, la spécification ainsi que les tests sont donc toujours à jour, et toujours disponibles.
- ✓ Fitnessse gère les versions. Il est possible de revenir en arrière sur un test.



# Utilisation de FitNesse

## Inconvénients

- ✗ L'interface est assez laide, et vieillotte.
- ✗ Il pourrait être pratique de spécifier un certain nombre de paramètre via l'interface au lieu que ce soit fait dans du code Wiki (Le type de test (FIT ou SLIM), le path pour les fichiers binaires, le path pour les JARs externes, ...).
- ✗ Il n'y a pas d'éditeur WYSIWYG intégré à Fitnessse pour l'écriture des tests. Il existe cependant un plugin-in plus ou moins buggé pour remédier à cela.
- ✗ On fait souvent des fautes de frappe entre le nom des entêtes dans le Wiki et le nom des fonctions ou des classes dans le code JAVA.
- ✗ L'intégration avec maven est très délicate. Il n'y a pas de solution « officielle », chacun y va de sa solution plus ou moins bancale

# Plan

- 1 Test du code hérité
- 2 Tests fonctionnels avec FitNesse
- 3 Tests unitaires et couverture du code**
- 4 Selenium et JUnit

# Couverture du code

## Tester quoi ?

- ➡ Plusieurs opinions sur le sujet
  - ❗ Absolument tout
  - ❗ L'interface publique
  - ❗ Les morceaux complexes, critiques
  - ❗ Comportement vs. état
- ➡ Règles générales
  - ❗ S'assurer que tout le code écrit fonctionne

# Couverture du code

## Définition

- ➡ Taux de code source exécuté par les tests
  - 📖 Fournit une valeur quantitative permettant de mesurer de manière indirecte la qualité des tests
  - 📖 Met en évidence les parties d'un programme qui ne sont pas testées
  - 📖 Permet d'ajouter de nouveaux tests

## Idée générale

- ➡ On veut **tester tous les chemins d'exécution possibles**
  - 📖 Plus on a de tests, moins on risque de régression
  - 📖 Pas besoin de tester le code trivial (getter / setter)

# Critères de couverture

## Statement Coverage

- ➡ Est-ce que toutes les instructions du code ont été exécutées ?
- ➡ Est-ce qu'un statement coverage de 100% garantit que les tests sont complets ?

```
int addAndCount(List list) {  
    if (list != null) {  
        list.add("Sample_text");  
    }  
    return list.size();  
}
```

- 🚩 Un Test avec `list != null` donne une couverture de 100%
- 🚩 Et si `list == null`??

## Avertissement

Ne pas écrire des cas de tests juste pour satisfaire votre outil de couverture de code

# Critères de couverture

## Decision Coverage

- Est-ce que les conditions dans les structures de contrôle ont été évaluées à true et false ?
- Similaire : Branch coverage

## Path Coverage

- Est-ce que tous les chemins d'exécution possibles au sein de chaque méthode ont été empruntés ?

```
if (A) {  
    if (B) { }  
}  
if (C) {  
}
```

- Le nombre de chemin augmente de manière exponentiel avec le nombre de branches. (10 `if` impliquent 1024 chemins possibles)
- Parfois tous les chemins ne sont pas atteignables.

# L'outil JaCoCo

## Présentation générale




- JaCoCo (Java Code Coverage ) est une bibliothèque de couverture de code Java gratuite distribuée sous la licence publique Eclipse (open source)
- Il peut être installé depuis Eclipse Market Place ou bien intégré à l'aide de Maven
- Mesures de couverture supportées
  - ☞ Instruction (bytecode instruction)
  - ☞ Branches
  - ☞ Lines
  - ☞ Methods
  - ☞ Classes (si au moins une méthode est exécutée)
  - ☞ Cyclomatic Complexity (Mesure de la complexité de la structure d'une fonction, différent du nombre de branches et du nombre de chemins)
- Site officiel : depuis <http://www.eclemma.org/jacoco>

# L'outil JaCoCo

## Installation

- ➡ Aller au site [www.mvnrepository.com](http://www.mvnrepository.com) et récupérer les méta-données relatives au plugin JaCoCo

**JaCoCo :: Maven Plugin » 0.8.6**

The JaCoCo Maven Plugin provides the JaCoCo runtime agent to your tests and allows basic report creation.

