





# Tests unitaires

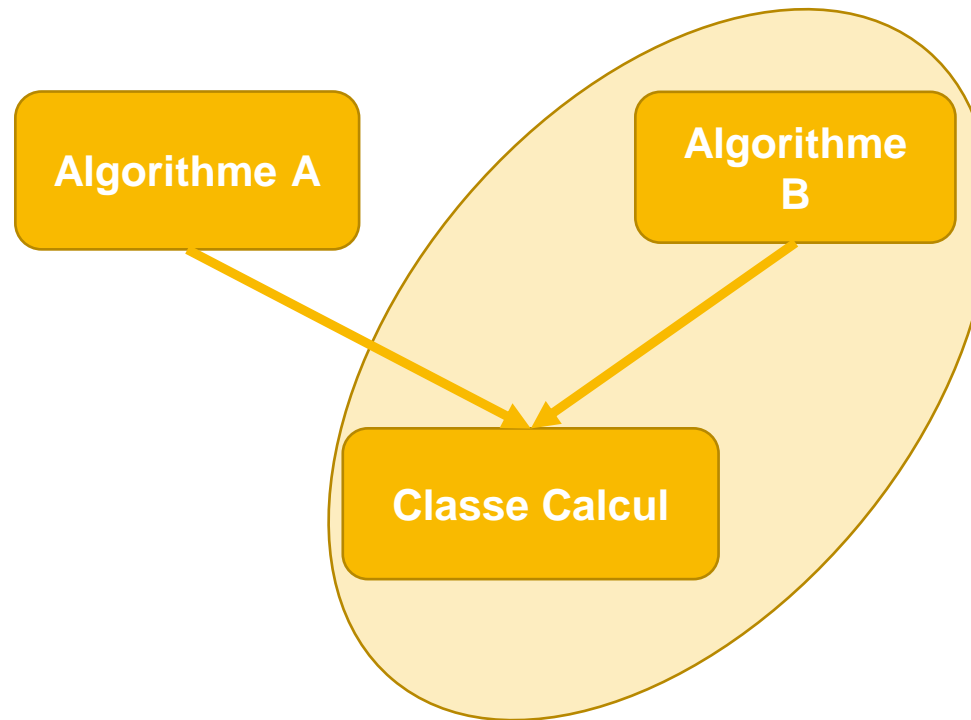
Avec JUnit

# Problème courant

1. Un nouvel **algorithme B** nécessite la modification d'une classe **Calcul**

2. Après modification de **Calcul**, l'algorithme B fonctionne très bien 

3. Le développeur a oublié l'existence de l'algorithme A qui ne fonctionne plus 



Solution: avoir des tests automatisés, qui testent toutes les méthodes, et capables de détecter des régressions.

# Définition

## ➤ Définition

- ☐ Un **test unitaire** permet de vérifier le bon fonctionnement d'une partie d'une application appelée « unité ».
  - ☐ En Java, l'**unité** est la **méthode**.
- Un **test unitaire** est une **portion de code qui en teste une autre**.
- Les tests unitaires sont dans une **classe de tests** indépendante qu'on appelle **classe de tests unitaires**.

# Bonnes pratiques

- **Source:** [Xebia](#)
- Un test unitaire ne teste qu'une unité, i.e. il teste **une méthode** d'une classe.
- Une classe difficile à tester est l'indicateur d'une classe mal écrite. Il faut faire du "refactoring".
- Un test unitaire doit s'exécuter le plus rapidement possible
  - Proscrire l'accès à des fichiers, bases de données ou services externes.
  - Un test qui dialogue avec une base de données est un **test d'intégration**.
- Ne testez qu'un comportement à la fois. Soyez raisonnable et gardez le test simple, lisible et concentré sur ce comportement.

# Bonnes pratiques

- Source: [Xebia](#)
- Pensez à utiliser un [framework de mocks](#) pour simuler l'accès à un fichier, une base de données ou service externe.
- Ne vous concentrez pas sur une couverture de code à 100%.
- Ne développez pas vos tests unitaires « plus tard ». Si vous n'utilisez pas l'approche TDD, développez-les le plus tôt possible.

