

Test Driven Development en Java

Le développement piloté par les tests

Mai 2024

Présentation du formateur

Présentation du formateur

- Docteur et ingénieur en informatique
 - 📅 +15 ans d'expérience
- Expert et consultant en développement Java, C et C++
- Expert et consultant en systèmes embarqués
- Chercheur en tests et vérification des systèmes automatisés

Plan

- 1 Introduction
- 2 Tests automatisés avec le framework JUnit
- 3 Tests paramétrés
- 4 Bonnes pratiques
- 5 Les objets Mock et Stub

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.

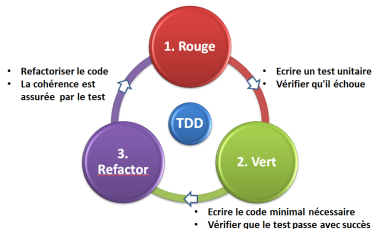
TDD : les trois bonnes règles

- ➡ 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 Introduction
- 2 Tests automatisés avec le framework JUnit
 - Le framework JUnit 4
 - Le framework JUnit 5
- 3 Tests paramétrés
- 4 Bonnes pratiques
- 5 Les objets Mock et Stub

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

```
1  @Test // Annotation designant la methode comme un test
2  public void testXXX() {
3      //Define
4          Instructions de mise en contexte
5      //When
6          Instruction sous test
7      //Then
8          Observation et verification de l'oracle
9  }
```

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)

```
1 public static int somme(int a,int b,int c) {  
2     return a+b-c;  
3 }
```

- Créer un Junit Test Case implémentant le test suivant :

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

- 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 4

Méthodes de test : vérification de levée d'une exception

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

```
1 @Test(expected = NullPointerException.class)
2 public void whenExceptionThrown_thenExpectationSatisfied() {
3     String test = null;
4     test.length();
5 }
```

➡ En utilisant l'annotation `@Rule`

📖 Elle permet de vérifier certaines propriétés de l'exception levée

```
1 @Rule
2 public ExpectedException exceptionRule = ExpectedException.none();
3
4 @Test
5 public void whenExceptionThrown_thenRuleIsApplied() {
6     exceptionRule.expect(NumberFormatException.class);
7     exceptionRule.expectMessage("For input string");
8     Integer.parseInt("1a");
9 }
```

- └ 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

```
1 package jlexemple1;
2 import static org.junit.Assert.*;
3 import org.junit.Assert;
4 import org.junit.Test;
5
6 class ValidPasswordTest {
7     @Test
8     void test() {
9         PasswordValidator pv=new PasswordValidator();
10        Assert.assertEquals(true,pv.isValid("123456"));
11    }
12 }
```

➤ 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

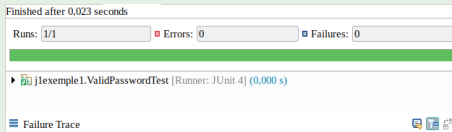


② Écrire le code applicatif

➡ On écrit le code applicatif pour faire passer le test

```
1 package j1exemple1;  
2  
3 public class PasswordValidator {  
4     public boolean isValid(String pw) {  
5         if (pw.length() >= 5 && pw.length() <= 10)  
6             return true;  
7         else  
8             return false;  
9     }  
10 }
```

➡ 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

```
1 package jlexemple1;
2 import static org.junit.Assert.*;
3 import org.junit.Assert;
4 import org.junit.Test;
5
6 class ValidPasswordTest {
7     @Test
8     void test() {
9         Assert.assertEquals("Verifier longueur mot de passe ",true,PasswordValidator.isValid("123456"
10         ));
11     }
12 }
```

➡ 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

```
1 package jlexemple1;
2
3 public class PasswordValidator {
4     static public boolean isValid(String pw) {
5         if (pw.length() >= 5 && pw.length() <= 10)
6             return true;
7         else
8             return false;
9     }
10 }
```

