

Les tests dans DevOps



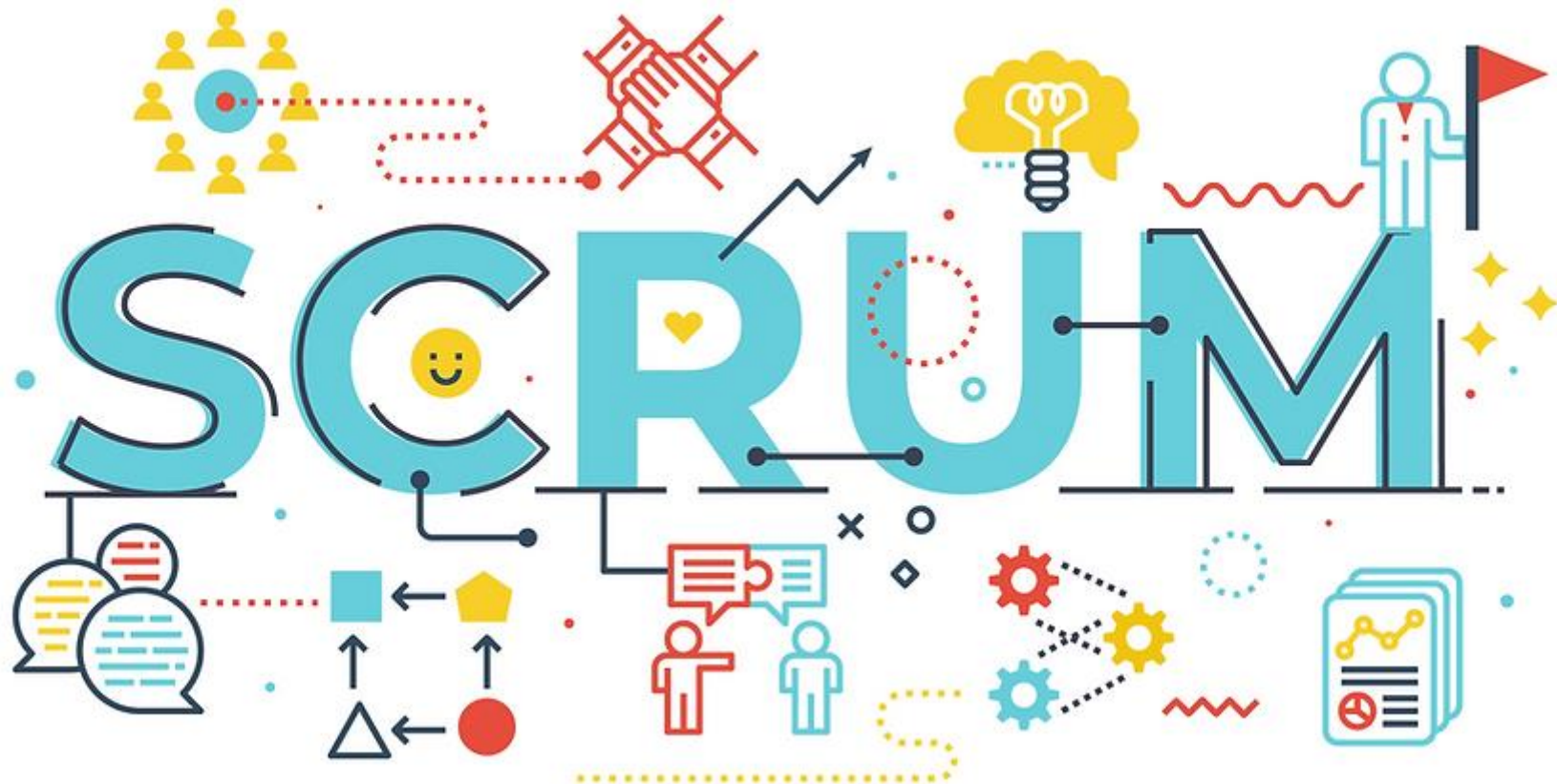
Bureau E204

Plan du cours

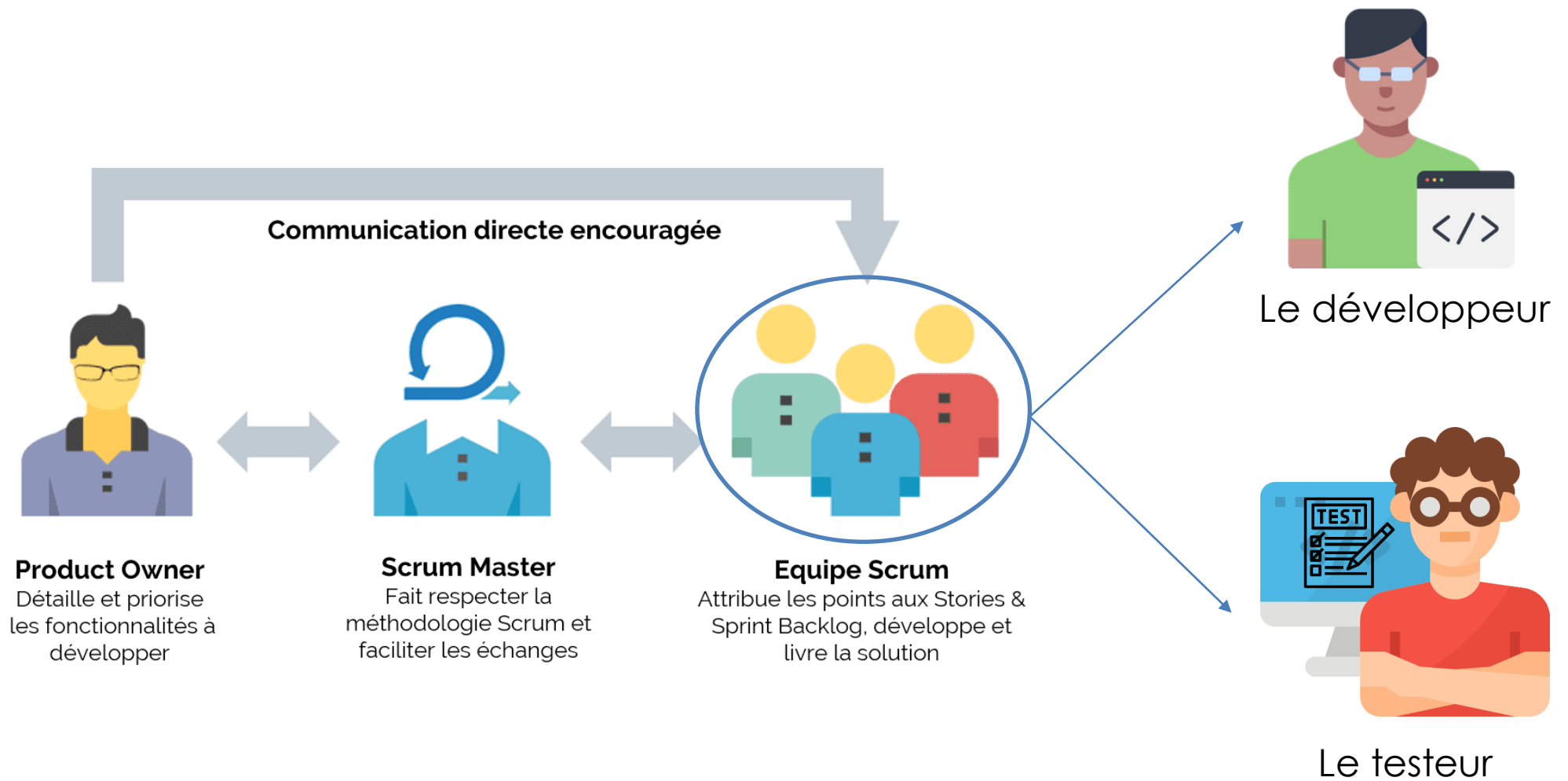
- Introduction
- Quelques types de tests
- Les tests dans le pipeline CI/CD
- Test unitaire
- Travail à faire