License	<a href="#">EPL 2.0</a>
Categories	<a href="#">Maven Plugins</a>
Date	(Sep 15, 2020)
Files	<a href="#">maven-plugin (52 KB)</a> <a href="#">View All</a>
Repositories	<a href="#">Central</a>
Used By	<b>59 artifacts</b>

[Maven](#) [Gradle](#) [SBT](#) [Ivy](#) [Grape](#) [Leiningen](#) [Buildr](#)

```
<dependency>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.6</version>
</dependency>
```


- ➡ Insérer les méta-données dans le fichier `POM.xml` et recompiler le projet



# L'outil JaCoCo

## Installation

- ➡ Aller au site [www.mvnrepository.com](http://www.mvnrepository.com) et récupérer les méta-données relatives au plugin JaCoCo

**JaCoCo :: Maven Plugin » 0.8.6**

The JaCoCo Maven Plugin provides the JaCoCo runtime agent to your tests and allows basic report creation.

License	<a href="#">EPL 2.0</a>
Categories	<a href="#">Maven Plugins</a>
Date	(Sep 15, 2020)
Files	<a href="#">maven-plugin (52 KB)</a> <a href="#">View All</a>
Repositories	<a href="#">Central</a>
Used By	<b>59 artifacts</b>

[Maven](#) [Gradle](#) [SBT](#) [Ivy](#) [Grape](#) [Leiningen](#) [Buildr](#)

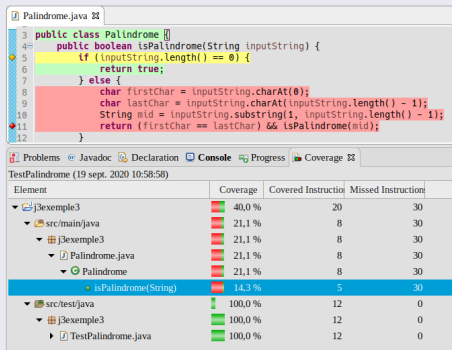
```
<dependency>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.6</version>
</dependency>
```

- ➡ Insérer les méta-données dans le fichier `pom.xml` et recompiler le projet

# L'outil JaCoCo

## Installation

➡ Exécuter JaCoCo à partir de menu contextuel Coverage As -> JUnit Test



```
3 public class Palindrome {
4     public boolean isPalindrome(String inputString) {
5         if (inputString.length() == 0) {
6             return true;
7         } else {
8             char firstChar = inputString.charAt(0);
9             char lastChar = inputString.charAt(inputString.length() - 1);
10            String mid = inputString.substring(1, inputString.length() - 1);
11            return (firstChar == lastChar) && isPalindrome(mid);
12        }
13    }
14 }
```

Problems @ Javadoc Declaration Console Progress Coverage

TestPalindrome (19 sept. 2020 10:58:58)

Element	Coverage	Covered Instruction	Missed Instruction
▼ j3exemple3	40,0 %	20	30
▼ src/main/java	21,1 %	8	30
▼ j3exemple3	21,1 %	8	30
▼ Palindrome.java	21,1 %	8	30
▼ Palindrome	21,1 %	8	30
isPalindrome(String)	14,3 %	5	30
▼ src/test/java	100,0 %	12	0
▼ j3exemple3	100,0 %	12	0
TestPalindrome.java	100,0 %	12	0

# L'outil JaCoCo

## Analyse d'un rapport

- ➡ Les rapports JaCoCo aident à analyser visuellement la couverture de code en utilisant des diamants avec des couleurs pour les branches et des couleurs d'arrière-plan pour les lignes
  - 🔴 Le losange rouge signifie qu'aucune branche n'a été exercée pendant la phase de test
  - 🟡 Le diamant jaune indique que le code est partiellement couvert ; certaines branches n'ont pas été exercées
  - 🟢 Le diamant vert signifie que toutes les branches ont été exercées au cours de la tester

