Les tests unitaires avec **JUnit 5**

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com

Plan

JUnit

JUnit ?

- Framework open source pour Java créé par Kent Beck et Erich Gamma
- Permettant d'automatiser les tests et de s'assurer que le programme répond toujours aux besoins
- Basé sur les assertions qui vérifient si les résultats de tests correspondent aux résultats attendus
- Membre de la famille XUnit (CPPUnit pour C++, CUnit pour C, PHPUnit pour PHP...)

Objectif

- Trouver un maximum de bugs pour les corriger

JUnit

TestCase (cas de test)

- Classe contenant un certain nombre de méthodes de tests
- Permettant de tester le bon fonctionnement d'une classe (en testant ses méthodes)

Remarques

- Si le test ne détecte pas d'erreur, ça ne veut pas dire qu'il n'y en a pas
- S'il détecte une erreur, il est incapable ni de la corriger ni de préciser sa source

JUnit

Étape

- Création d'un `Java Project`
- Création de deux `Package` : `org.eclipse.main` et `org.eclipse.test`
- Pour chaque classe créée dans `org.eclipse.main`, on lui associe une classe de test (dans `org.eclipse.test`)
- On prépare le test et ensuite on le lance : s'il y a une erreur, on la corrige et on relance le test.

JUnit

Création d'une première classe `Calcul`

```
package org.eclipse.main;

public class Calcul {
  public int somme(int x, int y) {
    return x + y;
  }
  public int division(int x, int y) {
    if (y == 0)
      throw new  ArithmeticException();
    return   x / y;
  }
}
```

JUnit

Pour créer une classe de test

- Faire un clic droit sur le package `org.eclipse.test`
- Aller dans `New > JUnit Test Case`
- Saisir le nom `CalculTest` dans `Name`
- Laisser cochées les 4 cases de `Which method stubs would you like to create ?`
- Cliquer sur `Browse` en face de `Class under test`
- Chercher `calcul`, sélectionner **`Calcul`** `- org.eclipse.main` et valider
- Cliquer sur `Next` puis cocher les cases correspondantes de `somme` et `division` dans `Calcul`
- Cliquer sur `Finish` puis sur `ok` (pour valider `Add JUnit 5 library to the build path` dans `Perform the following action:`)

Le code généré :

```
package org.eclipse.test;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class CalculTest {
  @BeforeAll
  static void setUpBeforeClass() throws Exception { }
  @AfterAll
  static void tearDownAfterClass() throws Exception { }
  @BeforeEach
  void setUp() throws Exception {        }
  @AfterEach
  void tearDown() throws Exception { }
  @Test
  void testSomme() {
    fail("Not yet implemented");
  }
  @Test
  void testDivision() {
    fail("Not yet implemented");
  }
}
```

JUnit

Nous parlerons de ces quatre méthodes dans une autre section

```
@BeforeAll
static void setUpBeforeClass() throws Exception {
}

@AfterAll
static void tearDownAfterClass() throws Exception {
}

@BeforeEach
void setUp() throws Exception {
}

@AfterEach
void tearDown() throws Exception {
}
```

JUnit

Pour tester

- Faire un clic droit sur le la classe de test
- Aller dans `Run As > JUnit Test`

JUnit

Pour tester

- Faire un clic droit sur le la classe de test
- Aller dans `Run As > JUnit Test`

Résultat

- 2 Exécutions : 2 Échecs : car les deux méthodes de test sont vides

JUnit

Implémentons la méthode `testSomme()` en testant chaque fois les cas particuliers

```
void testSomme() {
    Calcul calcul = new Calcul();
    if(calcul.somme(2,3)!=5)
          fail("faux pour deux entiers positifs");
    if(calcul.somme(-2,-3)!=-5)
      fail("faux pour deux entiers negatifs");
    if(calcul.somme(-2,3)!=1)
      fail("faux pour deux entiers de signe different");
    if(calcul.somme(0,3)!=3)
      fail("faux pour x nul");
    if(calcul.somme(2,0)!=2)
      fail("faux pour y nul");
    if(calcul.somme(0,0)!=0)
      fail("faux pour x et y nuls");
}
```

JUnit

Implémentons la méthode `testSomme()` en testant chaque fois les cas particuliers

```
void testSomme() {
    Calcul calcul = new Calcul();
    if(calcul.somme(2,3)!=5)
          fail("faux pour deux entiers positifs");
    if(calcul.somme(-2,-3)!=-5)
      fail("faux pour deux entiers negatifs");
    if(calcul.somme(-2,3)!=1)
      fail("faux pour deux entiers de signe different");
    if(calcul.somme(0,3)!=3)
      fail("faux pour x nul");
    if(calcul.somme(2,0)!=2)
      fail("faux pour y nul");
    if(calcul.somme(0,0)!=0)
      fail("faux pour x et y nuls");
}
```

En testant, plus d'échec pour `somme`. Implémentons donc `testDivision`.