- Ensemble de systèmes de tests unitaires dérivés de JUnit
- Ensemble de principes s'appliquant pour différents langages:
  - ☐ JsUnit pour Javascript
  - ☐ PyUnit pour Python
  - ☐ CppUnit pour C++
  - ☐ PhpUnit pour Php
  - ☐ JUnit pour Java
  - ☐ ...

# JUnit

# Vocabulaire

- **Unité** : bloc de code à tester
- **Assertion** : vérification d'un résultat attendu. Si la vérification échoue, une exception est lancée et le test courant s'arrête
- **Fixture** : initialisation / terminaison commune à tous les tests unitaires
- **Suite** : un ensemble de tests unitaires exécutables

# Règles de nommage tests unitaires

## Classe à tester

```
public class AnalyseDonnees {  
    public double extraireMax(List<Ligne> data){  
        double resultat = 0.0;  
        // Code à tester  
        return resultat;  
    }  
  
    public double agreger(){  
        double resultat = 0.0;  
        // Code d'aggrégation  
        return resultat;  
    }  
}
```

Import annotation  
@Test

## Classe de test unitaire

```
import static org.junit.Assert.*;  
import org.junit.Test;
```

Convention de  
nommage de la classe  
(suffixe « Test » )

```
public class AnalyseDonneesTest {
```

Annotation « @Test » qui  
indique que la méthode est  
un test à exécuter.

```
@Test  
public void testExtraireMax() {  
    // Test de la méthode extraireMax  
}
```

```
@Test  
public void testAggreger() {  
    // Test de la méthode agreger  
}
```

```
}
```



# Assertions

1. Création d'une instance de la classe à tester

```
public class AnalyseDonneesTest {  
    @Test  
    public void testExtraireMax() {  
        AnalyseDonnees analyse = new AnalyseDonnees();
```

2. Préparation des données de test

```
        List<Ligne> lignes = new ArrayList<>();  
        lignes.add(new Ligne("Versem. Société A", -12000, 2.35));  
        lignes.add(new Ligne("Encais. Société B", 8750, 1.28));  
        lignes.add(new Ligne("Encais. Société A", 2375, 0.34));
```

3. Exécution de la méthode à tester

```
        int resultat = analyse.extraireMax(lignes);  
        assertEquals(8750, resultat);
```

4. Assertion qui vérifie le résultat

```
    }  
  
    // ...  
}
```

# Assertions (test d'égalité)

- Pour vérifier qu'un objet retourné par une méthode est bien égal à un objet attendu.
- void **assertEquals** (Object expected, Object actual)
- void **assertEquals** (String message, Object expected, Object actual)
- void **assertEquals** (long expected, long actual)
- void **assertEquals** (String message, long expected, long actual)
- void **assertEquals** (String expected, String actual)
- void **assertEquals** (String message, String expected, String actual)
- ...

# Assertions (test de condition)

- Pour vérifier qu'une condition est vraie, ou fausse.
  - ❑ Exemple: vérifier qu'un résultat retourné par une méthode est  $> 10.0$
  - ❑ `assertTrue(resultat > 10.0);`
- `void assertTrue (boolean condition)`
- `void assertTrue (String message, boolean condition)`
- `void assertFalse (boolean condition)`
- `void assertFalse (String message, boolean condition)`

# Assertions (test de nullité)

- Pour vérifier qu'un objet attendu en retour d'une méthode doit être **null**
- void **assertNull** (Object object)
- void **assertNull** (String message, Object object)
  
- Pour vérifier qu'un objet attendu en retour d'une méthode doit être **non null**
- void **assertNotNull** (Object object)
- void **assertNotNull** (String message, Object object)

# Fail (test en échec)

- On va invoquer ces méthodes principalement dans un bloc catch pour indiquer que le test est en échec si une exception se produit.
- void **fail** ()
- void **fail** (String message)

