Annexe

Table des matières

Annexe

1 Cours sur les Tests Unitaires et JUnit

1.1 Introduction aux Tests Unitaires

Les tests unitaires sont une pratique essentielle en développement logiciel. Ils permettent de vérifier qu'une unité de code (généralement une fonction ou une méthode) fonctionne correctement. Dans ce cours, nous allons explorer :

— La définition et l'importance des tests unitaires.
— Le framework JUnit pour les tests en Java.
— La gestion des cas où des informations manquent ou doivent être ajoutées.

1.2 Qu'est-ce qu'un test unitaire ?

Un test unitaire est un test automatisé qui valide le fonctionnement d'une unité spécifique de code. Une unité est souvent définie comme :

— Une méthode dans une classe.
— Une fonction dans un programme.

1.2.1 Pourquoi effectuer des tests unitaires ?

Les tests unitaires présentent plusieurs avantages :

— **Détection précoce des bugs :** Les erreurs sont identifiées avant que le code ne soit intégré à d'autres modules.
— **Facilité de maintenance :** Lorsqu'une modification est apportée au code, les tests garantissent que cela n'introduit pas de nouvelles erreurs.
— **Documentation du code :** Les tests servent de référence pour comprendre comment chaque méthode est censée se comporter.

1.3 Présentation de JUnit

JUnit est un framework de test en Java largement utilisé pour écrire et exécuter des tests unitaires. Il est léger, rapide et facilement intégrable avec des outils modernes.

1.3.1 Annotations importantes dans JUnit

Voici les principales annotations utilisées dans JUnit :

— `@Test` : Identifie une méthode comme un test unitaire.
— `@BeforeEach` : Méthode exécutée avant chaque test, utile pour initialiser les données.
— `@AfterEach` : Méthode exécutée après chaque test, souvent utilisée pour nettoyer les ressources.
— `@BeforeAll` : Méthode exécutée une seule fois avant tous les tests.
— `@AfterAll` : Méthode exécutée une seule fois après tous les tests.
— `@Disabled` : Permet de désactiver temporairement un test.

1.3.2 Assertions principales

Les assertions sont des méthodes qui vérifient les résultats des tests :

- `assertEquals(expected, actual)` : Vérifie que la valeur réelle est égale à la valeur attendue.
- `assertTrue(condition)` : Vérifie qu'une condition est vraie.
- `assertFalse(condition)` : Vérifie qu'une condition est fausse.
- `assertThrows(exception.class, () -> {...})` : Vérifie qu'une exception spécifique est levée.

1.4 Exemple d'utilisation de JUnit

Voici un exemple simple qui teste une méthode d'addition dans une calculatrice.

1.4.1 Code de la classe `Calculator`

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

1.4.2 Code du test unitaire

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

class CalculatorTest {
    @Test
    void testAddition() {
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 3);
        assertEquals(5, result); // V rifie que 2 + 3 = 5
    }
}
```

2 Les Mock Objects avec Mockito

2.1 Qu'est-ce qu'un Mock Object ?

Un **mock object** (objet fictif) est une **simulation** d'une dépendance externe utilisée dans les tests unitaires. Il permet de simuler le comportement de cette dépendance sans exécuter réellement le code associé. Les mocks permettent ainsi de contrôler le comportement des dépendances et d'isoler la partie de code que l'on souhaite tester.

2.1.1 Pourquoi utiliser des Mock Objects ?

Les mocks sont utilisés principalement pour :

— **Isoler les tests** : Lorsqu'une classe ou une méthode dépend d'autres ressources externes (comme une base de données ou une API), il est difficile de tester avec ces dépendances réelles. Les mocks permettent de simuler ces ressources et de tester uniquement la logique de la classe.

— **Contrôler les retours des dépendances** : On peut définir exactement ce que les mocks doivent retourner (par exemple, une valeur spécifique ou une exception) pour tester différents scénarios.

— **Éviter les effets secondaires** : Les dépendances peuvent avoir des effets secondaires indésirables, comme envoyer des e-mails ou modifier une base de données réelle. Les mocks évitent ces effets secondaires pendant les tests.

2.2 Introduction à Mockito

Mockito est un framework Java populaire permettant de créer des **mock objects** dans le cadre des tests unitaires. Il est utilisé pour simuler les dépendances d'une classe ou d'une méthode. Grâce à Mockito, nous pouvons simuler des comportements complexes et vérifier si des méthodes ont été appelées, de manière simple et efficace.

2.2.1 Fonctionnalités principales de Mockito

Mockito offre plusieurs fonctionnalités essentielles pour travailler avec les mocks :

— **Création de mocks** : Mockito permet de créer facilement des mocks pour les dépendances d'une classe.

— **Stubbing** : Il permet de spécifier ce que les mocks doivent retourner lorsqu'une méthode est appelée.

— **Vérification** : Il permet de vérifier si certaines méthodes ont bien été appelées sur les mocks, ce qui est utile pour tester des interactions.

2.3 Exemples de base avec Mockito

Création d'un Mock :

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

public class UserServiceTest {
    @Test
    public void testGetUser() {
        UserRepository userRepository = mock(UserRepository.class);
        User mockUser = new User("John", "Doe");
```

```
        when(userRepository.findById(1)).thenReturn(mockUser);

        UserService userService = new UserService(userRepository);
        User user = userService.getUser(1);

        assertEquals("John", user.getFirstName());
    }
}
```

Conclusion

En conclusion, les tests unitaires et les mock objects jouent un rôle central dans le développement logiciel moderne. En utilisant des frameworks tels que JUnit et Mockito, les développeurs peuvent :

— Garantir la fiabilité et la qualité de leur code.
— Identifier et corriger rapidement les erreurs dans les unités individuelles.
— Simuler efficacement des dépendances externes pour isoler les tests.

Les tests unitaires favorisent une approche proactive de la gestion des bugs et permettent une documentation claire des fonctionnalités. Avec l'intégration des mock objects, il devient possible de tester des scénarios complexes tout en minimisant les risques liés aux dépendances externes. Ces outils constituent donc des atouts indispensables pour produire un logiciel robuste, maintenable et évolutif.