



**UNIVERSITÉ DE YAOUNDÉ 1**  
**DÉPARTEMENT INFORMATIQUE**



**INFORMATION AND COMMUNICATION  
TECHNOLOGIES FOR DEVELOPMENT**

**ICT304**

# **SOFTWARE TESTING AND DEPLOYMENT**

**Static Analysis  
& Dynamic  
Analysis**



**Réalisé par :**

- ➔ **MBIADA BAYON IDRIS DONALD**
- ➔ **NYADJOU LUCIE DANIELLE**
- ➔ **KAMELA PIERRICK DACK**
- ➔ **KEWOU MBOTCHAK CHRISTIAN**
- ➔ **NDOKOU ELAT DESIRE SAMIRA**

**21Q2915**

**21Q2301**

**21Q2493**

**21Q2402**

**21T2314**

# Sommaire

<b>A. TEST STATIQUE.....</b>	<b>3</b>
1. Définition.....	3
2. Types de Tests Statiques.....	3
a. Inspection de Code .....	3
b. Analyse Statique .....	3
c. Revue de Code.....	3
3. Objectifs des Tests Statiques.....	4
a. Détection Précoce des Erreurs .....	4
b. Amélioration de la Qualité du Code .....	4
c. Conformité aux Normes de Codage.....	4
d. Réduction des Coûts de Correction des Erreurs .....	4
4. Méthodes et Outils.....	4
a. Revue de Code Manuelle .....	4
b. Outils Automatisés.....	5
5. Avantages et Inconvénients.....	6
<b>B. TESTS DYNAMIQUES.....</b>	<b>7</b>
1. Test Unitaire.....	7
2. Test Asynchrone .....	8
3. Test d'Intégration.....	9
<b>C. RAPPORT DE TEST.....</b>	<b>10</b>
1. Introduction.....	10
2. Environnement de Test .....	10
3. Cas de Test .....	10
3.1. Test de la Méthode add .....	10
3.2. Test de la Méthode subtract.....	10
3.3. Test de la Méthode multiply.....	10
3.4. Test de la Méthode divide.....	11
3.5. Test de la Méthode fetchData.....	11
<b>CONCLUSION.....</b>	<b>11</b>

## **A. TEST STATIQUE**

### **1. Définition**

**Les tests statiques** sont une méthode de vérification du code source sans l'exécuter. Contrairement aux tests dynamiques, qui nécessitent l'exécution du programme pour vérifier son comportement, les tests statiques analysent le code pour identifier les erreurs potentielles, les problèmes de style, et les vulnérabilités avant l'exécution.

### **2. Types de Tests Statiques**

#### **a. Inspection de Code**

L'inspection de code est une revue minutieuse du code source par un ou plusieurs développeurs ou experts en programmation. Cette méthode est généralement formelle et suit un processus structuré pour identifier les erreurs, les violations de style, et les inefficacités.

#### **b. Analyse Statique**



L'analyse statique utilise des outils automatisés pour examiner le code source. Ces outils détectent des erreurs courantes, des failles de sécurité, des violations de normes de codage et des problèmes de performance potentiels.

**Exemples d'outils** : ESLint pour JavaScript, SonarQube pour plusieurs langages de programmation.

#### **c. Revue de Code**

La revue de code est un processus collaboratif où le code écrit par un développeur est examiné par ses pairs. Les revues de code peuvent être formelles ou informelles et sont essentielles pour partager des connaissances et améliorer la qualité du code.

### **3. Objectifs des Tests Statiques**

Les tests statiques ont plusieurs objectifs clés à savoir :

#### **a. Détection Précoce des Erreurs**

En identifiant les erreurs dans les premières étapes du développement, les tests statiques permettent de les corriger avant qu'elles ne deviennent coûteuses à résoudre.

#### **b. Amélioration de la Qualité du Code**

Les tests statiques contribuent à maintenir un code propre, bien structuré, et conforme aux normes de codage, ce qui facilite la maintenance et la compréhension du code.

#### **c. Conformité aux Normes de Codage**

Ils assurent que le code suit les normes de codage établies par l'organisation ou l'industrie, ce qui est crucial pour la lisibilité et la gestion du code sur le long terme.

#### **d. Réduction des Coûts de Correction des Erreurs**

Corriger les erreurs détectées tôt dans le cycle de développement est moins coûteux et moins perturbant que de les corriger après le déploiement.

### **4. Méthodes et Outils**

#### **a. Revue de Code Manuelle**

La revue de code manuelle implique des développeurs qui examinent le code ligne par ligne pour identifier les erreurs, discuter des améliorations potentielles, et partager des connaissances.