# Exemple: classe à tester

- Classe trouvée sur internet et qui permet d'évaluer une expression en polonaise inversée.
- Exemple: si polish="1, 2, +, 3, \*" alors la méthode retourne 9
- Mais fonctionne t'elle correctement ?

```
public class PolishUtils {  
    public static int eval(String polish) {  
        String operators = "+-*/";  
        Stack<String> stack = new Stack<String>();  
        for (String t : polish.split(",")) {  
            if (!operators.contains(t)) {  
                stack.push(t);  
            } else {  
                int a = Integer.valueOf(stack.pop());  
                int b = Integer.valueOf(stack.pop());  
                switch (t) {  
                    case "+":  
                        stack.push(String.valueOf(a + b));  
                        break;  
                    case ...:  
                        ...  
                }  
            }  
        }  
        return Integer.valueOf(stack.pop());  
    }  
}
```

# Exemple: classe de test unitaire

```
public class PolishUtilsTest {

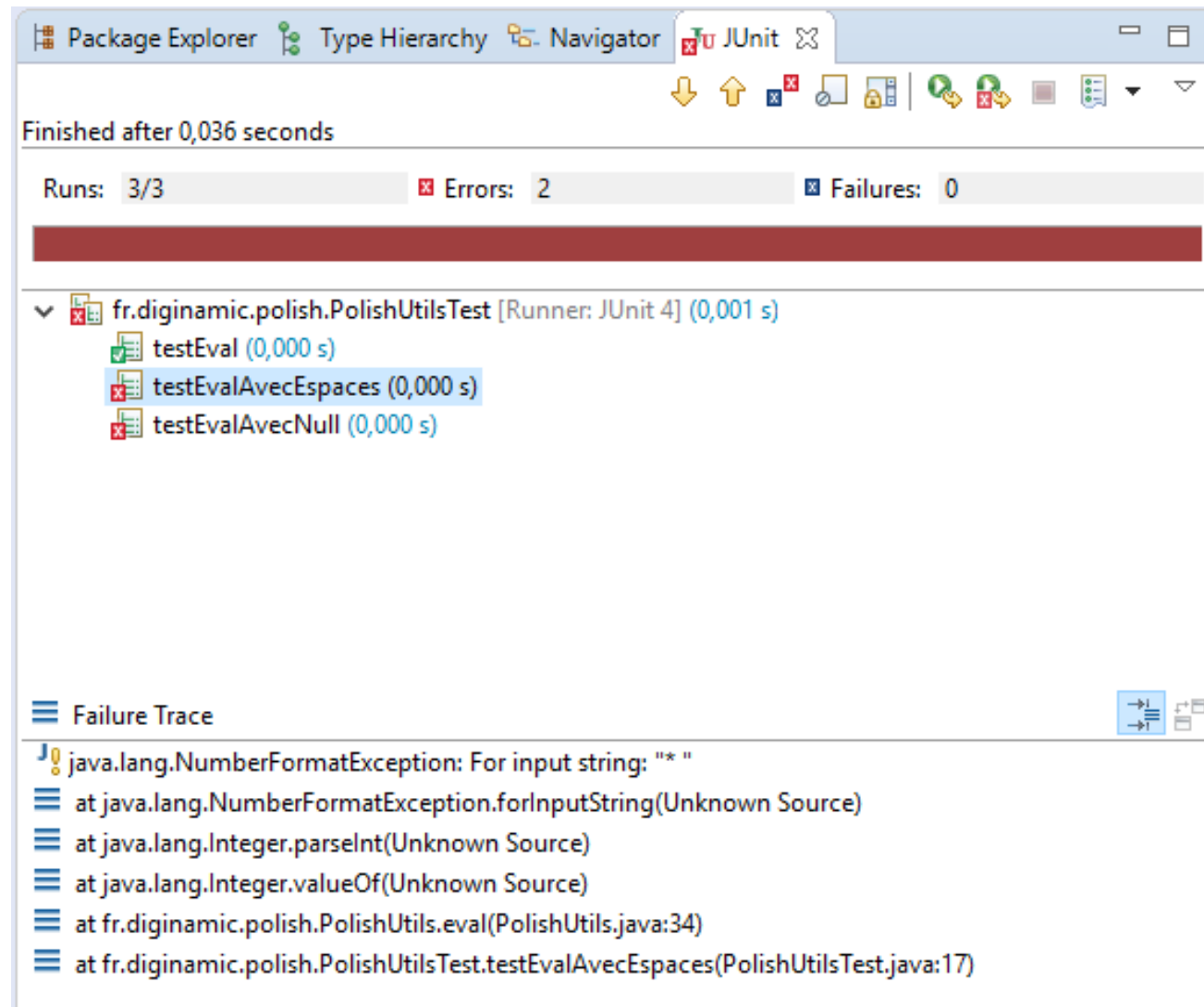
    @Test
    public void testEval() {
        int resultat = PolishUtils.eval("1,2,+,3,*");
        // Normalement la méthode doit retourner 9. Vérifions:
        assertEquals(9, resultat);
    }

    @Test
    public void testEvalAvecEspaces() {
        int resultat = PolishUtils.eval("1 ,2 ,+ ,3 ,* ");
        // Si la méthode a été bien pensée, elle doit être capable de
        // s'affranchir des caractères blancs. Vérifions:
        assertEquals(9, resultat);
    }

    @Test
    public void testEvalAvecNull() {
        int resultat = PolishUtils.eval(null);
        // Si la méthode est robuste, elle doit être capable de tout
        // supporter. Vérifions:
        assertEquals(0, resultat);
    }
}
```

