

# Les Tests en Génie Logiciel

---

Préparé par : Dr. Lakhrouit jihane

E-mail: [jihane.Lakhrouit@gmail.com](mailto:jihane.Lakhrouit@gmail.com)

- ▶ Erreur, défaut et défaillance
- ▶ Test boîte noir
- ▶ Test boîte blanche
- ▶ Test boîte grise
- ▶ Complémentarité test BN et BB
- ▶ Les Types de Test
- ▶ Les niveaux de Test
- ▶ Les Tests Unitaires

# Les Tests Logiciels

- ▶ Le test logiciel est le processus qui consiste à évaluer et à vérifier qu'un logiciel fait ce qu'il est censé faire sans bug

# Erreur, défaut et défaillance

- ▶ L'Erreur est humaine et est l'origine des défauts
- ▶ Les défauts sont des imperfections dans le logiciel. Ces défauts peuvent engendrer des défaillances mais ce n'est pas nécessaire.
- ▶ Les défaillances sont des événements au cours desquels le logiciel ne fait pas ce qu'il devrait. Pour simplifier, les défaillances sont ce que nous appelons couramment Bug ou Anomalie



# Erreur, défaut et défaillance

- ▶ Le programme suivant contient une erreur.
  - ▶ Ni défaut ni défaillance

```
public boolean fctTest(boolean a, int b){  
    int res;  
    if(a)  
        res = b;  
    else  
        res = b+10; //ici res doit recevoir b+20  
  
    return (res>30);  
}
```

## Ni défaut Ni défaillance

a = true et b = 50

Le résultat retourné est : true

Le résultat attendu est : true

# Erreur, défaut et défaillance

- ▶ Exemple
- ▶ Le programme suivant contient une erreur. Trouver un exemple d'exécution où l'erreur engendre :
  - ▶ Un défaut mais aucune défaillance

```
public boolean fctTest(boolean a, int b){  
    int res;  
    if(a)  
        res = b;  
    else  
        res = b+10; //ici res doit recevoir b+20  
  
    return (res>30);  
}
```

**Un défaut aucune défaillance**

a = false et b = 25

Le résultat retourné est : true

Le résultat attendu est : true

# Erreur, défaut et défaillance

- ▶ Exemple
- ▶ Le programme suivant contient une erreur. Trouver un exemple d'exécution où l'erreur engendre :
  - ▶ Un défaut et une défaillance

```
public boolean fctTest(boolean a, int b){  
    int res;  
    if(a)  
        res = b;  
    else  
        res = b+10; //ici res doit recevoir b+20  
  
    return (res>30);  
}
```

## Un défaut et une défaillance

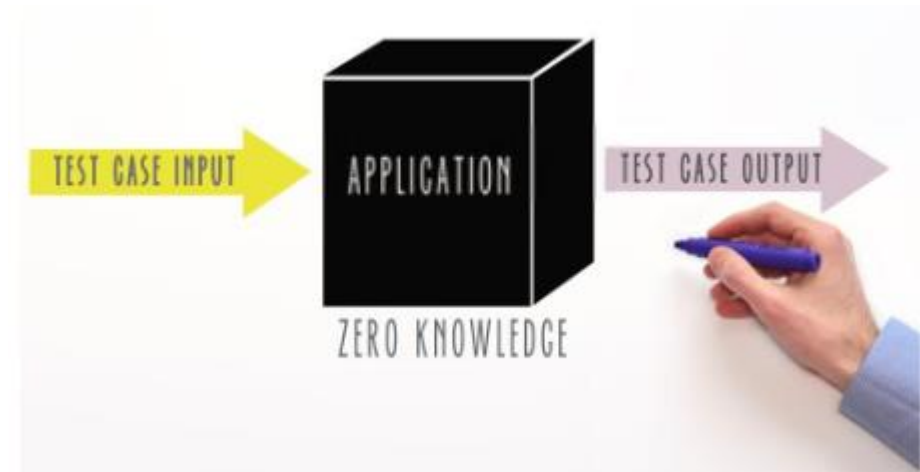
a = false et b = 11

Le résultat retourné est : true

Le résultat attendu est : false

# Test boîte noire

- ▶ Les tests en « boîte noire » consistent à examiner uniquement les fonctionnalités d'une application, c'est-à-dire si elle fait ce qu'elle est censée faire, peu importe comment elle le fait.
- ▶ Le testeur ignore les mécanismes internes. Il a un profil uniquement « utilisateur ».





# Test boîte noir – Exemple 1

- ▶ Un programme qui accepte en entrée les nombre de 01 à 10 et de 20 à 30
- ▶ Les tests doivent couvrir toutes les possibilités



Entrée	Résultat attendu	Résultat trouvé
-2	false	false
3	true	true
15	false	false
25	true	true
45	false	false

# Test boîte noir – Exemple 2

- ▶ Un formulaire qui contient trois champs obligatoires : Nom, Email et Message et un bouton « Envoyer »
- ▶ Le bouton Envoyer n'est actif que lorsque si tous les champs sont renseignés.
- ▶ Quelles sont toutes les possibilités de test ?

F : False  
T: True

Entrée	Test1	Test2	Test3	Test4	Test5	Test6	Test7	test8
Nom	F	T	F	T	F	T	F	T
Email	F	F	T	T	F	F	T	T
Message	F	F	F	F	T	T	T	T
Sortie								
Envoyer	F	F	F	F	F	F	F	T

# Exercice Boite Noir – Exemple 3

11

Écrire une fonction qui prend en entrée une chaîne de caractères et détermine si elle est un palindrome ou non.

Un palindrome est une chaîne qui se lit de la même manière de gauche à droite et de droite à gauche.

**Exemple :**

- Entrée : "radar"
- Sortie attendue : true

# Exercice Boite Noir – Exemple 3

12

**1. Test de cas de base** : Testez la fonction avec un palindrome.

- Entrée : "radar"
- Sortie attendue : true

**2. Test de cas avec un palindrome plus long** : Testez la fonction avec un palindrome plus long.

- Entrée : "racecar"
- Sortie attendue : true

**3. Test de cas avec une chaîne non palindrome** : Testez la fonction avec une chaîne qui n'est pas un palindrome.

- Entrée : "hello"
- Sortie attendue : false

**4. Test de cas avec une chaîne vide** : Testez la fonction avec une chaîne vide.

- Entrée : ""
- Sortie attendue : true (une chaîne vide est considérée comme un palindrome)

**5. Test de cas avec des caractères spéciaux** : Testez la fonction avec une chaîne contenant des caractères spéciaux.

- Entrée : "A man, a plan, a canal, Panama!"
- Sortie attendue : true

**6. Test de performance** : Testez la fonction avec une chaîne très longue pour vérifier si elle s'exécute dans un temps raisonnable.

- Entrée : Une chaîne très longue qui est un palindrome.
- Vérification : Vérifiez si la fonction se termine rapidement sans provoquer de débordement de mémoire ou de temps d'exécution excessif.

# Exercice Boite Noir – Exemple 4

- ▶ Un programme prend en entrée trois flottants, interprété comme les longueurs des côtés d'un triangle.

Le programme répond s'il s'agit d'un triangle scalène, isocèle ou équilatéral.

- ▶ On suppose que l'entête de la fonction est:

`TypeTriangle myProgramme(float a, float b, float c)`

où `TypeTriangle` est un type énuméré pouvant valoir `SCAL`, `ISOC`, `EQUIL`.

- ▶ Produire des scénarios de test pour ce programme.