➡ Résultat de l'exécution de test : succès

Finished after 0,015 seconds

Runs: 1/1	Errors: 0	Failures: 0
-----------	-----------	-------------

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

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

Tests automatisés avec le framework JUnit 5

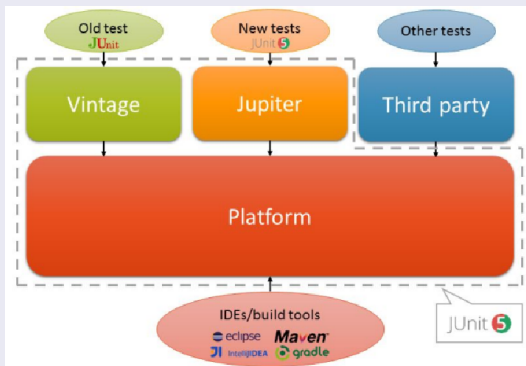
JUnit5 : la nouvelle version

- 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

```
1  @Test
2  public void whenExceptionThrown_thenAssertionSucceeds() {
3      Exception exception = assertThrows(NumberFormatException.class, () -> {
4          Integer.parseInt("1a");
5      });
6
7      String expectedMessage = "For input string";
8      String actualMessage = exception.getMessage();
9
10     assertTrue(actualMessage.contains(expectedMessage));
11 }
```

📌 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`.

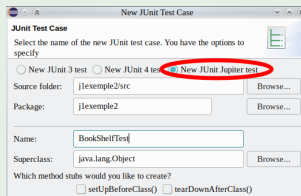
- └ Tests automatisés avec le framework JUnit
- └ Le framework JUnit 5

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

```
1 @Test
2 public void emptyBookShelfWhenNoBookAdded() {
3     BookShelf shelf = new BookShelf();
4     List<String> books = shelf.books();
5     assertTrue(books.isEmpty(), () -> "BookShelf should be empty.");
6 }
```

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

```
1 import java.util.Collections;
2 import java.util.List;
3
4 public class BookShelf {
5     public List<String> books() {
6         return Collections.emptyList();
7     }
8 }
```

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

```
1  @Test
2  void bookshelfContainsTwoBooksWhenTwoBooksAdded() {
3      BookShelf shelf = new BookShelf();
4      shelf.add("Programmer en Java");
5      shelf.add("Tester avant");
6      List<String> books = shelf.books();
7      assertEquals(2, books.size(), () -> "BookShelf should have two books.");
8  }
```


- └ 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

```
1 import java.util.ArrayList;
2 import java.util.List;
3 public class BookShelf {
4     private final List<String> books = new ArrayList<>();
5     public List<String> books() {
6         return books;
7     }
8     public void add(String bookToAdd) {
9         books.add(bookToAdd);
10    }
11 }
```

- └ 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

```
1  @Test
2  void booksReturnedFromBookShelfIsImmutableForClient() {
3      BookShelf shelf = new BookShelf();
4      shelf.add("Effective Java");
5      shelf.add("Code Complete");
6      List<String> books = shelf.books();
7      try {
8          books.add("The Mythical Man-Month");
9          fail(() -> "Should not be able to add book to books");
10     } catch (Exception e) {
11         assertTrue(e instanceof UnsupportedOperationException, () -> "Should throw
12             UnsupportedOperationException.");
13     }
14 }
```

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

```
1 public List<String> books() {  
2     return Collections.unmodifiableList(books);  
3 }
```

➡ 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

```
1 private BookShelf shelf;
2
3 @BeforeEach
4 void init() throws Exception {
5     shelf = new BookShelf();
6 }
7
8 @Test
9 public void emptyBookShelfWhenNoBookAdded() {
10     List<String> books = shelf.books();
11     assertTrue(books.isEmpty(), () -> "BookShelf should be empty.");
12 }
13
14 [...]
```

Tests automatisés avec le framework JUnit 5

Exercice de TDD avec JUnit 5

- ❶ 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
- ❷ On veut retourner une nouvelle liste triée en gardant la liste originale dans le même ordre

Tests automatisés avec le framework JUnit 5

Exemple illustratif du TDD avec JUnit 5

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

Tests automatisés avec le framework JUnit 5

Exemple du TDD avec JUnit 5

❶ Condition 1 : "A" → ""

📄 Code du test

```
1 import static org.junit.jupiter.api.Assertions.*;
2 import org.junit.jupiter.api.Test;
3
4 class TestRemoveAFirst {
5
6     @Test
7     void whenAThenEmpty() {
8         RemoveAFirst a = new RemoveAFirst();
9         assertEquals("", a.removeLetterA("A"));
10    }
11
12 }
```

📄 On propose une première solution pour le code applicatif

```
1
2 public class RemoveAFirst {
3     public String removeLetterA(String chaine) {
4         return "";
5     }
6 }
```


