Les bases du test logiciel

Pr. BE.ELBAGHAZAOUI

Qu’est-ce que le test logiciel?

Définition : Le test logiciel est un **processus systématique d'évaluation d'un produit logiciel** pour déterminer si il répond aux **exigences**, s'il fonctionne comme prévu et si il est prêt pour son utilisation finale.

- Le test logiciel, c’est une quête constante de la qualité.
- C’est une série de vérifications et de validations visant à garantir la qualité et la fiabilité du produit avant sa mise en production.

Objectifs Clés :

- Vérifier la conformité aux exigences
- Identifier et rapporter les défauts
- Évaluer la performance, la sécurité et l'expérience utilisateur
- Améliorer la qualité globale du logiciel

Quels rôles jouent les tests logiciels dans le cycle de développement logiciel ou d'une application?

- **Assurance de la Qualité** : Garantir que le logiciel répond aux normes de qualité
- **Détection Précoce des Erreurs** : Identifier les défauts tôt dans le cycle de développement pour réduire les coûts de correction
- **Amélioration Continue** : Fournir des retours pour améliorer les processus de développement et le produit lui-même
- **Support à la Livraison** : Aider à déterminer si le logiciel est prêt pour la mise sur le marché

Quels sont les missions du testeur?

- **Évaluation du Logiciel** : Exécuter des tests pour évaluer sa qualité et son fonctionnement
- **Rapport et Suivi des Défauts** : Identifier, documenter et suivre les erreurs jusqu'à leur résolution
- **Collaboration avec les Équipes** : Travailler en étroite collaboration avec les développeurs, les PO (Product Owners) et d'autres parties prenantes pour assurer l'alignement et l'amélioration continue
- **Contribution à l'Amélioration des Processus** : Fournir des retours pour améliorer les processus de test et de développement

La Qualité

Définition : La qualité fait référence au degré avec lequel un produit, un service ou un processus répond **aux attentes et exigences des utilisateurs ou des parties prenantes.**

Aspects clés :
- Conformité aux normes.
- Satisfaction client.
- Amélioration continue.

Importance :
- Fidélisation des clients.
- Réduction des coûts liés aux erreurs.
- Avantage concurrentiel.

Tests Statiques

Définition

- Les **tests statiques** sont des tests sans exécution de code.
- Ils s'appuient sur la relecture, l'inspection et la validation de livrables ou de documents produits avant et pendant le développement.

Exemples

1. **Relecture de spécifications fonctionnelles ou user stories**
   Permet de vérifier la cohérence et la clarté des exigences.
2. **Revue de code (Code Review)**
   Permet de détecter des erreurs de logique, de style, ou de sécurité avant l'intégration.
3. **Audit de bases de données et de diagrammes (UML, architecture, etc.)**
   Vérifie la structure technique et l'adéquation avec les besoins.
4. **Analyse de cahier des charges**
   Assure que toutes les exigences contractuelles sont prises en compte.

Tests Statiques

Avantages

- **Détection précoce des défauts** : Le coût de correction est plus faible quand un problème est repéré en amont.
- **Gain de temps en phase de développement** : Moins de retouches au code si les spécifications sont correctes.
- **Qualité documentaire améliorée** : Favorise la bonne compréhension du projet par toute l'équipe.

Limitations

- Ne vérifient pas le comportement réel de l'application.
- Nécessitent une bonne connaissance du domaine et une rigueur dans la lecture des documents.

Tests Dynamiques

Définition

- Les **tests dynamiques** impliquent l'exécution du code.
- Ils évaluent le comportement réel du produit, notamment lors de scénarios d'utilisation.

Types de tests dynamiques (exemples majeurs)

1. **Tests unitaires**
   - Vérifient le bon fonctionnement d'une fonction, d'une classe ou d'un composant isolé.
   - Rapides à exécuter et généralement automatisés.
2. **Tests d'intégration**
   - Vérifient l'interaction entre plusieurs composants ou modules.
   - S'assurent que les données circulent correctement.
3. **Tests système**
   - Évaluent l'ensemble du logiciel dans son intégralité.
   - Contrôlent le respect des spécifications globales (fonctionnelles et non-fonctionnelles).