# Correction

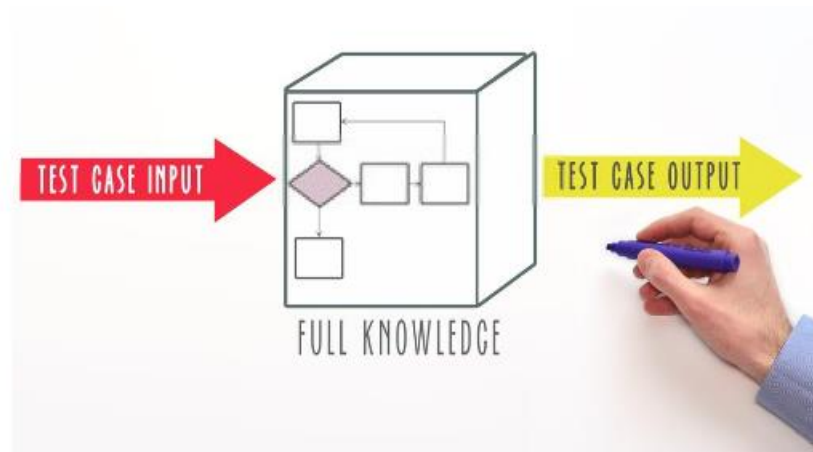
conditions											
c1: $a < b + c$ ?	F	T	T	T	T	T	T	T	T	T	T
c2: $b < a + c$ ?	-	F	T	T	T	T	T	T	T	T	T
c3: $c < a + b$ ?	-	-	F	T	T	T	T	T	T	T	T
c4: $a = b$ ?	-	-	-	T	T	T	T	F	F	F	F
c5: $a = c$ ?	-	-	-	T	T	F	F	T	T	F	F
c6: $b = c$ ?	-	-	-	T	F	T	F	T	F	T	F
a1: Pas un triangle	X	X	X								
a2: Scalène											X
a3: Isocèles							X		X	X	
a4: Équilatéral				X							
a5: Impossible					X	X		X			

# Test boîte blanche

- ▶ Les tests en « boîte blanche » consistent à examiner le fonctionnement d'une application et sa structure interne, ses processus, plutôt que ses fonctionnalités.
- ▶ Le testeur doit avoir une vue globale du fonctionnement de l'application, des éléments qui la composent, et naturellement de son code source.
- ▶ Le testeur ici a un profil développeur

# Test boîte blanche

- Pour réaliser un test en « boîte blanche », ce dernier doit donc avoir des compétences de programmation, afin de comprendre le code qu'il étudie.
- Il doit également avoir une vue globale du fonctionnement de l'application, des éléments qui la composent, et naturellement de son code source. Contrairement aux tests en « boîte noire », le testeur ici a un profil développeur, et non pas utilisateur.





# Test boîte blanche – Exemple -1

- Un programme qui vérifie si l'adresse mail contient un « . » et un « @ »

```
public boolean isValidEmail( String email ){  
    if(email.contains(".") && email.contains("@")) return true;  
    else return false;  
}
```

## Cas de test:

usergmailcom --> false

user@gmailcom --> false

usergmail.com --> false

user@gmail.com --> true

user.name@gmail --> true

## Test boîte blanche – Exemple -2

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

- Pour tester cette classe en utilisant le concept de test boîte blanche, nous pouvons écrire des cas de test pour vérifier si la méthode add fonctionne correctement en fournissant différentes entrées et en vérifiant les sorties correspondantes.

# Test boîte blanche – Exemple -2

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

- ▶ Dans cet exemple, nous avons écrit trois cas de test pour la méthode add de la classe Calculator. Pour chaque cas de test, nous appelons la méthode add avec des valeurs différentes et vérifions si le résultat retourné est celui attendu à l'aide de la méthode assertEquals de JUnit.
- ▶ Vous pouvez ajouter d'autres cas de test selon les différents scénarios que vous souhaitez tester pour vous assurer que la méthode add fonctionne correctement dans toutes les situations prévues.

# Test boîte blanche – Exemple -2

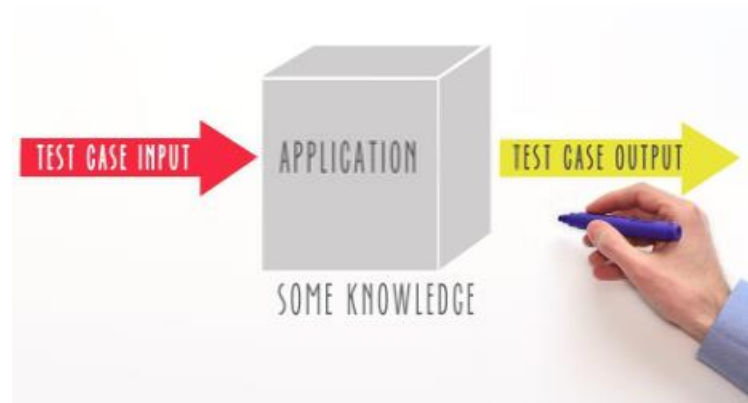
20

```
public class CalculatorTest {  
  
    @Test  
    public void testAdd() {  
        Calculator calculator = new Calculator();  
  
        // Test case 1  
        int result1 = calculator.add(3, 5);  
        assertEquals(8, result1);  
  
        // Test case 2  
        int result2 = calculator.add(-1, 1);  
        assertEquals(0, result2);  
  
        // Test case 3  
        int result3 = calculator.add(0, 0);  
        assertEquals(0, result3);  
  
        // Ajoutez d'autres cas de test selon les besoins  
    }  
}
```

- ▶ Dans cet exemple, nous avons écrit trois cas de test pour la méthode add de la classe Calculator. Pour chaque cas de test, nous appelons la méthode add avec des valeurs différentes et vérifions si le résultat retourné est celui attendu à l'aide de la méthode assertEquals de JUnit.
- ▶ Vous pouvez ajouter d'autres cas de test selon les différents scénarios que vous souhaitez tester pour vous assurer que la méthode add fonctionne correctement dans toutes les situations prévues.

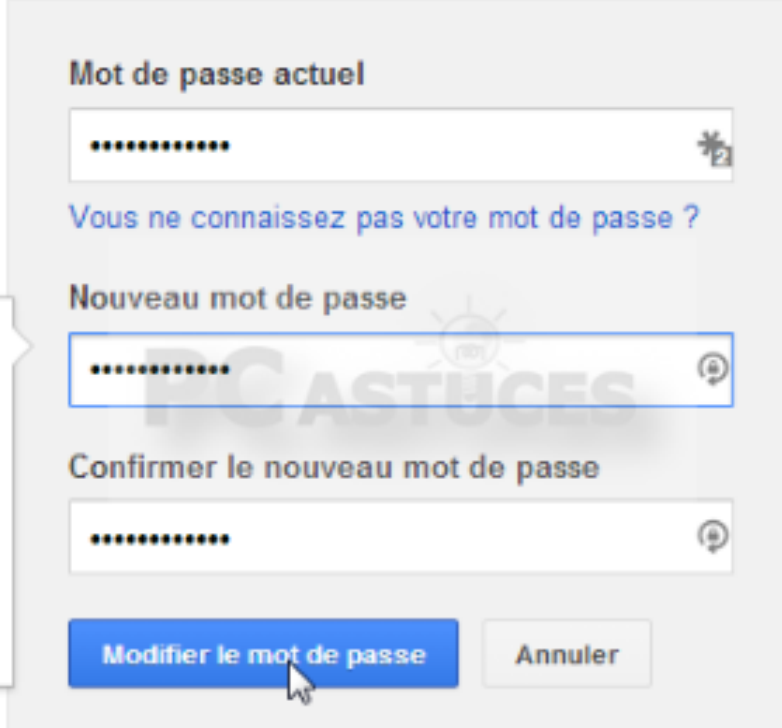
# Test boîte grise

- ▶ Les tests en « boîte grise » compilent ces deux précédentes approches :
- ▶ ils éprouvent à la fois les fonctionnalités et le fonctionnement d'un système.
- ▶ Le testeur connaît le rôle du système et de ses fonctionnalités cependant, il n'a pas accès au code source !



# Test boîte grise - Exemple

- ▶ Changement de mot de passe.
- ▶ L'utilisateur saisit le nouveau mot de passe sur le front-end
- ▶ vérifie si le nouveau mot de passe est sauvegardé en base de données.
- ▶ Plusieurs scénarios de test sont possibles.
- ▶ Le testeur n'a pas accès au code source!



Mot de passe actuel

.....

[Vous ne connaissez pas votre mot de passe ?](#)

Nouveau mot de passe

.....

Confirmer le nouveau mot de passe

.....

**Modifier le mot de passe** Annuler

The image shows a web form for changing a password. It has three text input fields: 'Mot de passe actuel', 'Nouveau mot de passe', and 'Confirmer le nouveau mot de passe'. Each field contains a series of dots representing masked text. There is a link 'Vous ne connaissez pas votre mot de passe ?' below the first field. At the bottom, there are two buttons: 'Modifier le mot de passe' (highlighted in blue) and 'Annuler' (grey). A mouse cursor is pointing at the 'Modifier le mot de passe' button. A large, semi-transparent watermark 'PCASTUCES' is visible across the middle of the form.