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

Tests automatisés avec le framework JUnit 5

Exemple du TDD avec JUnit 5

➡ Re-usinage

📄 Code du test

```
1 import static org.junit.jupiter.api.Assertions.*;
2 import org.junit.jupiter.api.Test;
3
4 class TestRemoveAFirst {
5     @Test
6     void whenAThenEmpty() {
7         assertEquals("", RemoveAFirst.removeLetterA("A"));
8     }
9 }
```

📄 Code applicatif

```
1 public class RemoveAFirst {
2     public static String removeLetterA(String chaine) {
3         return "";
4     }
5 }
```

Tests automatisés avec le framework JUnit 5

Exemple du TDD avec JUnit 5

② Condition 2 : "AB" → "B"

📄 Code du test

```
1  @Test
2  void whenABThenB() {
3      assertEquals("B", RemoveAFirst.removeLetterA("AB"));
4  }
```

📄 Code applicatif

```
1  public class RemoveAFirst {
2      public static String removeLetterA(String chaine) {
3          if (chaine.length() == 2)
4              return chaine.substring(1);
5          return "";
6      }
7  }
```

Tests automatisés avec le framework JUnit 5

Exemple du TDD avec JUnit 5

③ Condition 3 : "AABC" → "BC"

📄 Code du test

```
1  @Test
2  void whenAABCThenBC() {
3      assertEquals("BC", RemoveAFirst.removeLetterA("AABC"));
4  }
```

📄 Code applicatif

```
1  public class RemoveAFirst {
2      public static String removeLetterA(String chaine) {
3          String nvChaine="";
4          if (chaine.length()>2) {
5              if (chaine.charAt(0)=='A' && chaine.charAt(1)=='A')
6                  nvChaine=chaine.substring(2);
7          }
8          else if (chaine.length()==2) {
9              if (chaine.charAt(0)=='A')
10                 nvChaine=chaine.substring(1);
11          }
12          return nvChaine;
13      }
14  }
```

Tests automatisés avec le framework JUnit 5

Exemple du TDD avec JUnit 5

④ Condition 4 : "BACD" → "BCD"

📄 Code du test

```
1  @Test
2  void whenBACDThenBCD() {
3      assertEquals("BCD", RemoveAFirst.removeLetterA("BACD"));
4  }
```

📄 Code applicatif

```
1  public static String removeLetterA(String chaine) {
2      String nvChaine = "";
3      int pos1 = -1, pos2 = -1;
4      if (chaine.length() > 2) {
5          pos1 = chaine.indexOf('A');
6          pos2 = chaine.lastIndexOf('A');
7          if (pos1 == 0 && pos2 == 1)
8              nvChaine = chaine.substring(2);
9          else if (pos1 == 1)
10             nvChaine = chaine.charAt(0) + chaine.substring(2);
11         else
12             nvChaine = chaine;
13     } else if (chaine.length() == 2) {
14         pos1 = chaine.indexOf('A');
15         if (pos1 == 0)
16             nvChaine = chaine.substring(1);
17         else if (pos1 == 1)
18             nvChaine = Character.toString(chaine.charAt(0));
19     }
20     return nvChaine;
21 }
```

Tests automatisés avec le framework JUnit 5

Exemple du TDD avec JUnit 5

⑤ Condition 5 : "BBAA" → "BBAA"

📄 Code du test

```
1  @Test
2  void whenBBAAThenBBAA() {
3      assertEquals("BBAA", RemoveAFirst.removeLetterA("BBAA"));
4  }
```

Le test est au vert ! On passe au test suivant.

Tests automatisés avec le framework JUnit 5

Exemple du TDD avec JUnit 5

⑥ Condition 6 : "AABAA" → "BAA"

📄 Code du test

```
1 @Test
2 void whenAABAAThenBAA() {
3     assertEquals("BAA", RemoveAFirst.removeLetterA("AABAA"));
4 }
```

📄 Code applicatif

```
1 public static String removeLetterA(String chaine) {
2     String nvChaine="";
3     int pos=-1;
4     if (chaine.length()>2) {
5         pos=chaine.indexOf('A');
6         if (pos==0 && chaine.charAt(1)=='A')
7             nvChaine=chaine.substring(2);
8         else if (pos==1) nvChaine=chaine.charAt(0)+chaine.substring(2);
9         else nvChaine=chaine;
10    }
11    else if (chaine.length()==2) {
12        pos=chaine.indexOf('A');
13        if (pos==0) nvChaine=chaine.substring(1);
14        else if (pos==1) nvChaine=Character.toString(chaine.charAt(0));
15    }
16    return nvChaine;
17 }
```

