

Analyse et Conception

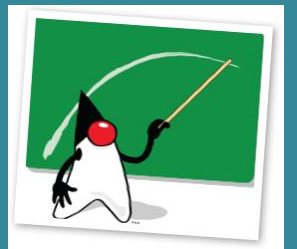
Module 8 – Les tests

Les tests unitaires avec JUnit



Objectifs

- Présentation du Framework JUnit
- Les annotations principales
- Créer une campagne de tests
- Valider la couverture des tests avec Eclemma
- Notion de bouchon et de mocking



Présentation

- JUnit est un framework de tests unitaires pour Java
- Un test unitaire se construit en 3 parties (AAA) :
 - Arrange : initialiser les données et l'état des objets dans l'objectif de tester le scénario visé.
 - Act: exécuter le traitement à vérifier (Le traitement est une partie du System Under Test / SUT)
 - Assert: vérifier les résultats du traitement et l'état des objets après traitement en fonction de ce qui est attendu
- Quelques principes à suivre:
 - Un test doit avoir un objectif unique – Si le test échoue, on doit savoir quel objectif a échoué.
 - Un test doit être le plus petit et le plus simple possible
 - Chaque test doit être isolé et ne pas dépendre d'un autre test
 - Les tests doivent être exécutés régulièrement (donc doivent être automatisés)

JUnit : les annotations principales

- JUnit utilise des annotations pour diriger l'exécution des tests :
 - **@Test**: (org.junit.jupiter.api.Test) permet de définir les méthodes de test
 - **@BeforeAll** : Déclenche l'exécution d'une méthode avant l'exécution du premier test de la classe.
Ceci peut servir à initialiser des ressources qui seront partagées par l'ensemble des tests
 - **@AfterAll** : Déclenche l'exécution d'une méthode après l'exécution du dernier test de la classe.
Ceci peut servir à fermer les différentes ressources partagées
 - **@BeforeEach** : Déclenche l'exécution d'une méthode avant l'exécution de chaque test de la classe.
 - **@AfterEach** : Déclenche l'exécution d'une méthode après l'exécution de chaque test de la classe
 - **@RunWith** et **@SuiteClasses**: permettent de définir une campagne de test

Comportement à l'exécution

```
public class DemoAnnotations {  
  
    @BeforeAll  
    public static void initAvantTousLesTests() {  
        System.out.println("Execution BeforeAll");  
    }  
  
    @BeforeEach  
    public void initAvantChaqueTest() {  
        System.out.println("Execution BeforeEach");  
    }  
  
    @Test  
    public void test1() {  
        System.out.println("Execution Test 1");  
    }  
  
    @Test  
    public void test2() {  
        System.out.println("Execution Test 2");  
    }  
  
    @AfterEach  
    public void apresChaqueTest() {  
        System.out.println("Execution AfterEach");  
    }  
  
    @AfterAll  
    public static void apresTousLesTests() {  
        System.out.println("Execution AfterAll");  
    }  
}
```



Execution BeforeAll
Execution BeforeEach
Execution Test 1
Execution AfterEach
Execution BeforeEach
Execution Test 2
Execution AfterEach
Execution AfterAll

Les méthodes de vérification

- La classe `org.junit.jupiter.api.Assertions` (JUnit V5)