- ▶ Une analogie est souvent utilisée pour différencier ces techniques, en comparant le système testé à une voiture.
- ▶ **En méthode « boîte noire »**, on vérifie que la voiture fonctionne en allumant les lumières, en klaxonnant et en tournant la clé pour que le moteur s'allume. Si tout se passe comme prévu, la voiture fonctionne.
- ▶ **En méthode « boîte blanche »**, on emmène la voiture chez le garagiste, qui regarde le moteur ainsi que toutes les autres parties (mécaniques comme électriques) de la voiture. Si elle est en bon état, elle fonctionne.
- ▶ **En méthode « boîte grise »**, on emmène la voiture chez le garagiste, et en tournant la clé dans la serrure, on vérifie que le moteur s'allume, et le garagiste observe en même temps le moteur pour s'assurer qu'il démarre bien selon le bon processus.

# Complémentarité test BN et BB

- ▶ Test boîte noire :
  - ▶ +test choisi à partir des spécifs (use-case, FSM, etc.)
  - ▶ spécifs parfois peu précises
- ▶ Test boîte blanche :
  - ▶ +test choisi à partir du code source
  - ▶ + description précise
- ▶ +test choisi à partir du code source
  - ▶ + description précise
- ▶ Les deux familles de tests sont complémentaires
  - ▶ BN : trouvent plus facilement les erreurs d'omission et de spécification
  - ▶ BB : trouvent plus facilement les défauts de programmation

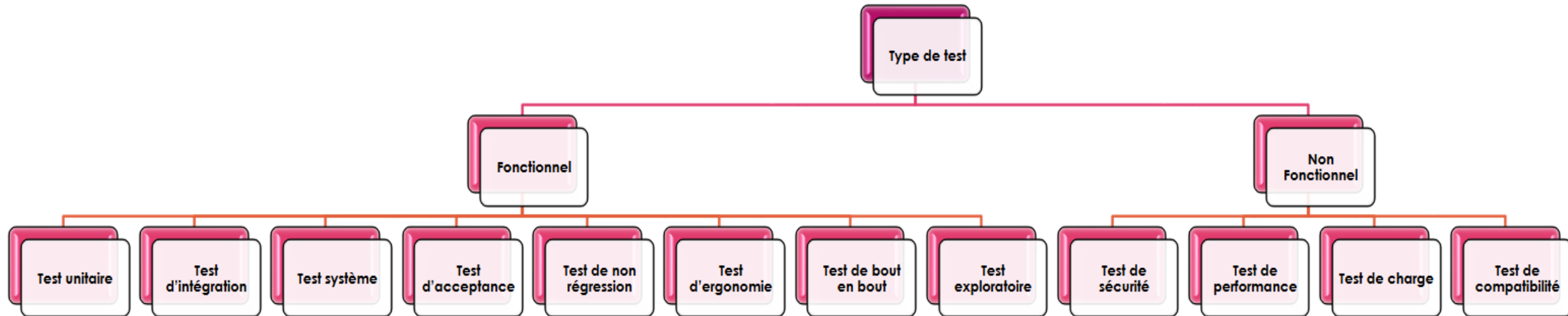


# Complémentarité test BN et BB

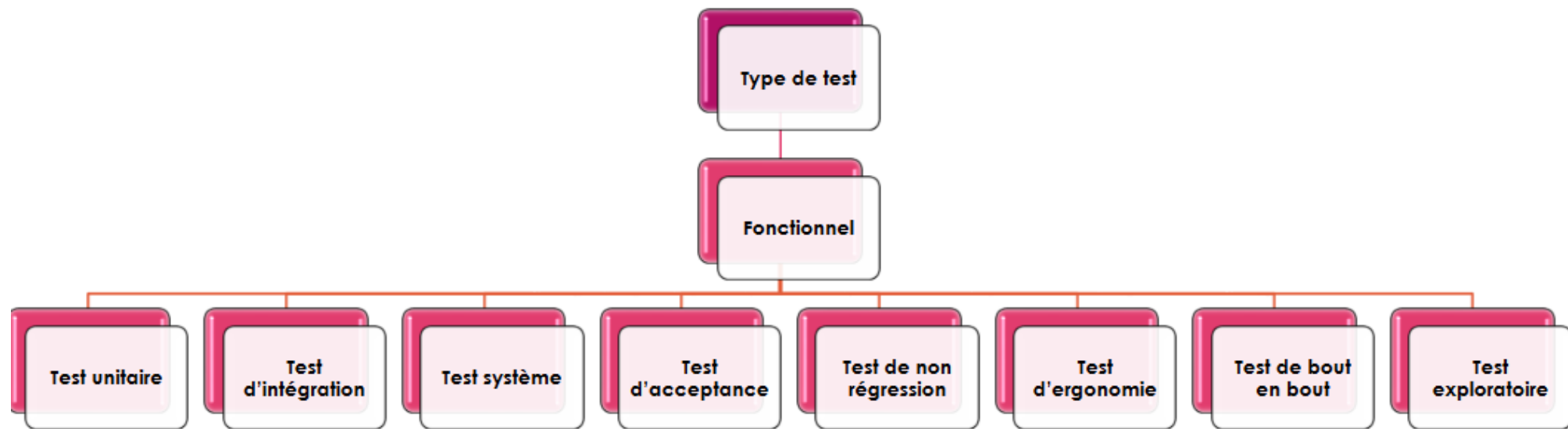
- ▶ Exemple : Soit le programme suivant censé calculer la somme de deux entiers :
- ▶ Une approche fonctionnelle détectera difficilement le défaut alors qu'une approche par analyse de code pourra produire la DT :  $x = 600, y = 500$

```
Function sum(x,y : integer) : integer;  
begin  
  if (x = 600) and (y = 500) then sum:= x-y  
  Else sum:= x+y;  
end
```

# Les Types de Tests



# Les Types de Tests



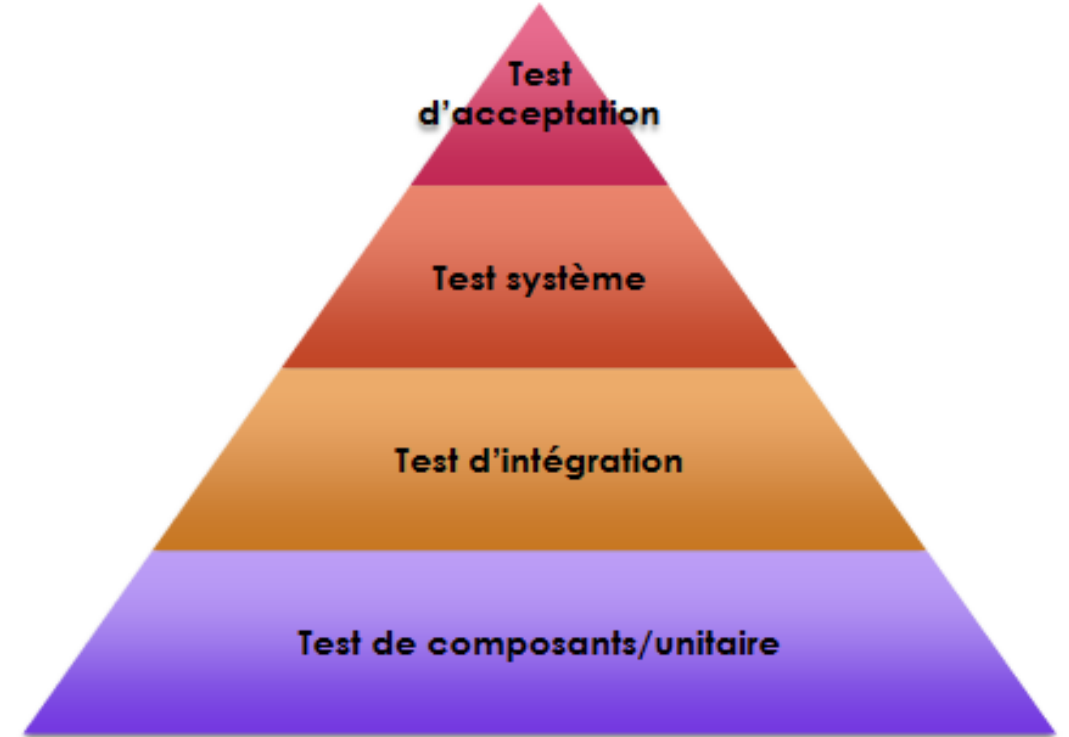
## Tests

Essayer le logiciel sur des données d'exemple pour s'assurer qu'il fonctionne correctement :

