# Les tests dans DevOps


**Bureau E204**

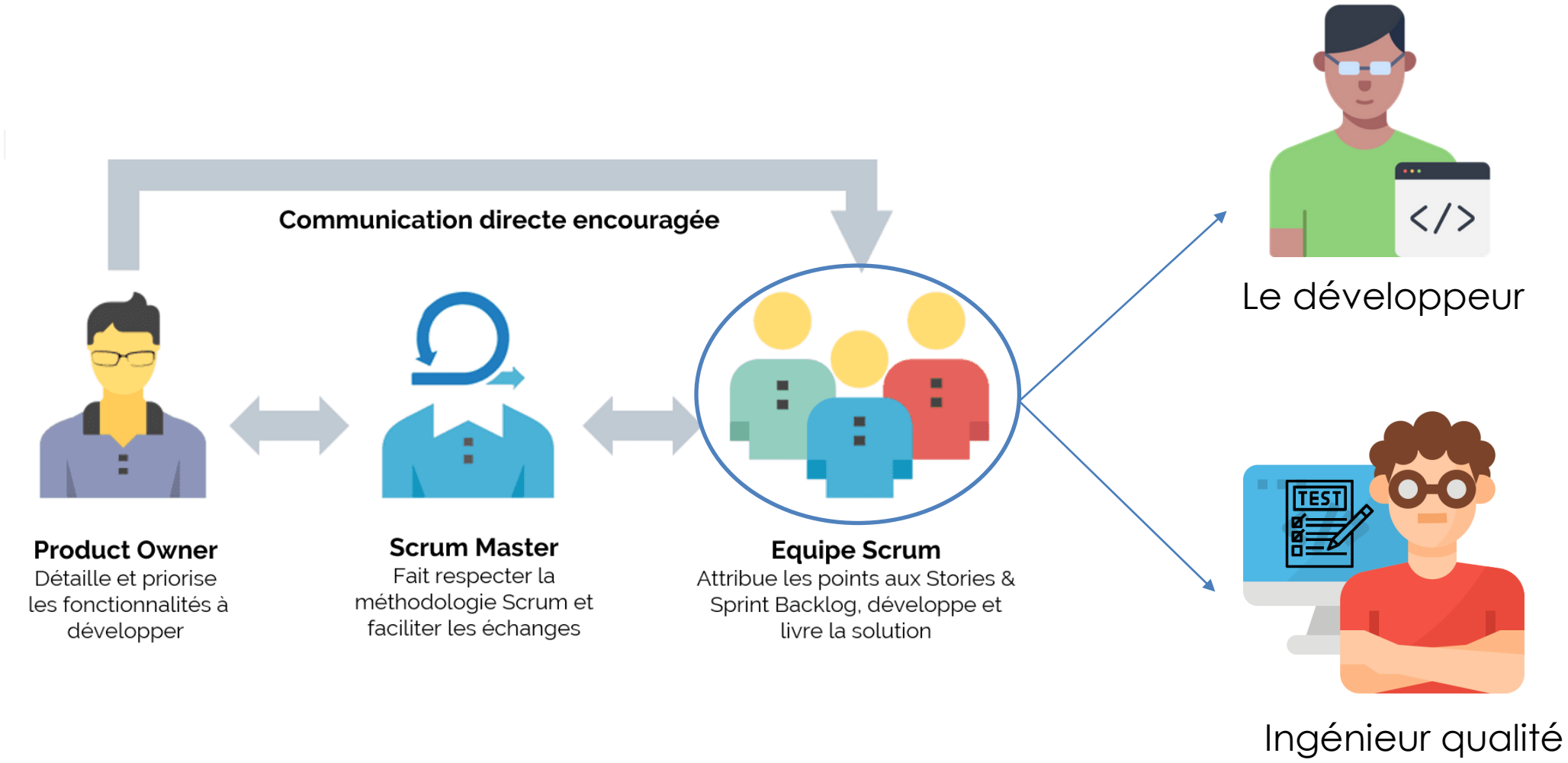
# Plan du cours

- Introduction
- Quelques types de tests
- Les tests dans le pipeline CI/CD
- Test unitaire
- Travail à faire

# Introduction

Quelle est la composition de base d'une équipe Scrum ?

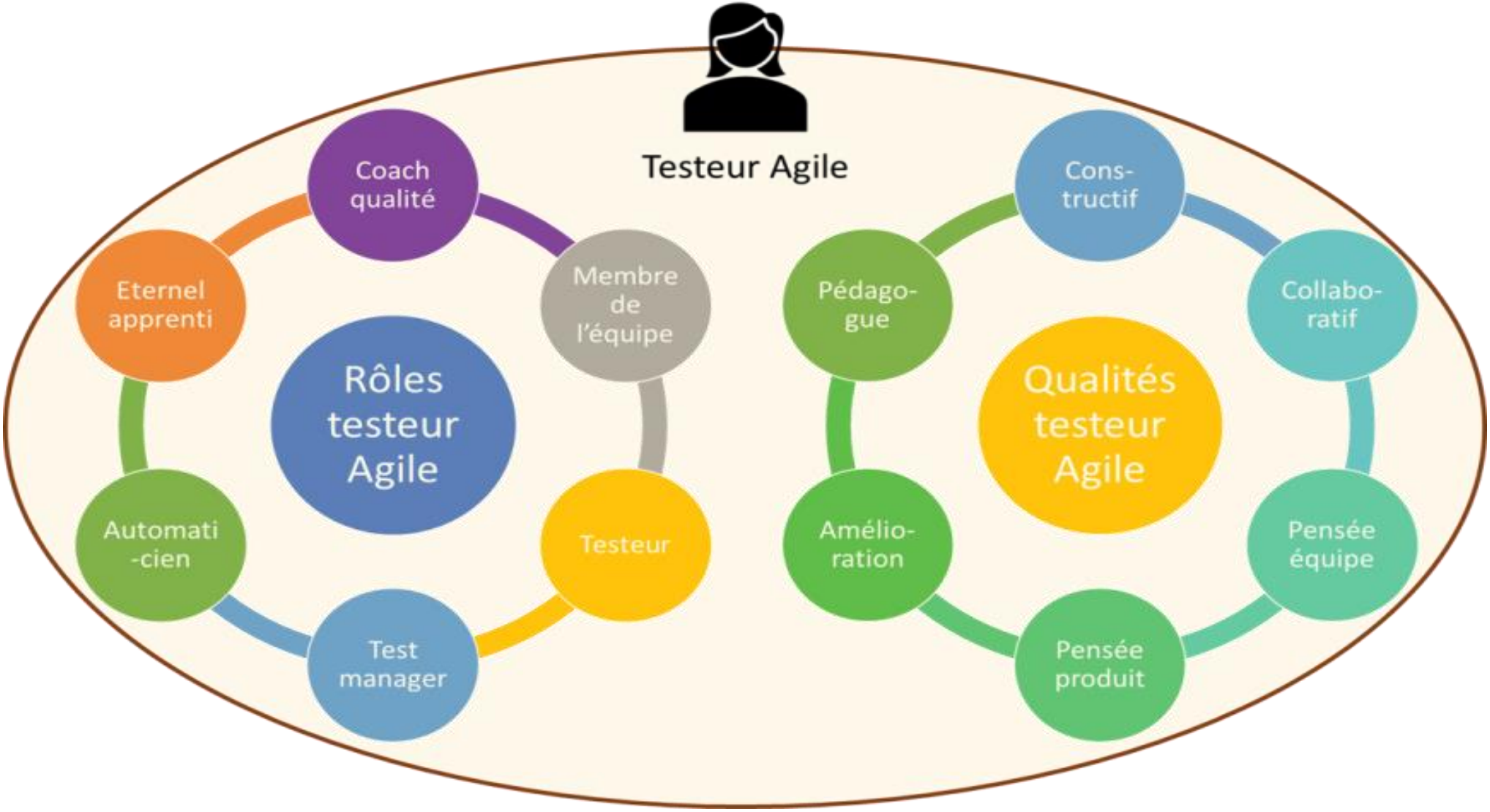
# Introduction



# Introduction

- Le rôle de l'ingénieur qualité (test) est de surveiller **la qualité du projet** au cours des différentes itérations de la réalisation.
- Pour être efficace, les ingénieurs Qualité doivent **communiquer** avec les membres de l'équipe.
- Ils doivent les **convaincre** de l'importance de respecter les procédures de correction afin que les produits proposés répondent aux recommandations du client.
- L'acquisition d'autres compétences est essentielle pour l'évolution professionnelle d'un ingénieur qualité.