JUnit

Nous pourrons faire des tests en utilisant d'autres méthodes qui peuvent remplacer `if ... fail`

- `assertTrue(message, condition)` : permet de vérifier que la condition fournie en paramètre est vraie
- `assertFalse(message, condition)` : permet de vérifier que la condition fournie en paramètre est fausse
- `assertEquals(message, expected, actual)` : permet de vérifier l'égalité (sa réciproque est `assertNotEquals`)
- `assertNotNull(message, object)` permet de vérifier, pour les paramètres utilisés, qu'une méthode ne retourne pas la valeur `null` (sa réciproque est `assertNull`)
- ...

JUnit

Remarques

- En l'absence d'un message explicite, un message d'erreur par défaut sera affiché.
- Les méthodes `assertX()` peuvent aussi avoir la signature suivante : `assertX(message, valeurAttendue, appelDeMéthodeATester)`

JUnit

Implémentons la méthode `testDivision()`

```
void testDivision() {
  Calcul calcul = new Calcul();
  assertFalse("2entiers positifs",calcul.division(6,3) == 0);
  assertEquals("2entiers negatifs", 2,calcul.division(-6,-3));
  assertNotNull("2 entiers de signe different",calcul.division(-6,3) );
  assertTrue("entier x nul",calcul.division(0,3) == 0);
  Throwable e = null;
  try {
    calcul.division(2,0);
  }
  catch (Throwable ex) {
    e = ex;
  }
  assertTrue(e instanceof ArithmeticException);
  e = null;
  try {
    calcul.division(0,0);
  }
  catch (Throwable ex) {
    e = ex;
  }
  assertTrue(e instanceof ArithmeticException);
}
```

JUnit

Ajouter des valeurs erronées pour avoir un échec

- Le message qui a été saisi sera affiché dans le panneau `Failure Trace` (vous pouvez cliquer sur l'icône d'écran, `Show Stack Trace in Console View`, en face `Failure Trace` pour visualiser les détails de l'erreur dans la console)
- Dans ce cas, il faut localiser l'erreur, la corriger et relancer

JUnit

Dans certains cas

- Avant de démarrer un test, il faut faire certains traitements :
  - instancier un objet de la classe,
  - se connecter à une base de données,
  - ouvrir un fichier...
- Après le test, il faut aussi fermer certaines ressources : connexion à une base de données, socket...

JUnit

Dans certains cas

- Avant de démarrer un test, il faut faire certains traitements :
  - instancier un objet de la classe,
  - se connecter à une base de données,
  - ouvrir un fichier...
- Après le test, il faut aussi fermer certaines ressources : connexion à une base de données, socket...

Pour ces cas

- On peut utiliser les méthodes `setUp()` et `tearDown()` qui sont respectivement exécutées avant et après l'appel de chaque méthode de test.

Reprenons les quatre méthodes précédentes et modifions le code

```
@BeforeAll
static void setUpBeforeClass() throws Exception {
  System.out.println("BeforeAll");
}

@AfterAll
static void tearDownAfterClass() throws Exception {
  System.out.println("AfterAll");
}

@BeforeEach
void setUp() throws Exception {
  System.out.println("BeforeEach");
}

@AfterEach
void tearDown() throws Exception {
  System.out.println("AfterEach");
}
```

JUnit

Comprenons les annotations de méthodes précédentes

- `@BeforeAll` : la méthode annotée sera exécutée seulement avant le premier test
- `@AfterAll` : la méthode annotée sera exécutée seulement après le dernier test
- `@BeforeEach` : la méthode annotée sera exécutée avant chaque test
- `@AfterEach` : la méthode annotée sera exécutée après chaque test