Introduction

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



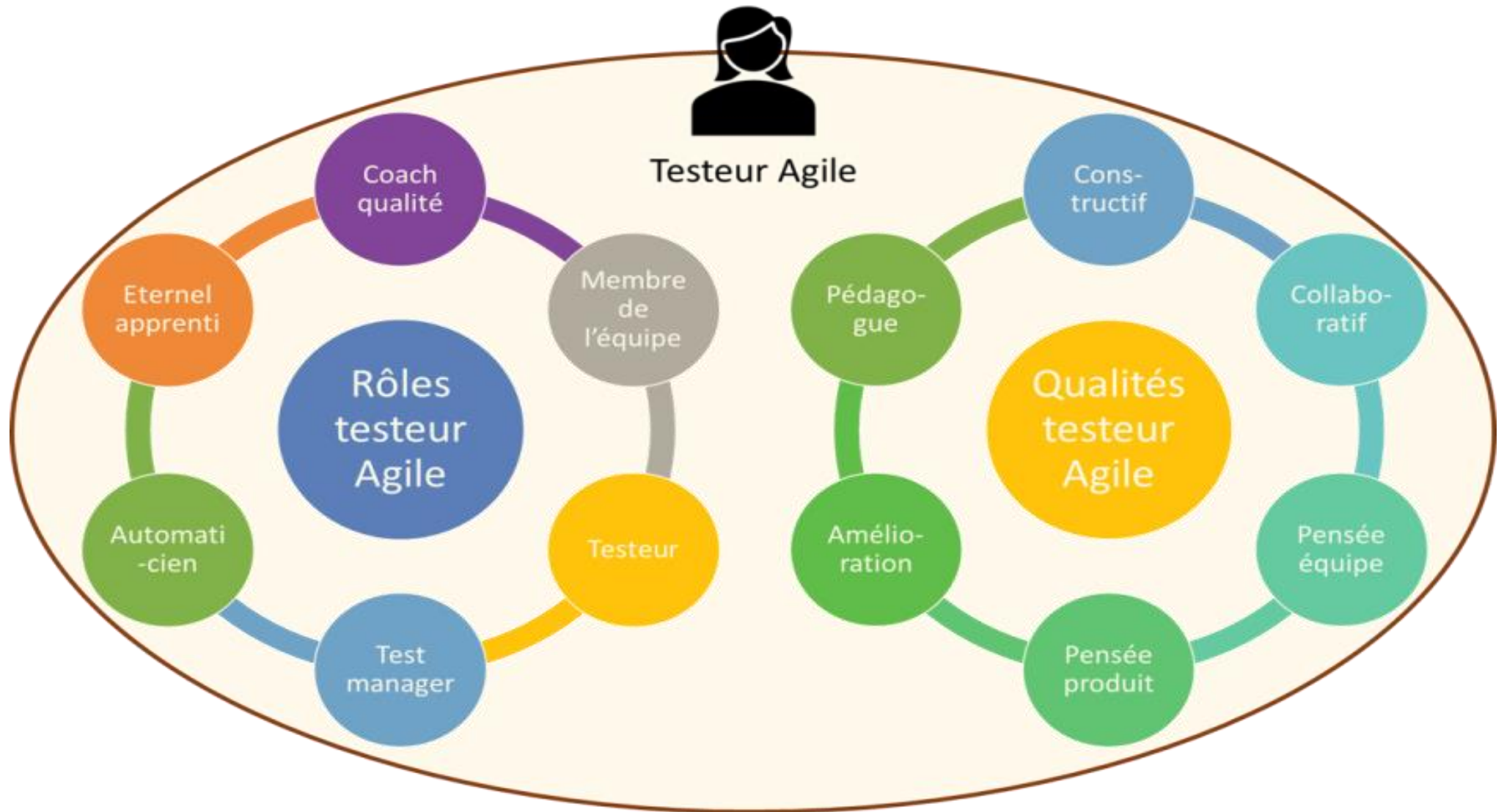
Introduction



Introduction

- Le rôle du testeur est de surveiller **la qualité du projet** au cours des différentes itérations de la réalisation.
- Pour être efficace, les testeurs doivent **communiquer** avec les membres de l'équipe.
- Il doit 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 testeur.