- Tests unitaires : faire tester les parties du logiciel par leurs développeurs
- Tests d'intégration : tester pendant l'intégration
- Tests de validation : pour acceptation par l'acheteur
- Tests système : tester dans un environnement proche de l'environnement d'utilisation.
- Tests alpha : faire tester par le client sur le site de développement
- Tests bêta : faire tester par le client sur le site d'exécution
- Tests de régression : enregistrer les résultats des tests et les comparer à ceux des anciennes versions pour vérifier si la nouvelle n'en a pas dégradé d'autres

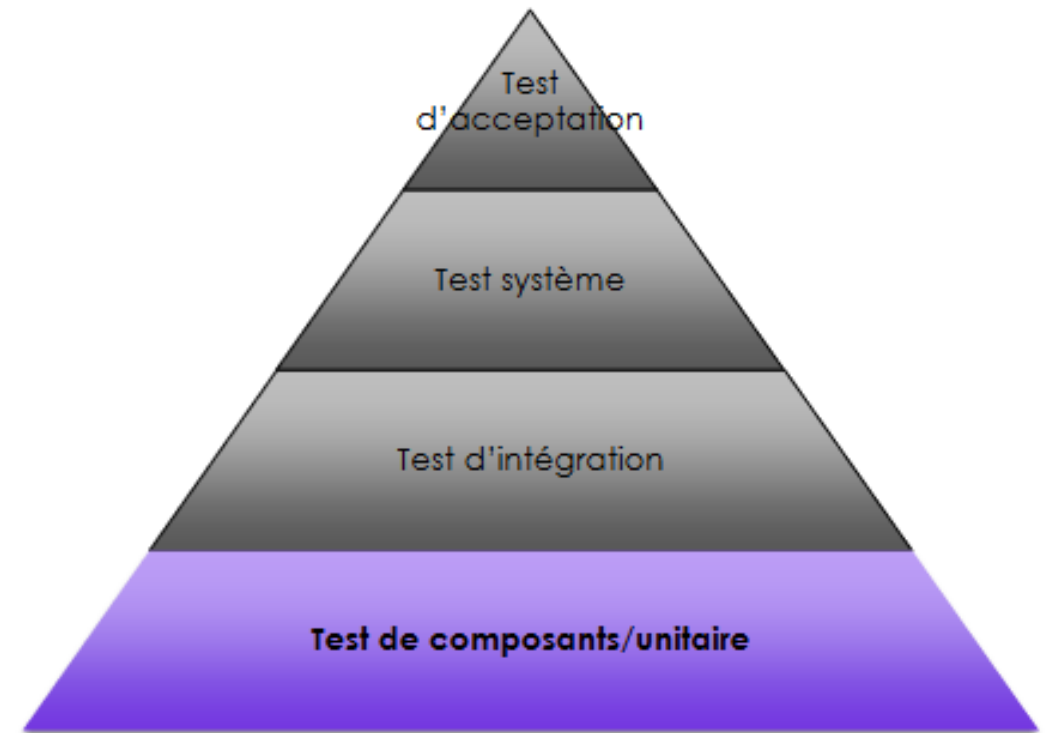
# Les niveaux de Test

- ▶ Les niveaux de test
- ▶ On définit quatre niveaux de tests permettant d'avoir une vision plus claire du test dans son ensemble.
- ▶ Chaque niveau de test a sa place et doit être effectué avec rigueur



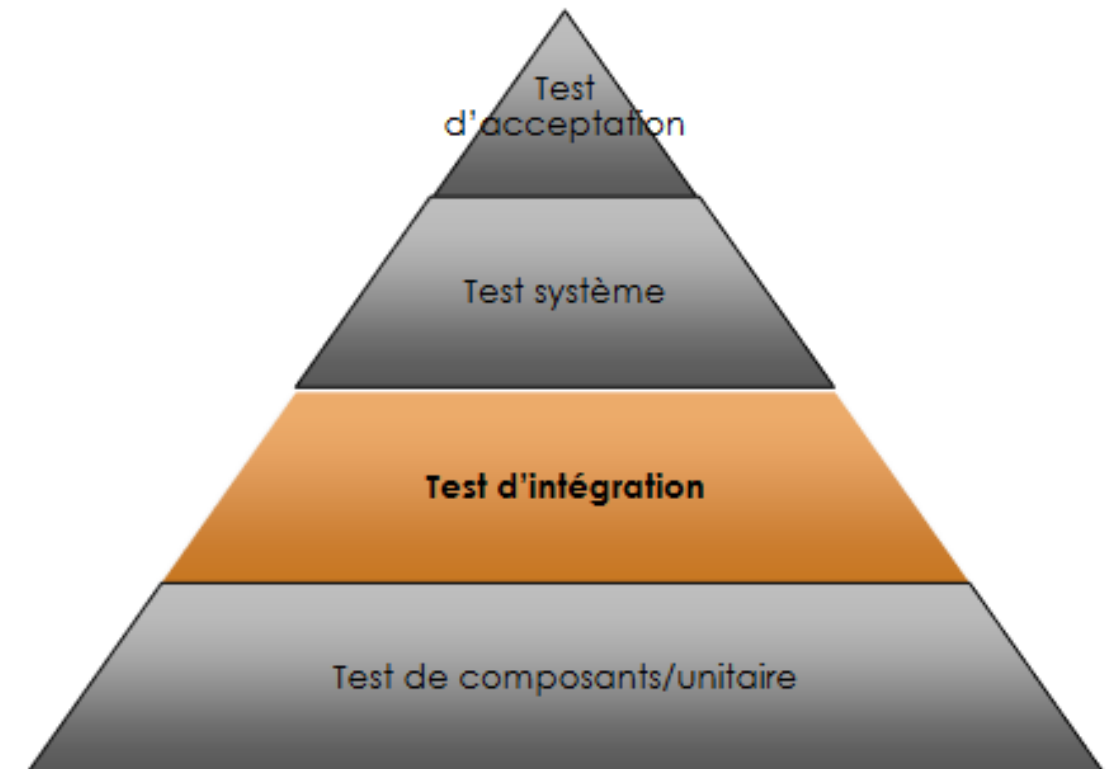
# Les niveaux de Test

- ▶ Test de composant / unitaire
- ▶ A pour but de tester les différents composants du logiciel séparément afin de s'assurer que chaque élément fonctionne comme spécifié.
- ▶ Ces tests sont écrits et exécutés par le développeur qui a écrit le code du composant.
- ▶ Exemple : Pour une authentification, le bouton « se connecter » est un composant.



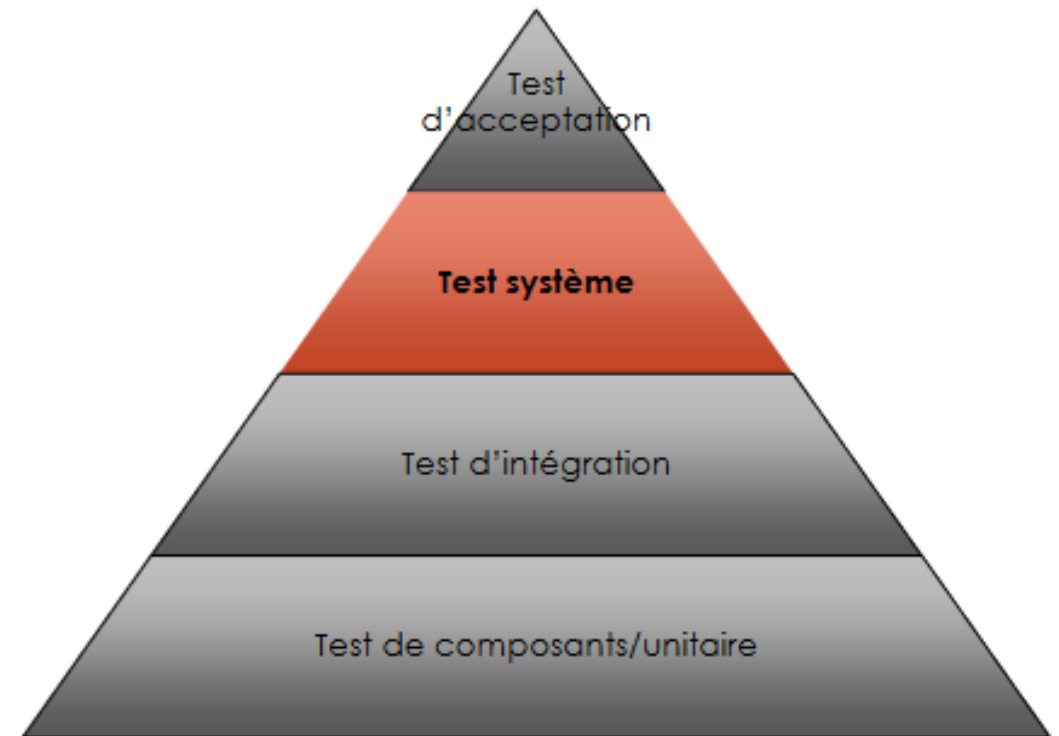
# Les niveaux de Test

- ▶ Test d'intégration
- ▶ Les tests d'intégrations sont des tests effectués entre les composants afin de s'assurer du fonctionnement des interactions et de l'interface entre les différents composants.
- ▶ **Ces tests sont également gérés, en général, par des développeurs.**
- ▶ Exemple : Pour une authentification on vérifie que le message envoyé après l'appui sur le bouton « se connecter » est bien reçu par le serveur d'authentification.