# Introduction




Rôle d'un testeur

# Quelques types de test

- Il existe différents niveaux de test :
  - Test d'intégration
  - Test de régression
  - Test de montée en charge
  - Test de sécurité
  - Test unitaire
  - ….

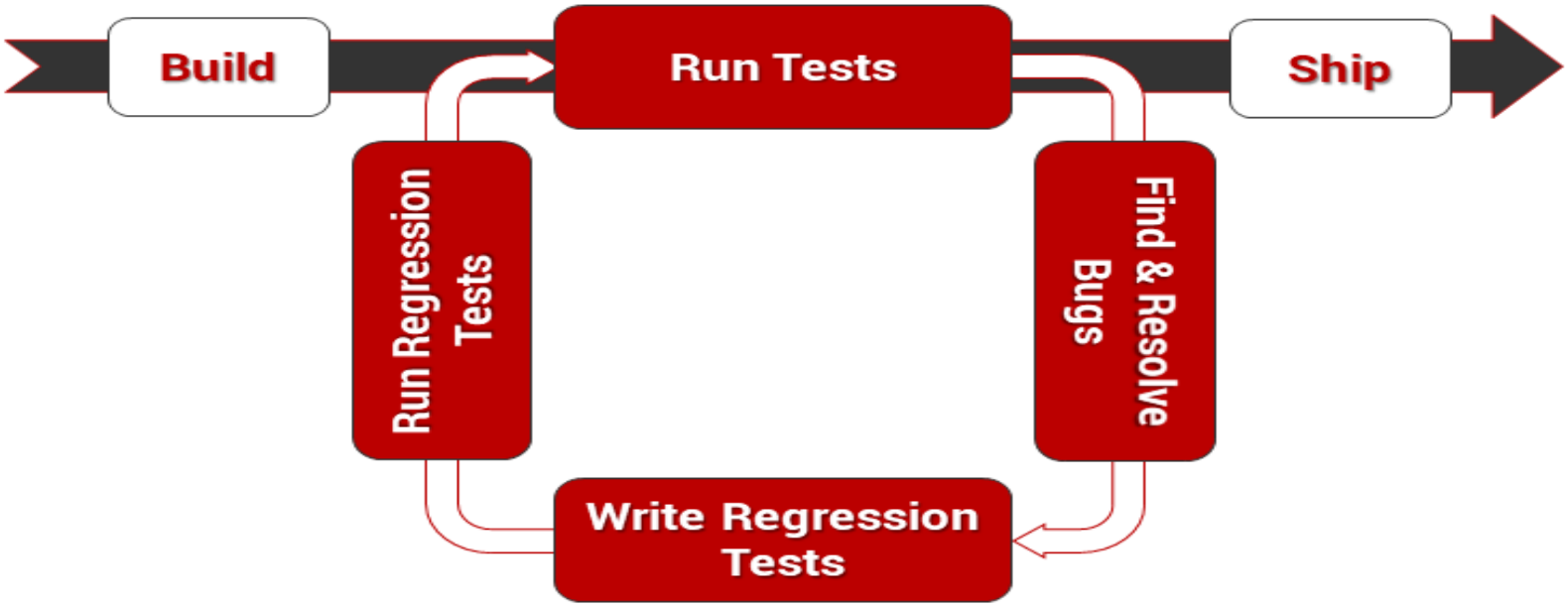
# Quelques types de test : Test d'intégration

- L'intégration, c'est assembler plusieurs composants logiciels élémentaires pour réaliser un composant de plus haut niveau.

  **Exemple**: Une classe Client et une classe Produit pour créer un module de commande sur un site marchant, c'est de l'intégration !
- **Un test d'intégration** vise à s'assurer du bon fonctionnement de la mise en œuvre conjointe de plusieurs unités de programme, testés unitairement au préalable.

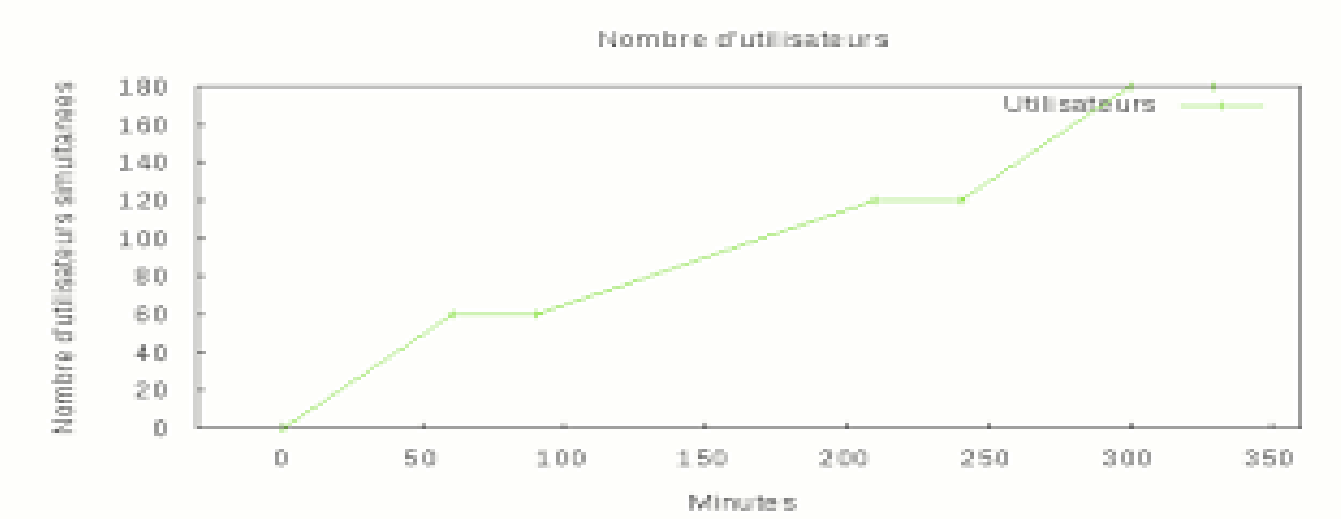
# Quelques types de test : Test de regression

- **Les tests de régression** sont les tests exécutés sur un programme préalablement testé mais qui a subit une ou plusieurs modifications.



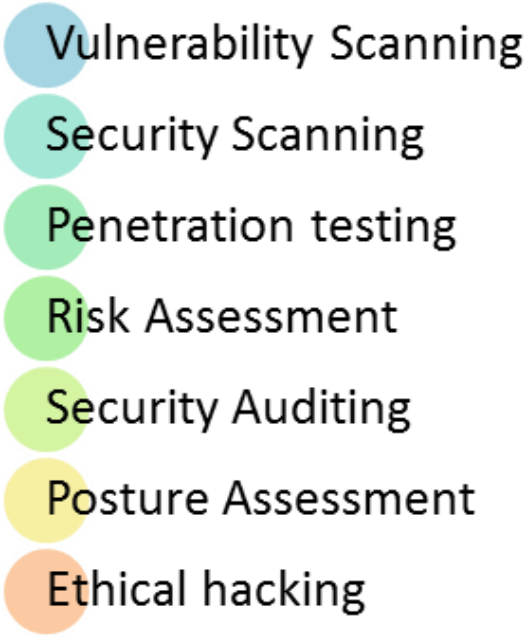
# Quelques types de test : Test de montée en charge

- **Test de montée en charge** (Test de capacité ) : il s'agit d'un test au cours duquel on va simuler un nombre d'utilisateurs sans cesse croissant de manière à déterminer quelle charge limite le système est capable de supporter.



# Quelques types de test : Test de sécurité

- **Le test de sécurité** est un type de test de logiciel qui vise à découvrir les vulnérabilités du système et à déterminer que ses données et ressources sont protégées contre d'éventuels intrus.



# Les tests dans le pipeline CI/CD

**Intégration continue**

- Dans une chaine d'intégration DevOps, il est primordiale de définir et de sélectionner les tests qui y seront incluses sans pour autant rendre le déploiement lent.
- Pour la phase d'intégration continue, il est recommandé d'avoir au moins les **tests unitaires** et les **tests du qualité** des codes.
- N'étant pas un développeur, **le rôle de l'ingénieur qualité dans cette phase est d'automatiser la génération de rapports détaillés pour les développeurs**.