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

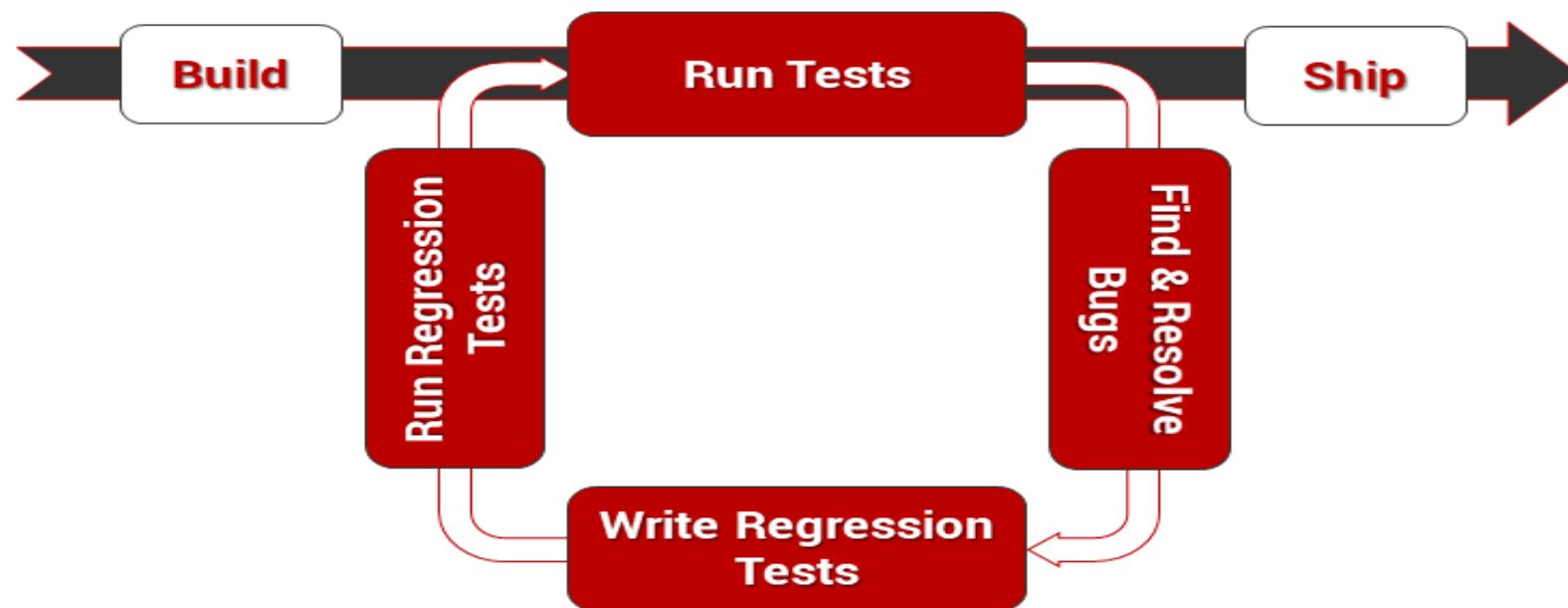
- L'intégration, c'est assembler plusieurs composants logiciels élémentaires pour réaliser un composant de plus haut niveau.

Exemple: Une classe Client et une classe Produit pour créer un module de commande sur un site marchand, 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.

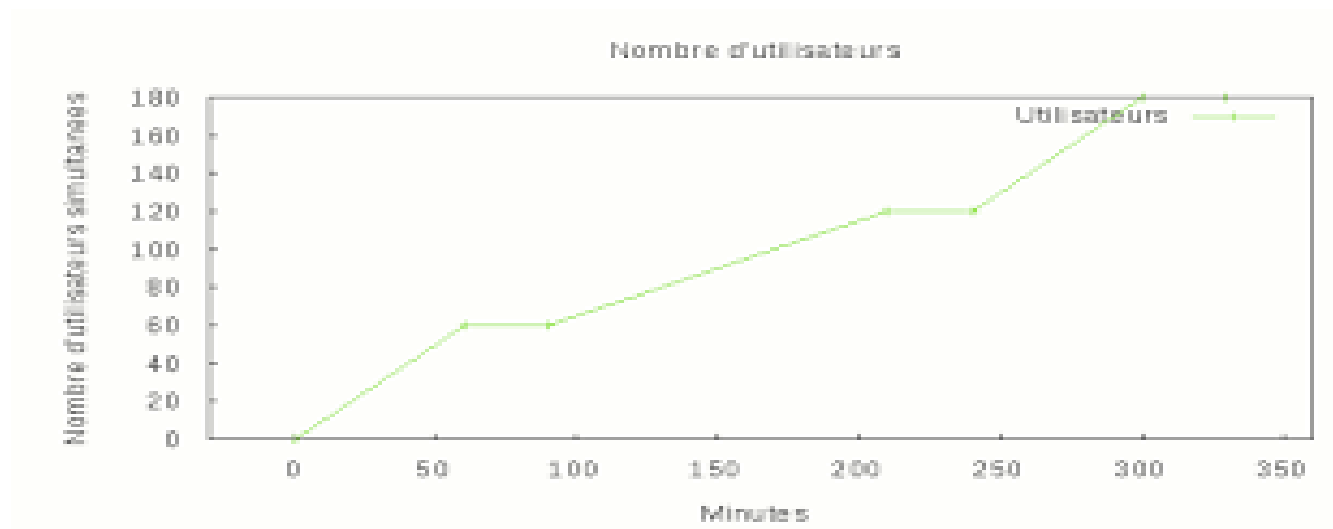
Quelques types de test : Test de régression

