FRAMEWORK JUNIT & MOCKITO

esprit
Se former autrement

PLAN DU COURS

- Introduction
- Quelques Types de Tests
- Tests Unitaires
- Utilisation de JUnit / Mockito
- Place à la Pratique

INTRODUCTION

- Un Testeur est un rôle à part entière (Testeur, QA, …).
- Membre de la Team dans une équipe Scrum.
- Certification ISTQB
- Détecter le maximum d'anomalies (Bug) et de disfonctionnements (lenteur, faille de sécurité, …).

INTRODUCTION

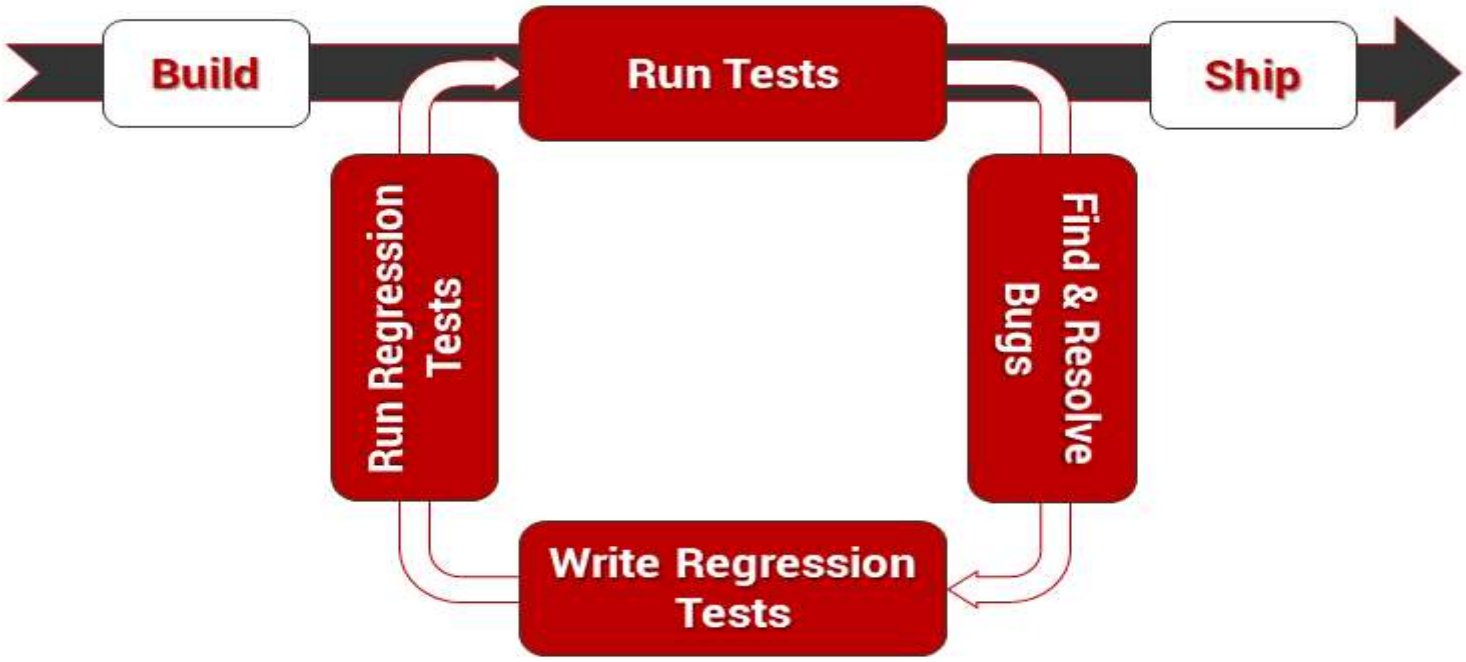
- Il existe différents niveaux de test :
  - Test Unitaire
  - Test d'Intégration
  - Test de Qualité de Code
  - Test de Montée en Charge
  - Test Fonctionnel (Métier)
  - Test Sécurité
  - ….