4. **Tests end-to-end (E2E)**
   - Vérifient la totalité d'un parcours utilisateur, du front-end au back-end.
   - Simulent un scénario complet (ex. : achat en ligne, inscription, etc.).
5. **Tests de régression**
   - Assurent qu'aucun bug n'a été introduit après une modification ou une mise à jour.
   - Souvent automatisés pour être exécutés fréquemment.
6. **Tests de performance et de charge**
   - Mesurent les temps de réponse, la stabilité, la robustesse sous diverses charges.
   - Cruciaux pour les applications à trafic élevé ou à enjeux critiques (finances, santé, etc.).
7. **Tests de sécurité**
   - Visent à détecter les vulnérabilités (injection SQL, XSS, CSRF, etc.).
   - Protègent les données et garantissent la conformité (RGPD, PCI-DSS, etc.).

Tests Dynamiques

Avantages

- **Validation du comportement réel** : Permet d'identifier les bugs dans des conditions proches de la production.
- **Réduction des risques** : Garantit que le produit répond correctement aux exigences fonctionnelles et non-fonctionnelles.

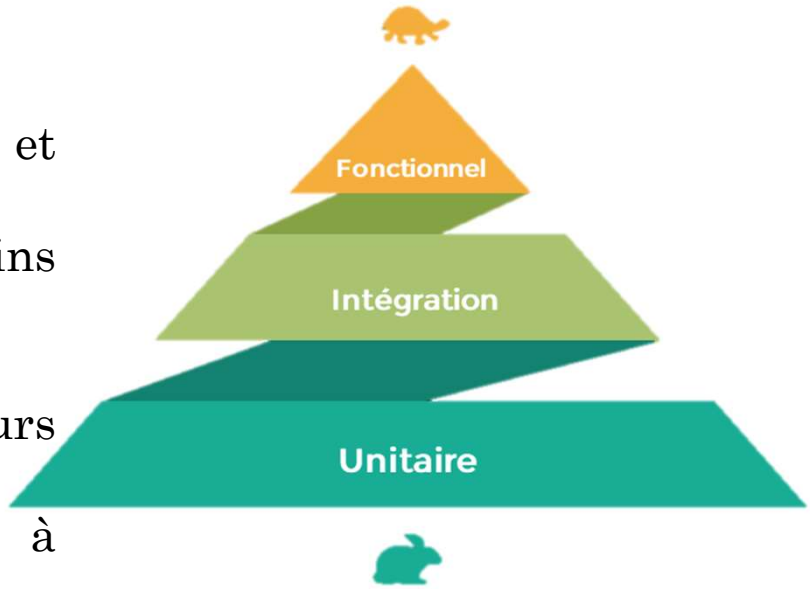
Limitations

- **Coût et temps d'exécution** : Souvent plus longs et plus complexes à mettre en place (surtout E2E).
- **Maintenance** : Les tests doivent évoluer avec le produit (risque de "tests cassés" si l'application change souvent).

La Pyramide de Mike Cohn

Pyramide de Mike Cohn illustre comment répartir et prioriser les types de tests automatisés pour optimiser la qualité du logiciel.

1. **Base : Les Tests Unitaires**
   - Représentent la plus grande partie des tests.
   - Rapides, faciles à automatiser, couvrent les briques logiques élémentaires.
2. **Niveau intermédiaire : Tests d'Intégration / API**
   - Vérifient la communication entre les composants et services (ex. microservices, API).
   - Plus complexes que les tests unitaires, mais moins que les E2E.
3. **Sommet : Tests End-to-End (E2E)**
   - Vérifient des flux métier complets (parcours utilisateur).
   - Généralement plus lents et plus coûteux à maintenir.
   - Représentent une plus petite partie de l'automatisation, mais sont essentiels pour la validation finale.



Tests Unitaires

Pr. BE.ELBAGHAZAOUI

Qu'est-ce qu'un Test Unitaire ?

Définition :
Un test unitaire vérifie le **bon fonctionnement d'une unité de code** (une méthode, une classe) de manière isoler.