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



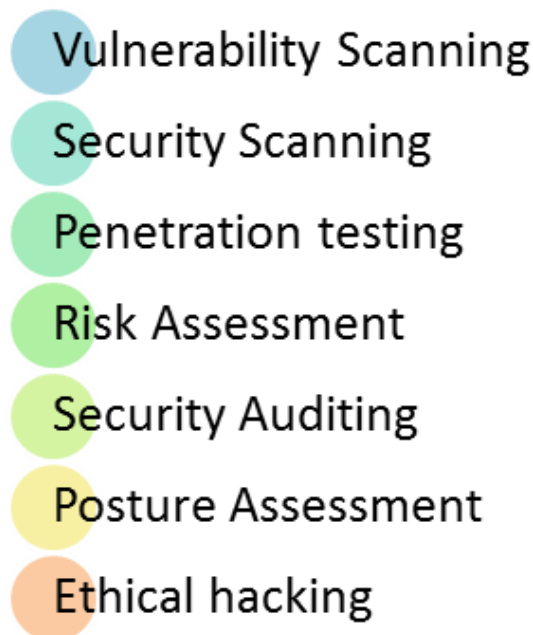
Quelques types de test : Test de montée en charge

- **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 chaîne 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 du testeur 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 du testeur 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) consolide les tests initialement faits.

Les tests dans le pipeline CI/CD

Déploiement continu

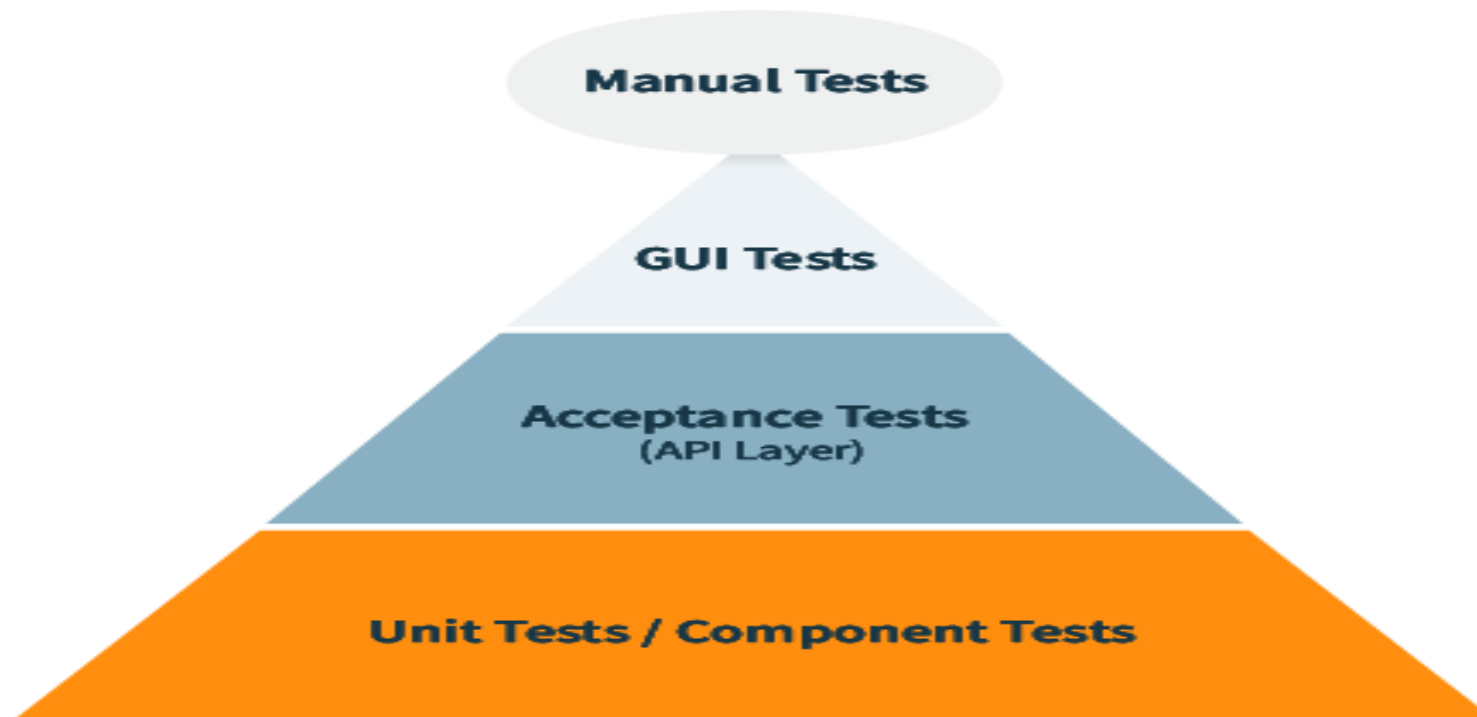
- Les tests non fonctionnels (tests de **performance** (Jmeter, etc..) génèrent des métriques très intéressantes pour améliorer la qualité du livrable.
- Les tests de **sécurité** (Fortify, etc..) détectent des failles de sécurité.
- L'objectif du testeur dans cette phase est de **minimiser** la batterie 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.



