DIGINAMIC

FORMATION
Java, tests unitaires et bonnes pratiques - 1 JOUR

PLAN DE COURS

Chapitre 1 : Notions générales

Chapitre 2 : Découverte de JUnit

Chapitre 3 : Organisation et bonnes pratiques

Chapitre 4 : Mocking avec Mockito

Chapitre 5 : Qualité et couverture des tests

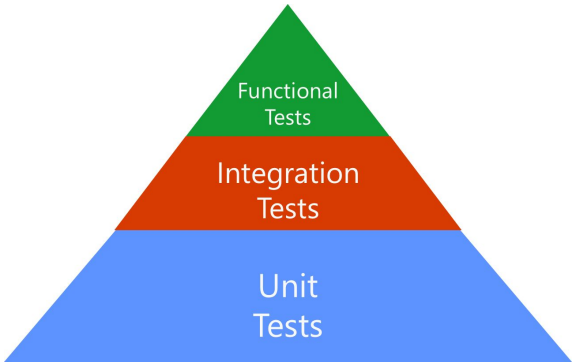
Chapitre 1
Notions générales

Pourquoi tester ?

- Améliorer la qualité du code : détection rapide des anomalies
- Faciliter la maintenance : les tests garantissent que les modifications n’introduisent pas de régressions.
- Fiabiliser le logiciel avant la mise en service.
- Instaurer une confiance : développeurs et clients sont rassurés sur la robustesse du produit.

Conseils pour tester son code

- Adopter une approche structurée
- Couvrir différents types de tests
- S’assurer que toutes les parties fonctionnent comme prévu



Tests fonctionnels

- Vérifient le comportement de l'application du point de vue de l'utilisateur final
- Simulent des actions utilisateur, comme remplir un formulaire ou naviguer sur le site
- Outils : Selenium, Playwright, Cypress

Tests d’intégration

- Vérifient que les différentes parties de l'application fonctionnent bien ensemble.
- Simulent l’interaction entre les couches (contrôleurs, BDD, services…).

Tests unitaires

- Ciblent des parties spécifiques du code, comme des méthodes spécifiques.
- Vérifient que la logique métier fonctionne correctement de manière isolée.

Focus sur les tests unitaires

- Vérifie qu'une fonction ou une méthode se comporte comme attendu.
- Caractéristiques :
  - isolé (sans dépendances externes),
  - rapide à exécuter,
  - déterministe (mêmes résultats à chaque exécution)

Exemple : tester qu'une fonction addition(a, b) retourne toujours a + b.

Tests unitaires : bonnes pratiques

- Automatiser les tests (CI/CD)
- Rédiger des tests simples, lisibles et rapides
- Couvrir les cas normaux et les cas limites/erreurs.
- Maintenir les tests au même titre que le code.
- Utiliser la couverture de code (code coverage) comme indicateur, mais pas comme objectif absolu.

Limites des tests unitaires

- Ne garantissent pas l’absence de bugs.
- Ne testent pas l’interaction entre plusieurs modules.
- Peuvent devenir coûteux si mal écrits ou trop nombreux
- Risque de tests fragiles (trop liés aux détails d’implémentation)

Frameworks de tests unitaires

JUnit

NUnit, xUnit

Jest, Mocha,
Jasmine

unittest,
pytest

PHPUnit,
Pest

Chapitre 2
Découverte de JUnit

Le framework JUnit pour Java

- Framework de test open source pour Java.
- Standard dans l'écosystème Java.
- S'intègre aux IDE (Eclipse, IntelliJ, VS Code…)
- Compatible avec Maven, Gradle et CI/CD

Rôle de JUnit

- Automatiser l’exécution des tests unitaires.
- Fournir des assertions pour valider les résultats
- Générer des rapports de succès/échec.
- Faciliter le TDD (Test Driven Development).

Installation de JUnit

- Avec Maven : https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html
- Puis mvn test pour lancer les tests

Annotations principales

- @Test : méthode de test.
- @BeforeEach : exécute du code avant chaque test.
- @AfterEach : nettoyage après chaque test.
- @BeforeAll / @AfterAll : initialisation/fermeture globale.

