Tests et Qualité Logicielle

Voici la section sur les **Tests et Qualité Logicielle**, accompagnée d'exemples et d'explications pour chaque concept :

---

1. Test-Driven Development (TDD)

Définition : TDD est une approche de développement où les tests sont écrits avant le code. Cela aide à s'assurer que le code répond aux exigences dès le départ.

Exemple de Code

JavaScript avec Jest :

```
// Test (test.js)
describe('Addition function', () => {
    it('should add two numbers correctly', () => {
        expect(add(1, 2)).toBe(3);
    });
});

// Fonction (app.js)
function add(a, b) {
    return a + b; // Fonction à tester
}
```

- Ici, le test est écrit d'abord pour la fonction `add`, qui est ensuite créée pour répondre à ce test.

Java avec JUnit :

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

public class CalculatorTest {
    @Test
    public void testAdd() {
        assertEquals(3, Calculator.add(1, 2)); // Test avant la création de la
méthode
    }
}

// Fonction (Calculator.java)
public class Calculator {
    public static int add(int a, int b) {
        return a + b; // Fonction à tester
```

```
    }
}
```

- Le même principe s'applique, avec les tests écrits avant le développement de la méthode `add`.

2. Unit Testing

Définition : Les tests unitaires sont des tests qui vérifient le comportement d'une petite unité de code, généralement une fonction ou une méthode. Ils permettent de s'assurer que chaque partie du code fonctionne comme prévu.

Exemple de Code

JavaScript avec Jest :

```
describe('Subtraction function', () => {
    it('should subtract two numbers correctly', () => {
        expect(subtract(5, 2)).toBe(3); // Test unitaire
    });
});

function subtract(a, b) {
    return a - b; // Fonction à tester
}
```

Java avec JUnit :

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

public class SubtractionTest {
    @Test
    public void testSubtract() {
        assertEquals(3, Calculator.subtract(5, 2)); // Test unitaire
    }
}

// Fonction (Calculator.java)
public class Calculator {
    public static int subtract(int a, int b) {
        return a - b; // Fonction à tester
    }
}
```

3. Integration Testing

Définition : Les tests d'intégration vérifient que plusieurs composants ou systèmes fonctionnent correctement ensemble. Cela peut inclure des tests sur des bases de données, des API ou d'autres systèmes externes.

Exemple de Code

JavaScript avec Jest :

```
describe('API Integration', () => {
    it('should return data from the API', async () => {
        const data = await fetchDataFromAPI(); // Fonction à tester
        expect(data).toBeDefined(); // Vérifie que les données sont définies
    });
});

async function fetchDataFromAPI() {
    const response = await fetch('https://api.example.com/data');
    return response.json(); // Fonction à tester
}
```

Java avec JUnit :

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

public class ApiIntegrationTest {
    @Test
    public void testFetchData() {
        String data = ApiClient.fetchData(); // Fonction à tester
        assertNotNull(data); // Vérifie que les données ne sont pas null
    }
}

// Fonction (ApiClient.java)
public class ApiClient {
    public static String fetchData() {
        // Appel à l'API
        return "data"; // Simule une réponse de l'API
    }
}
```

4. Code Reviews et Pair Programming

Définition : Les revues de code et le pair programming sont des pratiques collaboratives qui améliorent la qualité du code. La revue de code consiste à faire examiner le code par un autre développeur avant de l'intégrer. Le pair programming implique deux développeurs travaillant ensemble sur le même code, ce qui favorise l'échange d'idées et la détection des erreurs.

Exemple de Processus

Code Review :

1. Un développeur écrit du code et le soumet à une revue de code.
2. Un ou plusieurs développeurs examinent le code, commentent et suggèrent des améliorations.

3. Les modifications sont apportées en fonction des commentaires, et le code est fusionné dans la base de code principale.

Pair Programming :

- Un développeur, le "driver", tape le code, tandis que l'autre, le "observer", examine et propose des idées.
- Cela permet une discussion continue sur le design et la logique, améliorant ainsi la qualité et la compréhension du code.