TEST D'INTÉGRATION

- L'intégration, c'est assembler plusieurs composants logiciels élémentaires pour réaliser un composant de plus haut niveau.
- **Exemple 1**: Au niveau d'une même application : Intégrer une classe Client et une classe Produit pour créer un module de commande sur un site marchant, c'est de l'intégration.
- **Exemple 2**: Intégration au niveau de deux applications : Une application expose des WS, l'autre les consomme. Il faut vérifier que les 2 applications fonctionnent bien ensemble (exposition et consommation).
- **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.
- Outils : **Jenkins**, …

TEST DE RÉGRESSION

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



- Outils : **Quality Center**, …

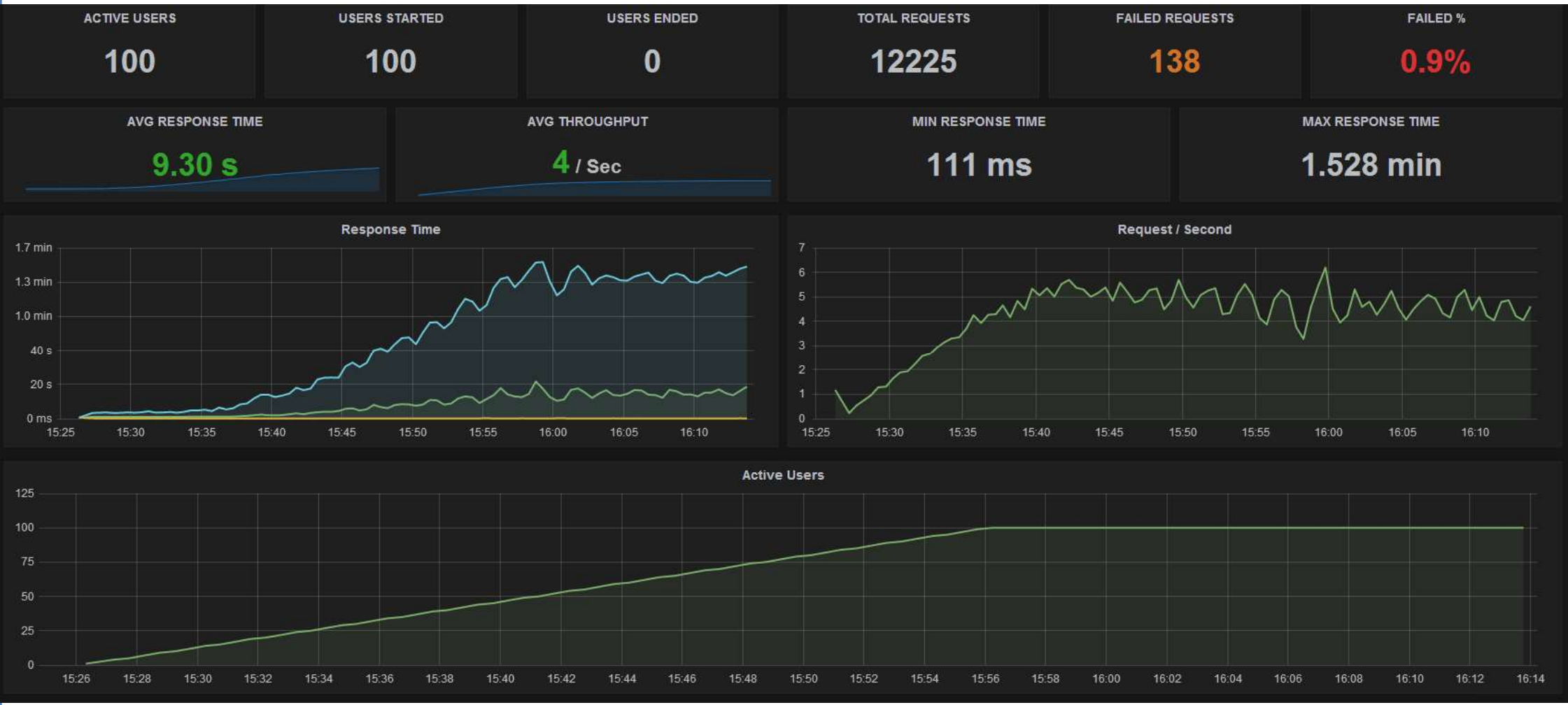
TEST DE MONTÉE EN CHARGE

- **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.

- Outil : **JMeter**, …

TEST DE MONTÉE EN CHARGE

- Nombre utilisateurs, temps de réponse, requêtes perdues … :



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 (SQL Injection, …).

TEST FONCTIONNEL (MÉTIER)