# Les tests dans le pipeline CI/CD

**Livraison continue**

- Les tests unitaires sont utiles dans la phase d'intégration continue mais avoir des bouts de code testés à part n'est pas suffisant pour la phase de livraison continue.
- Il faut impérativement inclure des tests beaucoup plus avancés (**test d'intégration**, etc..) même si le process prendra beaucoup plus de temps.
- Le rôle de l'ingénieur qualité est beaucoup plus important dans cette phase (**optimiser le temps d'exécution**, assurer l'**automatisation** des différents tests, vérification des différents environnements).

# Les tests dans le pipeline CI/CD

**Déploiement continu**

- L'environnement de production étant très sensible, des tests beaucoup plus avancés sont nécessaires dans un environnement identique à celui de production ( environnement staging).
- Le test fonctionnel vérifiant que **la partie IHM** ne contient pas de bugs. **Selenium** est l'outil le plus recommandé pour ce genre de tests.
- Les **UAT** (User acceptance test) faites par les business analyst consolide les tests initialement faits.

# Les tests dans le pipeline CI/CD

**Déploiement continu**

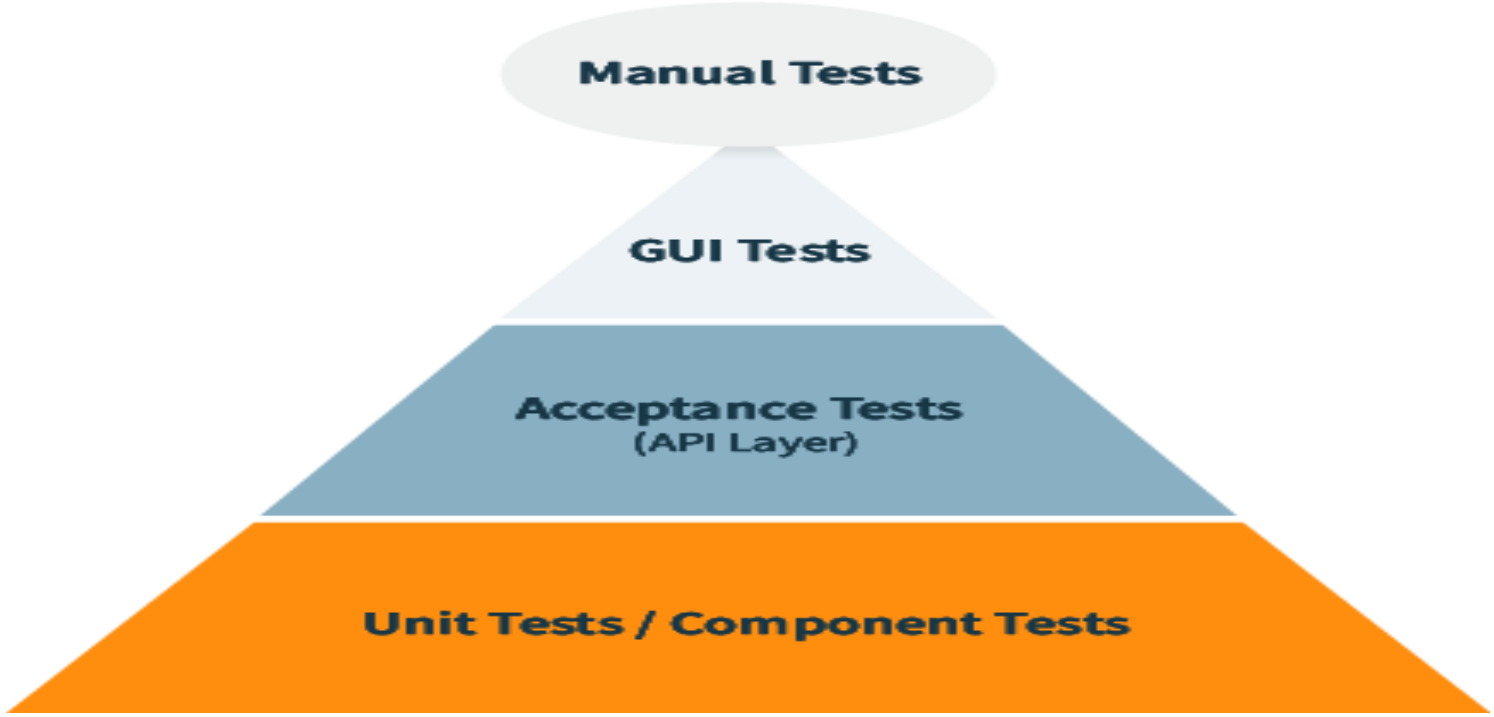
- Les tests non fonctionnels ( tests de **performance** (Jmeter, etc..) génèrent des métriques très intéressantes pour améliorer la qualité du livrable.
- Les tests de **sécurité** (Fortify, etc..)détectent des failles de sécurité.
- L'objectif du testeur dans cette phase est de **minimiser** la batterie de test tout en assurant les mêmes performances et de sélectionner les outils de test les plus performants.

# Les tests dans le pipeline CI/CD

- Nous constatons donc que les tests sont présents dans tous les process DevOps.
- Les tests les plus rapides (**temps d'exécution réduit**) sont présents au début du pipeline (rythme d'intégration continu très important). Le retour de ces tests améliore considérablement la **productivité**.
- Les tests nécessitant beaucoup plus de temps d'exécution arrivent beaucoup plus tard dans le pipeline(Livraison ou déploiement continue).

# Les tests dans le pipeline CI/CD

- Le choix d'exécution des tests est fortement lié à la pyramide de tests.