Plan

- 1 Introduction
- 2 Tests automatisés avec le framework JUnit
- 3 Tests paramétrés**
- 4 Bonnes pratiques
- 5 Les objets Mock et Stub

Tests paramétrés

Tests paramétrés

- Junit 5 permet les tests paramétrés 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 (shorts), byte (bytes), int (ints), long (longs), float (floats), double (doubles), char (chars), java.lang.String (strings)`

Tests paramétrés

Exemple : un seul argument

➡ Soit à tester la méthode suivante

```
1 public static boolean isOdd(int number) {  
2     return number % 2 != 0;  
3 }
```

➡ Code de la méthode de test

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

Tests paramétrés

Un seul argument

- ➡ Spécifier la valeur `null`

`@NullSource`

- ➡ Pour le type `String`

- ▣ Spécifier une chaîne de caractères vide

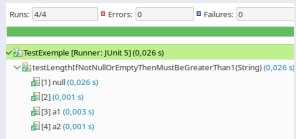
`@EmptySource`

- ▣ Spécifier une chaîne de caractères vide, et la valeur `null`

`@NullAndEmptySource`

- ➡ Exemple

```
1 @ParameterizedTest
2 @NullAndEmptySource
3 @ValueSource(strings = { "a1", "a2" })
4 void testLengthIfNotNullOrEmptyThenMustBeGreaterThan1(String word) {
5     if (word==null || word=="") return;
6     assertTrue(word.length()>1,"Length must be > 1");
7 }
```



Tests paramétrés

Plusieurs arguments

- L'annotation `@CsvSource` permet de spécifier des données au format CSV
 - 📖 Définie dans le paquet `org.junit.jupiter.params.provider`
- La conversion est effectuée systématiquement à partir du type `String`
- Principaux attributs
 - 📖 `delimiter` : spécifier le caractère de séparation utilisé. La valeur par défaut est la virgule (,)
 - 📖 `quoteCharacter` : spécifier le caractère de guillemet utilisé
 - 📖 `numLinesToSkip` : le nombre de lignes à ignorer au début

Tests paramétrés

Exemple 1 : plusieurs arguments

➡ La méthode à tester

```
1 public static int somme(int a, int b, int c) {  
2     return a+b+c;  
3 }
```

➡ Code de la méthode de test

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

Exemple 2 : plusieurs arguments

```
1 @ParameterizedTest  
2 @CsvSource(value = {"test:test", ";", "'':''"}, delimiter = ';', quoteCharacter='\'')  
3 void toLowerCase_ShouldGenerateTheExpectedLowercaseValue(String input, String expected) {  
4     if (input==null) return;  
5     String actualValue = input.toLowerCase();  
6     assertEquals(expected, actualValue);  
7 }
```

Tests paramétrés

Exemple 3 : plusieurs arguments

➡ L'exemple pour tester la classe `RemoveAFirst`

➡ Code de la méthode de test

```
1 @ParameterizedTest
2 @CsvSource({ "", "A", "B, AB", "BC, AABC", "BCD, BACD", "BBAA, BBAA", "BAA, AABAA" })
3 void whenValueThenExpected(String expected, String value) {
4     assertEquals(expected, RemoveAFirst.removeLetterA(value));
5 }
```

Tests paramétrés

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
- Accepte les mêmes attributs que `CsvSource`

Exemple 1 : données à partir d'un fichier

- Code de la méthode de test

```
1 @ParameterizedTest
2 @CsvFileSource(resources = "../test.csv", numLinesToSkip = 1)
3 void testSommeShouldReturnSumofNumbersfromFile(int a,int b,int c) {
4     assertEquals(a+b+c,Operations.somme(a,b,c));
5 }
```

- Exemple de contenu du fichier `test.csv`

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

Tests paramétrés

Exemple 2 : données à partir d'un fichier

➡ Exemple pour tester la classe RemoveAFirst

```
1 @ParameterizedTest
2 @CsvFileSource(resources = "../test.csv", numLinesToSkip = 1)
3 void whenValueThenExpectedFromCSV(String expected,String value) {
4     assertEquals(expected,RemoveAFirst.removeLetterA(value));
5 }
```