JUnit

Pour mieux comprendre

- Lancer le test JUnit

JUnit

Pour mieux comprendre

- Lancer le test JUnit

Le résultat

```
BeforeAll
BeforeEach
AfterEach
BeforeEach
AfterEach
AfterAll
```

Utilisons ces méthodes pour restructurer la classe `CalculTest`

```
class CalculTest {
  Calcul calcul;
  @BeforeAll
  static void setUpBeforeClass() throws Exception { }
  @AfterAll
  static void tearDownAfterClass() throws Exception { }
  @BeforeEach
  void setUp() throws Exception {
    calcul = new Calcul();
  }
  @AfterEach
  void tearDown() throws Exception {
    calcul = null;
  }
  @Test
  void testSomme() {
    // le code precedent sans l'instanciation de calcul
  }
  @Test
  void testDivision() {
    // le code precedent sans l'instanciation de calcul
  }
}
```

JUnit

Mock ?

- Objet factice (fake object)
- permettant de reproduire le comportement d'un objet réel non implémenté

JUnit

Mock ?

- Objet factice (fake object)
- permettant de reproduire le comportement d'un objet réel non implémenté

Mockito ?

- Framework open source pour Java
- Générateur automatique de doublures
- Un seul type de Mock possible et une seule façon de le créer

JUnit

Exemple, supposant qu'on

- a une interface `CalculService` ayant une méthode `carre()`
- veut développer une méthode `sommeCarre()` dans `Calcul` qui utilise la méthode `carre()` de cette interface `CalculService`

JUnit

L'interface `CalculService`

```
package org.eclipse.main;

public interface CalculService {

  public int carre(int x);

}
```

JUnit

La classe `Calcul`

```
package org.eclipse.main;

public class Calcul {

  CalculService calculService;
  public Calcul(CalculService calculService) {
    this.calculService = calculService;
  }

  public int sommeCarre(int x, int y) {
    return somme(calculService.carre(x), calculService.
      carre(y));
  }

  // + le code precedent
```

JUnit

Pour tester la classe `Calcul` dans `TestCalcul`, il faut commencer par instancier `CalculService`

```
class CalculTest {
  Calcul calcul;
  CalculService calculService;

  @BeforeEach
  void setUp() throws Exception {
    calcul = new Calcul(calculService);
  }

  @Test
  void testSommeCarre() {
    assertTrue("calcul␣exact",calcul.sommeCarre(2,3) ==
       13);
  }
  // + le code precedent
```

JUnit

En testant, on aura l'erreur suivante

```
java.lang.NullPointerException
    at org.eclipse.main.Calcul.sommeCarre(Calcul.java:6)
    at org.eclipse.test.CalculTest.testSommeCarre(CalculTest.
       java:26)
```

JUnit

En testant, on aura l'erreur suivante

```
java.lang.NullPointerException
    at org.eclipse.main.Calcul.sommeCarre(Calcul.java:6)
    at org.eclipse.test.CalculTest.testSommeCarre(CalculTest.
       java:26)
```

Explication

- La source de l'erreur est l'appel de la méthode `carre(x)` qui n'est pas implémenté.
- Pour corriger cette erreur, on peut utiliser les STUB
- STUB (les bouchons en français) : classes qui renvoient en dur une valeur pour une méthode invoquée

JUnit

Pour tester la classe `Calcul` dans `TestCalcul`, il faut commencer par instancier `CalculService`

```
class CalculTest {
  Calcul calcul;

  CalculService calculService =  new CalculService()
      {

    @Override
    public int carre(int x) {
      // TODO Auto-generated method stub
      return x*x;
    }
  };
  // + le code precedent
```

JUnit

En testant

- tout se passe bien et le test est passé.

JUnit

En testant

- tout se passe bien et le test est passé.