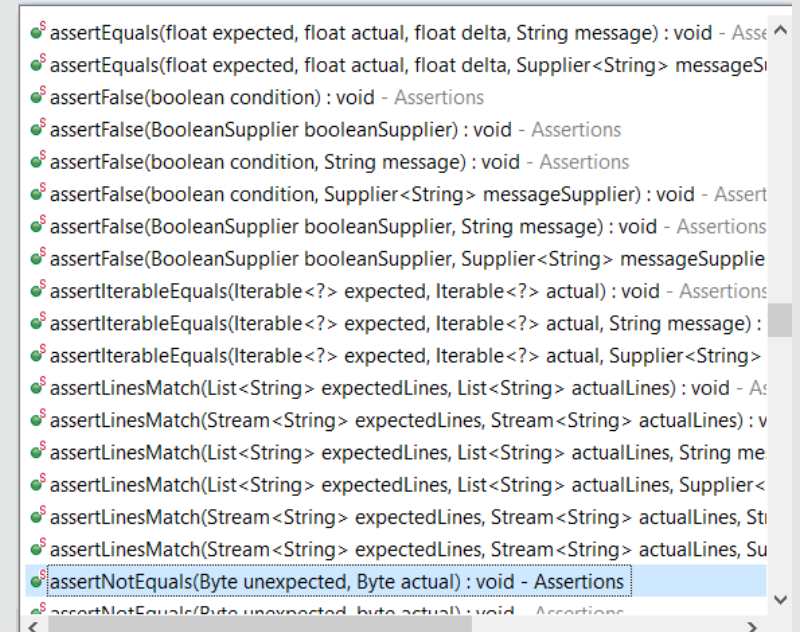
propose un ensemble de méthodes permettant de tester l'exactitude

des résultats des traitements testés

Si une des assertions d'un test n'est pas vérifiée, le test passe au rouge (échec), sinon il passe au vert (réussite)

- Il est recommandé d'utiliser les imports statiques pour utiliser ces méthodes, Exemple :

```
import static org.junit.jupiter.api.Assertions.assertEquals;
```



```
assertEquals(float expected, float actual, float delta, String message) : void - Assertions
assertEquals(float expected, float actual, float delta, Supplier<String> messageSupplier) : void - Assertions
assertFalse(boolean condition) : void - Assertions
assertFalse(BooleanSupplier booleanSupplier) : void - Assertions
assertFalse(boolean condition, String message) : void - Assertions
assertFalse(boolean condition, Supplier<String> messageSupplier) : void - Assertions
assertFalse(BooleanSupplier booleanSupplier, String message) : void - Assertions
assertFalse(BooleanSupplier booleanSupplier, Supplier<String> messageSupplier) : void - Assertions
assertIterableEquals(Iterable<?> expected, Iterable<?> actual) : void - Assertions
assertIterableEquals(Iterable<?> expected, Iterable<?> actual, String message) : void - Assertions
assertIterableEquals(Iterable<?> expected, Iterable<?> actual, Supplier<String> messageSupplier) : void - Assertions
assertLinesMatch(List<String> expectedLines, List<String> actualLines) : void - Assertions
assertLinesMatch(Stream<String> expectedLines, Stream<String> actualLines) : void - Assertions
assertLinesMatch(List<String> expectedLines, List<String> actualLines, String message) : void - Assertions
assertLinesMatch(List<String> expectedLines, List<String> actualLines, Supplier<String> messageSupplier) : void - Assertions
assertLinesMatch(Stream<String> expectedLines, Stream<String> actualLines, String message) : void - Assertions
assertLinesMatch(Stream<String> expectedLines, Stream<String> actualLines, Supplier<String> messageSupplier) : void - Assertions
assertNotEquals(Byte unexpected, Byte actual) : void - Assertions
assertNotEquals(Byte unexpected, Byte actual, Supplier<String> messageSupplier) : void - Assertions
```

Vérifier l'envoi d'une exception

- La méthode `assertThrows` permet de vérifier la levée d'une exception :

```
@Test
public void testDiviserCasNegatif() throws DivisionParZero {
    //Arrange
    Calculatrice calc = new Calculatrice();
    double nombre1 = 30;
    double nombre2 = 0;

    //Act / Assert
    assertThrows(DivisionParZero.class,
        () -> calc.diviser(nombre1, nombre2));
}
```