Caractéristiques :
- Rapides à exécuter
- Automatisables
- Confiance dans le code

Exemple simple :
Tester la méthode

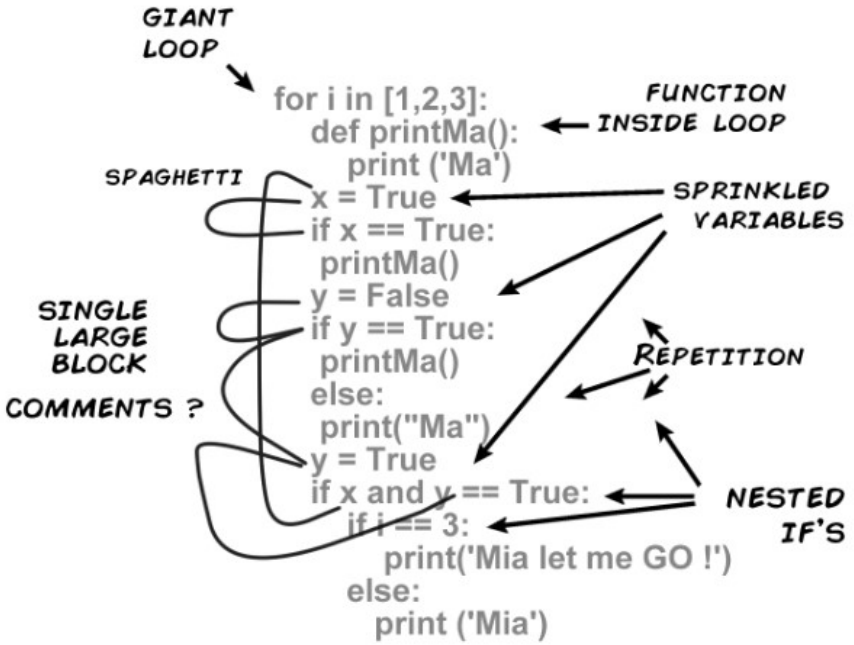
calculateSum(int a, int b)

et vérifier qu'elle renvoie

$a + b$

Pourquoi Tester ?

1. **Détection rapide de bugs**
   Permet de trouver des problèmes avant qu'ils n'impactent la production.
2. **Qualité du code améliorée**
   Encourage la conception modulaire et évite le *spaghetti code*.
3. **Maintenabilité**
   - Les tests servent de documentation.
   - Facilitent les refactorings futurs.
4. **Confiance de l'équipe**
   Évite la "peur de casser" du code existant.



Junit (Java)

Définition :

- Framework de test unitaire le plus utilisé en Java.
- Intégré à la plupart des IDE (*Eclipse, IntelliJ, VS Code*) et outils de build (*Maven, Gradle*).

Version :

- Dernière version majeure : JUnit 5 (JUnit Jupiter).

Concepts clés :

- Annotations : *@Test, @BeforeEach, @AfterEach, @BeforeAll, @AfterAll*
- Assertions : *Assertions.assertEquals, assertTrue, assertThrows*

Structure d'un Projet Test

Arborescence classique :

```
src
├── main
│   └── java (code métier)
└── test
    └── java (code de test)
```

- Fichier pom.xml : ajouter la dépendance JUnit 5.

Exemple de dépendance Maven (JUnit 5) :

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.8.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Exemple de Test Basique

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

class CalculatorTest {

    @Test
    void testAddition() {
        Calculator calc = new Calculator();
        int result = calc.add(2, 3);
        assertEquals(5, result, "2 + 3 devrait être égal à 5");
    }
}
```

- **@Test** : Indique que cette méthode est un test.
- **assertEquals** : Vérifie l'égalité entre la valeur attendue et le résultat.

Cycle de Vie des Tests

- **@BeforeAll** : Exécuté une seule fois avant tous les tests (méthode statique).
- **@BeforeEach** : Exécuté avant chaque test (initialisation).
- **@Test** : Méthode de test.
- **@AfterEach** : Exécuté après chaque test (nettoyage).
- **@AfterAll** : Exécuté une seule fois après tous les tests (méthode statique).