## **b. Outils Automatisés**

Les outils d'analyse statique automatisés sont utilisés pour scanner le code et détecter les erreurs de manière efficace. Voici quelques outils populaires :

- **ESLint** : Un outil de linting pour JavaScript qui analyse le code pour trouver des problèmes de style et des erreurs.



- **SonarQube** : Un outil de qualité de code qui prend en charge plusieurs langages et offre une analyse approfondie des vulnérabilités et des problèmes de qualité.



- **Pylint** : Un outil pour analyser le code Python, détecter les erreurs et appliquer des normes de codage.



## **5. Avantages et Inconvénients**

### **Avantages**

- ✓ **Détection Précoce des Erreurs** : Permet d'identifier et de corriger les erreurs avant l'exécution, réduisant ainsi les coûts et les efforts nécessaires pour les résoudre plus tard.
- ✓ **Amélioration de la Qualité** : En forçant le respect des normes de codage et en identifiant les inefficacités, les tests statiques améliorent la maintenabilité et la qualité du code.
- ✓ **Réduction des Coûts de Maintenance** : En détectant les erreurs tôt, les tests statiques contribuent à réduire les coûts associés aux corrections et à la maintenance du code.

### **Inconvénients**

- **Temps Nécessaire** : Les revues de code manuelles et les inspections formelles peuvent être chronophages, retardant parfois le développement.
- **Faux Positifs** : Les outils d'analyse statique peuvent générer des alertes pour des problèmes qui ne sont pas réellement des erreurs, nécessitant une évaluation manuelle pour les filtrer.
- **Complexité des Configurations** : La configuration initiale des outils d'analyse statique pour qu'ils correspondent aux normes spécifiques de l'organisation peut être complexe et nécessite du temps.

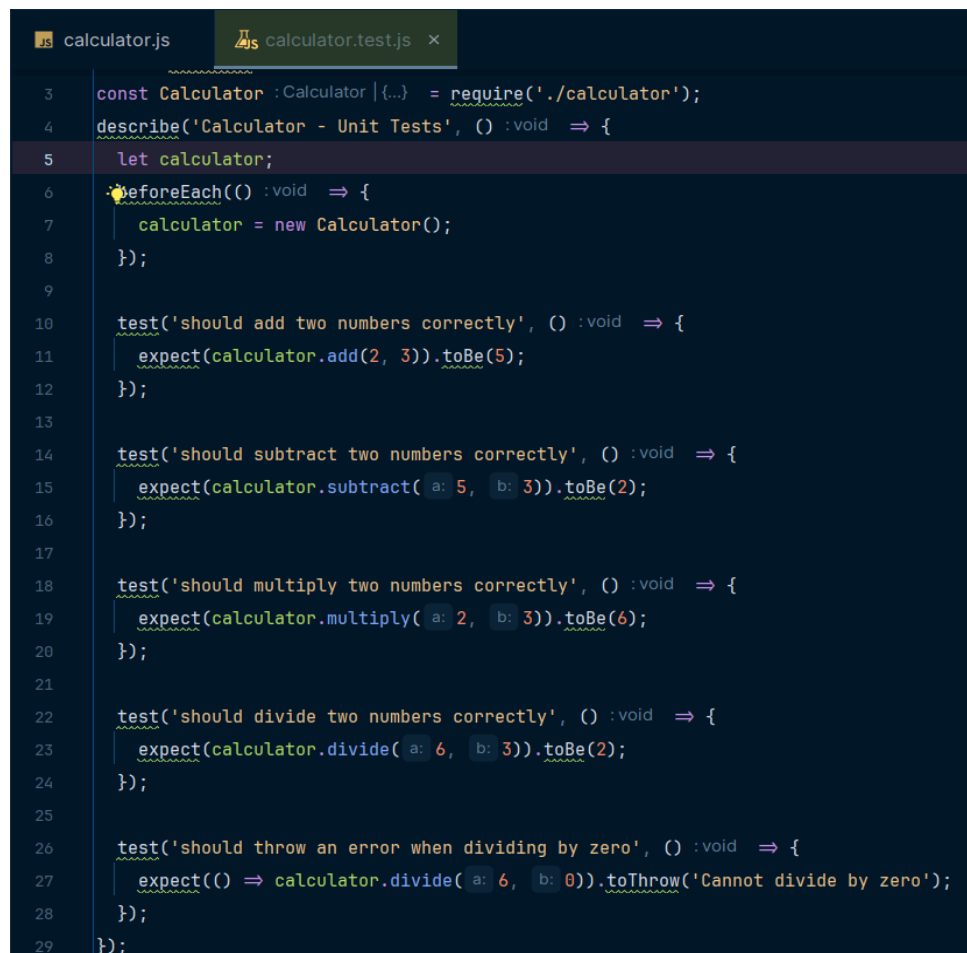
## B. TESTS DYNAMIQUES

Les Techniques de Test Dynamique : Test Unitaire, Test Asynchrone, Test d'Intégration.