# Les niveaux de Test

- ▶ Test système
- ▶ **C'est les tests qui sont effectués par les ingénieurs de tests(Responsable fonctionnel).**
- ▶ Leurs but est de vérifier que le système répond aux exigences définies dans les spécifications.
- ▶ Ces tests peuvent être manuels ou automatisés, en général un mixte de tests automatisés et de tests manuels est ce qui a le meilleur retour sur investissement.
- ▶ Exemple : Pour une authentification on vérifie que la page d'accueil s'affiche comme décrit dans la spécification.

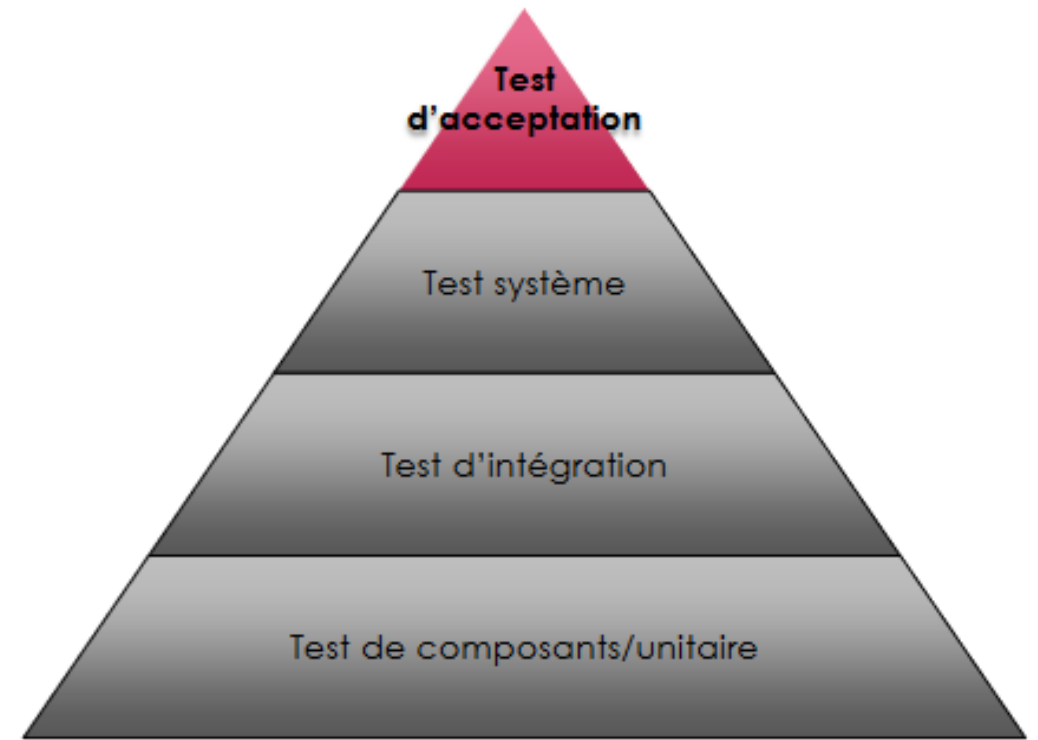




# Les Tests Unitaires

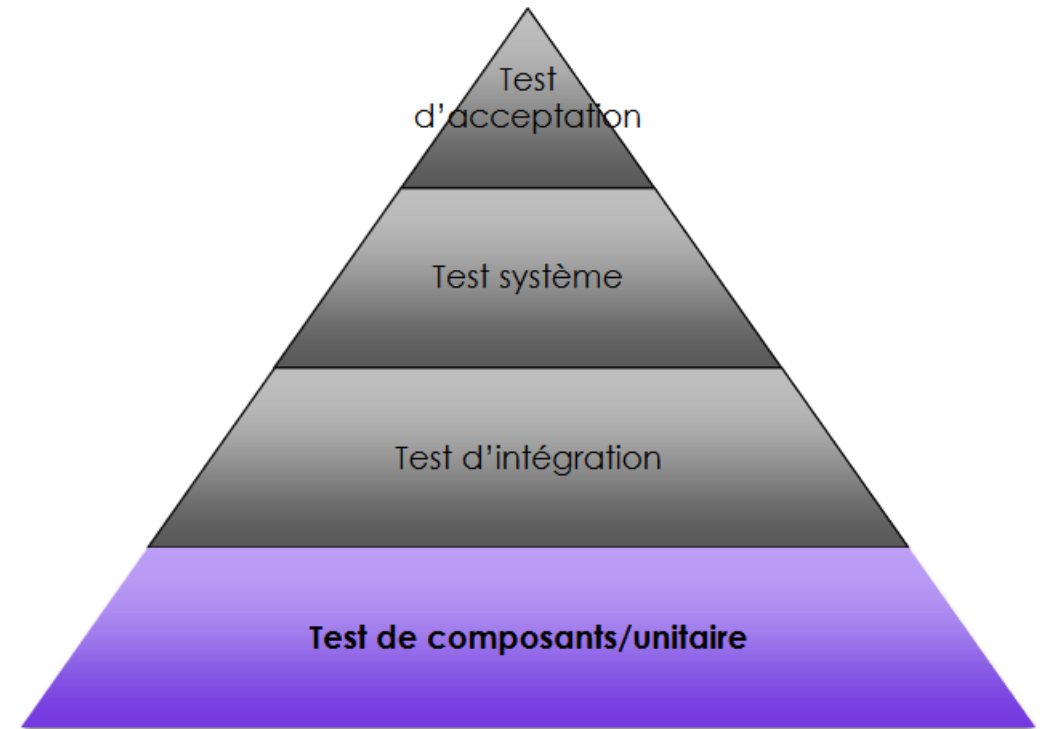
# Les niveaux de test :

- ▶ Test d'acceptation
- ▶ Les tests « finaux » effectués par le métier ou les utilisateurs finaux,
- ▶ Leur but est de confirmer que le produit final correspond bien aux besoins des utilisateurs finaux.
- ▶ Les tests d'acceptation sont des tests manuels.
- ▶ Exemple : l'authentification correspond bien à ce à quoi le métier ou les clients finaux s'attendent (un champ authentification trop petit peut être problématique par exemple)



# Les Tests Unitaires

- ▶ Le test unitaire est le niveau le plus bas des tests,
- ▶ Le test unitaire permet de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme (une méthode par exemple).
- ▶ Le test unitaire, doit être petit, rapide et très ciblé afin d'identifier le bloc de code défaillant.
- ▶ Les tests unitaires sont rédigés par des développeurs de logiciels, et non par des testeurs de logiciels.