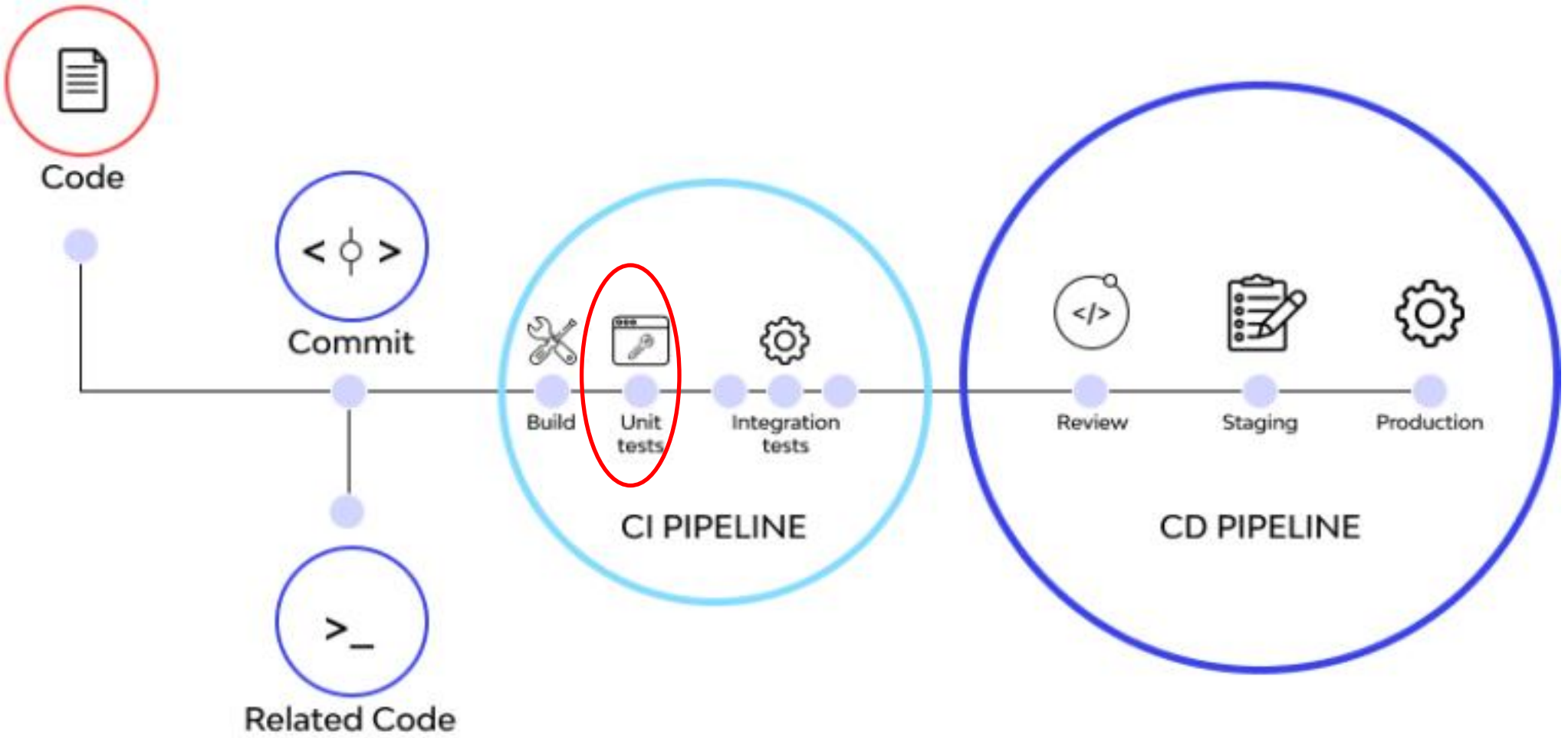

**Automate as Many Tests as You Can !**

# Tests manuelles vs tests automatiques

- Les tests manuelles sont assez **fastidieuses** et **répétitives**.
- Vu la monotonie des taches, l'ingénieur qualité peut manquer de rigueur et engendrer la non-détection de quelques bugs.
- Les tests automatiques peuvent être réalisés en parallèle si les capacités de la machine le permettent.
- Les tests automatiques nécessitent un certain temps de mise en place. Ce temps est facilement récupéré lors des prochaines itérations surtout avec le rythme de livraison continu assez rapide.

# Test unitaire

- Dans notre cours, on va s'intéresser aux tests unitaires.



# Test unitaire : Définition

- C’est un type de test logiciel dans lequel une seule unité ou un seul composant de logiciel est testé. L'objectif est de vérifier que chaque unité de code logiciel fonctionne comme prévu. Les unités peuvent être des fonctions, des méthodes, des procédures, des modules ou des objets individuels.

- Dans des différents cycles de développement, les tests unitaires sont le premier niveau de test effectué avant les tests d'intégration.

# Test unitaire : Définition

- Les tests unitaires sont une technique de test en **boîte blanche**, généralement effectuée par des développeurs. Cependant, dans le monde réel, les ingénieurs QA (Quality Assurance) effectuent également des tests unitaires.
- Une bonne couverture de tests permet d'être sûr que les fonctionnalités développées fonctionnent bien avant la livraison, mais aussi de vérifier que le code développé sur une version précédente **s'exécute toujours correctement** sur la version courante.

# Test unitaire : Les phases

- Le test unitaire comporte 3 phases:
  - D'abord, il initialise une petite partie de l'application à tester.
  - Ensuite, il applique ensuite un stimulus au système à tester (généralement en appelant une méthode)
  - Enfin, il observe le comportement résultant.

  Si le comportement observé est celui attendu, le test unitaire réussit, sinon il échoue, indiquant un problème quelque part dans le système testé.

  → Ces trois phases de tests unitaires sont également appelées **Arrange**, **Act** et **Assert**, ou **AAA** en abrégé.

# Test unitaire : Avantage et intérêt

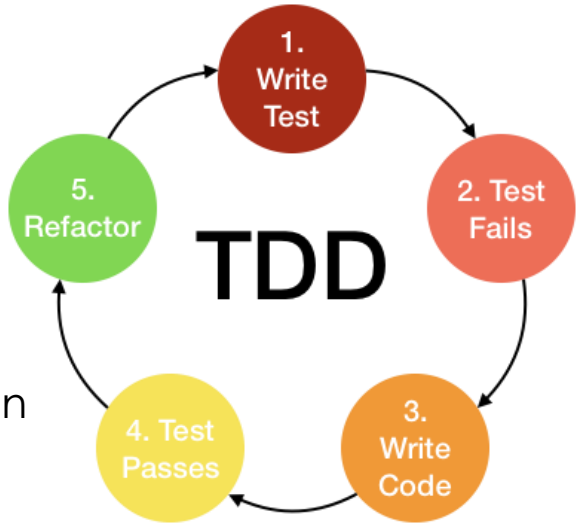
- Garantie la non régression
- Détection de bug plus facile
- Aide a isoler les fonctions
- Aide a voir l'avancement d'un projet (TDD*)

– Le **test-driven development** (TDD) ou en français le développement piloté par les tests est une technique de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel.

• Les étapes de TDD:

1. Écrire un test échoué
2. Écrire le code le plus simple pour que le test soit réussi
3. Supprimer la duplication (code/test) et améliorer la lisibilité (Refactor)

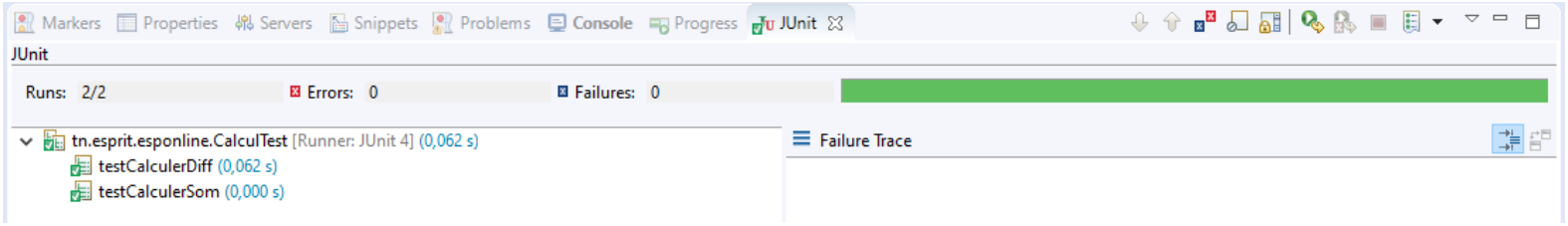
→ Le TDD est une approche avec laquelle on cherche à cadrer la production de code par petits incréments qui sont testés.



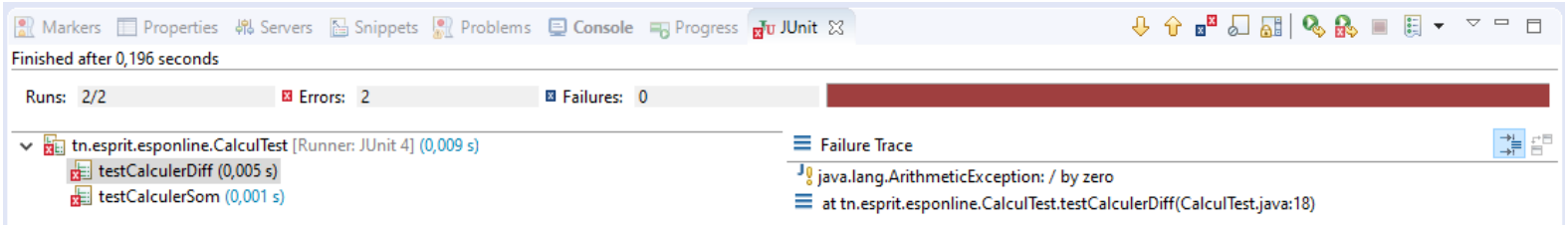
# Test unitaire : Outils

# Test unitaire : Résultats

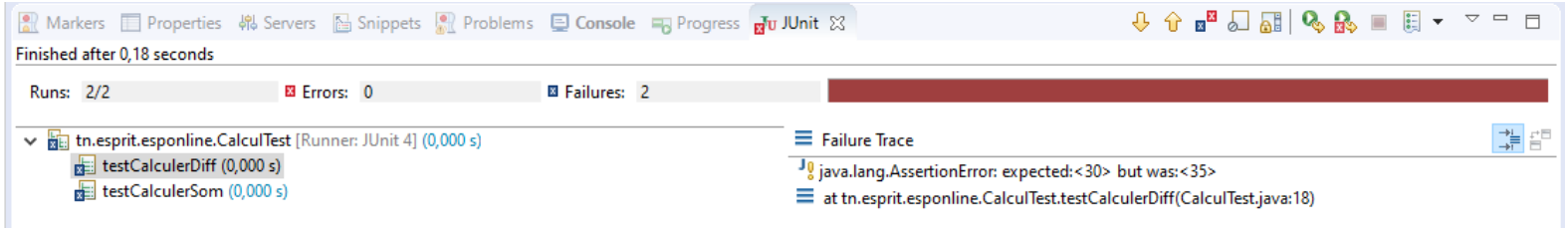
- Un test unitaire peut renvoyer 3 résultats différents :
  - Success : test réussi




  - Error : erreur inattendue à l'exécution (Exception)




  - Failure : au moins une assertion est fausse



# Test unitaire: JUnit - Définition

- JUnit est un framework open source pour le développement et l'exécution de tests unitaires automatisables. Le principal intérêt est de s'assurer que le code répond toujours aux besoins même après d'éventuelles modifications.