On peut utiliser les mocks

- pour créer un objet factice de `CalculService`

JUnit

Démarche

- Aller dans `https://jar-download.com/artifacts/org.mockito` et télécharger `mockito-core`
- Créer un répertoire `lib` à la racine du projet
- Décompresser l'archive `mockito-core` et copier les 4 `.jar` dans `lib`
- Ajouter les `.jar` de `lib` au `path` du projet

JUnit

Commençons par importer `mockito` dans la classe `CalculTest`

```
import static org.mockito.Mockito.*;
```

JUnit

Commençons par importer `mockito` dans la classe `CalculTest`

```
import static org.mockito.Mockito.*;
```

Remplaçons l'instanciation de `CalculService` par un `mock`

```
CalculService calculService = mock(CalculService.class);
```

JUnit

Commençons par importer `mockito` dans la classe `CalculTest`

```
import static org.mockito.Mockito.*;
```

Remplaçons l'instanciation de `CalculService` par un `mock`

```
CalculService calculService = mock(CalculService.class);
```

La méthode `testSommeCarre()` devient ainsi

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue("calcul␣exact",calcul.sommeCarre(2,3) == 13)
     ;
}
```

JUnit

En testant

- tout se passe bien et le test est passé.

JUnit

En testant

- tout se passe bien et le test est passé.

Pour vérifier que notre mock a bien été appelé

- on peut utiliser la méthode `verify`

JUnit

En testant

- tout se passe bien et le test est passé.

Pour vérifier que notre mock a bien été appelé

- on peut utiliser la méthode `verify`

Ajoutons la méthode `verify` à `testSommeCarre()`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue("calcul␣exact",calcul.sommeCarre(2,3) ==13);
  verify(calculService).carre(2);
}
```

JUnit

Et si on n'appelle plus la méthode `carre()` de `CalculService` dans `Calcul`

```
public int sommeCarre(int x, int y) {
  return somme(x*x, y*y);
}
```

JUnit

Et si on n'appelle plus la méthode `carre()` de `CalculService` dans `Calcul`

```
public int sommeCarre(int x, int y) {
  return somme(x*x, y*y);
}
```

Testons de nouveau `testSommeCarre()`

```
@Test
void testSommeCarre() {
  when(calculService.carre(2)).thenReturn(4);
  when(calculService.carre(3)).thenReturn(9);
  assertTrue("calcul␣exact",calcul.sommeCarre(2,3) ==13);
  verify(calculService).carre(2);
}
```

JUnit

Le résultat est

```
Wanted but not invoked:
calculService.carre(2);
-> at org.eclipse.test.CalculTest.testSommeCarre(
   CalculTest.java:39)
Actually, there were zero interactions with this mock.
```

JUnit

Le résultat est

```
Wanted but not invoked:
calculService.carre(2);
-> at org.eclipse.test.CalculTest.testSommeCarre(
   CalculTest.java:39)
Actually, there were zero interactions with this mock.
```

Explication

- tout mock créé doit être invoqué.

JUnit

On peut aussi utiliser les annotations

```
class CalculTest {
  Calcul calcul;
  @Mock
  CalculService calculService;

  @Rule
  MockitoRule rule = MockitoJUnit.rule();

  @BeforeEach
  void setUp() throws Exception {
    MockitoAnnotations.initMocks(this);
    calcul = new Calcul(calculService);
  }
  @Test
  void testSommeCarre() {
    when(calculService.carre(2)).thenReturn(4);
    when(calculService.carre(3)).thenReturn(9);
    assertTrue("calcul␣exact",calcul.sommeCarre(2,3) == 13);
    verify(calculService).carre(2);
  }
  // + les autres tests
```

JUnit

Commençons par créer un `Java Project` avec `Maven`

- Aller dans `File > New > Other`
- Chercher puis sélectionner `Maven Project`
- Cliquer sur `Next`
- Choisir `maven-archetype-quickstart`
- Remplir les champs
  - `group Id` avec `org.eclipse`
  - `artifact Id` avec `FirstMavenJunit`

JUnit

Vérifier l'existence des deux répertoires

- `/src/main/java` : code source
- `/src/test/java` : code source de test
- ...

S'il n'y a pas de `src/main/java` ou `src/test/java`

- Faire clic droit sur le nom du projet
- Aller dans `Build Path > Configure Build Path...`
- Cliquer sur `Order and Export`
- Cocher les trois case `Maven Dependencies`, `Apache Tomcat vX.X` et `JRE System Library`
- Cliquer sur `Apply and Close`

JUnit

Ajouter les dépendances suivantes dans `pom.xml` dans la section `dependencies`

```
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>1.2.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.3.1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>2.23.0</version>
    <scope>test</scope>
</dependency>
```