Les tests unitaires avec JUnit

Mise en place et exemple d'utilisation

Démonstration



Les tests unitaires avec JUnit

TP Coffre fort

TP



Campagne de tests

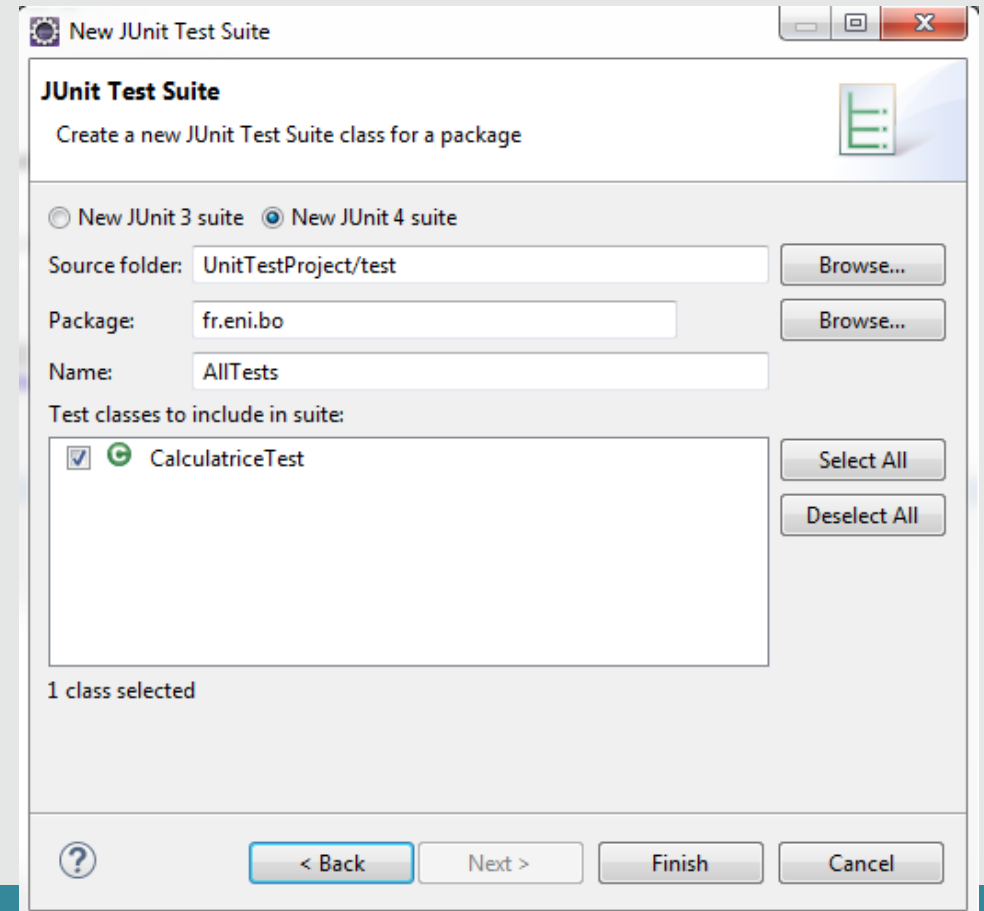
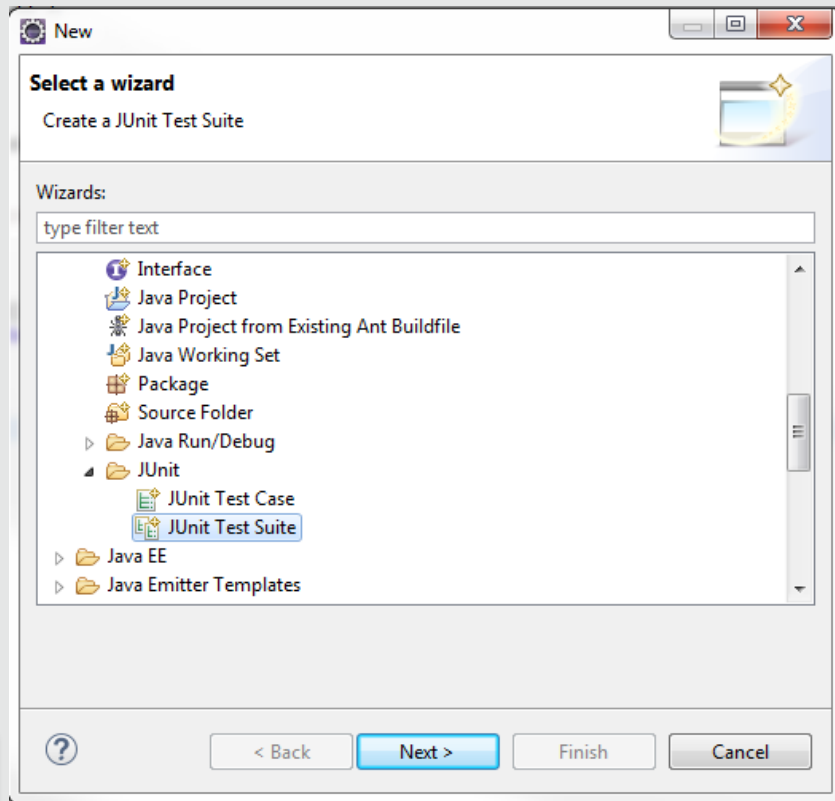
- Les tests sont souvent situés dans différentes classes
- Le regroupement de ces classes pour l'exécution des tests écrits à l'intérieur s'appelle une campagne de test
- JUnit offre la possibilité de regrouper un certain nombre de classe de test dans une campagne de test
- Il faut utiliser pour cela les annotations **@RunWith** et **@SuiteClasses**

```
@RunWith(Suite.class)
@SuiteClasses({ CalculatriceTest.class })
public class AllTests {

}
```