- Il n'y a pas de limite au nombre de tests au sein de notre classe de test.
- On écrit **au moins** un test par méthode de la classe testée.
- Pour désigner une méthode comme un test, il suffit d'utiliser l'annotation **@Test** (à partir de JUnit4).

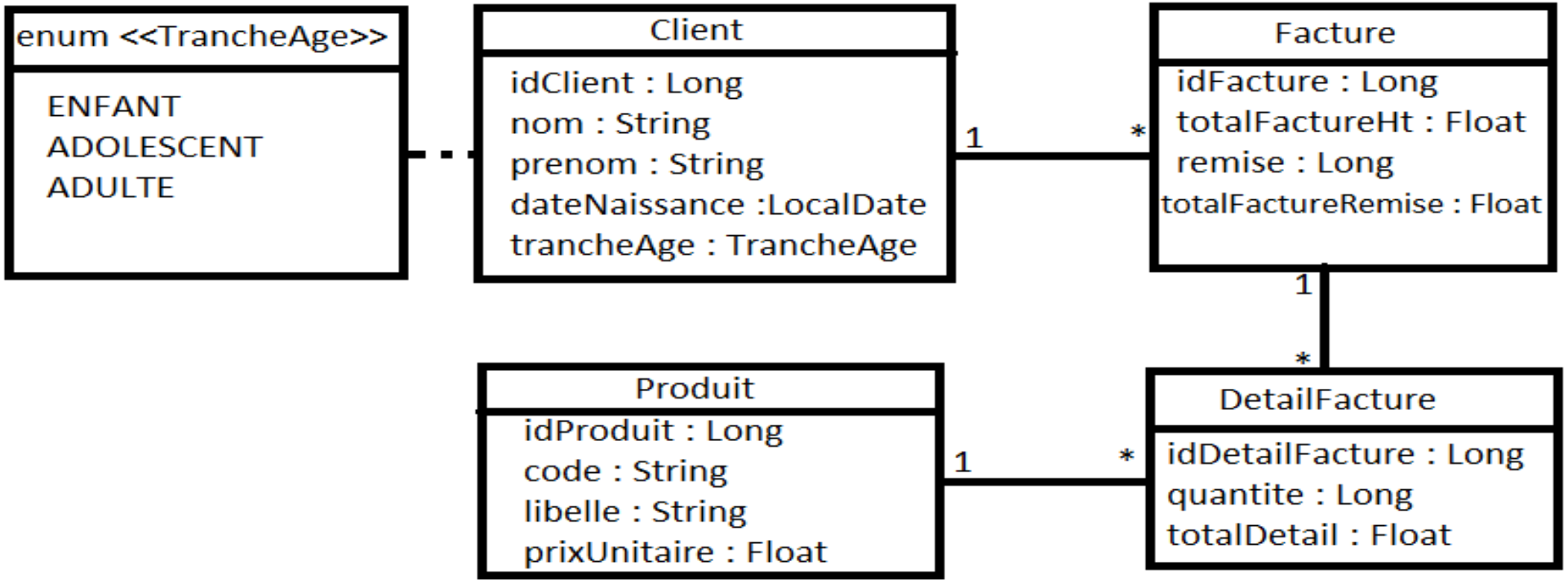
# Test unitaire: JUnit - Dépendance

- Dans le fichier « pom.xml », nous pouvons voir la dépendance starter suivante qui inclut les bibliothèques pour tester les applications Spring Boot.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

# Test unitaire: JUnit – Mise en place

- Pour mieux comprendre l'implémentation des tests unitaires avec JUnit afin de tester les méthodes unitairement, nous allons implémenter une petite application de gestion de la facturation d'une boutique
- Le diagramme des classes de notre application est le suivant :



# Test unitaire: JUnit – Mise en place

- En ce qui concerne l'implémentation des méthodes de test, le projet Spring Boot génère automatiquement un exemple de classe de test dans le package **src/test/java** (le package qui qui va contenir toutes les classes des tests unitaires). Nous allons maintenant créer nos propres classes de test en suivant les bonnes pratiques et supprimer cette classe par défaut.

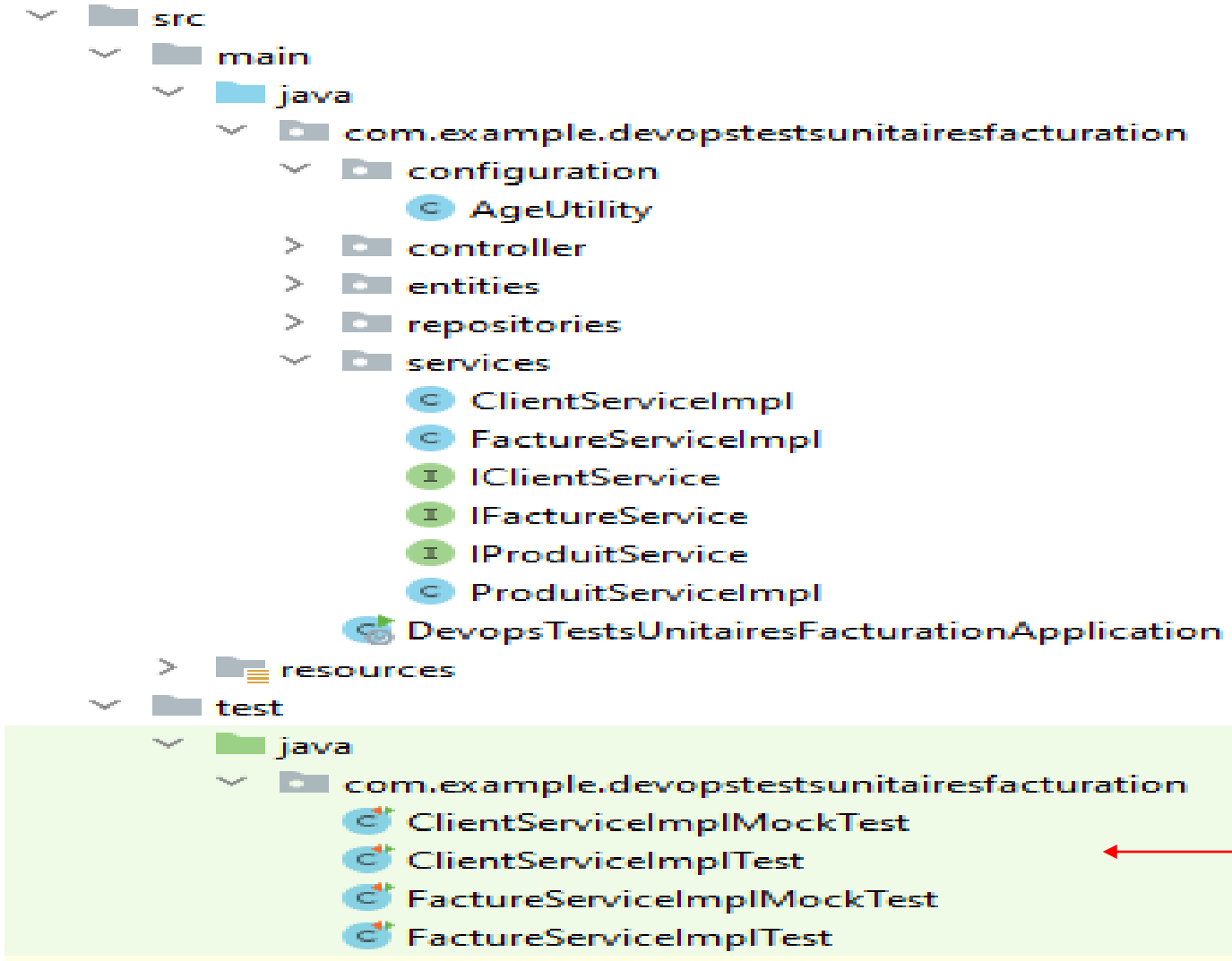
```
@RunWith(SpringRunner.class)
@SpringBootTest
class SpringBootWithUnitTestApplicationTests {

    @Test
    void contextLoads() {
    }

}
```