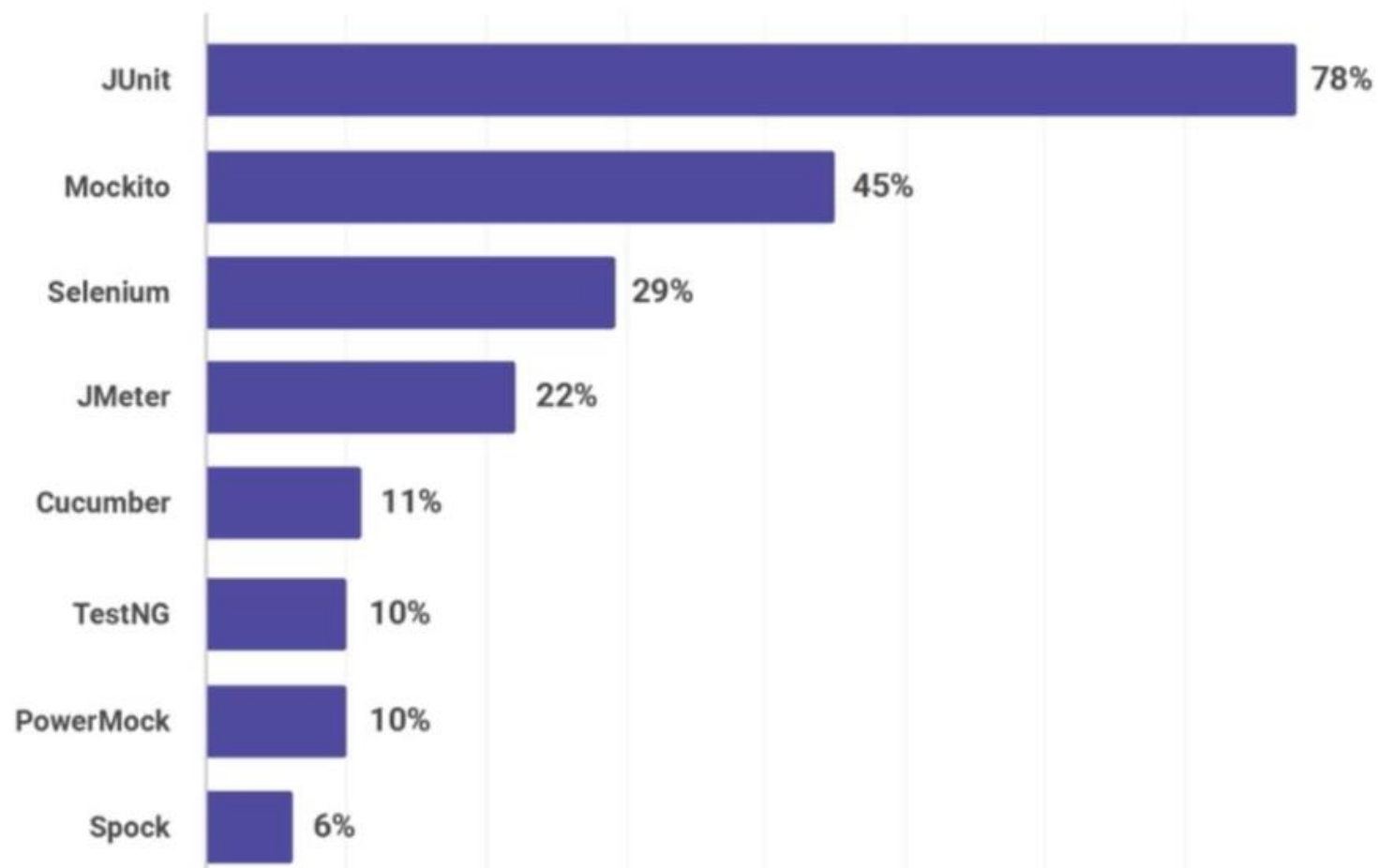
# Avantages des tests unitaires

- ▶ L'avantage d'un test unitaire sont:
  - ▶ Augmenter la confiance durant le développement.
  - ▶ Avoir un code propre et réutilisable.
  - ▶ Réduire le cout des erreurs

# Frameworks de tests unitaires

Langage de programmation	FrameWork de test unitaire
JAVA	JUNIT, JTEST
.NET	MSTEST, NUNIT
PHP	PhpUnit
PYTHON	Unittest, Pytest
JAVASCRIPT	Jasmine, JEST
C, C++	Embunit
Jquery	Qunit

# Les frameworks de test les plus utilisés



# JUnit

- ▶ JUnit est un framework de test unitaire pour le langage de programmation
  - ▶ Java, créé par Kent Beck et Erich Gamma.
  - ▶ Compatible avec java 8 ou supérieur.



# Exemple

```
public static Double asCelsius(Double tempF) {  
    return ((tempF - 32)*5)/9;  
}
```

Bloc de code à tester unitairement



```
@Test  
public void should_returnCorrectTemperatureAsCelsius() {  
    assertEquals(5, Calculator.asCelsius(41.0));  
}
```

Le test unitaire pour vérifier la  
conversion de 41 F en C  
41F == 5 C ??



# Exemple

Annotation pour indiquer  
le test à exécuter

La méthode de teste  
doit être public void

Le nom de la méthode décrit  
ce que le test doit vérifier.

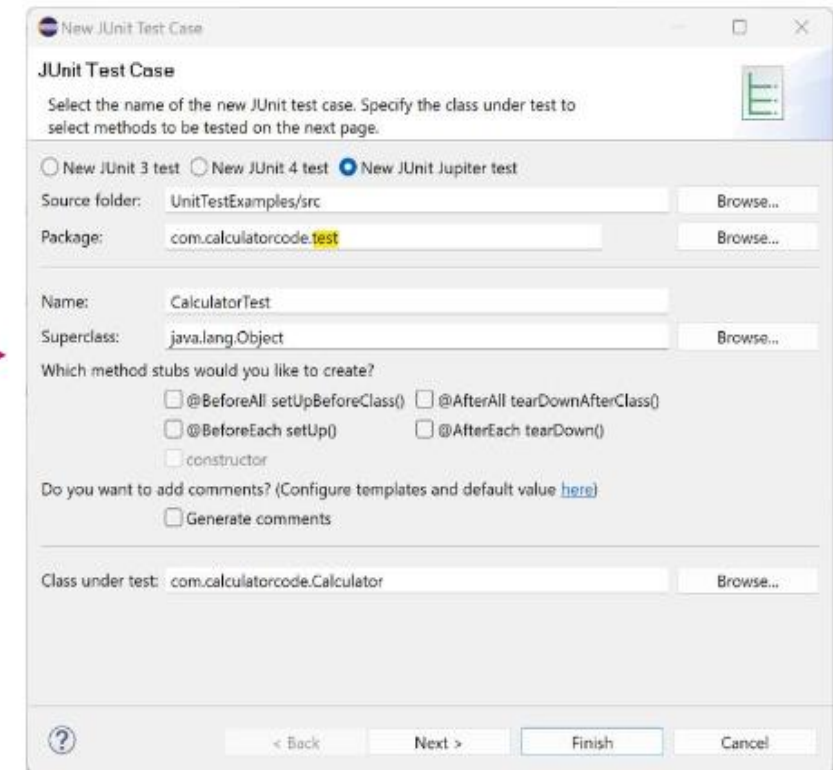
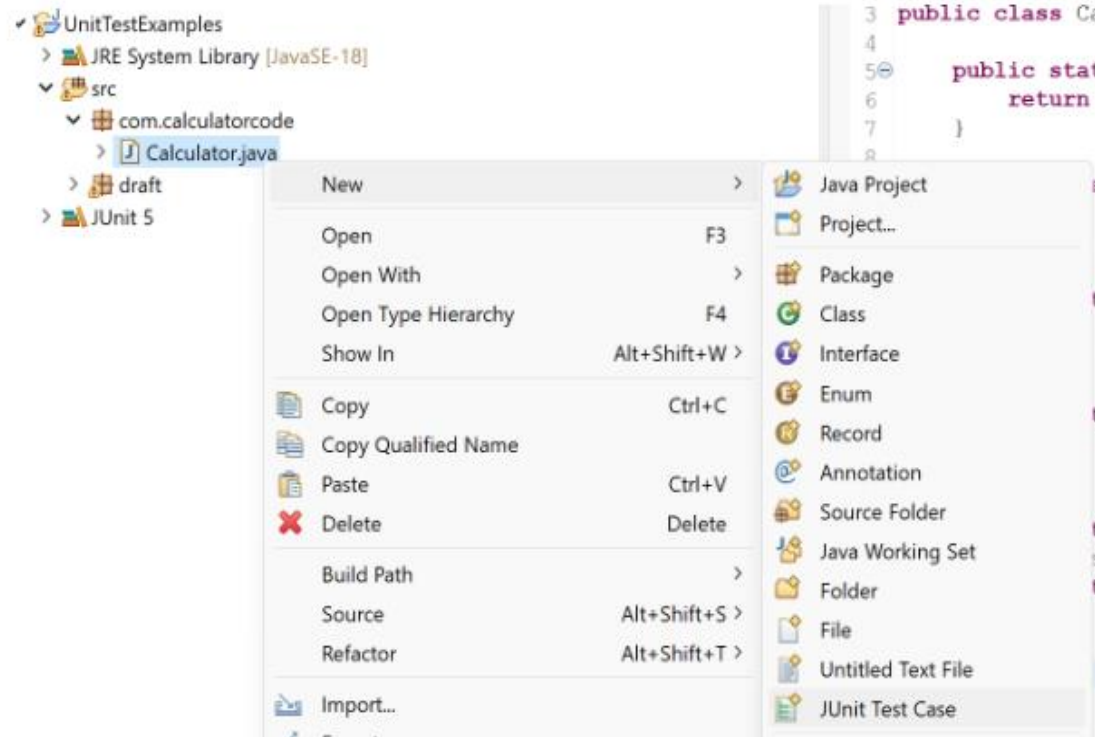
```
@Test  
public void should returnCorrectTemperatureAsCelsius() {  
    assertEquals(5, Calculator.asCelsius(41.0));  
}
```