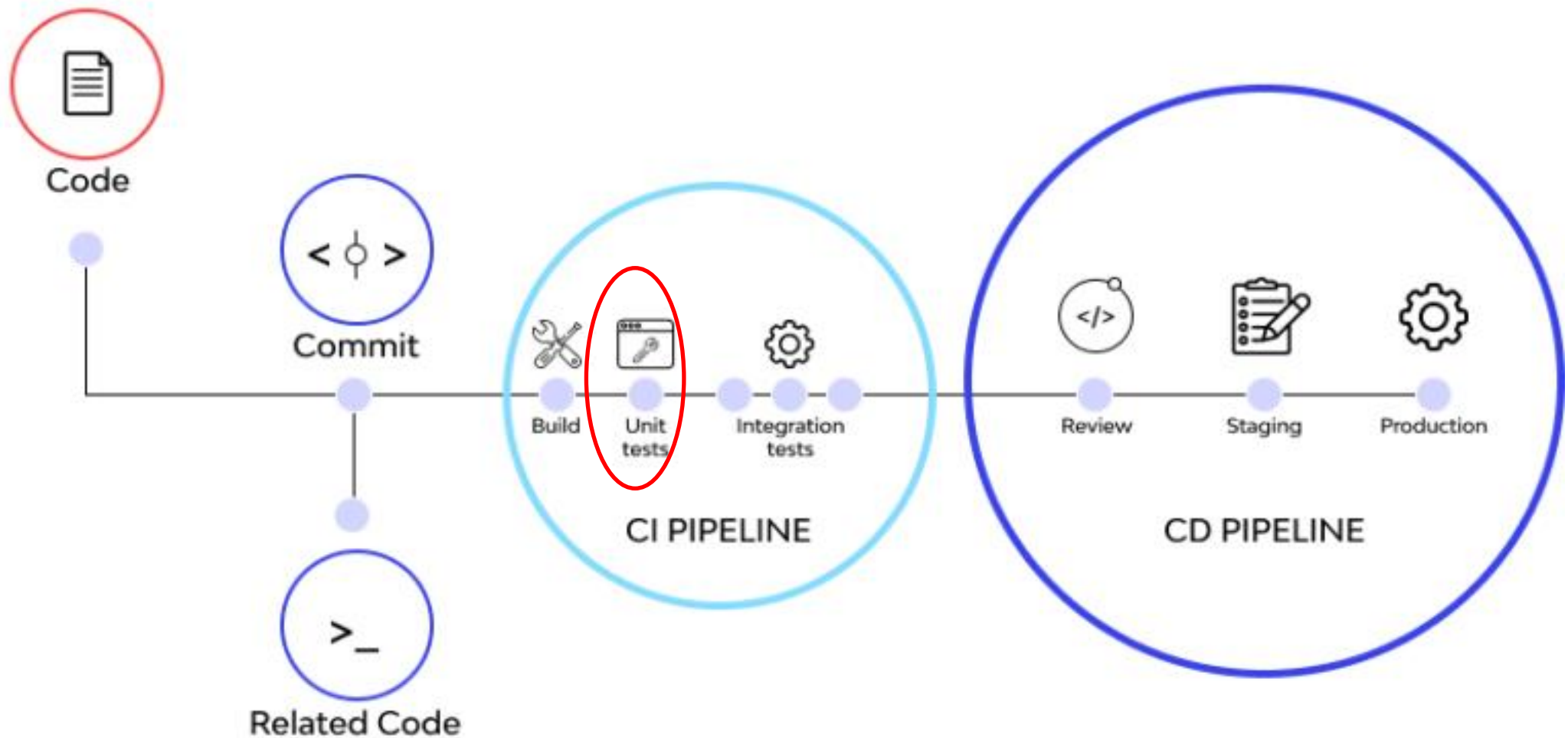
Automate as Many Tests as You Can !

Tests manuelles vs tests automatiques

- Les tests manuelles sont assez **fastidieuses** et **répétitives**.
- Vu la monotonie des tâches, le testeur 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

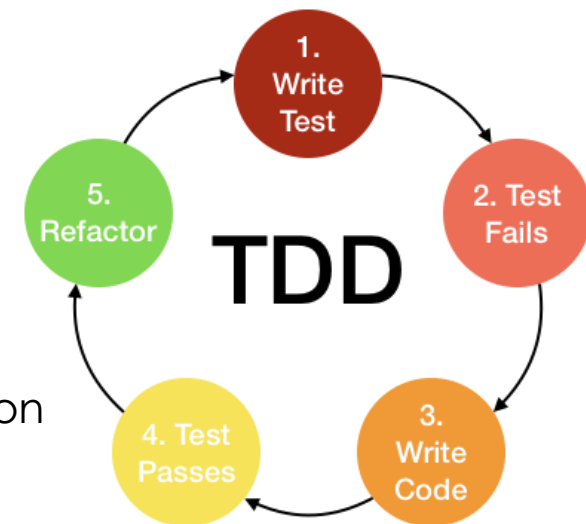
- Le test unitaire comporte 3 phases:
 - D'abord, il initialise une petite partie de l'application à tester.
 - Ensuite, il applique ensuite un stimulus au système à tester (généralement en appelant une méthode)
 - Enfin, il observe le comportement résultant.