Création d'une campagne de tests

- Nécessaire lorsqu'on souhaite exécuter les tests contenus dans plusieurs classes de test



Création d'une campagne de tests

- Une classe « AllTests » est créée

- Elle recense les classes de test à exécuter

```
package fr.eni.bo;  
  
import org.junit.runner.RunWith;  
  
@RunWith(Suite.class)  
@SuiteClasses({ CalculatriceTest.class })  
public class AllTests {  
  
}
```

- Cette classe peut être le point d'entrée pour exécuter des tests

Exécution d'une campagne de tests

- L'exécution se fait de la même manière que pour une classe de test unique
- L'écran des résultats montre le résultat de l'exécution de chacune des classes contenues dans la campagne de tests

Les tests unitaires avec JUnit

Création d'une campagne de tests

Démonstration

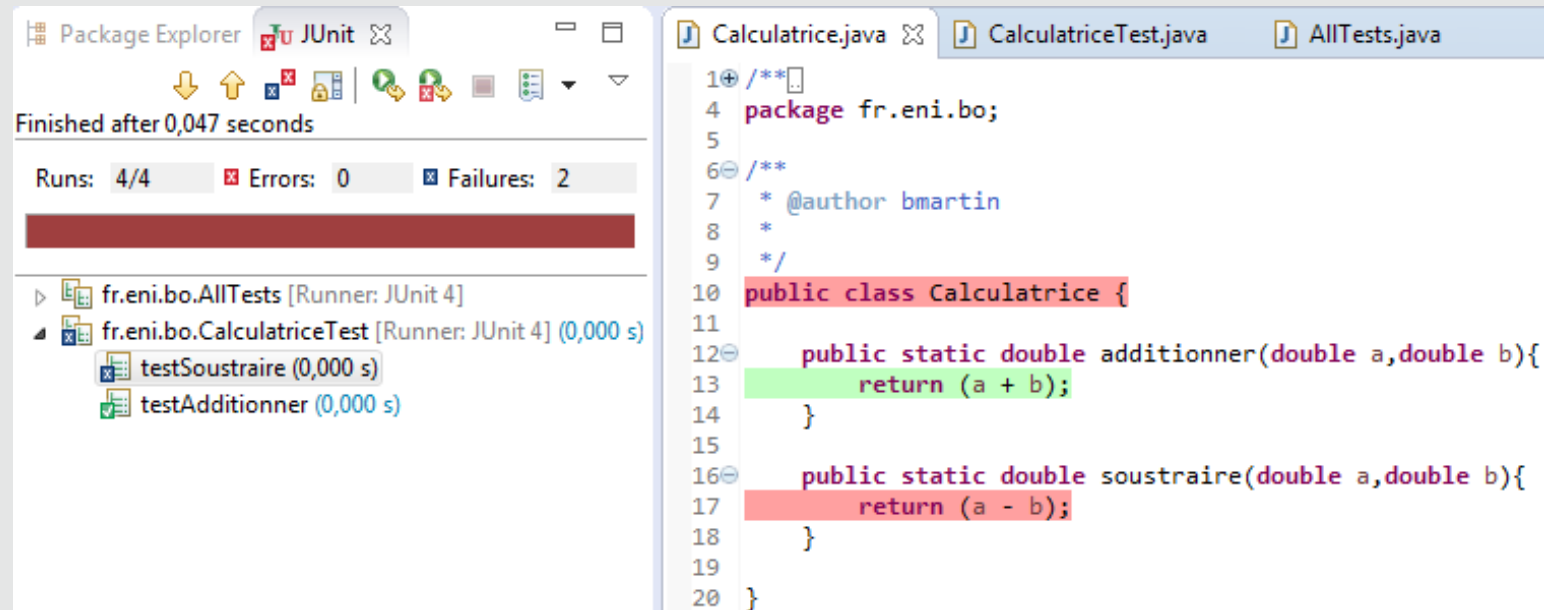


Mesurer la couverture des tests avec Ecclema

- Mesurer le pourcentage de code vérifié par les tests
- Identifier les lignes de codes testées
- Identifier les lignes de codes non testés

Mesurer la couverture des tests avec Eclemma

- A partir du projet : Menu Coverage As -> JUnit Test
- Un écran de résultat apparaît:











The screenshot displays an IDE interface. On the left, the 'JUnit' tab shows test results: 'Finished after 0,047 seconds', 'Runs: 4/4', 'Errors: 0', and 'Failures: 2'. Below this, a tree view shows the test hierarchy: 'fr.eni.bo.AllTests [Runner: JUnit 4]' containing 'fr.eni.bo.CalculatriceTest [Runner: JUnit 4] (0,000 s)', which in turn contains 'testSoustraire (0,000 s)' and 'testAdditionner (0,000 s)'. On the right, the 'CalculatriceTest.java' file is open, showing the following code:

```
1+ /**
4 package fr.eni.bo;
5
6+ /**
7  * @author bmartin
8  *
9  */
10 public class Calculatrice {
11
12+     public static double additionner(double a,double b){
13         return (a + b);
14     }
15
16+     public static double soustraire(double a,double b){
17         return (a - b);
18     }
19
20 }
```


Une vue d'ensemble de la couverture

- La vue Coverage

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
▲ UnitTestProject	 60,6 %	20	13	33
▲ src	 36,4 %	4	7	11
▲ fr.eni.bo	 36,4 %	4	7	11
▲ Calculatrice.java	 36,4 %	4	7	11
▲ Calculatrice	 36,4 %	4	7	11
● soustraire(double, double)	 0,0 %	0	4	4
● additionner(double, double)	 100,0 %	4	0	4
▶ test	 72,7 %	16	6	22

La granularité des tests unitaires

- Un test unitaire est un test d'un seul composant indépendamment des autres composants de l'application
- Une méthode appelle bien souvent d'autres méthodes situées dans la même classe ou dans d'autres classes. Ces autres classes sont alors appelées dépendances.
- Le choix doit être fait de la granularité :
 - On peut tester un service métier avec ses dépendances, notamment les appels à la couche d'accès aux données
 - OU on peut tester le composant en isolation complète de ses dépendances
 - Dans le deuxième cas, il y aura nécessité de fournir des **bouchons (stub en anglais)** permettant de simuler le comportement des dépendances en fonction des scénarios de test.