Assertion pour vérifier que la  
condition est vraie

# Comment générer la classe de Test?

► Pour générer la classe de test sous eclipse:

1. Clic droit sur la classe java
2. Cliquer sur le menu « New » > « JUnit Test Case »
3. Dans la boîte de dialogue « New JUnit Test Case » renseigner le sous package « **test** » où organiser les tests
4. Cliquer sur le bouton « Confirmer » de la boîte de dialogue.



## ► @Test

- l'annotation de test indique à junit que la méthode public void à laquelle elle est attachée peut être exécutée comme cas de test

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

class CalculatorTestBackup {

    @Test
    public void should_return_correct_sum_value() {
        System.out.println("Sum test is running");
        assertEquals(Calculator.sum(20, 80), 100);
    }

    @Test
    public void should_return_correct_substraction_value() {
        System.out.println("Substraction test is running");
        assertEquals(-1, Calculator.substraction(1, 2));
    }

    @Test
    public void should_return_correct_pow_value() {
        System.out.println("Power test is running");
        assertEquals(16, Calculator.pow(2, 4));
    }
}
```

# Les annotations

- ▶ **@BeforeEach** : permet d'exécuter une méthode avant **chaque** test. C'est un très bon emplacement pour installer ou **organiser** un prérequis pour vos tests.
  - ▶ Exemple : une connexion à la base de données, Ouverture de fichier, un transfert ftp...
- ▶ **@AfterEach** : permet d'exécuter une méthode après **chaque** test. C'est un très bon emplacement pour nettoyer ou satisfaire à une postcondition.
  - ▶ Exemple: fermeture de connexion à la base de données, fermeture de fichier, fin du transfert ftp...

```
class CalculatorTestBackup {
    Calculator cal;

    @BeforeEach
    public void setup() {
        cal = new Calculator();
        System.out.println("Test has started.");
    }

    @AfterEach
    public void tearDown() {
        System.out.println("Test is finished.\n");
    }

    @Test
    public void should_return_correct_div_value() {
        System.out.println("Div test is running");
        assertEquals(5, cal.div(25, 5));
    }

    @Test
    public void should_return_correct_multi_value() {
        System.out.println("Mult test is running");
        assertEquals(48, cal.mult(12, 4));
    }
}
```



```
Console x Problems @
<terminated> CalculatorTestBackup
Test has started.
Div test is running
Test is finished.

Test has started.
Mult test is running
Test is finished.
```

# Les annotations

- ▶ **@BeforeAll** : Désignez une méthode statique pour qu'elle soit exécutée avant *tous* vos tests. Vous pouvez l'utiliser pour installer d'autres variables statiques pour vos tests.
- ▶ **@AfterAll** : Désignez une méthode statique pour qu'elle soit exécutée après *tous* vos tests. Vous pouvez utiliser ceci pour nettoyer les dépendances statiques.


```
class CalculatorTestBackup {
    Calculator cal;

    @BeforeAll
    public static void setupAll() {
        System.out.println("Befor all tests.");
    }

    @AfterAll
    public static void teardownAll() {
        System.out.println("After all tests.");
    }

    @Test
    public void should_return_correct_sum_value() {
        System.out.println("Sum test is running");
        assertEquals(Calculator.sum(20, 80), 100);
    }

    @Test
    public void should_return_correct_substraction_value() {
        System.out.println("Substraction test is running");
        assertEquals(-1, Calculator.substraction(1, 2));
    }
}
```



```
Console x Problems @ Javadoc
<terminated> CalculatorTestBackup [JUnit] C:\
Befor all tests.
Substraction test is running
Sum test is running
After all tests.
```

# Les annotations

- ▶ **@BeforeAll** : Désignez une méthode statique pour qu'elle soit exécutée avant *tous* vos tests. Vous pouvez l'utiliser pour installer d'autres variables statiques pour vos tests.
- ▶ **@AfterAll** : Désignez une méthode statique pour qu'elle soit exécutée après *tous* vos tests. Vous pouvez utiliser ceci pour nettoyer les dépendances statiques.

```
class CalculatorTestBackup {  
    Calculator cal;  
  
    @BeforeAll  
    public static void setupAll() {  
        System.out.println("Befor all tests.");  
    }  
  
    @AfterAll  
    public static void teardownAll() {  
        System.out.println("After all tests.");  
    }  
  
    @Test  
    public void should_return_correct_sum_value() {  
        System.out.println("Sum test is running");  
        assertEquals(Calculator.sum(20, 80), 100);  
    }  
  
    @Test  
    public void should_return_correct_substraction_value() {  
        System.out.println("Substraction test is running");  
        assertEquals(-1, Calculator.substraction(1, 2));  
    }  
}
```



Console × Problems @ Javadoc  
<terminated> CalculatorTestBackup [JUnit] C:\  
Befor all tests.  
Substraction test is running  
Sum test is running  
After all tests.



# Les annotations

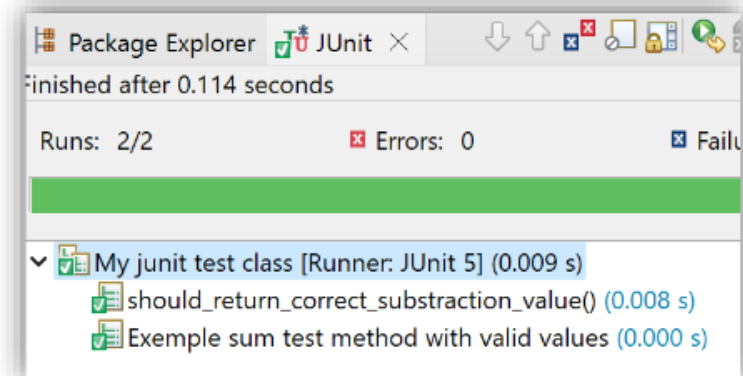
## ► @DisplayName

- Permet de définir un libellé pour la classe ou la méthode de test annotée.

```
@DisplayName("My junit test class")
class CalculatorTestBackup {
    Calculator cal;

    @Test
    @DisplayName("Exemple sum test method with valid values ")
    public void should_return_correct_sum_value() {
        System.out.println("Sum test is running");
        assertEquals(Calculator.sum(20, 80), 100);
    }

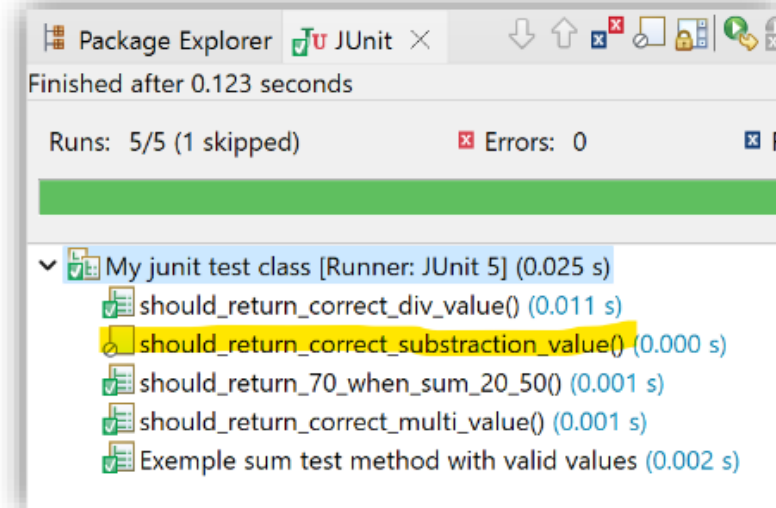
    @Test
    public void should_return_correct_substraction_value() {
        System.out.println("Substraction test is running");
        assertEquals(-1, Calculator.substraction(1, 2));
    }
}
```



## ► @Disabled

- Permet de désactiver une classe ou méthode de test.

