Java Avancé

Cours 2 : Tests Unitaires

Arsène Lapostolet & Nada Nahle

Tests unitaires en Java

Tests automatiques

- Du code qui vérifie le fonctionnement de l'application
- Détecter le bugs majeurs en 1 clic
- On est plus confiant quand on doit changer le code

Meilleure qualité & fiabilité du logiciel

Tests unitaires

- Fin niveau de granularité
- Teste une **unité** de code

Unité : méthode, classe, petit groupe de classe ayant un fort lien logique.

Aide a trouver les bugs, mais ne permet pas de dire qu'il n'y en a pas

Structure d'un test

- **Arrangement** : mis en place
- **Action** : exécuter le code que l'on veut tester
- **Affirmation** : vérifier que le résultat est bien le bon

Etant donné <arrangement>, quand <action>, alors <affirmation>

Tests avec JUnit 5

- Méthode de tests: Méthode avec l'annotation `@Test`
- Suite de tests: Classe contenant des méthodes de test
- Une suite de test par **Unité**
- Classes rangées sous `src/test/java`

Méthode de test

```java
@Test
void wordCount_whenMultipleWords_returnsRightCount(){
    // Given
    var input = "bonjour le monde";

    // When
    var result = App.countWords(input);

    // Then
    assertEquals(3, result);
}
```

Cas de test

Des points pivots divisent les flux d'exécution potentiels.

Points pivots :

- `if`
- `switch`

Analyser ce qui a du sens d'un PDV fonctionnel

Des questions ?

Pseudo Entités

Pseudo Entités

Utilité : remplacer une dépendance pour faciliter les tests.

- **Faux (fake)** : implémentation cohérente mais simplifiée
- **Simulacre (mock)** : coquille vide avec un comprtement paramètré

Faux (fake)

```java
public class FakeRepository implements UserRepository {

    private final Map<String, User> data;

    public FakeUsersRepository(Map<String, User> data){
        this.data = data;
    }

    public User findByUsername(String username){
        return this.data.get(username);
    }
}
```

Simulacre (mock) avec Mockito

```java
public interface GithubApiClient {
    List<GithubRepo> getUserRepository(String username);
}
```

```java
final var apiClientMock = mock(GithubApiClient.class);
when(apiClientMock.getUserRepository(testUsername))
    .thenReturn(
        List.of(
            new GithubRepo("test repo 1", 32),
            new GithubRepo("test repo 2", 12)
        )
    );
```

Des questions ?

Qualité de tests

Conception de tests et couplage

Ecrire un test sanctuarise une interface (si on refactor on doit refactorer le test aussi)

Cela a du sens pour :

- Algorithmes
- IO
- Frontière des unités

La conception des unités est importante

Code testable

Injection de dépendances : Externaliser des comportement dans des classes (dépendances), et les fournir en paramètre du constructeur.

Inversion de dépendance : Abstraire les dépendance par un contrat de service (une interface)

- Code modulaire
- Couplages moins forts

-> Le code est plus facile et test, maintenir, refactorer

Des questions ?

Développement dirigé par les tests (TDD)

Développement dirigé par les tests (TDD)

TDD (tests driven development) : Méthode de développment “test-first”.

Cycle RED-GREEN-REFACTOR :

- RED: écrire un test qui ne passe pas
- GREEN: écrire le code minimal qui suffit à faire passer le test
- REFACTOR: retravailler le code écrit pour l’améliorer

Pourquoi le TDD ?

- **Vitesse:** valider plus vite les idées, passer moins de temps à débugger manuellement
- **Confiance:** tests + fiables et pertinents, vraie spécification exécutable. Meilleure sécurité contre la régression
- **Qualité :** force la réflexion autour des interfaces, on détecte ainsi les problèmes de conception plus tôt. On est forcé à refactorer plus souvent, donc on produit du meilleur code

Des questions ?