JUnit

Ajouter cette section dans `pom.xml` **avant la section** `dependencies`

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.19.1</version>
      <dependencies>
        <dependency>
          <groupId>org.junit.platform</groupId>
          <artifactId>junit-platform-surefire-provider</
            artifactId>
          <version>1.0.1</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
</build>
```

JUnit

Pour tester

- Supprimer les packages générés
- Copier les deux packages `org.eclipse.main` et `org.eclipse.test` dans `src/main/java` et `src/test/java`
- Annoter les classes de test par `@RunWith(JUnitPlatform.class)`

JUnit

Pour tester

- Supprimer les packages générés
- Copier les deux packages `org.eclipse.main` et `org.eclipse.test` dans `src/main/java` et `src/test/java`
- Annoter les classes de test par `@RunWith(JUnitPlatform.class)`

La classe `CalculTest`

```
@DisplayName("Test de la classe Calcul")
@RunWith(JUnitPlatform.class)
class CalculTest {
  // + tout le code precedent
```

JUnit

Pour désactiver un test, on utilise la notation `@Disabled` : le test faux même s'il échoue

```
@Disabled
@Test
void testSomme() {
  if(calcul.somme(2,3)!=6)
    fail("faux pour deux entiers positifs");
}
```

JUnit

Utiliser `@RepeatedTest` pour répéter un test plusieurs fois

```
@RepeatedTest(3)
void testSomme(RepetitionInfo repetitionInfo) {
  assertNotEquals(7, calcul.somme(repetitionInfo.
    getCurrentRepetition(),3));
}
```

JUnit

Utiliser `@RepeatedTest` pour répéter un test plusieurs fois

```
@RepeatedTest(3)
void testSomme(RepetitionInfo repetitionInfo) {
  assertNotEquals(7, calcul.somme(repetitionInfo.
    getCurrentRepetition(),3));
}
```

Explication

- Il faut remplacer `@Test` par `@RepeatedTest`
- Pour récupérer l'index de l'itération courante, on déclare un objet de type `RepetitionInfo`

JUnit

Utiliser `@DisplayName` pour utiliser un nom d'affichage personnalisé pour la classe de test ou la méthode de test (peut donc contenir des espaces et des caractères spéciaux).

```
@DisplayName("Test de la classe Calcul")
class CalculTest {
    ...
  @RepeatedTest(3)
  @DisplayName("Trois tests de la methode somme")
  void testSomme(RepetitionInfo repetitionInfo) {
  ...
```

JUnit

Utiliser `@ParameterizedTest` pour paramétrer un test

```
@DisplayName("Test de la methode somme")
@ParameterizedTest
@ValueSource(ints = {2,3})
      void testSomme(int t) {
        assertNotEquals(5, calcul.somme(t,1));
}
```

JUnit

Utiliser `@ParameterizedTest` pour paramétrer un test

```
@DisplayName("Test de la methode somme")
@ParameterizedTest
@ValueSource(ints = {2,3})
      void testSomme(int t) {
        assertNotEquals(5, calcul.somme(t,1));
}
```

Explication

- Il faut remplacer `@Test` par `@ParameterizedTest`
- Cette méthode sera testée deux fois, une fois pour la valeur `2` et une fois pour la valeur `3`
- `@ValueSource` indique les valeurs à injecter dans `t` lors de chaque appel
- Il existe aussi `strings`, `longs` et `doubles`

JUnit

Pour utiliser l'annotation `@ParameterizedTest`, il faut ajouter la dépendance suivante :

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.3.1</version>
    <scope>test</scope>
</dependency>
```

JUnit

Pour activer ou désactiver un test selon le système d'exploitation, on utilise soit `@DisabledOnOs` soit `@EnabledOnOs`

```
@Test
@DisabledOnOs(MAC)
void testSommeCarre() {
  when(calculService.Carre(2)).thenReturn(4);
  when(calculService.Carre(3)).thenReturn(9);
  assertTrue("calcul␣exact",calcul.sommeCarre(2,3) == 13)
     ;
  verify(calculService).Carre(2);
}
```

JUnit

Pour activer ou désactiver un test selon le système d'exploitation, on utilise soit `@DisabledOnOs` soit `@EnabledOnOs`

```
@Test
@DisabledOnOs(MAC)
void testSommeCarre() {
  when(calculService.Carre(2)).thenReturn(4);
  when(calculService.Carre(3)).thenReturn(9);
  assertTrue("calcul␣exact",calcul.sommeCarre(2,3) == 13)
     ;
  verify(calculService).Carre(2);
}
```