# Exemple: rapport de tests JUnit

- Plugin d'exécution de Test intégré à Eclipse





# Exemple: modification après test

## ➤ Ajout de 2 modifications pour rendre la classe plus robuste

```
public class PolishUtils {  
    public static int eval(String polish) {  
        if (polish==null){  
            return 0;  
        }  
        String operators = "+-*/";  
        Stack<String> stack = new Stack<String>();  
        for (String token : polish.split(",")) {  
            String tokenMod = token.trim();  
            if (!operators.contains(tokenMod)) { //push to stack if it is a number  
                stack.push(tokenMod);  
            } else { //pop numbers from stack if it is an operator  
                int a = Integer.valueOf(stack.pop());  
                int b = Integer.valueOf(stack.pop());  
                switch (tokenMod) {  
                    case "+":  
                        stack.push(String.valueOf(a + b));  
                        break;  
                }  
            }  
        }  
        return Integer.valueOf(stack.pop());  
    }  
}
```

# Fixtures

## @BeforeClass

Exécution de code  
avant l'instanciation  
de la classe de test

## @Before

Exécution de code  
avant l'exécution de  
chaque test

## @After

Exécution de code  
après l'exécution de  
chaque test

## @AfterClass

Exécution après  
l'exécution de tous les  
tests

```
public class MaClasseTest {  
  
    @BeforeClass  
    public static void setUpBeforeClass() {  
        // implémentation  
    }  
  
    @Before  
    public void setUp() {  
        // implémentation  
    }  
  
    @Test  
    public void testMaPremiereMethode() {  
        // une implémentation  
    }  
  
    @Test  
    public void testMaSecondeMethode() {  
        // une implémentation  
    }  
  
    @After  
    public void tearDown() {  
        // implémentation  
    }  
  
    @AfterClass  
    public static void tearDownAfterClass() {  
        // implémentation  
    }  
}
```

# Paramètres optionnels de @Test

Le résultat attendu du test est le **lancement d'une exception**.