Les tests unitaires avec JUnit

Granularité des tests et notion de bouchon

Démonstration



Notion de Mock

- Pour faciliter la gestion des bouchons, des frameworks ont été créés. Ces outils vont créer automatiquement les instances de bouchon. Ces instances seront appelés des Mocks.
- To mock = simuler, reproduire le comportement
- Quoi ? Le retour des méthodes invoquées dans la méthode testée unitairement
 - Retour de la fonction (Type objet ou type primitif)
 - Exception
- Complémentaire aux tests unitaires dans le cas de méthodes ayant des dépendances avec d'autres objets
 - Exemple:

```
private UtilisateurDAO utilisateurDAO;  
public boolean authentifier(String user, String password) {  
    //du code interne à authentifier à tester en isolation  
    //...  
    return utilisateurDAO.authentifier(user, password);  
}
```

Pour tester la méthode **authentifier** en isolation il faut mocker `utilisateurDAO.authentifier(...)`

Mockito : présentation

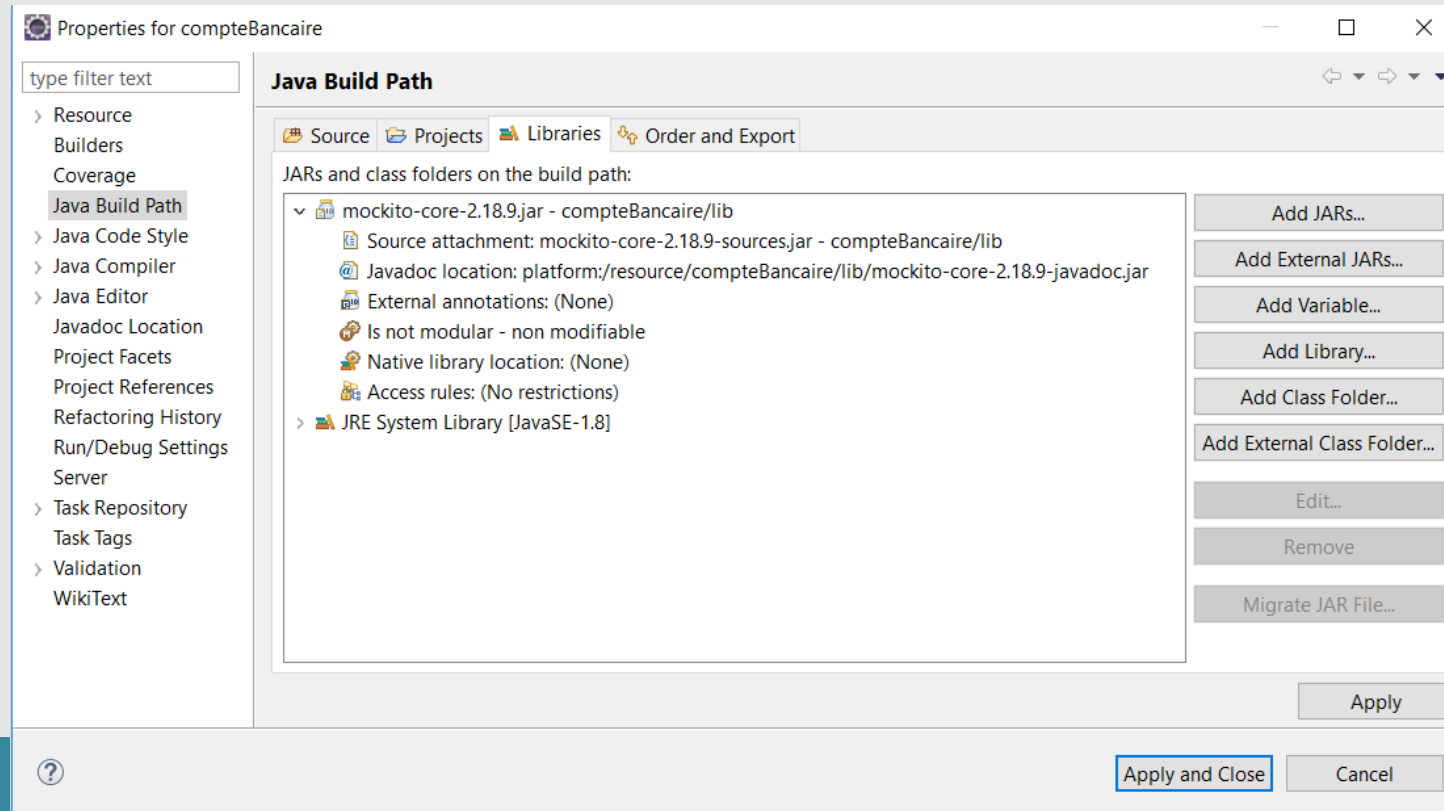
- Mockito est un framework de mocking pour écrire des tests unitaires en java
- Ce framework s'appuie sur JUnit
- <http://site.mockito.org/>