- https://docs.junit.org/current/user-guide/

Assertions

- assertEquals(expected, actual) : compare deux valeurs.
- assertNotEquals(unexpected, actual) : vérifie que les valeurs sont différentes.
- assertTrue(condition) / assertFalse(condition) : vérifie une condition booléenne.
  assertNull(object) / assertNotNull(object) : vérifie si un objet est (non) nul.
  assertSame(expected, actual) / assertNotSame() : vérifie si deux références pointent le même objet.
- assertThrows(Exception.class, () -> {...}) : vérifie qu'une exception est levée.
- fail("message") : force l'échec d'un test (utile si un cas inattendu se produit).

- https://docs.junit.org/5.0.1/api/org/junit/jupiter/api/Assertions.html

Premier test d’une classe

Exemple de test

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

class UserTest {

    @Test
    void testUserName() {
        User user = new User("Alice");

        assertNotNull(user.getName());        // Vérifie que le nom n'est pas null
        assertEquals("Alice", user.getName()); // Vérifie que le nom est correct
    }
}
```

Gestion des cas d’erreurs / exceptions

- On utilise assertThrows pour vérifier qu’une exception est levée.
- Vérifie le type d’exception et éventuellement le message.

Un second test

Exempe de test

```java
// Fichier MathUtils.java
public class MathUtils {

    public int divide(int a, int b) {
        return a / b; // Lève ArithmeticException si b == 0
    }
}

// MathUtilsTest.java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class MathUtilsTest {

    @Test
    void testDivisionValid() {
        MathUtils math = new MathUtils();
        int result = math.divide(10, 2);
        assertEquals(5, result);  // Vérifie que 10 / 2 = 5
    }

    @Test
    void testDivisionByZero() {
        MathUtils math = new MathUtils();
        assertThrows(ArithmeticException.class, () -> {
            math.divide(10, 0);  // Provoque une ArithmeticException
        });
    }
}
```

Exercice

Chapitre 3
Organisation et bonnes pratiques

digiamic FORMATION

Convention de nommage des classes de test

- Nommer la classe de test comme la classe testée + Test.
- Exemple : le test de MathUtils est MathUtilsTest
- Placer les tests dans un package/dossier séparé (src/test/java).
- Conserver une structure parallèle à celle du code source.

Convention de nommage des méthodes de test

- Nom explicite décrivant le comportement attendu.

- Formats possibles :
  - methodName_expectedBehavior()
  - Exemple : divide_returnsQuotient()
  - shouldExpectedBehavior_whenCondition()
  - Exemple : shouldThrowException_whenDividingByZero()

- Éviter test1(), fooTest() → pas assez parlants.

Organisation du code de test

- Utiliser @BeforeEach / @AfterEach pour initialiser et nettoyer les ressources.
- Grouper les tests par fonctionnalité.
- Garder chaque méthode de test courte et lisible (au maximum).

Approche Given / When / Then dans un test

- Given : état initial, préparation des données
- When : action exécutée
- Then : résultat attendu, vérification avec des assertions

Exemple

```java
@Test
void shouldReturnQuotient_whenDividingTwoNumbers() {
    // Given : un objet MathUtils et des valeurs d'entrée
    MathUtils math = new MathUtils();
    int a = 20;
    int b = 4;

    // When : on exécute la méthode
    int result = math.divide(a, b);

    // Then : on vérifie le résultat attendu
    assertEquals(5, result);
    assertTrue(result > 0);     // Vérifie une propriété
    assertNotNull(result);      // Vérifie que le résultat n'est pas nul
}
```

Tests paramétrés avec JUnit

- Permettent d'exécuter le même test avec plusieurs valeurs.
- Évitent la duplication du code de test.
- Utilisent l'annotation @ParameterizedTest.
- Sources de paramètres :
  - @ValueSource (valeurs simples)
  - @CsvSource (valeurs combinées)
  - @MethodSource (méthodes qui fournissent les données)

Tests paramétrés : exemple

Exemple

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

class MathUtilsTest {

    @ParameterizedTest
    @ValueSource(ints = {2, 4, 6, 8})
    void shouldBeEvenNumbers(int number) {
        assertEquals(0, number % 2);
    }
}
```

Exercice

Chapitre 4
Mocking avec Mockito

Introduction au mocking

- Permet de simuler une dépendance (un objet factice)
- Utilisé quand :
    - La vraie dépendance est lente, ou complexe, ou externe
    - On veut totalement isoler une classe, un comportement
- Très utile pour contrôler le comportement et les réponses de la dépendance ciblée

Pourquoi mocker dans les tests unitaires ?