Il faut importer la constante `MAC`

```
import static org.junit.jupiter.api.condition.OS.MAC;
```

JUnit

Pour activer ou désactiver un test selon le système d'exploitation, on utilise soit `@DisabledOnOs` **soit** `@EnabledOnOs`

```
@Test
@DisabledOnOs(MAC)
void testSommeCarre() {
  when(calculService.Carre(2)).thenReturn(4);
  when(calculService.Carre(3)).thenReturn(9);
  assertTrue("calcul␣exact",calcul.sommeCarre(2,3) == 13)
     ;
  verify(calculService).Carre(2);
}
```

Il faut importer la constante `MAC`

```
import static org.junit.jupiter.api.condition.OS.MAC;
```

Autres constantes possibles : `LINUX`, `WINDOWS`

JUnit

Pour activer ou désactiver un test selon la version de `JRE`, on utilise soit `@DisabledOnJre` soit `@EnabledOnJre`

```
@Test
@DisabledOnOs(MAC)
@EnabledOnJre(JAVA_10)
void testSommeCarre() {
  when(calculService.Carre(2)).thenReturn(4);
  when(calculService.Carre(3)).thenReturn(9);
  assertTrue("calcul␣exact",calcul.sommeCarre(2,3) == 13);
  verify(calculService).Carre(2);
}
```

JUnit

Pour activer ou désactiver un test selon la version de `JRE`, on utilise soit `@DisabledOnJre` soit `@EnabledOnJre`

```
@Test
@DisabledOnOs(MAC)
@EnabledOnJre(JAVA_10)
void testSommeCarre() {
  when(calculService.Carre(2)).thenReturn(4);
  when(calculService.Carre(3)).thenReturn(9);
  assertTrue("calcul␣exact",calcul.sommeCarre(2,3) == 13);
  verify(calculService).Carre(2);
}
```

Il faut importer la constante `JAVA_10`

```
import static org.junit.jupiter.api.condition.JRE.JAVA_10;
```

JUnit

Pour activer ou désactiver un test selon la version de `JRE`, on utilise soit `@DisabledOnJre` soit `@EnabledOnJre`

```
@Test
@DisabledOnOs(MAC)
@EnabledOnJre(JAVA_10)
void testSommeCarre() {
  when(calculService.Carre(2)).thenReturn(4);
  when(calculService.Carre(3)).thenReturn(9);
  assertTrue("calcul␣exact",calcul.sommeCarre(2,3) == 13);
  verify(calculService).Carre(2);
}
```

Il faut importer la constante `JAVA_10`

```
import static org.junit.jupiter.api.condition.JRE.JAVA_10;
```

Autres constantes possibles : `JAVA_8`, `JAVA_9`

JUnit

Pour activer ou désactiver un test selon la valeur d'un test logique (on peut même utiliser une expression régulière), on utilise soit `@DisabledIf` soit `@EnabledIf`

```
@Test
@DisabledOnOs(MAC)
@DisabledIf("2 * 3 > 4")
void testSommeCarre() {
  when(calculService.Carre(2)).thenReturn(4);
  when(calculService.Carre(3)).thenReturn(9);
  assertTrue("calcul exact",calcul.sommeCarre(2,3)
    == 13);
  verify(calculService).Carre(2);
}
```

JUnit

Objectif

- On veut vérifier si nos tests couvrent l'intégralité de notre code
- Ou s'il y a des zones inaccessibles dans notre code

JUnit

Objectif

- On veut vérifier si nos tests couvrent l'intégralité de notre code
- Ou s'il y a des zones inaccessibles dans notre code

Pour cela, il faut installer un plugin **Eclipse** `EclEmma`

- Aller dans `Help > Install New Software > Add`
- Saisir `EclEmma` dans `Name` et `http://update.eclemma.org/` dans `Location`
- Cocher la case `EclEmma` et cliquer sur `Next`
- Terminer l'installation et redémarrer **Eclipse**

JUnit

Pour tester le recouvrement du code

- Aller dans `Window > Show View > Other ...`
- Saisir `Coverage` et valider
- Aller dans `Run`
- Cliquer sur `Coverage` et vérifier si votre code est bien couvert par les tests