```
@BeforeEach
void setup() {
   // Initialiser les ressources pour chaque test
}

@AfterEach
void teardown() {
   // Libérer les ressources
}
```

```java
public class CycleDeVieTest {
    @BeforeAll
    static void setUpBeforeAll() {
        System.out.println("Avant tous les tests");
    }
    @BeforeEach
    void setUp() {
        System.out.println("Avant chaque test");
    }
    @Test
    void testMultiplication() {
        int result = 2 * 3;
        assertEquals(6, result);
        System.out.println("Test de multiplication exécuté");
    }
    @AfterEach
    void tearDown() {
        System.out.println("Après chaque test");
    }
    @AfterAll
    static void tearDownAfterAll() {
        System.out.println("Après tous les tests");
    }
}
```

Tests Paramétrés (Parametrized Tests)

Utilité :

- Tester plusieurs valeurs d'entrée dans un même test.
- Éviter la duplication de code.

Exemple :

```
@ParameterizedTest
@ValueSource(ints = {2, 4, 6, 8})
void testIsEven(int number) {
    assertTrue(Calculator.isEven(number),
               number + " devrait être pair");
}
```

Avantages :

- Gain de temps et de clarté.
- Meilleure couverture de test.

Mocks et Stubs avec Mockito

Problématique et Contexte

Dans les tests unitaires, on veut **isoler** la classe ou la méthode testée.
Or, certaines classes (DAO, services externes, APIs, etc.) sont :

- Complexes à instancier (accès base de données, réseau…).
- Coûteuses en temps d'exécution.
- Parfois non disponibles (service tiers hors-ligne).

Objectif : Éviter de dépendre de ces composants dans nos tests.

Présentation de Mockito

Mockito est le framework de référence pour **mock** en Java.
Avantages :

- Syntaxe fluide et lisible.
- Forte intégration avec JUnit 5 (ou JUnit 4).
- Possibilité de configurer finement les comportements (retours, exceptions).
- Vérification des interactions (méthodes appelées, nombre d'appels, arguments).

Stub

1. **Définition :**
   - Un ***stub*** est un objet factice qui retourne systématiquement des données prédéfinies. Il n'y a pas de logique avancée ou de vérification d'interaction avec l'objet stub.
2. **Utilisation :**
   - Principalement pour **stabiliser** un test en évitant des appels externes (base de données, API) qui peuvent être indisponibles.
   - On veut juste **remplacer** l'implémentation réelle par un résultat **constant**, afin de se concentrer sur le code testé.
3. **Exemple :**
   - Ici, aucune logique ne varie en fonction des paramètres.

```
// Stub renvoie toujours la même liste de produits
class ProductRepositoryStub implements ProductRepository {
    @Override
    public List<Product> findAll() {
        return Arrays.asList(new Product("Pencil"), new Product("Eraser"));
    }
}
```

Mock

1. **Définition :**
   - Un **mock** est un objet factice configurable, dont on peut **vérifier** les interactions (méthodes appelées, nombre d'appels, paramètres, etc.).
2. **Utilisation :**
   - Tester l'**interaction** entre une classe et sa dépendance
   - **Contrôler** précisément les retours de la dépendance
   - **Vérifier** la fréquence et la façon dont les méthodes sont appelées
3. **Exemple** :
   - On configure le résultat retourné et on vérifie que la méthode *findAll()* est appelée exactement une fois.

```
@Mock
private ProductRepository productRepository;
@Test
void testFindAll() {
    // Configuration du mock (retour spécifique)
    when(productRepository.findAll())
        .thenReturn(Arrays.asList(new Product("Pencil"), new Product("Eraser")));
    // Vérification de l'interaction
    verify(productRepository, times(1)).findAll();
}
```

Exemple : Injection et Utilisation d’un Mock

```
@ExtendWith(MockitoExtension.class) // Active l’extension Mockito
class UserServiceTest {
    @Mock // Crée un objet factice pour userRepository
    private UserRepository userRepository;
    @InjectMocks // Injecte automatiquement les mocks dans userService
    private UserService userService;
    @Test
    void testGetUserById() {
        // Configure la valeur de retour du mock
        when(userRepository.findById(1))
            .thenReturn(Optional.of(new User(1, "Alice")));
        // Appelle la méthode à tester
        User user = userService.getUserById(1);
        // Vérifie que l'utilisateur n'est pas null et que son nom est "Alice"
        assertNotNull(user, "L'utilisateur ne doit pas être null");
        assertEquals("Alice", user.getName(), "Le nom de l'utilisateur devrait être
'Alice'");
        // Vérifie que la méthode findById(1) a été appelée une fois
        verify(userRepository, times(1)).findById(1);
    }
}
```

Installer et configurer un outil de build