# Test unitaire: JUnit – Mise en place

- La structure de notre projet est la suivante :




Tests unitaires avec Junit et Mockito

# Test unitaire: JUnit – Mise en place

- En suivant les bonnes pratiques de développement, nous devons créer une classe de test pour chaque classe service. Nous avons ainsi créer la classe de test **ClientServiceImplTest** pour la classe service ClientServiceImpl et la classe **FactureServiceImplTest** pour la classe FactureServiceImpl

- Pour créer une classe de test, on crée une classe avec le même nom que la classe service en ajoutant le mot **Test** à la fin.
- Pour créer une méthode de test, on crée une méthode en gardant le même nom de la méthode en la précédant avec le mot **test**
- **exemple** : ajouterFacture (Facture f) => **test**AjouterFacture(Facture f)

# Test unitaire: JUnit – Mise en place

- Les annotations à ajouter à la classe de test sont les suivantes :

```
@ExtendWith(SpringExtension.class)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@SpringBootTest
```

**@ExtendWith** : pour indiquer que nous allons travailler avec des tests unitaires en utilisant JUnit

**@TestMethodOrder** : pour pouvoir ordonner le lancement des tests selon une logique bien précise

**@SpringBootTest** : pour indiquer qu'il s'agit d'une classe de test

# Test unitaire: JUnit – Mise en place

**Les annotations que nous pouvons utiliser dans JUnit 5 sont les suivantes :**

- **@Test :** C'est comme dans JUnit4, cette annotation est utilisée pour dire que c'est une méthode de test.
- **@Order :** Pour imposer l'exécution des tests dans un ordre spécifique (Pour pouvoir utiliser cette annotation, on doit ajouter @TestMethodOrder(MethodOrderer.OrderAnnotation.class) au-dessus de la classe de test)
- **@RepeatedTest (n) :** La méthode annotée sera exécutée n fois comme si le même test avait été écrit n fois.

# Test unitaire: JUnit – Mise en place

- **@BeforeEach @AfterEach :** Méthodes qui sont exécutées avant et après chaque méthode de test dans la classe.
- **@BeforeAll @AfterAll :** Ces annotations sont utilisées pour signaler que les méthodes statiques annotées doivent être exécutée avant et après tous les tests @Test

# Test unitaire: JUnit – Mise en place

**Les assertions :**

Lorsqu'on implémente les tests unitaires, nous pouvons utiliser **les assertions**. Ce sont des méthodes utilitaires dont le résultat de l'expression booléenne indique un succès ou une erreur pour supporter l'assertion de conditions dans les tests. Ces méthodes sont accessibles à travers la classe **Assertions**, dans JUnit 5.

Afin d'augmenter la lisibilité du test et des assertions elles-mêmes, il est toujours recommandé d'importer statiquement la classe respective. De cette façon, nous pouvons nous référer directement à la méthode d'assertion elle-même sans la classe représentative comme préfixe.

# Test unitaire: JUnit – Mise en place

## Quelques assertions disponibles :

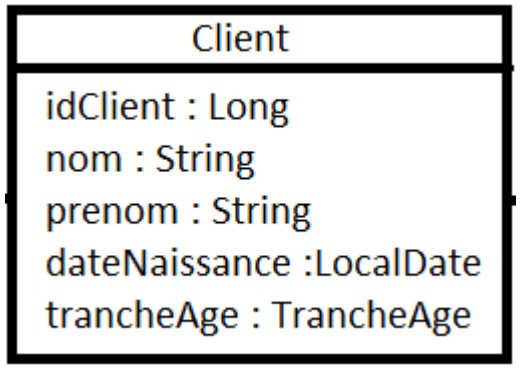
| Assertions | Actions | Implémentation |
|---|---|---|
| assertArrayEquals | Vérifier l'égalité entre deux tableaux | assertArrayEquals (valeur attendue, valeur à tester); |
| assertEquals | Vérifier l'égalité entre deux entiers, réels, chaines de caractères, … | assertEquals ("Test unitaire", "Test unitaire"); |
| assertNotEquals | Vérifier l'inégalité entre deux entiers, réels, chaines de caractères, … | assertEquals ("Test unitaire", "test unitaire"); |
| assertTrue | Vérifier que les conditions fournies sont vraies | assertTrue (5 > 4) |
| assertFalse | Vérifier que les conditions fournies sont fausses | assertFalse(5 > 6) |
| assertSame | Vérifier l'égalité entre deux objets. | assertSame(new Client(), new Client()) |

# Test unitaire: JUnit – Mise en place

| Assertions | Actions | Implémentation |
|---|---|---|
| assertNotSame | Vérifier l'inégalité entre deux objets. | assertNotSame(new Client(), new Magasin()) |
| assertArrayEquals | Vérifier l'égalité entre deux tableaux | assertArrayEquals(int[] expected, int[] actual) |
| assertAll | Permet la création d'assertions groupées | assertAll( "heading",<br>() -> assertEquals(4, 2 * 2),<br>() -> assertEquals("java","JAVA".toLowerCase()),<br>() -> assertEquals(null, null)<br>); |
| assertTimeout | Affirmer que l'exécution d'un exécutable fourni se termine avant une date donnée. | assertTimeout( ofSeconds(2), () -> { // code necéssite maximum 2 minutes pour s'exécuter Thread.sleep(1000); } ); |

# Test unitaire: JUnit – Mise en place

Nous allons maintenant implémenter à titre d'exemple deux méthodes de test :




Commençons par la première méthode de test de l'ajout d'un client.

ce test permet de vérifier que le **nom** et le **prénom** du client ont une taille logique ainsi que la tranche d'âge (calculable) est bien associé à son âge en respectant les conditions suivantes :

**si** age < 10 : ENFANT **sinon si** age entre 10 et 19 : ADOLESCENT

**sinon** (age supérieur à 19) ADULTE

Nous vérifions aussi que le client a bien été ajouté à la table Client.

# Test unitaire: JUnit – Mise en place

Le code de notre méthode service est le suivant. La méthode currentAgeAffectation associe le client à sa tranche d'âge selon le retour de la méthode calculateAge

```
@Override
public Client ajouterClient(Client c) {
    Client client = currentAgeAffectation(c);
    return clientRepository.save(client);
}
1 usage
private Client currentAgeAffectation(Client c) {
    int currentAge = AgeUtility.calculateAge(c.getDateNaissance());
    if(currentAge<10)
    { c.setTrancheAge(TrancheAge.ENFANT); }
    else if(currentAge>=10 && currentAge < 19)
    { c.setTrancheAge(TrancheAge.ADOLESCENT); }
    else { c.setTrancheAge(TrancheAge.ADULTE);}
    return c;
}
```

# Test unitaire: JUnit – Mise en place

La méthode de test associée est présentée ainsi avec les commentaires explicatifs

```
public void testAjouterClient() {
    // création client avec un age qui le classifie dans la trancheAge adulte
    Client adultClient=  Client.builder().nom("bouslama").prenom("ahmed")
            .dateNaissance(LocalDate.parse( text: "1990-04-01")).build();
    // Ajout Client
    Client savedClient = clientService.ajouterClient(adultClient);
    // Vérifier que le client a bien été ajouté dans la table client
    Assertions.assertNotNull(savedClient.getIdClient());
    TrancheAge expectedTrancheAge = TrancheAge.ADULTE;
    // vérifier que le client est associé à la bonne tranche d'age
    Assertions.assertEquals(expectedTrancheAge, savedClient.getTrancheAge());
    // vérifier que la taille des noms et prénoms est logique
    Assertions.assertTrue( condition: savedClient.getNom().length()<15);
    Assertions.assertTrue( condition: savedClient.getPrenom().length()<10);
    // nettoyer la base
    clientService.supprimerClient(savedClient.getIdClient());
}
```

# Test unitaire: JUnit – Mise en place

Passons maintenant au deuxième test unitaire d'ajout d'une facture

Le test de l'ajout facture consiste à vérifier que le total de chaque détail facture ainsi que le total ht remisé de la facture soit bien calculé en prenant en considération le prix du produit, la quantité et le pourcentage de la remise. Nous vérifions aussi que la facture et ses détails sont bien insérés dans la base de données.

