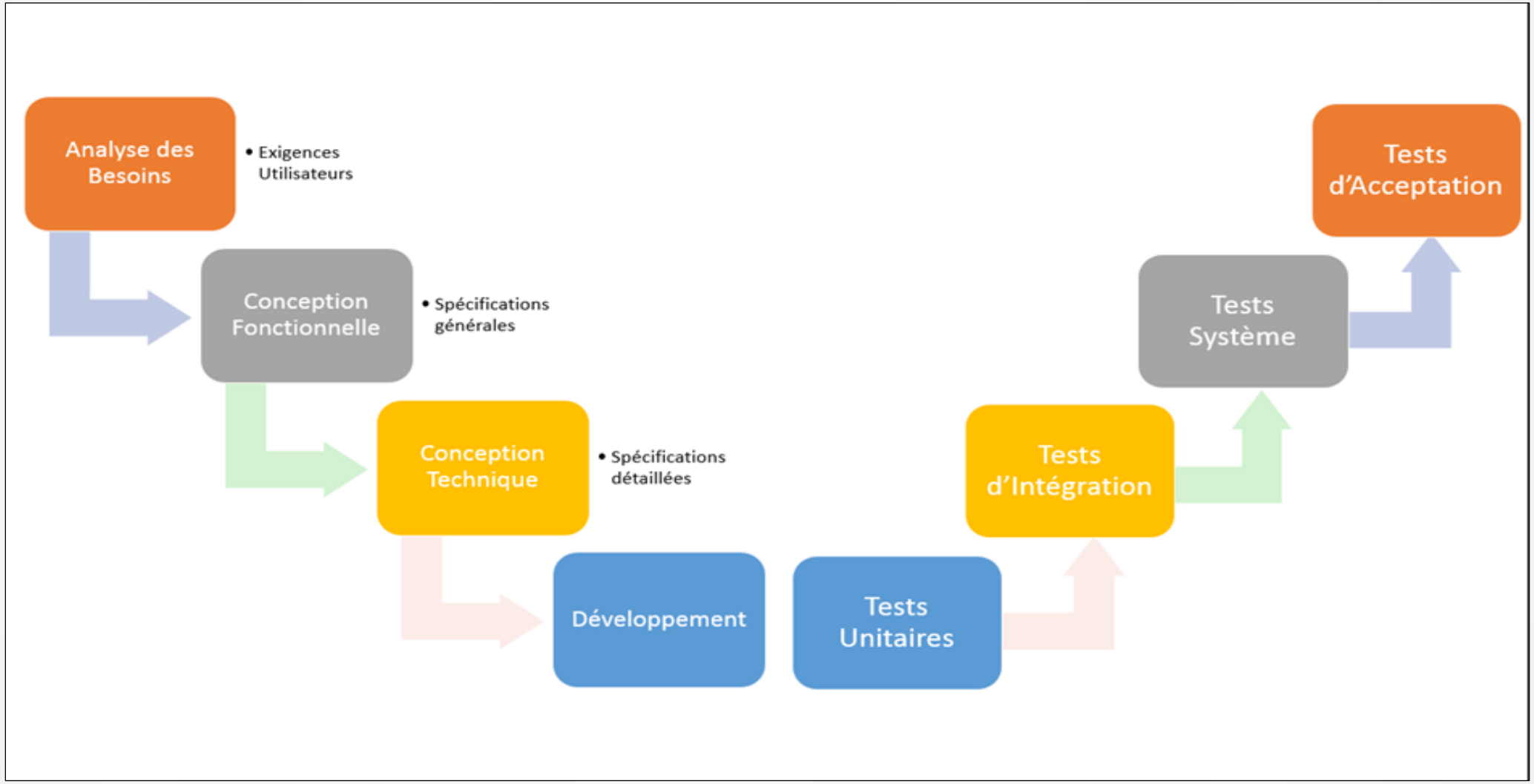
Se former autrement

PIDEV
Sprint Java

Workshop Test unitaire

C’est quoi un test unitaire ?



Unit Test ?

Un test unitaire est une technique de test logiciel qui consiste à vérifier le bon fonctionnement d'une unité de code de manière isolée.

Une unité peut être une méthode, une classe ou un composant simple.

Les tests unitaires permettent de **détecter** rapidement les erreurs, de **valider** la logique métier et d'assurer la qualité du code tout au long du développement.