```xml
<dependencies>
   <dependency>
     <groupId>org.junit.jupiter</groupId>
     <artifactId>junit-jupiter</artifactId>
     <version>5.8.2</version>
     <scope>test</scope>
   </dependency>
   <dependency>
     <groupId>org.mockito</groupId>
     <artifactId>mockito-core</artifactId>
     <version>5.0.0</version>
     <scope>test</scope>
   </dependency>
</dependencies>
```

1. **Lancer les tests**
    1. **Avec Maven :**
        - Depuis la ligne de commande : *mvn test*
        - Ou depuis l'IDE (Eclipse, IntelliJ, VS Code) : faites un clic droit sur le projet ou la classe de test, puis Run as ou Run '*ExampleTest*'.
    2. **Avec Gradle :**
        - Depuis la ligne de commande : *gradle test*
        - Depuis l'IDE, même principe : clic droit et Run.
2. **Analyser les résultats**
    1. Les outils de build (Maven/Gradle) affichent généralement un récapitulatif des tests :
        - Nombre de tests *passés, échoués, ignorés*.
    2. Les IDE proposent un tableau de résultats permettant de visualiser lesquels ont échoué ou réussi.

Tests Intégration

Pr. BE.ELBAGHAZAOUI

Tests d'Intégration

Qu'est-ce qu'un test d'intégration ?

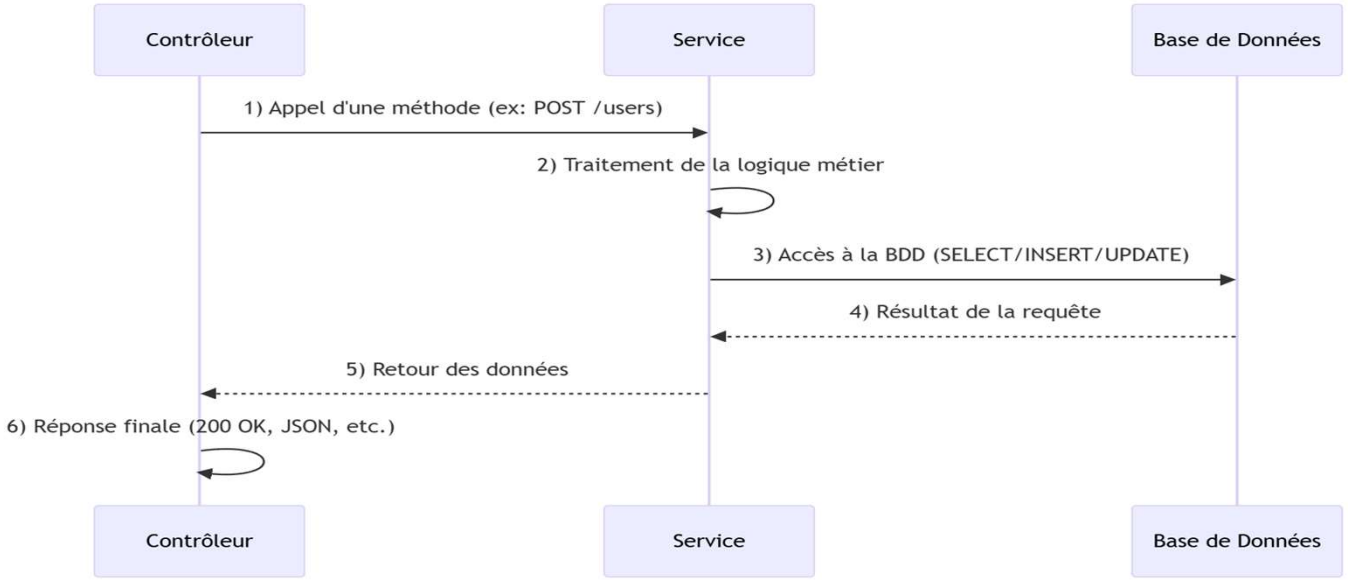
- Vérifie que plusieurs composants ou modules d'une application fonctionnent correctement **ensemble**.

Objectif principal :

- Détecter les **problèmes d'interfaçage** entre les différentes parties du système (base de données, APIs, microservices, etc.).

Exemple :

- Tester le flux de données entre un contrôleur, un service et la base de données.



Pourquoi des Tests d'Intégration ?

1. **Identifier les problèmes de compatibilité**
   - Paramètres incorrects, formats de données non conformes, etc.
2. **Évaluer la cohérence de l'application**
   - Les composants échangent-ils correctement leurs informations ?
3. **Détection de bugs plus tôt**
   - Mieux vaut repérer une anomalie d'intégration avant la phase finale de tests E2E.
4. **Complément des tests unitaires**
   - Même si chaque module fonctionne isolément, leur intégration peut révéler de nouveaux bugs.

Approches Courantes

1. **Test d'Intégration Top-Down**
   - Tester d'abord les modules de plus haut niveau (front-end, contrôleurs),
   - Puis descendre vers la couche service et DAO.
2. **Test d'Intégration Bottom-Up**
   - Partir des composants bas niveau (DAO, services) et remonter vers la couche la plus haute (contrôleurs, UI).
3. **Test d'Intégration Big Bang**
   - Assembler tous les modules en même temps et tester.
   - Rapide à mettre en place, mais **difficile d'isoler l'origine** d'un bug.
4. **Approche Modulaire / Incrementale**
   - Intégrer les modules par petites itérations.
   - Plus systématique et plus facile à diagnostiquer en cas d'erreur.

Outils et Configuration

Spring Boot Test (Java)

Annotation

- *@SpringBootTest* pour charger le contexte Spring.
- *@AutoConfigureMockMvc* pour tester les endpoints REST sans déployer l'app.

TestContainers

Pour lancer des bases de données Docker ou des services externes (Kafka, Elasticsearch) dans un environnement de test isolé.

WireMock

Simule des endpoints HTTP externes.

Database

Utiliser une base de données embarquée (H2) ou un conteneur Docker pour éviter d'impacter la BDD de production.

Conseil : Veiller à isoler l'environnement de test pour qu'il soit reproductible sur toutes les machines.