**NB** : prix ht remisé = prix ht - remise

# Test unitaire: JUnit – Mise en place

Les données que nous allons utiliser dans ce test sont les suivantes :

**Client** : ahmed slimi né le 01-05-2000

Produits : **produit 1** dentifrise code 05 prix : **25** dt

**produit 2** pèse personne code 19 prix **250** dt

La facture contient les deux détails suivants pour une remise globale de **10%**, un prix ht de **300** dinars et un prix ht remisé égale à **270** dt :

**Detail Facture 1** : dentifrice quantité : 2 prix total : **50** dt

**Detail Facture 2** : pèse personne quantité : 1 prix total : **250** dt

Nous assertons dans notre scénario principal de test que le total facture remisé ht est égal à 250+50 –30 (remise) =**270** dt

# Test unitaire: JUnit – Mise en place

La méthode service de l'ajout de la facture est la suivante :

```
public Facture ajouterFacture(Facture f) {
    // récupérer les détails pour les mettre à jour
    List<DetailFacture> detailFactures =  f.getDetailsFacture();
    // initialiser le totalttc facture à zéro
    AtomicReference<Float> total = new AtomicReference<>(Float.valueOf( f: 0));
    detailFactures.stream().forEach(detailFacture ->
    {
        // récupérer un par un le produit de chaque detail
        Produit p = produitRepository.findById(detailFacture.getProduit().getIdProduit()).get()
        // calculer le total de la ligne de commande : prix unitaire * qte
        detailFacture.setTotalDetail(p.getPrixUnitaire()* detailFacture.getQuantite());
        total.set(total.get() + detailFacture.getTotalDetail());
    });
```

# Test unitaire: JUnit – Mise en place

```
// affecter les details facture mis à jour
f.setDetailsFacture(detailFactures);
f.setTotalFactureHt(total.get());
// calcul du total avec remise
f.setTotalFactureHtRemise(f.getTotalFactureHt()- (f.getTotalFactureHt()*f.getRemise()/100));
// sauvegarde des nouveaux valeurs de la facture ainsi que ses détails mis à jour
Facture savedFacture = factureRepository.save(f);
detailFactures.stream().forEach(
        detailFacture -> {
            detailFacture.setFacture(f);
            detailFactureRepository.save(detailFacture);
        }
);
return f;
```

# Test unitaire: JUnit – Mise en place

La méthode de test associée au service de l'ajout de la facture expliquée  est la suivante :

```
public void testAjouterFacture() {
    //Création Client
    Client client = Client.builder().prenom("ahmed").nom("slimi")
            .dateNaissance(LocalDate.parse( text: "2000-05-01")).
            trancheAge(TrancheAge.ADULTE).build();
    // sauvegarde client
    Client savedClient= clientService.ajouterClient(client);
    // création des produits
    Produit produit1 = Produit.builder().libelle("dentifrice")
            .code("05").prixUnitaire(Float.valueOf( f: 25)).build();
    Produit produit2 = Produit.builder().libelle("pèse personne")
            .code("19").prixUnitaire(Float.valueOf( f: 250)).build();
    // sauvegarde des produits
    Produit savedProduct1= produitService.ajouterProduit(produit1);
    Produit savedProduct2= produitService.ajouterProduit(produit2);
```

# Test unitaire: JUnit – Mise en place

```
// création des détails factures
DetailFacture detailFacture1 = DetailFacture.builder().
produit(savedProduct1).quantite(2L).build();
DetailFacture detailFacture2 = DetailFacture.builder().
        produit(savedProduct2).quantite(1L).build();
List<DetailFacture> detailFactures = new ArrayList<>();
detailFactures.add(detailFacture1);
detailFactures.add(detailFacture2);
//Création de la facture associé aux client, produits et détails déja crées
Facture facture = Facture.builder().client(savedClient)
        .remise(10L).detailsFacture(detailFactures).build();
// sauvegarder la facture
Facture persistedFacture = factureService.ajouterFacture(facture);
```

# Test unitaire: JUnit – Mise en place

```
    // vérifier que la facture et ses détails ont bien été ajoutées dans la base
    Assertions.assertNotNull(persistedFacture.getIdFacture());
    persistedFacture.getDetailsFacture().forEach(detailFacture ->
    {
        Assertions.assertNotNull(detailFacture.getIdDetailFacture());
    });
   // totalFactureHtRemise = (25*2 +250) *0.9 = 270 -- réduction remise 10%
    Float expectedTotalFacture = Float.valueOf( f: 270);
     Assertions.assertEquals(expectedTotalFacture,persistedFacture.getTotalFactureHtRemise());
    // nettoyer la base de donnée en supprimant toutes les données insérées dans ce test
     // supprimer la facture avec ses détails (cascade)
     factureService.supprimerFacture(persistedFacture.getIdFacture());
    // supprimer les produits
      produitService.supprimerProduit(produit1.getIdProduit());
     produitService.supprimerProduit(produit2.getIdProduit());
    // supprimer le client
     clientService.supprimerClient(client.getIdClient());
} }
```

# Test unitaire: JUnit – Bonnes pratiques

Pour réaliser un test unitaire réussi, il y a un ensemble de bonnes pratiques à respecter :

- En cas d'ajout de nouvelles lignes, il faut vérifier à la fois l'existence de cette nouvelle ligne (id existant) et que les champs insérés sont pertinents et le scénario métier est bien réalisé
- L'état de la base de données doit être le même avant et après l'exécution des différents tests unitaires (chaque ligne ajouté dans un test doit être éliminé après)
- Le scénario de test doit être logique et complet

# Test unitaire: Mockito – Mise en place

**Mockito :**

Les tests unitaires doivent être de petits tests, légers et rapides. Cependant, l'objet testé peut avoir des dépendances sur d'autres objets. Il peut avoir besoin d'interagir avec une base de données, de communiquer avec un service Web.

Dans la réalité, tous ces services ne sont pas disponibles pendant les tests unitaires. Même s'ils sont disponibles, les tests unitaires de l'objet testé et de ses dépendances peuvent prendre beaucoup de temps. Que faire si ?

- Le service web n'est pas joignable.
- La base de données est en panne pour maintenance.
- La file d'attente des messages est lourde et lente.

# Test unitaire: Mockito – Mise en place

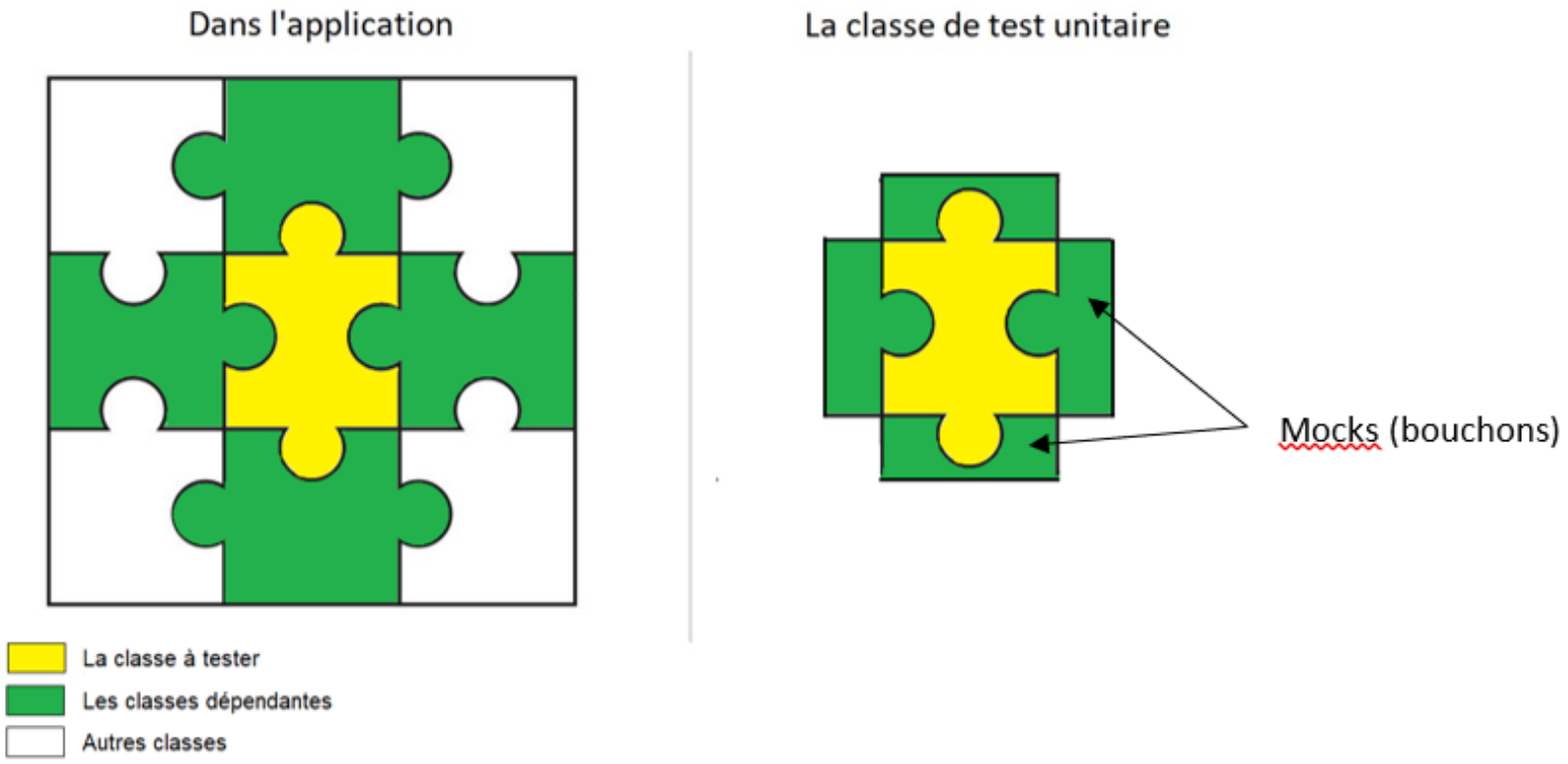
Tout cela va à l'encontre de l'objectif des tests unitaires qui sont atomiques, légers et rapides.