Les tests dynamiques sont réalisés en exécutant le code et en observant son comportement. Ils sont essentiels pour valider que le système fonctionne comme attendu lorsqu'il est en fonctionnement. Voici une explication des trois techniques principales de tests dynamiques qui sont les tests unitaires, les tests asynchrones et les tests d'intégration.

### 1. Test Unitaire

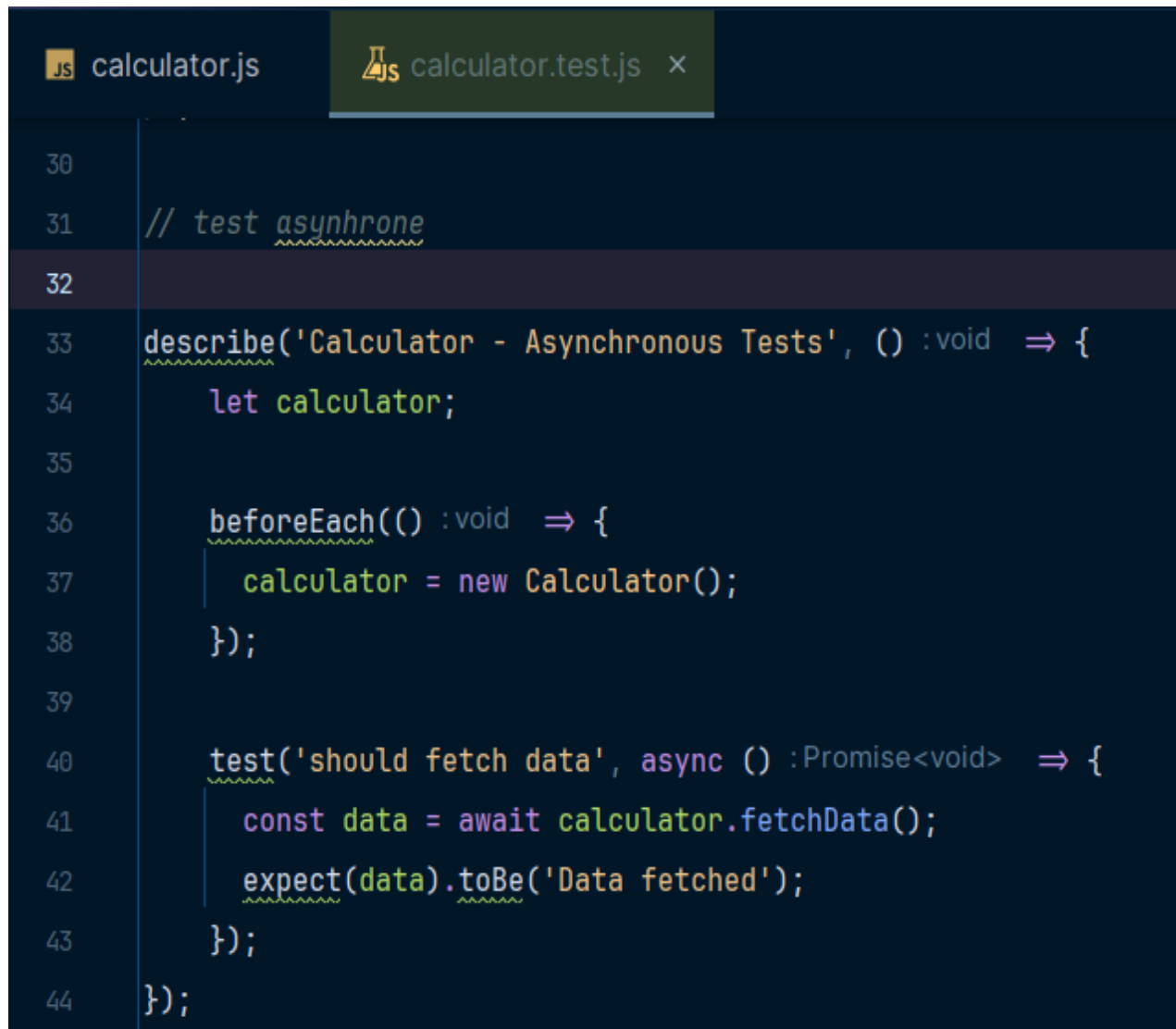
Les tests unitaires valident le bon fonctionnement des plus petites unités de code, généralement des fonctions ou des méthodes, de manière isolée.



```
calculator.js  calculator.test.js x
3  const Calculator : Calculator | {...} = require('./calculator');
4  describe('Calculator - Unit Tests', () : void => {
5    let calculator;
6    beforeEach(() : void => {
7      calculator = new Calculator();
8    });
9
10   test('should add two numbers correctly', () : void => {
11     expect(calculator.add(2, 3)).toBe(5);
12   });
13
14   test('should subtract two numbers correctly', () : void => {
15     expect(calculator.subtract(a: 5, b: 3)).toBe(2);
16   });
17
18   test('should multiply two numbers correctly', () : void => {
19     expect(calculator.multiply(a: 2, b: 3)).toBe(6);
20   });
21
22   test('should divide two numbers correctly', () : void => {
23     expect(calculator.divide(a: 6, b: 3)).toBe(2);
24   });
25
26   test('should throw an error when dividing by zero', () : void => {
27     expect(() => calculator.divide(a: 6, b: 0)).toThrow('Cannot divide by zero');
28   });
29 });
```