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

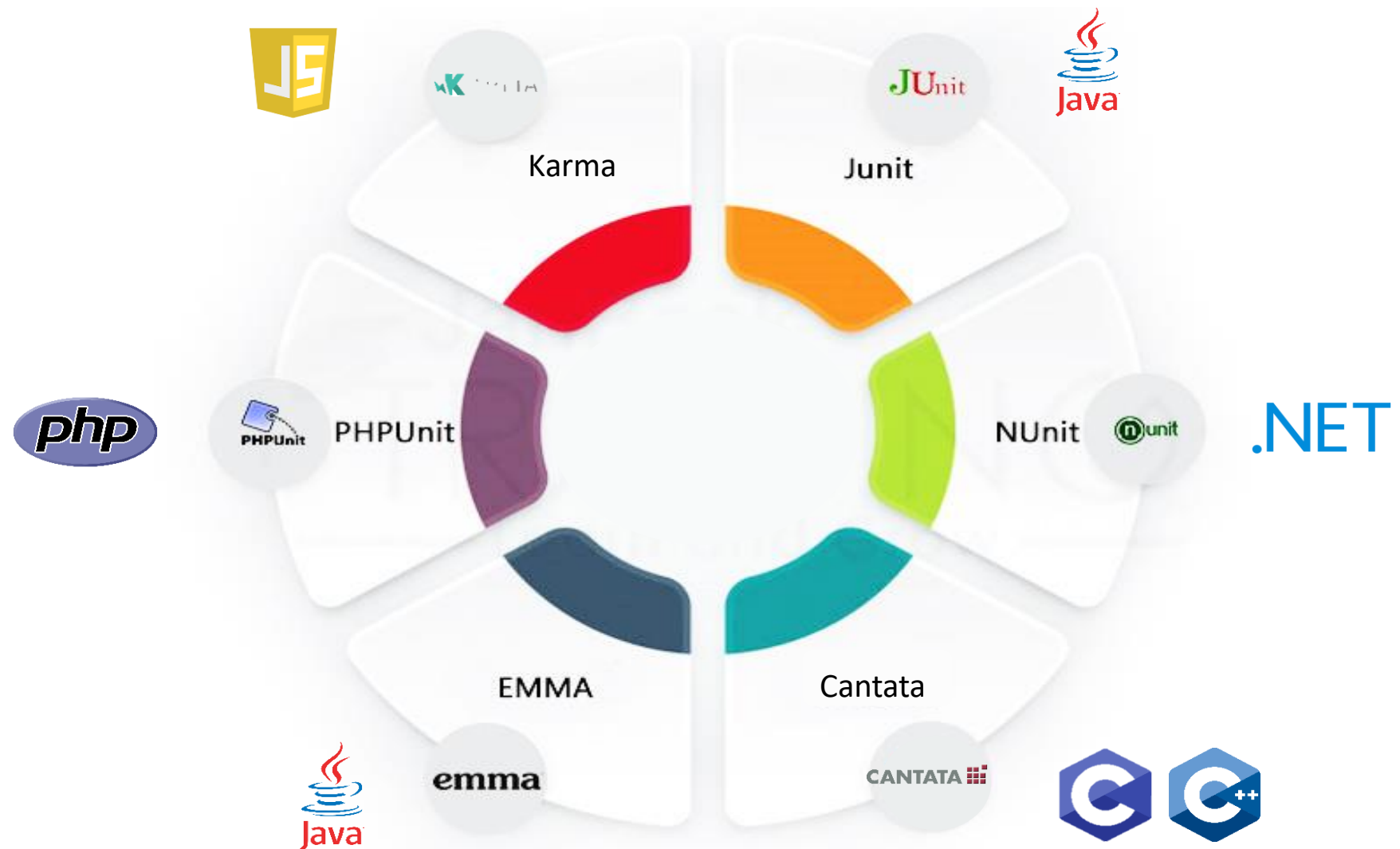
→ Ces trois phases de tests unitaires sont également appelées **Arrange**, **Act** et **Assert**, ou **AAA** en abrégé.

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.