```java
@SpringBootTest  // Lance l'application Spring Boot, chargement du contexte
@AutoConfigureMockMvc // Active l'injection de MockMvc pour simuler les requêtes HTTP
class UserControllerIntegrationTest {
    @Autowired
    private MockMvc mockMvc; // Pour simuler les appels HTTP
    @Autowired
    private UserRepository userRepository; // Pour valider ce qui est enregistré en BDD
    @BeforeEach
    void setup() {
        userRepository.deleteAll(); // Nettoyage de la base de données avant chaque test
    }
    @Test
    void testCreateUserAndCheckDB() throws Exception {
        // 1) Appeler l'endpoint POST /users pour créer un utilisateur
        mockMvc.perform(post("/users")
                .contentType(MediaType.APPLICATION_JSON)
                .content("{\"name\":\"Alice\", \"email\":\"alice@example.com\"}"))
            // 2) Vérifier la réponse HTTP
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.id").exists())
            .andExpect(jsonPath("$.name").value("Alice"));
        // 3) Vérifier que l'utilisateur est bien enregistré en base
        var usersInDb = userRepository.findAll();
        assertEquals(1, usersInDb.size(), "Un seul utilisateur doit être présent en base");
        assertEquals("Alice", usersInDb.get(0).getName(), "Le nom enregistré doit être 'Alice'");
    }
}
```

Tests Fonctionnels

Pr. BE.ELBAGHAZAOUI

Tests Fonctionnels

Définition :

- Les tests fonctionnels vérifient qu'une ou plusieurs **fonctionnalités** respectent les **spécifications** et fournissent le résultat attendu.

Exemple :

- Tester qu'une fonctionnalité de création de compte utilisateur valide correctement les champs et renvoie le message approprié.

Caractéristiques des Tests Fonctionnels

1. **Basés sur les spécifications**
   - S'appuient sur le *cahier des charges*, les *user stories* ou les *cas d'utilisation*.
2. **Orientés "boîte noire"**
   - Se concentrent sur les **entrées** et **sorties** plutôt que sur le code interne.
3. **Impliquent souvent l'interface utilisateur**
   - Peuvent tester des fonctionnalités à travers l'UI (web, mobile, etc.) ou via un point d'accès applicatif (API).
4. **Critère d'acceptation**
   - Le critère principal est la **conformité** au comportement prévu (fonctionnel).

Outils Courants

1. **Selenium / Cypress / Playwright**
   - Pour automatiser les interactions avec l'interface web.
   - Vérifier le comportement de l'application depuis un navigateur simulé.
2. **Postman / Newman**
   - Pour tester les **APIs** REST (fonctionnalités sans interface graphique).
3. **Cucumber**
   - Pour écrire des tests fonctionnels en **langage Gherkin** (BDD).
   - Facilite la communication entre équipes techniques et non techniques.
4. **JUnit / TestNG** (Java) ou **pytest** (Python)
   - Peuvent aussi être utilisés pour des tests fonctionnels "back-end" si on teste des services métier.

Exemple Simple (Selenium)