➡ Contenu du fichier test.csv

```
1 expected,value
2 "",A
3 B,AB
4 BC,AABC
5 BCD,BACD
6 BBAA,BBAA
7 BAA,AABAA
```

Plan

- 1 Introduction
- 2 Tests automatisés avec le framework JUnit
- 3 Tests paramétrés
- 4 Bonnes pratiques**
- 5 Les objets Mock et Stub

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

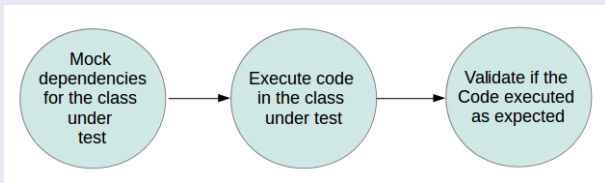
Plan

- 1 Introduction
- 2 Tests automatisés avec le framework JUnit
- 3 Tests paramétrés
- 4 Bonnes pratiques
- 5 Les objets Mock et Stub**

Présentation de Mockito

Pourquoi le mocking ?

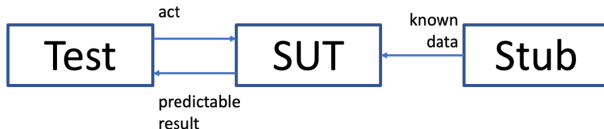
- Certains objets "réels" dans les tests sont difficiles à instancier et à les configurer
- Parfois, seulement les interfaces existent 🚫 Pas d'implémentation prête



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** (substitut, 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`

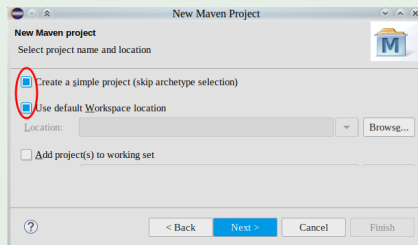
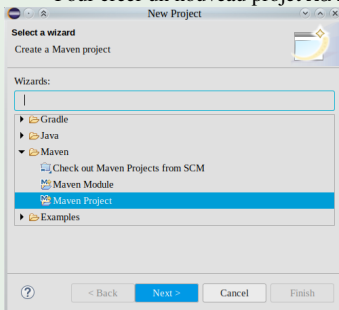


Mockito supporte JUnit 5 à partir de la version 2.16.3

Utilisation de Mockito

Exemple illustratif : Intégration de Mockito dans un projet Java

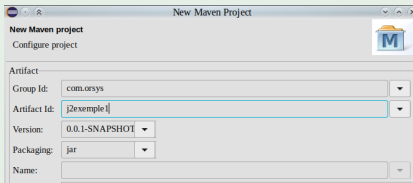
➡ Pour créer un nouveau projet Maven, aller au menu **File->New->Project**



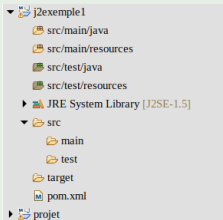
Utilisation de Mockito

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 et récupérer les méta-données relatives au framework Mockito

Maven Gradle SBT Ivy Grape Leining

```
<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
 - ❶ 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

Select Dependency

Group Id:

Artifact Id:

Version: Scope:

Enter groupId, artifactId or sha1 prefix or pattern ("):

⚠ Index downloads are disabled, search results may be incomplete.

Search Results:

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

```
1 public interface Calcul {
2     int Somme(int x,int y);
3 }
4 /*****/
5 import org.junit.Test;
6 import static org.mockito.Mockito.*;
7
8 public class ExempleTest {
9     @Test
10     public void testMock() {
11         Calcul monMock=mock(Calcul.class);
12         assertTrue(true);
13     }
14 }
```

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 `@Mock`, qui se trouvent dans la classe passée en paramètre.