Tasty mocking framework for unit tests in Java



Installation manuelle

- Télécharger les jars depuis le site de Mockito ()
- Ajouter les jars au classpath du projet



Installation via gestionnaire de dépendance

- Gradle : Ajouter les dépendances suivantes dans le build.gradle :

```
testImplementation 'junit:junit:4.13.2'
```

```
testImplementation 'org.mockito:mockito-all:1.10.19'
```

- Maven : Ajouter la dépendance suivante dans le pom.xml :

```
<dependency>  
    <groupId>org.mockito</groupId>  
    <artifactId>mockito-all</artifactId>  
    <version>1.10.19</version>  
    <type>pom</type>  
</dependency>
```

Créer un mock

- Par annotation : @Mock
 - Initialiser les mock : classe MockitoAnnotations
 - Exemple :

```
@Mock
private UtilisateurDAO utilisateurDAO;

@Before
public void init() {
    MockitoAnnotations.initMocks(this);
}
```

- En utilisant la méthode statique mock()

```
import static org.mockito.Mockito.*;
...
utilisateurDAO = mock( UtilisateurDAO.class);
```


Définir le comportement du mock : Stubbing

- L'objectif est de définir le comportement attendu lors de l'appel d'un objet mocké
- Exemple de scénario positif : un utilisateur s'authentifie avec un login et mot de passe correct :

@Test

```
public void testAuthentifierCasReussie() throws BLLException, SQLException {  
    //Arrange  
    String user = "bob";  
    String mp = "azerty";  
    UtilisateurMger mger = new UtilisateurMger();  
    mger.setUtilisateurDAO(this.utilisateurDAO);  
    when(utilisateurDAO.authentifier(any(String.class), any(String.class))).thenReturn(true);  
  
    //Act  
    boolean authentication = mger.authentifier(user, mp);  
  
    //Assert  
    Assert.assertTrue(authentication);  
    verify(utilisateurDAO).authentifier(any(String.class), any(String.class));  
}
```

Définir le comportement du mock : Stubbing

- Exemple de scénario négatif : un utilisateur s'authentifie avec un login et mot de passe **incorrect** :

```
@Test(expected=BLLEException.class)
public void testAuthentifierCasException() throws BLLEException, SQLException {
    //Arrange
    String user = "bob";
    String mp = "azerty";
    UtilisateurMger mger = new UtilisateurMger();
    mger.setUtilisateurDAO(this.utilisateurDAO);
    when(utilisateurDAO
        .authentifier(any(String.class), any(String.class)))
        .thenThrow(BLLEException.class);

    //Act
    mger.authentifier(user, mp);
}
```

Les tests unitaires avec JUnit

Comptes bancaires

TP