## 2. Test Asynchrone

Les tests asynchrones vérifient le comportement du code asynchrone, comme les appels API, les timers, et les promesses.

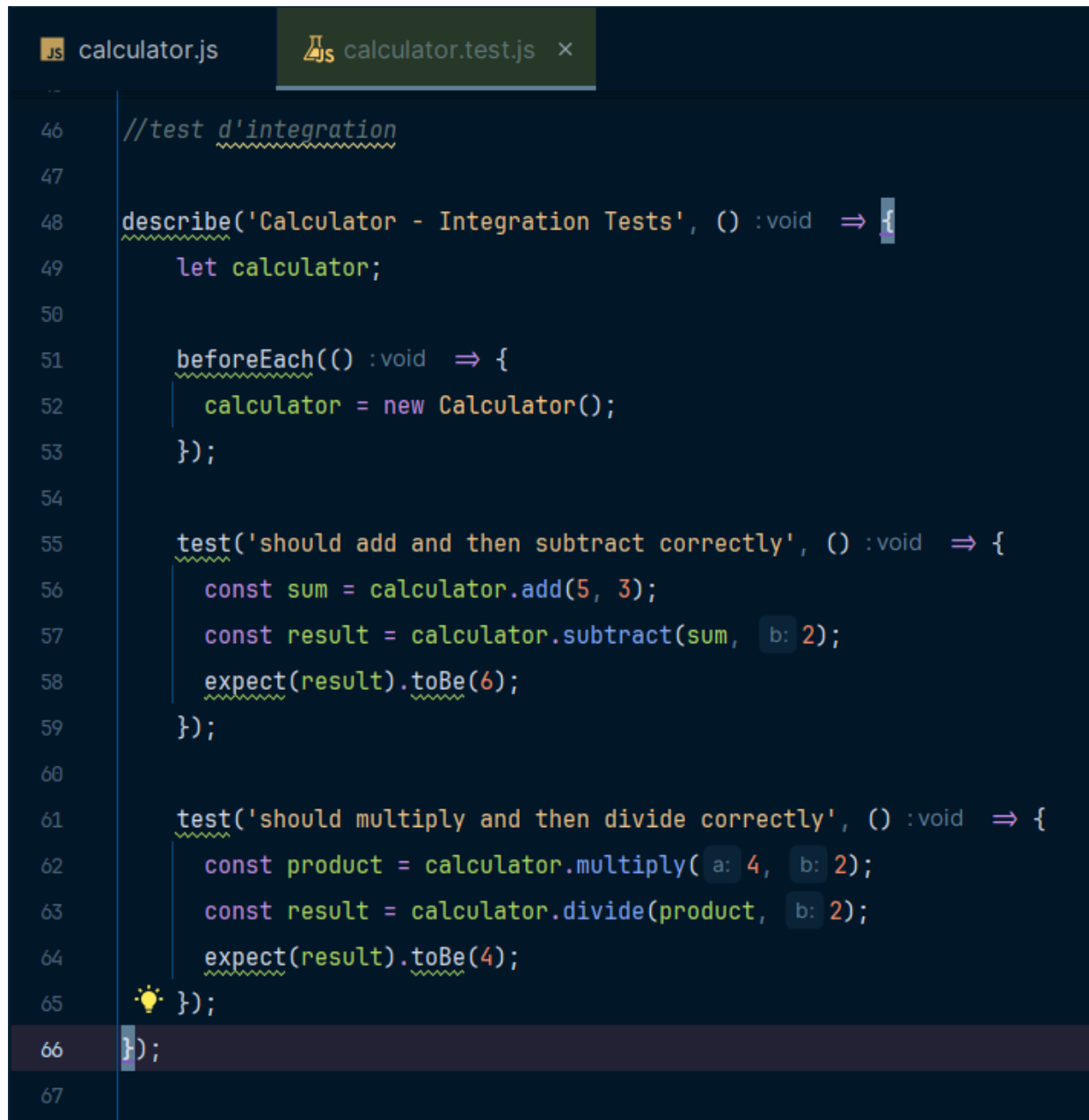


```
JS calculator.js  JS calculator.test.js x
30
31 // test asynchrone
32
33 describe('Calculator - Asynchronous Tests', () :void => {
34     let calculator;
35
36     beforeEach(() :void => {
37         calculator = new Calculator();
38     });
39
40     test('should fetch data', async () :Promise<void> => {
41         const data = await calculator.fetchData();
42         expect(data).toBe('Data fetched');
43     });
44 });
```