```
@Disabled
@Test
public void should_return_correct_substraction_value() {
    System.out.println("Substraction test is running");
    assertEquals(-1, Calculator.substraction(1, 2));
}
```





## ► @RepeatedTest

- Permet exécuter la méthode de test plusieurs fois.

```
@RepeatedTest(10)
public void should_return_correct_multi_value() {
    System.out.println("Mult test is running");
    assertEquals(48, cal.mult(12, 4));
}
```



```
▼ [icon] should_return_correct_multi_value() (0.002 s)
  [icon] repetition 1 of 10 (0.002 s)
  [icon] repetition 2 of 10 (0.002 s)
  [icon] repetition 3 of 10 (0.003 s)
  [icon] repetition 4 of 10 (0.001 s)
  [icon] repetition 5 of 10 (0.001 s)
  [icon] repetition 6 of 10 (0.001 s)
  [icon] repetition 7 of 10 (0.001 s)
  [icon] repetition 8 of 10 (0.001 s)
  [icon] repetition 9 of 10 (0.000 s)
  [icon] repetition 10 of 10 (0.003 s)
```

## ► **@ParameterizedTest :**

- Permet d'exécuter le même test avec plusieurs entrants **@ValueSource**.

```
@ParameterizedTest
@ValueSource(ints = {10,6,80})
void should_return_true_when_positive_number(int nb) {
    assertEquals(true, Calculator.isPositiveNb(nb));
}
```



```
▼ [1] should_return_true_when_positive_number(int) (0.038 s)
  [1] 10 (0.038 s)
  [2] 6 (0.002 s)
  [3] 80 (0.001 s)
```

- ▶ Les assertions sont des méthodes statiques de la classe

**public class Assert extends java.lang.Object**

- ▶ **assertTrue** : vérifie qu'une condition est vraie. Si elle ne l'est pas, l'exception `AssertionError`, avec le message donné, est générée.
- ▶ **assertFalse** : vérifie qu'une condition est fausse. Si elle ne l'est pas, l'exception `AssertionError`, avec le message donné, est générée.

```
@Test
void should_return_true_when_positive_number() {
    assertTrue(Calculator.isPositiveNb(100));
}

@Test
void should_return_true_when_negative_number() {
    assertFalse(Calculator.isPositiveNb(-88));
}
```

## ► **assertEquals :**

- vérifie que deux objets sont égaux. Si ce n'est pas le cas, un `AssertionError`, avec le message donné, est générée.

```
@Test
public void should_return_correct_div_value() {
    System.out.println("Div test is running");
    assertEquals(5, cal.div(25, 5));
}
```

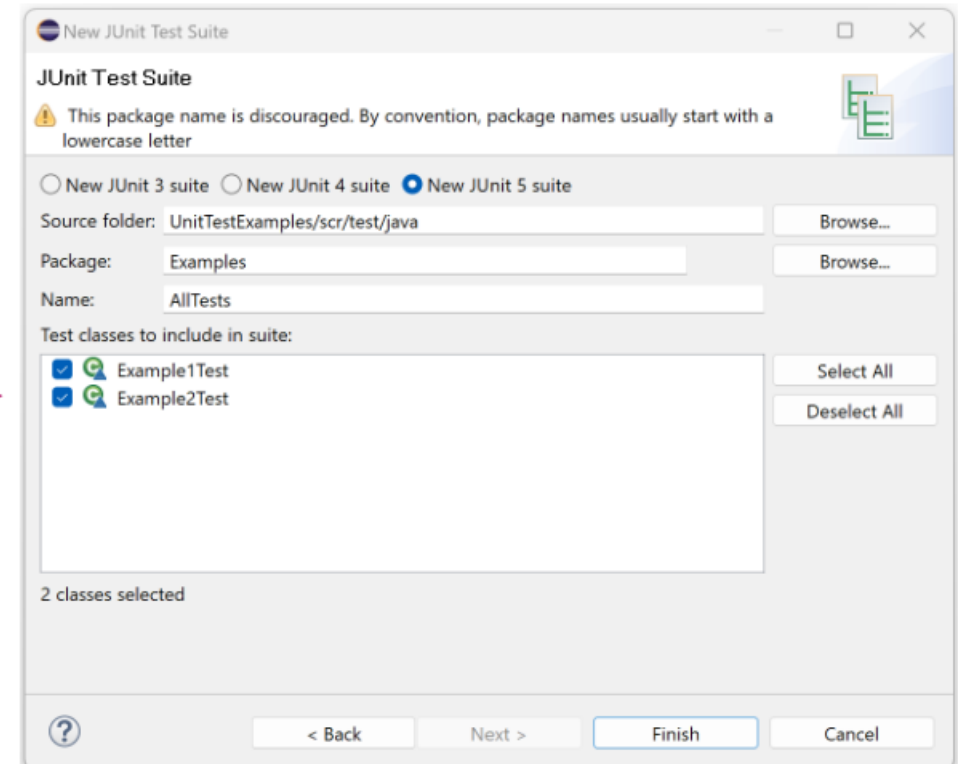
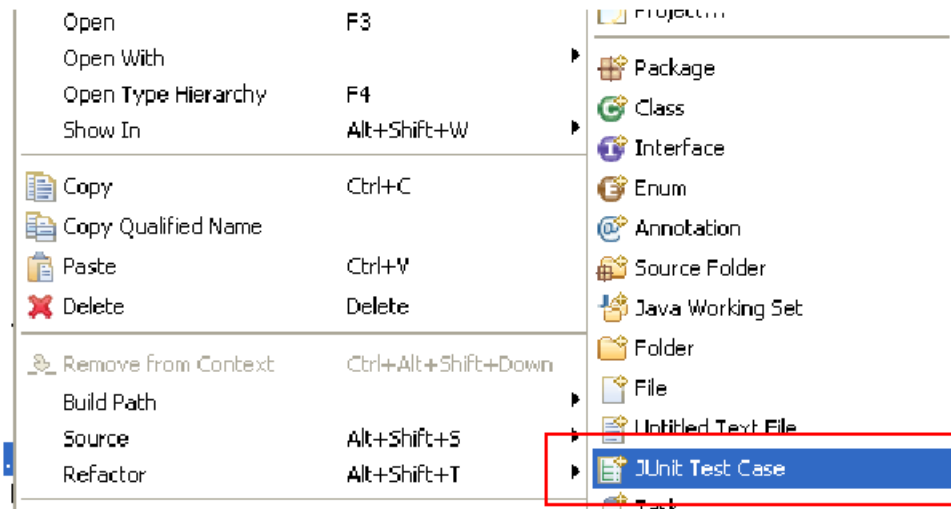
## ► **assertNull**

- vérifie que l'objet fourni en paramètre est null. Si ce n'est pas le cas, un `AssertionError`, avec le message donné, est générée.

```
@Test
void sould_return_null_object() {
    assertNull(cal.getObj());
}
```

# Générer test suite

- ▶ Test suite est utilisée pour regrouper quelques cas de test unitaires et les exécuter ensemble.



# Exemple de test suite

AllTests.java

```
1 package Examples;
2
3 import org.junit.platform.suite.api.SelectClasses;
4
5
6 @Suite
7 @SelectClasses({ Example1Test.class, Example2Test.class })
8 public class AllTests {
9
10 }
11
```

Les classes de test à exécuter

# Annexe : Les livrables en Génie logiciel



## **Plan de test du logiciel**

- Décrit les procédures de tests appliquées au logiciel pour contrôler son bon fonctionnement
  - ✓ tests de validation
  - ✓ ...

## **Plan d'assurance qualité**

- Décrit les activités mises en œuvre pour garantir la qualité du logiciel

## **Manuel utilisateur**

- Mode d'emploi pour le logiciel dans sa version finale

## **Code source**

- Code complet du produit fini

## **Rapport des tests**

- Décrit les tests effectués et les réactions du système

## **Rapport des défauts**

- Décrit les comportements des systèmes qui n'ont pas satisfait le client
- Il s'agit le plus souvent de défaillance du logiciel ou d'erreurs