Rôle et importance des tests unitaires

Les tests unitaires jouent un rôle fondamental dans le développement logiciel :

- **Détecter les erreurs très tôt**
- **Garantir la qualité du code**
- **Réduire le coût de correction des bugs**

Un code bien testé est plus robuste, fiable et évolutif.

Étapes pour réussir un test unitaire

Voici les étapes essentielles pour mettre en place un test unitaire efficace :

1. **Identifier l’unité à tester**
2. **Définir le comportement attendu**
3. **Préparer les données d’entrée**
4. **Exécuter l’unité de code**
5. **Comparer le résultat obtenu avec le résultat attendu**
6. **Corriger le code si le test échoue**
7. **Réexécuter les tests jusqu’à réussite**

Mise en place

+ JUnit 5

Outil utilisé : JUnit 5

Pourquoi JUnit ?

- ✓ standard Java
- ✓ simple à utiliser
- ✓ intégré à IntelliJ

Annotations principales :

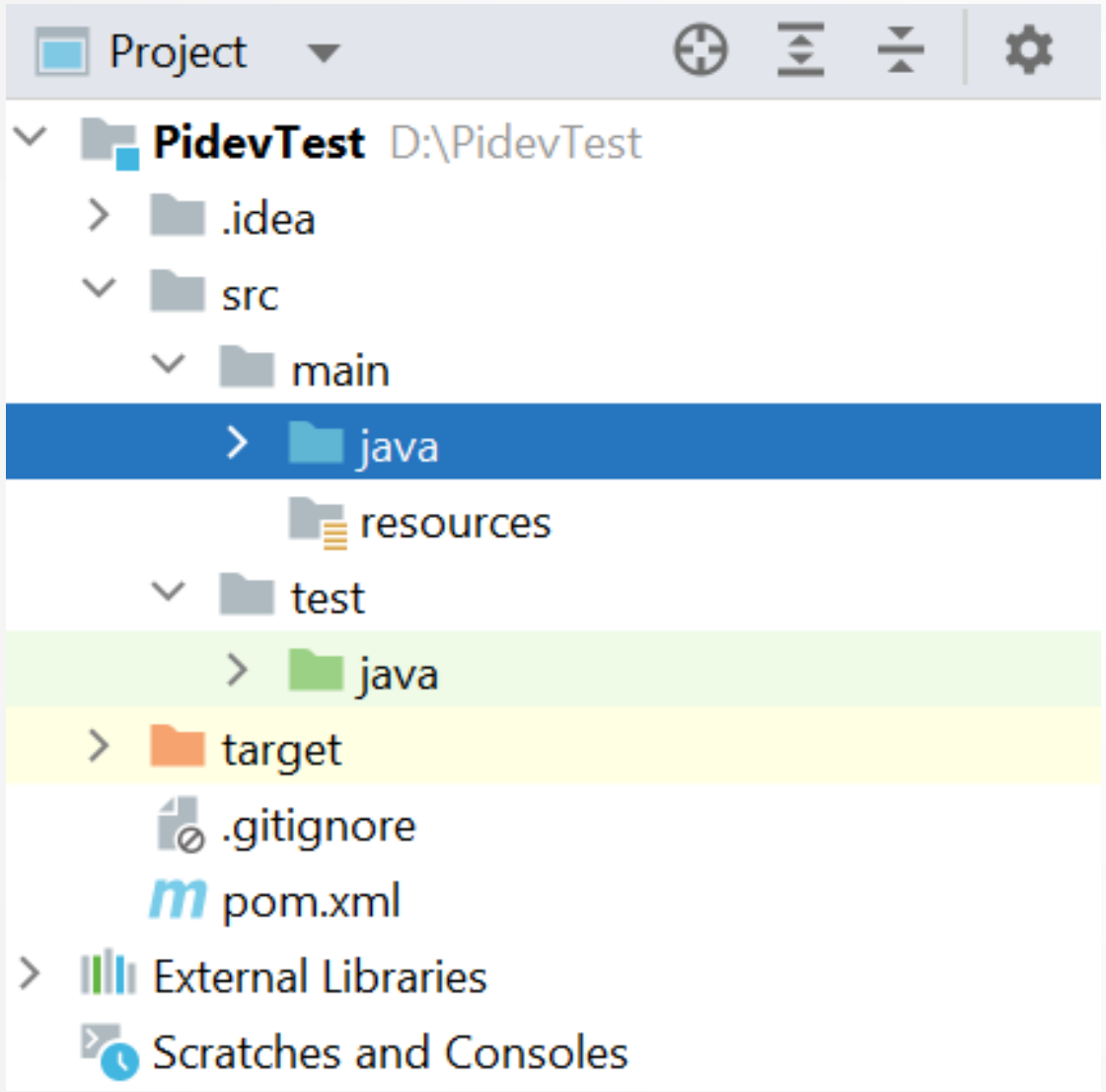
- ➢ **@Test** Indique que la méthode est un **test unitaire**. (Sans @Test, la méthode n'est jamais exécutée par JUnit.)
- ➢ **@BeforeEach** Exécute une méthode **avant chaque test**.
- ➢ **@AfterEach** Exécute une méthode **après chaque test**.