### 3. Test d'Intégration

Les tests d'intégration valident que plusieurs composants fonctionnent correctement ensemble. Ils sont plus complexes que les tests unitaires et peuvent impliquer des interactions avec des bases de données, des API, ou d'autres systèmes.



```
calculator.js  calculator.test.js x

46  //test d'integration
47
48  describe('Calculator - Integration Tests', () :void => {
49      let calculator;
50
51      beforeEach(() :void => {
52          calculator = new Calculator();
53      });
54
55      test('should add and then subtract correctly', () :void => {
56          const sum = calculator.add(5, 3);
57          const result = calculator.subtract(sum, b: 2);
58          expect(result).toBe(6);
59      });
60
61      test('should multiply and then divide correctly', () :void => {
62          const product = calculator.multiply(a: 4, b: 2);
63          const result = calculator.divide(product, b: 2);
64          expect(result).toBe(4);
65      });
66  });
67
```

## **C. RAPPORT DE TEST**

### **1. Introduction**

**Objectif** : Cette partie décrit les tests unitaires pour la classe Calculator qui inclut les méthodes add, subtract, multiply, divide, et fetchData.

**Portée** : Les tests couvrent les opérations arithmétiques de base et une opération asynchrone.

### **2. Environnement de Test**

Langage : JavaScript

Framework de Test : Jest

Outils de Linting : ESLint

### **3. Cas de Test**

#### **3.1. Test de la Méthode add**

Description : Teste l'addition de deux nombres.

ID de test	Description du test	Entrée	Sortie attendue
TC1	Addition de deux nombres positifs	Add(1,2)	3
TC2	Addition d'un nombre positif et un négatif	Add(1,-1)	0
TC3	Addition de deux nombres négatifs	Add(-1,-1)	-2

#### **3.2. Test de la Méthode subtract**

Description : Teste la soustraction de deux nombres.

ID de test	Description du test	Entrée	Sortie attendue
TC1	Soustraction de deux nombres positifs	Substract(5, 3)	2
TC2	Soustraction d'un nombre positif et un négatif	Substract(5, -3)	8
TC3	Soustraction de deux nombres négatifs	Substract(-5,-3)	-2

#### **3.3. Test de la Méthode multiply**

Description : Teste la multiplication de deux nombres.

ID de test	Description du test	Entrée	Sortie attendue
TC1	Multiplication de deux nombres positifs	multiply (5, 3)	15

TC2	Multiplication d'un nombre positif et un négatif	multiply (5, -3)	-15
TC3	Multiplication de deux nombres négatifs	multiply(-5,-3)	-15

### **3.4. Test de la Méthode divide**

Description : Teste la division de deux nombres.

ID de test	Description du test	Entrée	Sortie attendue
TC1	Division de deux nombres positifs	divide(15, 3)	5
TC2	Division d'un nombre positif et un négatif	divide (15, -3)	-5
TC3	Division de deux nombres négatifs	divide (-15,-3)	5
TC13	Division par zéro (doit lancer une erreur)	divide(10, 0)	Erreur lancée

### **3.5. Test de la Méthode fetchData**

Description : Teste l'opération asynchrone fetchData.

ID de test	Description du test	Entrée	Sortie attendue
TC1	Vérifie le retour de la promesse	fetchData()	"Data fetched"

## **CONCLUSION**

Les tests statiques sont essentiels pour maintenir une haute qualité de code et détecter les erreurs dès les premières étapes du développement. En utilisant des méthodes manuelles et des outils automatisés, les développeurs peuvent s'assurer que leur code est propre, efficace, et conforme aux normes de codage, ce qui réduit les coûts de maintenance et améliore la fiabilité des logiciels.