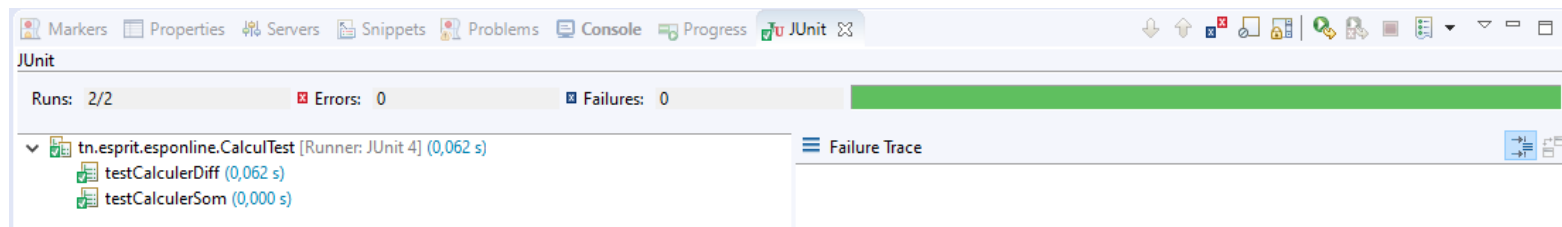


Test unitaire : Outils

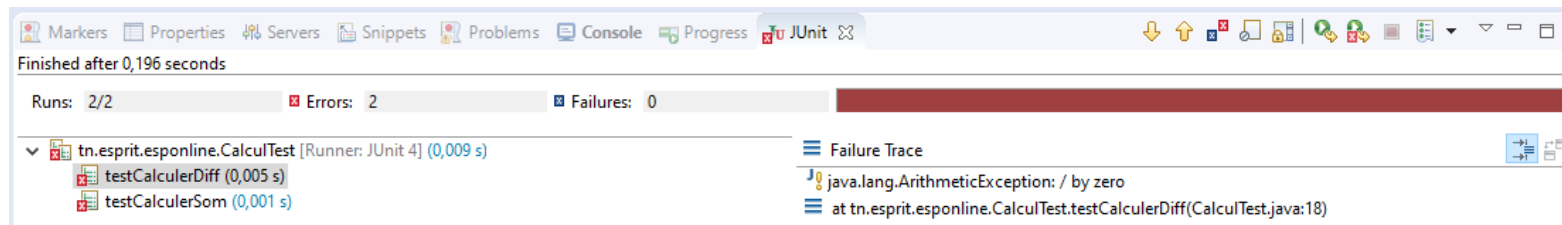


Test unitaire : Résultats

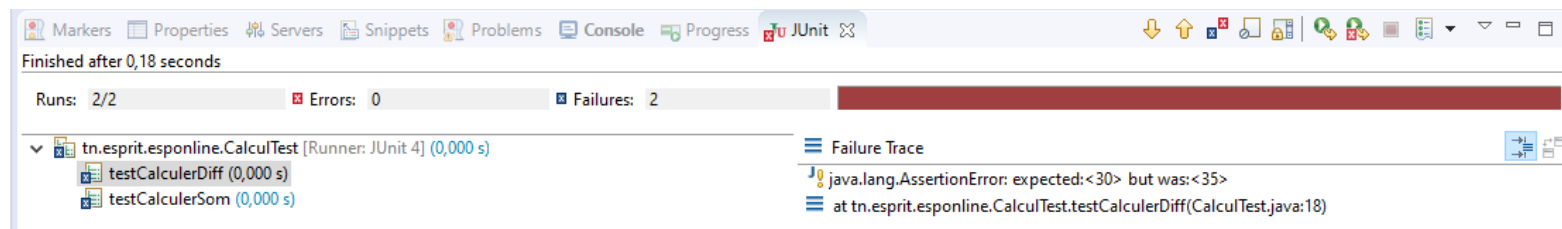
- Un test unitaire peut renvoyer 3 résultats différents :
 - Success : test réussi



- 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 – Mettre en place

- Pour mieux comprendre comment nous pouvons utiliser JUnit pour tester les méthodes unitairement, nous allons implémenter une petite application qui contient une seule entité nommée « Magasin »

```
@Entity
@Getter
@Setter
@FieldDefaults(level = AccessLevel.PRIVATE)
public class Magasin {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    long id;
    String nom;
    String adresse;
    long telephone;
    String email;

}
```

- Dans la couche service, nous avons implémenté les CRUDs qui sont générés par l'interface JpaRepository.

Test unitaire: JUnit – Mettre 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 va contenir toutes les classes des tests unitaires).
- Concentrons-nous sur la classe de test suivante :

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

    @Test
    void contextLoads() {
    }

}
```

Test unitaire: JUnit – Mettre en place

- En fait, la seule chose qu'on doit changer est :
 - l'annotation **RunWith** (JUnit invoquera la classe spécifiée dans cette annotation en tant qu'exécuteur de test au lieu d'exécuter l'exécuteur par défaut.), car elle provient de JUnit 4.
- Après le changement, la classe de test devrait ressembler à :

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
class SpringBootTestApplicationTests {

    @Test
    void contextLoads() {
    }

}
```

Test unitaire: JUnit – Mettre en place

Les annotations de JUnit 5 :

- **@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 – Mettre 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 – Mettre 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 – Mettre en place

Quelques assertions disponibles :

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

Test unitaire: JUnit – Mettre en place

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

Test unitaire: JUnit – Mettre en place

Exemples d'utilisation des assertions pour tester les méthodes de la couche repository :

```
@Test
@Order(0)
public void ajouterMagasinTest() {
    m = magasinRepository.save(m);
    log.info(m.toString());
    Assertions.assertNotNull(m.getId());
}
```

```
@Test
@Order(1)
public void modifierMagasinTest() {
    m.setAdresse("Ariana");
    m = magasinRepository.save(m);
    log.info(m.toString());
    Assertions.assertNotEquals(m.getAdresse(), actual: "Tunis");
}
```

```
@Test
@Order(5)
public void compter() {
    long taille = magasinRepository.count();
    Assertions.assertEquals(taille, magasinRepository.findAll().size());
}
```

Test unitaire: JUnit – Mettre 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: JUnit – Mettre en place