Nous voulons que les tests unitaires s'exécutent en quelques millisecondes. Si les tests unitaires sont lents, les constructions deviennent lentes, ce qui affecte la productivité de l'équipe de développement.

La solution consiste à utiliser le **mocking**, un moyen de fournir des doubles de test pour vos classes à tester.

# Test unitaire: Mockito – Mise en place

Pour simplifier le mock, on peut la considérer comme un bouchon qui va isoler la classe à tester unitairement comme décrit dans la figure ci-dessous :



# Test unitaire: Mockito - Dépendance

- Dans le fichier « pom.xml », nous devons ajouter la dépendance suivante :

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
</dependency>
```

# Test unitaire: Mockito – Mise en place

Pour pouvoir utiliser le mock, il suffit d'ajouter au niveau de la classe de test : `@ExtendWith(MockitoExtension.class)`

Dans notre cas, nous allons travailler sur les mêmes méthodes (ajout facture, ajout client) sans persister les données dans la base de données (données mock)

# Test unitaire: Mockito – Mise en place

La première méthode qui teste l'ajout du client suit la même logique que le test avec JUnit sans accès à la base de données

```
@Test
public void testAjouterClient() {
    // Création client avec un age qui le classifie dans la trancheAge adulte
    Client adultClient=  Client.builder().nom("bouslama").prenom("ahmed")
            .dateNaissance(LocalDate.parse( text: "1990-04-01")).build();
      // Ajout Client
    Mockito.when(clientRepository.save(Mockito.any(Client.class)))
            .thenReturn(adultClient);
    Client savedClient = clientService.ajouterClient(adultClient);
    TrancheAge expectedTrancheAge = TrancheAge.ADULTE;
    // vérifier que le client est associé à la bonne tranche d'age
    Assertions.assertEquals(expectedTrancheAge, savedClient.getTrancheAge());
    // vérifier que la taille des noms et prénoms est logique
    Assertions.assertTrue( condition: savedClient.getNom().length()<15);
    Assertions.assertTrue( condition: savedClient.getPrenom().length()<10);
    verify(clientRepository).save(Mockito.any(Client.class));
}
```

# Test unitaire: Mockito – Mise en place

La deuxième méthode de test de l'ajout facture est la suivante :

```
public void testAjouterFacture() {
    //Création Client
    Client client = Client.builder().prenom("ahmed").nom("slimi")
            .dateNaissance(LocalDate.parse( text: "2000-05-01")).
            trancheAge(TrancheAge.ADOLESCENT).build();
    // création des produits
    Produit produit1 = Produit.builder().idProduit(1L).libelle("dentifrice")
            .code("05").prixUnitaire(Float.valueOf( f: 25)).build();
    Produit produit2 = Produit.builder().idProduit(2L).libelle("pèse personne")
            .code("19").prixUnitaire(Float.valueOf( f: 250)).build();
    // création des détails factures
    DetailFacture detailFacture1 = DetailFacture.builder().
    produit(produit1).quantite(2L).build();
    DetailFacture detailFacture2 = DetailFacture.builder().
            produit(produit2).quantite(1L).build();
    List<DetailFacture> detailFactures = new ArrayList<>();
```

# Test unitaire: Mockito – Mise en place

```
  detailFactures.add(detailFacture1);
  detailFactures.add(detailFacture2);
  //Création de la facture associé aux client, produits et détails déja crées
  Facture facture = Facture.builder().client(client)
          .remise(10L).detailsFacture(detailFactures).build();
// Initialiser le résultat d'appel aux repos par des données mock
  Mockito.when(factureRepository.save(Mockito.any(Facture.class))).thenReturn(facture);
  Mockito.when(produitRepository.findById(1L)).thenReturn(Optional.ofNullable(produit1));
  Mockito.when(produitRepository.findById(2L)).thenReturn(Optional.ofNullable(produit2));
  Mockito.when(detailFactureRepository.save(detailFacture1)).thenReturn(detailFacture1);
  Mockito.when(detailFactureRepository.save(detailFacture2)).thenReturn(detailFacture2);
  // faire appel à la méthode de sauvegarder facture
  Facture mockedFacture = factureService.ajouterFacture(facture);
  // totalFactureHtRemisé = (25*2 +250) *0.9 = 270 -- réduction remise 10%
  Float expectedTotalFacture = Float.valueOf( f: 270);
  Assertions.assertEquals(expectedTotalFacture,mockedFacture.getTotalFactureHtRemise());
  // vérifier l'appel à la méthode save du facture repository
  verify(factureRepository).save(Mockito.any(Facture.class)); }}
```

# Test unitaire: Mockito – Mise en place

Les méthodes « **when** » et « **thenReturn** » de Mockito permettent de paramétrer le mock en affirmant que si la méthode findById() est appelée sur la classe (mockée) « **FactureRepository** », alors on retourne l'objet «**m**» comme résultat .

C'est grâce à cette ligne que l'on isole bien le test du service. Aucun échec de test ne peut venir de « **FactureRepository** », car on le simule, et on indique comment simuler.

# Test unitaire: Mockito – Mise en place

On peut aussi vérifier, si on utilise la méthode **verify()** de Mockito, que la classe (mockée) « **FactureRepository** » a été utilisée, en particulier la méthode « **save()** ».

```
Float expectedTotalFacture = Float.valueOf( f: 270);
Assertions.assertEquals(expectedTotalFacture,mockedFacture.getTotalFactureHtRemise());
// vérifier l'appel à la méthode save du facture repository
verify(factureRepository).save(Mockito.any(Facture.class)); }}
```

Vérifier que la méthode save du FactureRepository a été appelée

# Test unitaire: Résumé

- La phase de tests unitaire c'est une phase primordiale dans la réalisation d'un projet.

- Un **mock** est un type de doublure de test simple à créer, et qui vous permet également de tester comment on interagit avec lui.

- **Mockito** vous permet de définir comment vos mocks doivent se comporter (avec when) et vérifier si ces mocks sont bien utilisés (avec verify).

# Travail à faire

- Pour chaque module du projet, réaliser quelques tests unitaires avec JUnit et mockito et ajouter un nouveau « Stage » dans Jenkins pour lancer les tests unitaires automatiquement.
- **Les tests doivent etre consistants et ne doivent pas traiter un simple crud**

Si vous avez des questions, n'hésitez pas à nous contacter :

**Département Informatique**
**UP ASI**

Bureau E204

# Les tests dans DevOps