- Isolation des tests : tester une classe sans sa base de données, API, etc.
- Rapidité : pas besoin de connexion externe à la dépendance
- Fiabilité : mêmes résultats à chaque exécution (principe majeur d'un TU)
- Focus : on teste la logique métier, pas l'environnement.

Mock, Stub et spy : différences

- Stub : objet avec réponse prédéfinie (pas intelligent).
- Mock : objet simulé qui peut être programmé et vérifié.
- Spy : objet réel que l'on peut partiellement surveiller.

Introduction à Mockito

- Framework de mocking pour Java facile à intégrer à JUnit.
- Annotations principales :
  - @Mock permet de créer un mock
  - @InjectMocks pour injecter les mocks dans la classe testée.
  - @ExtendWith(MockitoExtension.class) pour activer Mockito avec JUnit.

Mockito en pratique : when()

- Définit le comportement d’un mock
- Permet de simuler la dépendance externe

Exemple

```
when(api.getData()).thenReturn("fake data");
```

Mockito en pratique : verify()

- Permet de vérifier qu'un mock a bien été utilisé comme prévu.
- Contrôle les interactions (appels de méthodes, nombre de fois, arguments).

Exemple

```
@Test
void shouldCallApiOnce() {
    ApiClient mockClient = mock(ApiClient.class);
    ApiService service = new ApiService(mockClient);

    service.fetchData();

    verify(mockClient).getData();              // Vérifie l'appel
    verify(mockClient, times(1)).getData();    // Vérifie le nombre d'appels
}
```

Exemple : test d’un service API

Exemple

```
class ApiService {
    private final ApiClient client;
    ApiService(ApiClient client) { this.client = client; }

    String fetchData() {
        return client.getData().toUpperCase();
    }
}
```

Exemple

```
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.mockito.junit.jupiter.MockitoExtension;
import org.junit.jupiter.api.extension.ExtendWith;

@ExtendWith(MockitoExtension.class)
class ApiServiceTest {

    @Test
    void shouldReturnUppercaseData() {
        ApiClient mockClient = mock(ApiClient.class);
        when(mockClient.getData()).thenReturn("hello");

        ApiService service = new ApiService(mockClient);

        assertEquals("HELLO", service.fetchData());
        verify(mockClient).getData(); // Vérifie l’appel
    }
}
```

Exercice

Chapitre 5
Qualité et couverture de tests

La couverture de code avec JaCoCo

- Outil de mesure de code coverage (% du code couvert par des tests) pour Java.
- Intégré avec JUnit pour analyser quels morceaux de code sont exécutés.
- Rapports disponibles en HTML / XML.
- S'utilise en ligne de commande, avec IDE, ou en CI/CD.

Types de couverture de code

- Des lignes : pourcentage de lignes exécutées.
- Des branches : pourcentage des chemins (if/else, switch).
- Des instructions : granularité plus fine.
- Des méthodes/classes : vérifie que tout est invoqué.

Les limites de la couverture

- 100% couverture ≠ 0 bug
- Exemples :
  - Cas non vérifiés (valeurs limites, exceptions)
  - Tests superficiels (juste exécuter sans vérifier la pertinence du test)

La couverture est un indicateur, pas un objectif absolu.

Intégration JaCoCo + JUnit en CI/CD

- Génération automatique des rapports :
  - Exécution des tests JUnit puis JaCoCo collecte les données.
  - Génération rapport HTML lisible par devs.

- Suivi dans le temps :
  - CI/CD (Jenkins, GitHub Actions, GitLab CI) = exécution à chaque commit.
  - Rapports stockés permettant le suivi de l'évolution du code coverage
  - Permet d'éviter la régression de la qualité des tests.

Intégration : exemple

Exercice

TP : Validation des acquis