Dépendance JUnit dans le pom.xml

Dans cette étape, nous allons ajouter la dépendance JUnit afin de pouvoir **écrire et exécuter des tests unitaires.**

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.10.1</version>
    <scope>test</scope>
</dependency>
```

Structure d’un projet Maven avec tests



Le code de test est séparé du code métier.

Code de l’application

Tests unitaires

Classe de test : PersonneServicesTest.java

Cette classe se trouve dans : **src/test/java/tn/esprit/services**

Objectif de cette classe

Vérifier automatiquement que les méthodes CRUD fonctionnent sur la base de données MySQL.

Nous allons tester :

- ajouter(Personne)
- afficher()
- modifier(Personne)
- supprimer(int id)

Déclaration du package et imports JUnit

Dans la classe PersonneServicesTest:

`import org.junit.jupiter.api.*;` **Cet import permet d'utiliser les annotations JUnit pour définir les tests.**

- `@Test`
- `@BeforeAll`
- `@AfterEach`
- `@Order`

`import static org.junit.jupiter.api.Assertions.*;` **Il permet d'utiliser :**

- `assertTrue`
- `assertFalse`
- `assertEquals`
- `assertNotNull`

Les assertions servent à vérifier si le résultat obtenu est correct.

Initialisation du service (@BeforeAll)

```java
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class PersonneServiceTest {
    static PersonneServices service;

    @BeforeAll
    static void setup() {
        service = new PersonneServices();
    }
}
```

permet de **définir l'ordre d'exécution des méthodes de test**, en se basant sur l'annotation **@Order** placée sur chaque test.

Elle est utile lorsque certains tests doivent être exécutés dans un ordre précis (par exemple, ajouter avant supprimer).

Test 1 : Ajouter une personne

```
@Test
@Order(1)
void testAjouterPersonne() throws SQLException {
    Personne p = new Personne("TestNom", "TestPrenom",22);
    service.ajouter(p);
    List<Personne> personnes = service.afficher();
    assertFalse(personnes.isEmpty());
    assertTrue(
            personnes.stream().anyMatch(pers ->
pers.getNom().equals("TestNom")
            )
    );
}
```

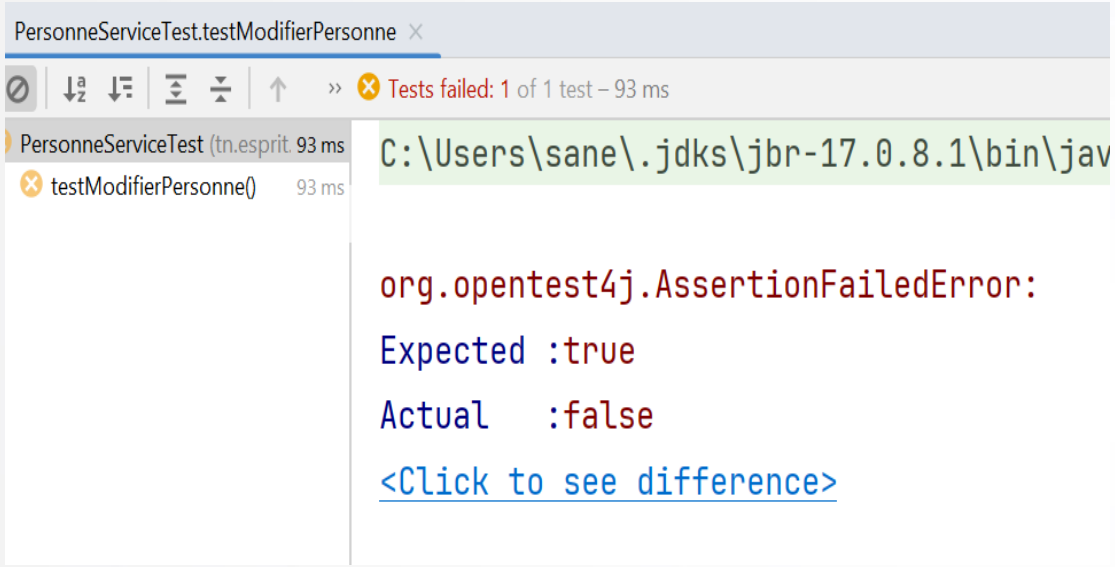
Test 1 : Ajouter une personne