```
1 public void FooTest {
2     private Foo foo;
3     @Mock
4     private MaClasse monMock;
5     @Before
6     public void setUp() {
7         MockitoAnnotations.initMocks(this);
8         foo = new Foo(monMock);
9         ... // Testing
10    }
11
12 }
```

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

```
1  @RunWith(MockitoJUnitRunner.class)
2  public void FooTest {
3      private Foo _foo;
4      @Mock
5      private MaClasse _monMock;
6      @Before
7      public void setUp() {
8          _foo = new Foo(_monMock);
9          ... // Testing
10     }
11     ...
12 }
```

Création d'une doublure avec l'annotation @Mock

Initialisation par la règle MockitoRule

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

```
1 public void FooTest {  
2     private Foo _foo;  
3     @Mock  
4     private MaClasse _monMock;  
5     @Rule  
6     public MockitoRule mockitoRule = MockitoJUnit.rule();  
7     @Before  
8     public void setUp() {  
9         _foo = new Foo(_monMock);  
10    }  
11    ...  
12 }
```

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

```
1 public class ExempleTest {
2     @Test
3     public void testMock() {
4         Calcul monMock=mock(Calcul.class);
5         when(monMock.Somme(4, 3)).thenReturn(7);
6         when(monMock.Incrementer()).thenReturn(0).thenReturn(1);
7
8         System.out.println("Somme(4,3) retourne : "+monMock.Somme(4, 3));
9         System.out.println("Incrementer : "+monMock.Incrementer());
10        System.out.println("Incrementer : "+monMock.Incrementer());
11        assertTrue(true);
12    }
13 }
```

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

```
1 @Test(expected = BidonException.class)  
2 public void should_throw_exception() {  
3     monMock.methode();  
4 }
```

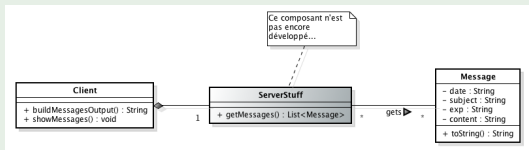
Exemple

```
1 @Test(expected=IllegalArgumentException.class)  
2 public void testMockException() {  
3     Calcul monMock=mock(Calcul.class);  
4     when(monMock.Diviser(5,0)).thenThrow(new IllegalArgumentException("Argument nul"));  
5     when(monMock.Diviser(5,2)).thenReturn(2);  
6     System.out.println("Diviser (5,2) =" +monMock.Diviser(5,2));  
7     System.out.println("Diviser (5,0) =" +monMock.Diviser(5,0));  
8 }
```


Exemple 1 : utilisation de Mockito

Exemple 1 : utilisation de Mockito

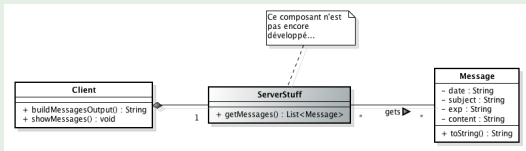
On considère la structure de classes ci-après décrivant un système de messagerie simplifié



- 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`

Exemple 1 : utilisation de Mockito

Exemple 1 : utilisation de Mockito



➡ L'objectif est de tester la classe `Client` et vérifier quelle retourne la liste des messages correctement sous forme d'une chaîne de caractères

📖 Si aucun message est reçu, on aura la chaîne suivante :

Pas de message!

📖 Si plusieurs messages sont reçus, on aura la chaîne suivante :

```
1 Message 1
2 sujet1
3 Reçu le 7/2/2024
4 Expéditeur bob@bob
5 Contenu1
6 Message 2
7 sujet2
8 Reçu le 7/2/2024
9 Expéditeur emilie@emilie
10 Contenu2
```

Exemple 1 : utilisation de Mockito

Code de la classe Message

```
1 public class Message {
2     Date date;
3     String subject;
4     String exp;
5     String content;
6
7     public Message(Date date, String subject, String exp, String content) {
8         this.date = date;
9         this.subject = subject;
10        this.exp = exp;
11        this.content = content;
12    }
13
14    public String toString() {
15        return subject + "\nReçu le : " + date + "\nExpéditeur : " + exp + "\n" + content;
16    }
17
18 }
```

Code de l'interface ServerStuff!!

```
1 import java.util.List;
2
3 public interface ServerStuff {
4     List<Message> getMessages();
5 }
```

Exemple 1 : utilisation de Mockito

Code de la classe Client

```
1 public class Client {
2     ServerStuff serverStuff;
3     public Client(ServerStuff serverStuff) {
4         this.serverStuff = serverStuff;
5     }
6
7     public String buildMessagesOutput() {
8         List<Message> messages = this.serverStuff.getMessages();
9         String output = "";
10        if (messages.size()>0) {
11            int i=1;
12            for (Message message messages) {
13                output += "Message "+i+"\n";
14                output += message + "\n";
15                i++;
16            }
17        }
18        else {
19            output="Pas de nouveau message!";
20        }
21        return output;
22    }
23 }
```

Code de la classe de test