Si l'exception n'est pas lancée, le test échoue.

```
public class MaClasseTest {
```

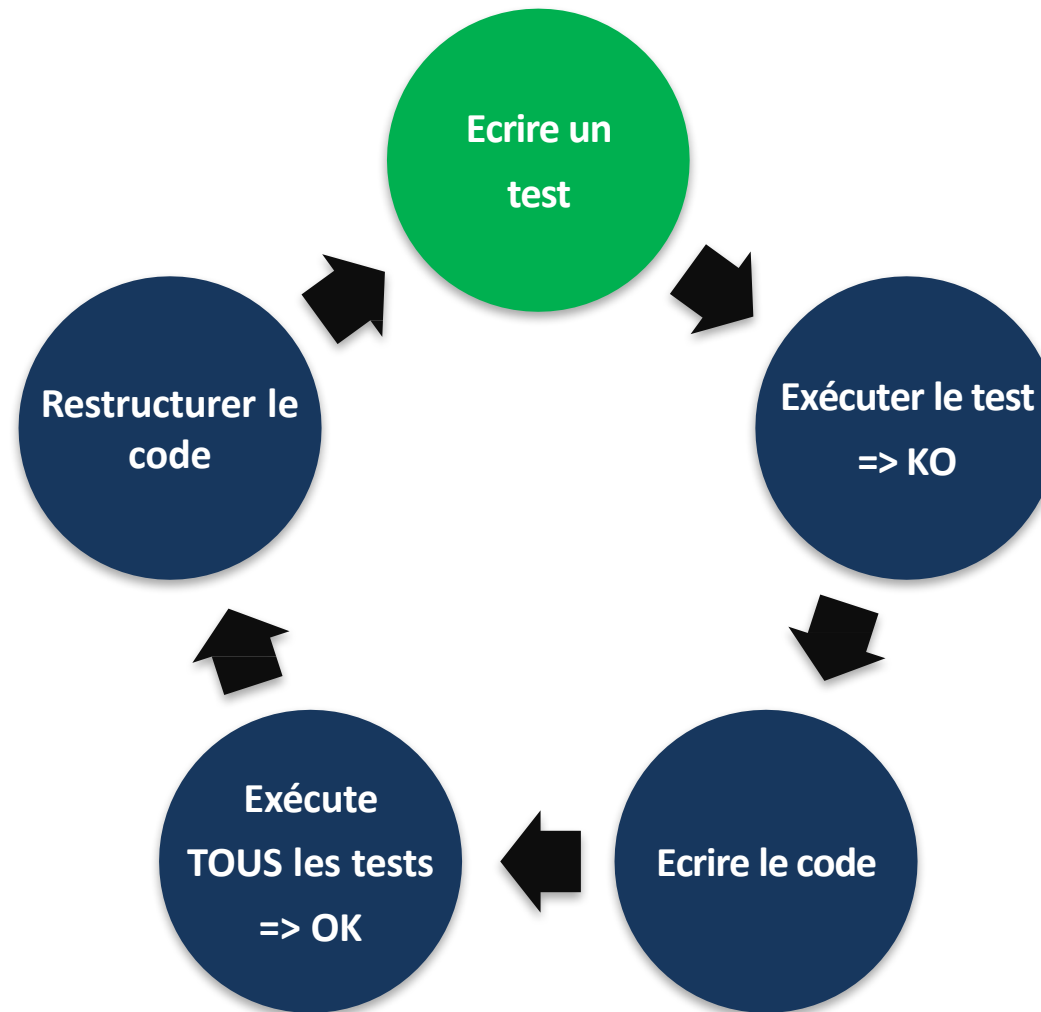
```
    @Test (expected = NullPointerException.class)
    public void testMaPremiereMethode() {
        // une implémentation
    }
```

Le test échoue si l'exécution prend plus de 1000 ms.