```
@Test
void testUserLogin() {
    // 1. Ouvrir le navigateur
    WebDriver driver = new ChromeDriver();
    driver.get("https://mon-application.com/login");
    // 2. Saisir les identifiants
    driver.findElement(By.id("username")).sendKeys("alice");
    driver.findElement(By.id("password")).sendKeys("secret");
    driver.findElement(By.id("loginBtn")).click();
    // 3. Vérifier que la page d'accueil est affichée
    String welcomeMessage =
driver.findElement(By.id("welcomeMsg")).getText();
    assertEquals("Bienvenue Alice!", welcomeMessage);
    // 4. Fermer le navigateur
    driver.quit();
}
```

Introduction aux Tests d'Acceptation

Pr. BE.ELBAGHAZAOUI

Tests Fonctionnels

- **Définition** :Les tests d'acceptation (UAT : User Acceptance Testing) sont menés par les utilisateurs finaux ou des représentants métier afin de valider que le logiciel répond aux exigences et au besoin réel.
- **Objectif** :Confirmer que le produit est prêt à être mis en production, du point de vue fonctionnel et métier.
- **Pourquoi des Tests d'Acceptation ?**

1. **Validation finale** :
   - Permet de s'assurer que les fonctionnalités développées correspondent à la **réalité du terrain**.
2. **Retour d'expérience utilisateur** :
   - Identifier les problèmes d'ergonomie, de performance, ou de compréhension.
3. **Réduction du risque** :
   - Éviter de livrer un produit non conforme aux attentes, réduisant ainsi les retours négatifs ou échecs après mise en production.
4. **Implication des parties prenantes** :
   - Les équipes métier, les clients ou les utilisateurs finaux s'approprient la solution et s'assurent de sa pertinence.

Différents Types d'Acceptation

1. **Acceptation par le Client (Customer Acceptance Testing)**
   - Réalisée par le client ou commanditaire.
   - Valide la conformité au cahier des charges et aux termes du contrat.
2. **Acceptation par l'Utilisateur (UAT)**
   - Menée par un panel d'utilisateurs finaux dans un environnement proche de la production.
   - Vise l'adéquation aux besoins réels et l'ergonomie.
3. **Beta Testing**
   - Version préliminaire du produit (release candidate) testée par un **groupe restreint** d'utilisateurs réels.
   - Retour d'expérience en conditions quasi-réelles.
4. **Acceptation Opérationnelle (Operational Acceptance Testing)**
   - Vérifie la **compatibilité** avec les processus d'exploitation, la sécurité, la performance dans l'environnement de production.

1. Code Source : Le développeur écrit et pousse son code dans le dépôt.
2. Tests Unitaires : Vérifient des composants isolés (fonctions, classes).
3. Tests d'Intégration : Vérifient la communication entre plusieurs modules.
4. Tests Fonctionnels : Vérifient le respect des spécifications (fonctions attendues).
5. Tests End-to-End : Évaluent un parcours utilisateur complet, du front-end au back-end.
6. Tests d'Acceptation : Réalisés par les utilisateurs/clients pour valider que le logiciel répond au besoin réel.
7. Mise en Production : Publication du logiciel validé sur l'environnement final.