```
1 import java.util.ArrayList;
2 import java.util.Arrays;
3 import java.util.Date;
4 import java.util.List;
5 import org.junit.jupiter.api.BeforeEach;
6 import org.junit.jupiter.api.Test;
7 class TestClient {
8     private Client client;
9     private ServerStuff serverStuff;
10    @BeforeEach
11    void setUp() throws Exception {
12        // Mocking
13        serverStuff = mock(ServerStuff.class);
14        // Instancier le client
15        this.client = new Client(serverStuff);
16    }
17    @Test
18    void whenClientHasTwoMessagesThenMessagesShouldBeDisplayed() {
19        // Stubbing
20        when(serverStuff.getMessages()).thenReturn((List) Arrays.asList(
21            new Message("7/2/2024", "sujet1", "bob@bob", "Contenu1"),
22            new Message("7/2/2024", "sujet2", "emilie@emilie", "Contenu2")));
23        assertEquals("Message 1\n" + "sujet1\n" + "Reçu le : 7/2/2024\n"
24            + "Expéditeur : bob@bob\n" + "Contenu1\n" + "Message 2\n" + "sujet2\n"
25            + "Reçu le : 7/2/2024\n" + "Expéditeur : emilie@emilie\n"
26            + "Contenu2\n", client.buildMessagesOutput());
27    }
28    @Test
29    void whenClientHasNoMessageThenPasDeMessageShouldBeDisplayed() {
30        when(serverStuff.getMessages()).thenReturn((List) new ArrayList<>());
31        assertEquals("Pas de message!", client.buildMessagesOutput());
32    }
33 }
```

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 misee 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

```
1 public class Two {
2     public void doSomething() {
3         System.out.println("Un autre service fonctionne...");
4     }
5 }
6 public class One {
7     private Two _two;
8     public One( Two two ) {
9         _two = two;
10    }
11    public void work() {
12        System.out.println("Un premier service fonctionne");
13        _two.doSomething();
14    }
15 }
16 ///////////////////////////////////////////////////
17 public class OneTest {
18     @Mock
19     private Two _two;
20     @InjectMocks
21     private One _one;
22     @Before
23     public void setup() {
24         MockitoAnnotations.initMocks(this);
25     }
26     @Test
27     public void oneCanWork() throws Exception {
28         _one.work();
29         Mockito.verify(_two).doSomething();
30     }
31 }
```

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`

```
1 public final class OneTest {
2     @Mock
3     private Two _two;
4     @InjectMocks
5     private One _one;
6     @Before
7     public void setup() {
8         MockitoAnnotations.initMocks(this);
9     }
10    @Test
11    public void oneCanWork() throws Exception {
12        _one.work();
13        Mockito.verify(_two).doSomething();
14    }
15 }
```

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.

```
1 package j2exemple2;
2 import java.sql.Connection;
3 import java.sql.DriverManager;
4 import java.sql.SQLException;
5
6 public class DatabaseService {
7     private Connection dbConnection;
8     public void getDbConnection() throws ClassNotFoundException, SQLException {
9         dbConnection = DriverManager.getConnection("http://127.0.0.1:3306/mockito_db","root"
10             , "passwd");
11     }
12     public int executeQuery(String query) throws SQLException {
13         return dbConnection.createStatement().executeUpdate(query);
14     }
15 }
```


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

```
1 package j2exemple2;
2 import static org.junit.jupiter.api.Assertions.*;
3 import java.sql.*;
4 import org.junit.jupiter.api.*;
5 import org.mockito.*;
6 class DatabaseServiceTest {
7     @InjectMocks
8     private DatabaseService dbConnection;
9     @Mock
10    private Connection mockConnection;
11    @Mock
12    private Statement mockStatement;
13    @BeforeEach
14    public void setUp() {
15        MockitoAnnotations.initMocks(this);
16    }
17    @Test
18    void test() throws Exception {
19        Mockito.when(mockConnection.createStatement()).thenReturn(mockStatement);
20        Mockito.when(mockConnection.createStatement().executeUpdate(Mockito.anyString())).
21            thenReturn(1);
22        int value=dbConnection.executeQuery("");
23        assertEquals(1, value);
24    }
```

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.

MERCI POUR VOTRE ATTENTION



Questions ?