# L'outil JaCoCo

## Exemple illustratif

- ➡ Afin d'atteindre une couverture de code à 100%, nous devons introduire des tests, qui couvrent les parties manquantes.

```
@Test
public void whenPalindrom__thenAccept() {
    Palindrome palindromeTester = new Palindrome();
    assertTrue(palindromeTester.isPalindrome("noon"));
}

@Test
public void whenNearPalindrom__thanReject() {
    Palindrome palindromeTester = new Palindrome();
    assertFalse(palindromeTester.isPalindrome("neon"));
}
```

# L'outil JaCoCo

## Installation

- ➡ Maintenant, toutes les lignes/branches/chemins de notre code sont entièrement couverts

```
3 public class Palindrome {
4     public boolean isPalindrome(String inputString) {
5         if (inputString.length() == 0) {
6             return true;
7         } else {
8             char firstChar = inputString.charAt(0);
9             char lastChar = inputString.charAt(inputString.length() - 1);
10            String mid = inputString.substring(1, inputString.length() - 1);
11            return (firstChar == lastChar) && isPalindrome(mid);
12        }
13    }
14 }
```

TestPalindrome (19 sept. 2020 11:11:35)

Element	Coverage	Covered Instruction	Missed Instruction
└ j3example3	100,0 %	68	0
└ src/main/java	100,0 %	38	0
└ j3example3	100,0 %	38	0
└ Palindrome.java	100,0 %	38	0
└ Palindrome	100,0 %	38	0
└ isPalindrome(String)	100,0 %	35	0
└ src/test/java	100,0 %	30	0

# Plan

- 1 Test du code hérité
- 2 Tests fonctionnels avec FitNesse
- 3 Tests unitaires et couverture du code
- 4 Selenium et JUnit**

# Présentation de Selenium

## Présentation de Selenium

- Selenium est une suite de tests automatisés (open source) pour les applications Web sur différents navigateurs et plates-formes.
- Selenium se concentre sur l'automatisation des applications Web. Les tests effectués à l'aide de l'outil Selenium sont généralement appelés tests Selenium.
- Puisque Selenium ne support pas l'exécution du code en test cases, on peut utilisé TestNG (ou JUnit) avec Selenium Framework



# Présentation de Selenium

## Présentation de Selenium

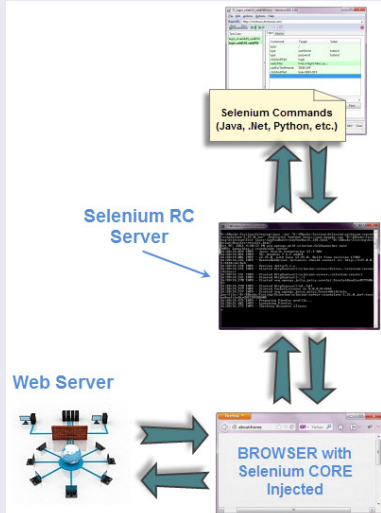
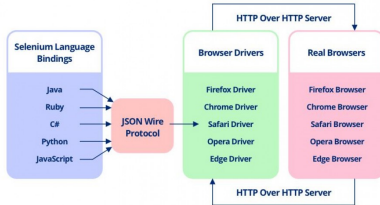
- Selenium est une famille de solution
- Selenium IDE
  - 🔴 Plugin navigateur pour enregistrer/exécuter les scénarios de tests
  - 🔴 Création des tests rapides
  - 🔴 Reproduire des bugs
  - 🔴 Absence de logs, metrics
- Selenium WebDriver
  - 🔴 Tests robustes en vue d'automatiser l'étude de la régression
  - 🔴 Passage à l'échelle des tests
  - 🔴 Clients en Java, C#, Python, Ruby, PHP, Javascript



# Présentation de Selenium

## Fonctionnement de Selenium

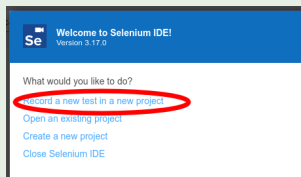
### Selenium WebDriver Architecture



# Utilisation de Selenium IDE

## Exemple : étude de cas simple avec Selenium IDE

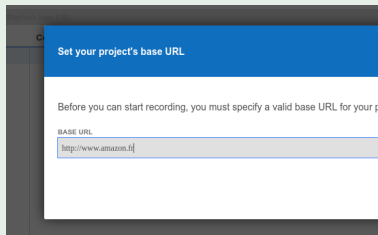
- On se propose d'enregistrer un scénario de test avec Selenium IDE. Ce scénario consiste à rechercher 'téléphone' dans le site `www.amazon.fr`
- Télécharger Selenium IDE selon le type du navigateur  
`https://www.selenium.dev/selenium-ide/`
  - 📖 Selenium IDE sera installé comme étant un Plugin navigateur pour enregistrer/exécuter les scénarios de tests
- Lancer Selenium IDE et enregistrer un test dans un nouveau projet