```
@Test
@Order(1)
void testAjouterPersonne() throws SQLException {
    Personne p = new Personne("TestNom", "TestPrenom",22);
    service.ajouter(p);
    List<Personne> personnes = service.afficher();
    assertFalse(personnes.isEmpty());
    assertTrue(
     personnes.stream().anyMatch(pers ->
pers.getNom().equals("TestNom"))
    );
}
```

Test 2 : Modifier une personne

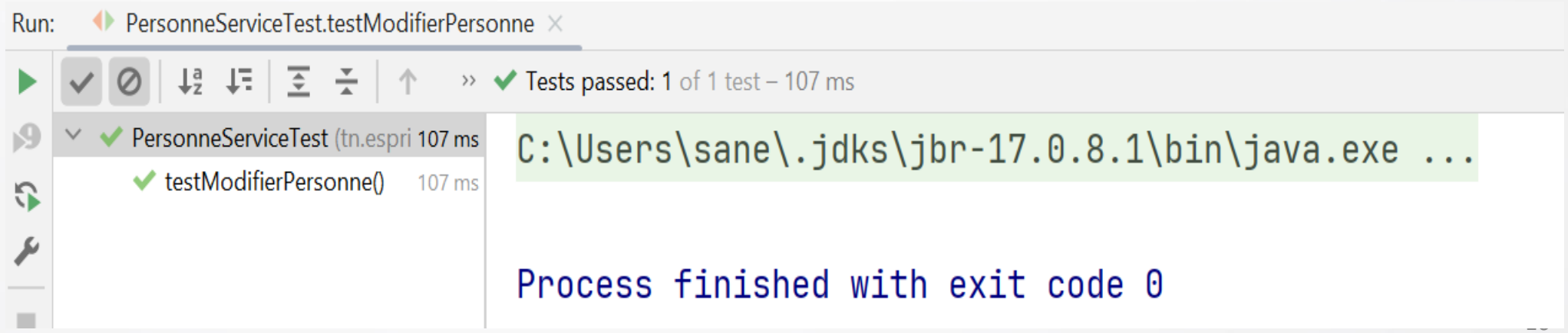
```
@Test
@Order(2)
void testModifierPersonne() throws SQLException {
    Personne p = new Personne();
    p.setId(idPersonneTest);
    p.setNom("NomModifie");
    p.setPrenom("PrenomModifie");
    service.modifier(p);
    List<Personne> personnes = service.afficher();
    boolean trouve = personnes.stream()
            .anyMatch(per ->
per.getNom().equals("NomModifie"));
    assertTrue(trouve);
}
```

Test 2 : Modifier une personne



Donc soit :

- la modification **n'a pas été appliquée**
- soit afficher() ne retourne pas ce que tu crois
- soit le test dépend d'un état incorrect de la BD



Test 3 : Supprimer une personne

```
@Test
@Order(3)
void testSupprimerPersonne() throws SQLException {
    service.supprimer(idPersonneTest);
    List<Personne> personnes = service.afficher();
    boolean existe = personnes.stream().anyMatch(p -> p.getId() == idPersonneTest);
    assertFalse(existe);
}
```

Nettoyage automatique après chaque test

```
@AfterEach
void cleanUp() throws SQLException {
    List<Personne> personnes = service.afficher();
    if (!personnes.isEmpty()) {
    Personne last = personnes.get(personnes.size() - 1);
    service.supprimer(last.getId());
} }
```

Après chaque test, la base de données est nettoyée automatiquement.

Un bon test ne laisse aucune trace

Merci pour votre attention

+ JUnit 5