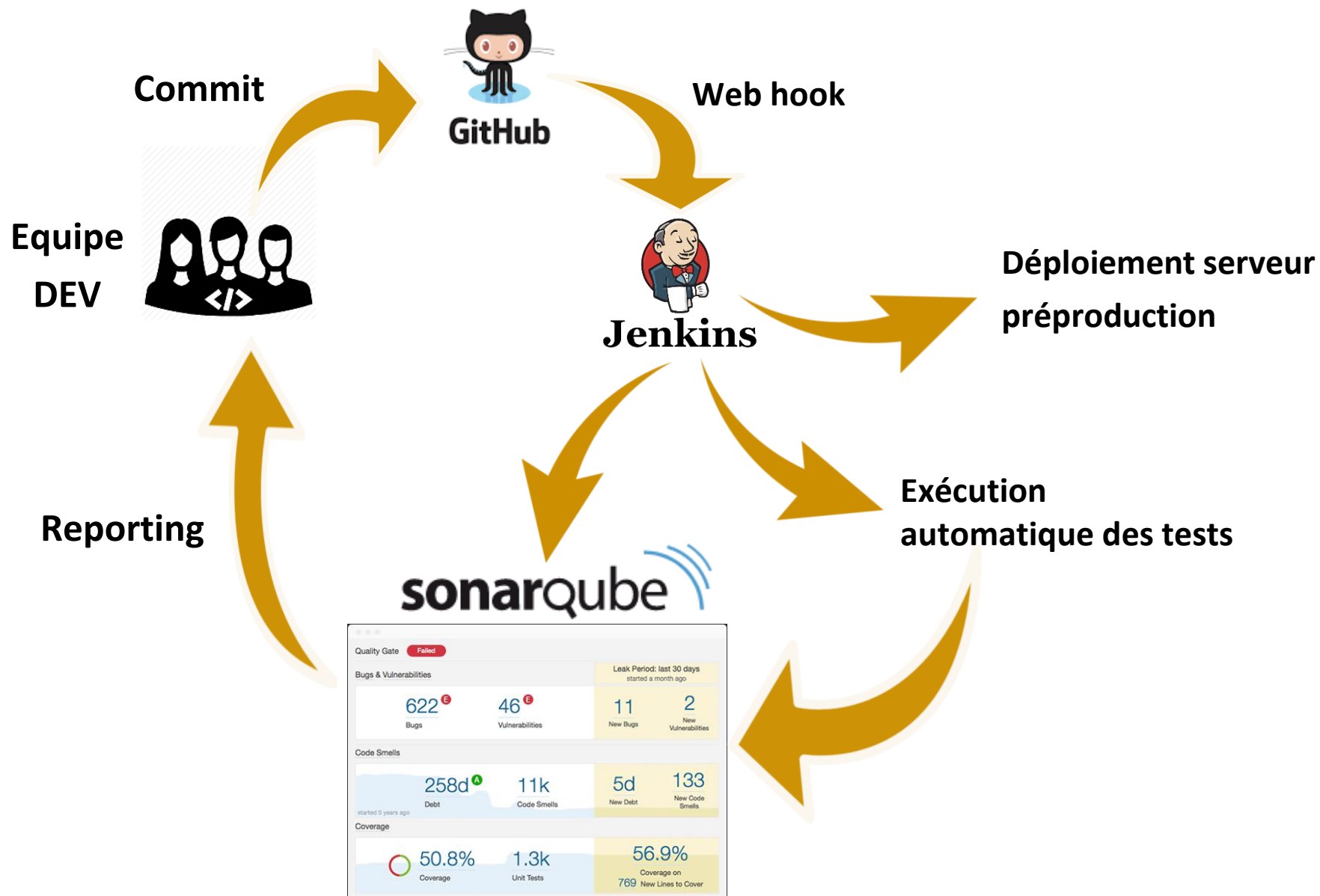
```
    @Test (timeout = 1000)
    public void testMaSecondeMethode() {
        // une implémentation
    }
```

```
}
```

# Test Driven Development - TDD



# Intégration continue



# Atelier (TP)

## OBJECTIFS :

Savoir développer des tests unitaires

DESCRIPTION : Dans ce TP vous allez apprendre à développer des tests unitaires avec JUnit.