# Utilisation de Selenium IDE

## Exemple : étude de cas simple avec Selenium IDE

- ➡ Spécifier l'adresse de site `www.amazon.fr`, puis effectuer la recherche demandée durant l'enregistrement du test

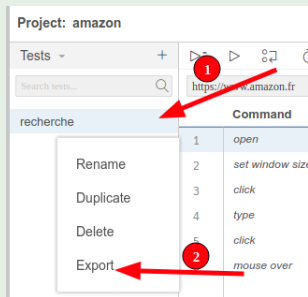


- ➡ Terminer l'enregistrement à partir de Selenium IDE

# Utilisation de Selenium IDE

## Exemple : étude de cas simple avec Selenium IDE

- ➡ Exporter en Java le scénario enregistré et l'intégrer dans un projet Java



- ➡ Essayer d'exécuter le scénario exporté depuis Eclipse

# Utilisation de Selenium IDE

## Exemple : étude de cas simple avec Selenium IDE

### ➡ Examiner la liste des actions enregistrées

	Command	Target	Value
1	open	/	
2	set window size	1440x757	
3	mouse over	css=area:nth-child(3)	
4	click	id=twotabsearchtextbox	
5	type	id=twotabsearchtextbox	téléphone
6	send keys	id=twotabsearchtextbox	{KEY_ENTER}

Command	type	
Target	id=twotabsearchtextbox	
Value	téléphone	
Description		

### ➡ Une commandeselenese est composée de :

- 📖 Command : ensemble prédéfini des types de commandes
- 📖 Target : comment identifier/référencer l'élément concerné (id, xpath, css, etc).
- 📖 Value : paramètre de la commande

# Selenium Web Driver

## Selenium Web Driver API

- Selenium Web Driver définit une API pour contrôler et interagir avec un navigateur web
- Navigateur générique : `RemoteWebDriver`
- Navigateur spécifique : `selenium.{nomnavigateur}`
  - 📖 Classes spécifiques pour interaction avec navigateur spécifique
- Interactions simples et complexes : `selenium.interactions`
  - 📖 `ClickAction`, `ClickAndHold`, `ContextClick`, `DoubleClick`, etc.
  - 📖 `KeyDownAction`, `KeyUpAction`, `SendKeys`
  - 📖 `touch.*`
- Localisation des éléments : Interface `Locators`
  - 📖 `FindBy{ClassName, CssSelector, Id, LinkText, Name, TagName, XPath}`

# Selenium Web Driver

## Exemple : Utiliser Selenium Web Driver avec JUnit

➡ On se propose de remplir le formulaire de la page html suivante

```
<!doctype html>
<html>
  <head>
    <script>
      function somme(){
        var nbr1, nbr2, sum;
        nbr1 = Number(document.getElementById("nbr1").value);
        nbr2 = Number(document.getElementById("nbr2").value);
        sum = nbr1 + nbr2;
        document.getElementById("sum").value = sum;
      }
    </script>
  </head>
  <body>
    <input id="nbr1"> + <input id="nbr2">
    <button onclick="somme()">Calculer la somme</button>
    = <input id="sum">
  </body>
</html>
```

# Selenium Web Driver

## Exemple : Utiliser Selenium Web Driver avec JUnit

```
@Test
public void test() {
    [...]
    WebElement element = driver.findElement(By.name("nbr1"));
    element.sendKeys("12");
    element.submit();
    WebElement element = driver.findElement(By.name("nbr2"));
    element.sendKeys("12");

    new WebDriverWait(driver, 10).until(ExpectedConditions.
        visibilityOfLocatedElement(By.id("sum")));
    assertEquals(driver.findElement(By.id("sum")).getText(),
        "24") ;
}
```



MERCI POUR VOTRE ATTENTION



Questions ?