
Java Avancé

Cours 2 : Tests Unitaires

Arsène Lapostolet

22 Janvier 2024

Tests automatiques

- Du code qui vérifie le fonctionnement de l'application
- Détecter les bugs majeurs en 1 clic
- Plus sûr de 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 à trouver les bug, 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

```
1  @Test
2  void wordCount_whenMultipleWords_returnsRightCount(){
3      // Given
4      var input = "bonjour le monde";
5
6      // When
7      var result = App.countWords(input);
8
9      // Then
10     assertEquals(3, result);
11 }
```

Java

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

Conception de tests et couplage

Ecrire un test -> sanctuariser 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

Pseudo Entités

Utilité : remplacer une dépendance pour faciliter les tests.

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

Faux (fake)

```
1 public class FakeRepository implements UserRepository { Java
2
3     private final Map<String, User> data;
4
5     public FakeUsersRepository(Map<String, User> data){
6         this.data = data;
7     }
8
9     public User findByUsername(String username){
10         return this.data.get(username);
11     }
12 }
```

Simulacre (mock) avec Mockito

```
1 public interface GithubApiClient {
2     List<GithubRepo> getUserRepository(String username);
3 }
```

Java

```
1 final var apiClientMock = mock(GithubApiClient.class);
2 when(apiClientMock.getUserRepository(testUsername))
3     .thenReturn(
4         List.of(
5             new GithubRepo("test repo 1", 32),
6             new GithubRepo("test repo 2", 12)
7         )
8     );
```

Java

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*

Développement dirigé par les tests (TDD)

TDD (tests driven development) : Méthode de développement “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

Pour le TDD ?

- **Vitesse:** valider plus vite les idées, passer moins de temps à débbugger 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