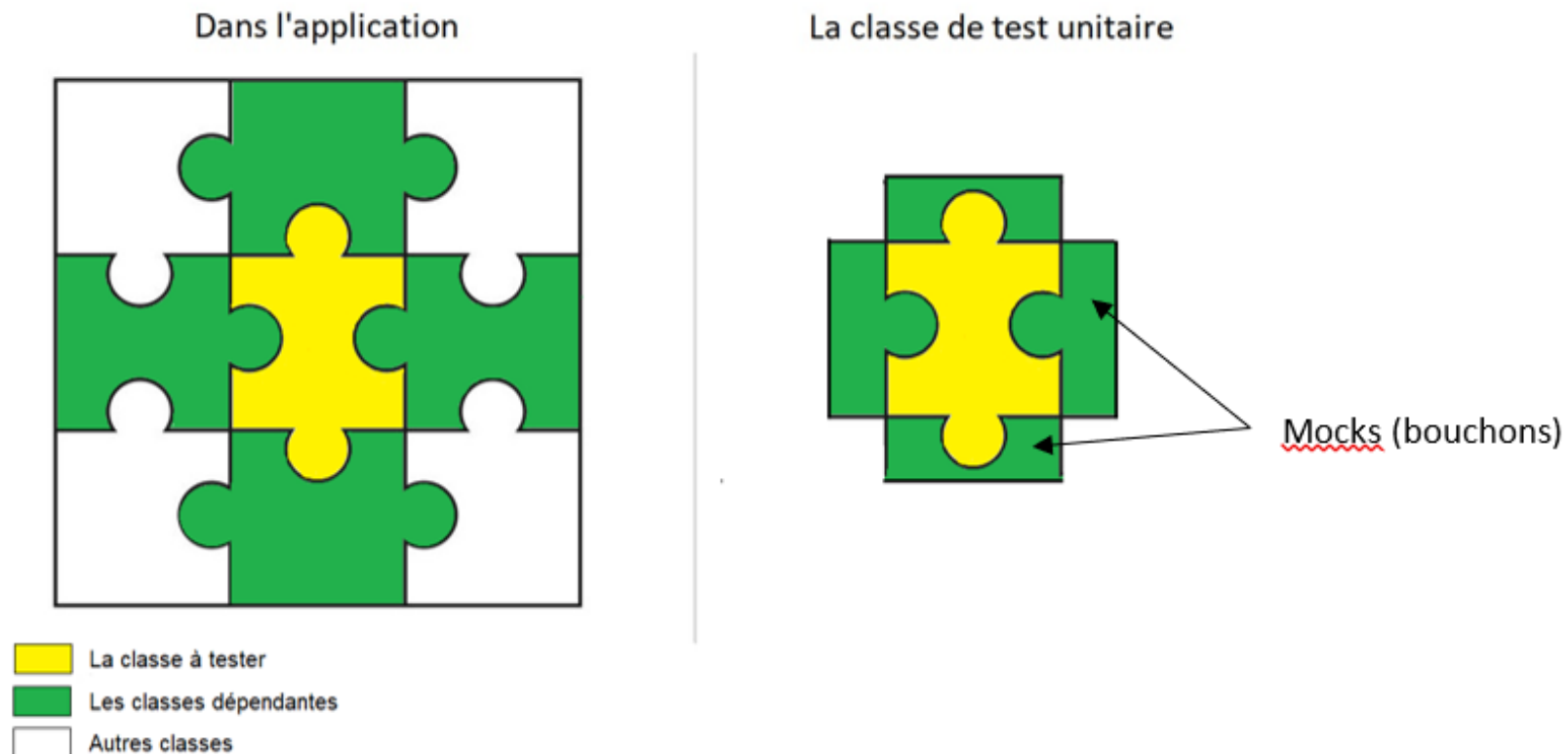
Tout cela va à l'encontre de l'objectif des tests unitaires qui sont atomiques, légers et rapides.

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: JUnit – Mettre 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: JUnit – Mettre en place

Pour pouvoir utiliser le mock, il suffit d'ajouter au niveau de la classe de test :

```
@ExtendWith(MockitoExtension.class)
```

Dans notre cas, la classe **MagasinServiceImpl** a besoin des méthodes de l'interface **MagasinRepository**. Au niveau de la classe de test, on va avoir l'implémentation suivante :

```
@Mock
MagasinRepository magasinRepo;
//ou
MagasinRepository m = Mockito.mock(MagasinRepository.class);
```

Pour déclarer le mock

```
@InjectMocks
MagasinServiceImpl magasinService;
```

Crée une instance de la classe et
injecte les mocks qui sont créés
avec l'annotation @Mock

Test unitaire: JUnit – Mettre en place

Après l'instanciation, on va entamer la phase de configuration des mocks. Pour cela, on va instancier l'objet « m » et la liste « list » en premier lieu.

```
Magasin m = Magasin.builder().nom("Le Bonus").email("bonus@gmail.com").adresse("Ghazella, Ariana").telephone(123456789).build();

List<Magasin> list= new ArrayList<Magasin>() {
    {
        add(Magasin.builder().nom("MG").email("mg@gmail.com").adresse("Ariana Soghra, Ariana").telephone(123456789).build());
        add(Magasin.builder().nom("Carrefour").email("carrefour@gmail.com").adresse("Nkhilet, Tunis").telephone(123456789).build());
    }
};
```

En deuxième lieu, on va implémenter la méthode de test « **retreiveMagasinTest** » en utilisant les paramètres « **when** » et « **thenReturn** » :

```
@Test
public void retreiveMagasinTest() {
    Mockito.when(magasinRepo.findById(Mockito.anyLong())).thenReturn(Optional.of(m));
    Magasin magasin = magasinService.chercherMagasinAvecId((long) 2);
    assertNotNull(magasin);
    log.info("get ==> " + magasin.toString());
}
```

Test unitaire: JUnit – Mettre en place

En fait, 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) « **MagasinRepository** », 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 « **MagasinRepository** », car on le simule, et on indique comment simuler.

Test unitaire: JUnit – Mettre en place

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

```
@Test
public void retrieveMagasinTest() {
    Mockito.when(magasinRepo.findById(Mockito.anyLong())).thenReturn(Optional.of(m));
    Magasin magasin = magasinService.chercherMagasinAvecId((long) 2);
    assertNotNull(magasin);
    log.info("get ==> " + magasin.toString());

    verify(magasinRepo).findById(Mockito.anyLong());
}
```



Vérifie si la méthode findById() 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 mockito et ajouter un nouveau « Stage » dans Jenkins pour lancer les tests unitaires automatiquement.



Les tests dans DevOps

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

Département Informatique
UP ASI

Bureau E204

Les tests dans DevOps