- Il ne s'agit pas de tests les méthodes une à une (cela s'appelle?), mais de tester un scenario complet (login, achat, paiement par exemple dans une application dd vente en ligne).
- Outil : **Selenium** (automatise des actions faite sur un navigateur en enregistrant les actions d'n utilisateur et ne les reproduisant), ...

```
describe('Visual testing of landing pages', () => {
    it('looks good responsive', () => {
        browser.url('/?utm_source=automated-testing');
        browser.execute('/*@visual.init*/', 'Responsive Test');
        browser.execute('/*@visual.snapshot*/', 'Home Page');

        browser.url('/selenium-java-sale/?utm_source=automated-testing');
        browser.execute('window.scrollTo(0,document.body.scrollHeight)');
        browser.execute('/*@visual.snapshot*/', 'Sales Page', {scrollAndStitchScreenshot: true});

        browser.url('/selenium-webdriver-java-course/?utm_source=automated-testing');
        browser.execute('window.scrollTo(0,document.body.scrollHeight)');
        browser.execute('/*@visual.snapshot*/', 'Selenium Java Landing Page', {scrollAndStitchScreenshot: true});

        browser.url('/selenium-java-2/?utm_source=automated-testing');
        browser.execute('window.scrollTo(0,document.body.scrollHeight)');
        browser.execute('/*@visual.snapshot*/', 'Selenium Java Landing Page v2', {scrollAndStitchScreenshot: true});
        browser.execute('/*@visual.end*/');
    });
});
```

TEST UNITAIRE : DÉFINITION

- Un test unitaire est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel. Il s'agit d'un code.
- En POO, on teste au niveau des classes
- Pour chaque classe (MyClass), on a une classe de test (MyClassTest).
- Outil : **JUnit**, **Mockito** …

TEST UNITAIRE VS TEST D'INTÉGRATION

TEST UNITAIRE : QUELQUES RÈGLES

- Doit être isolé : il doit être indépendant
- N'est pas un test de bout en bout : il agit que sur une portion de code
- Doit être déterministe : le résultat doit être le même pour les mêmes entrées
- Est le plus petit et le plus simple possible

TEST UNITAIRE : QUELQUES RÈGLES

- Ne teste pas d'enchainement d'actions
- Etre lancé le plus souvent possible : intégration continue
- Etre lancé le plus tôt possible : détection des bug plus rapidement
- Couvrir le plus de code possible
- Etre lancé a chaque modification

TEST UNITAIRE : AVANTAGE ET INTÉRÊT

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

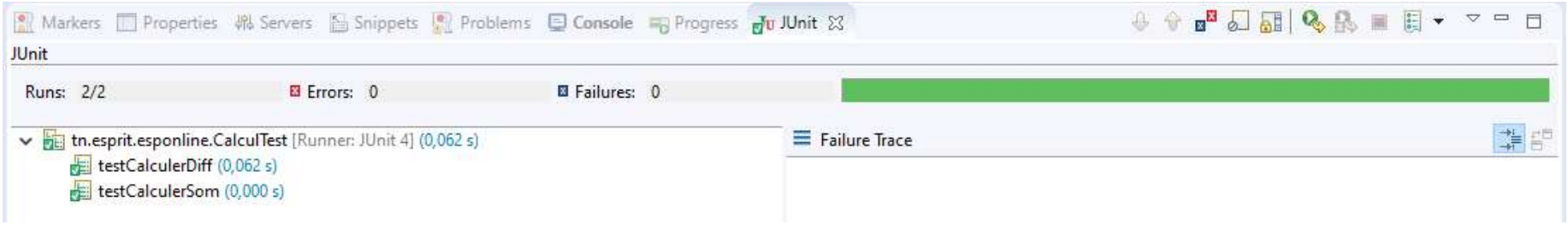
* Le **test-driven development** (**TDD**) ou en français développement piloté par les tests est une technique de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel.

TEST UNITAIRE : OUTIL DE TEST

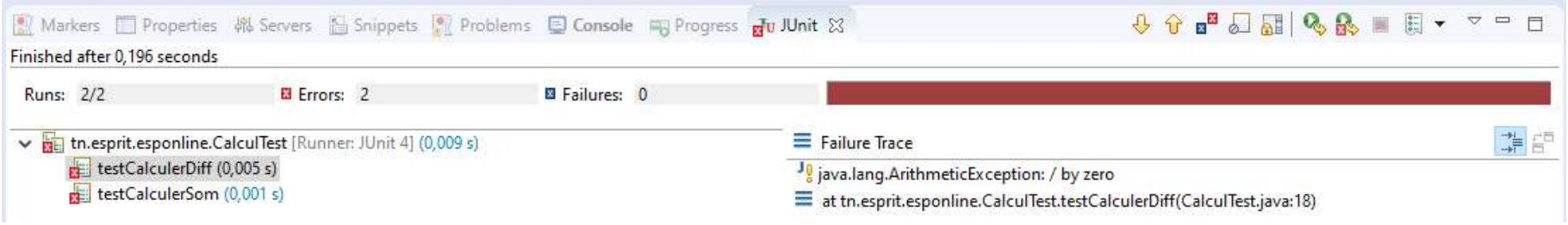
PHP	JS	SQL	JAVA
PHPUnit SimpleTest	JSUnit	SQLUnit	JUnit

TEST UNITAIRE : LES RÉSULTATS

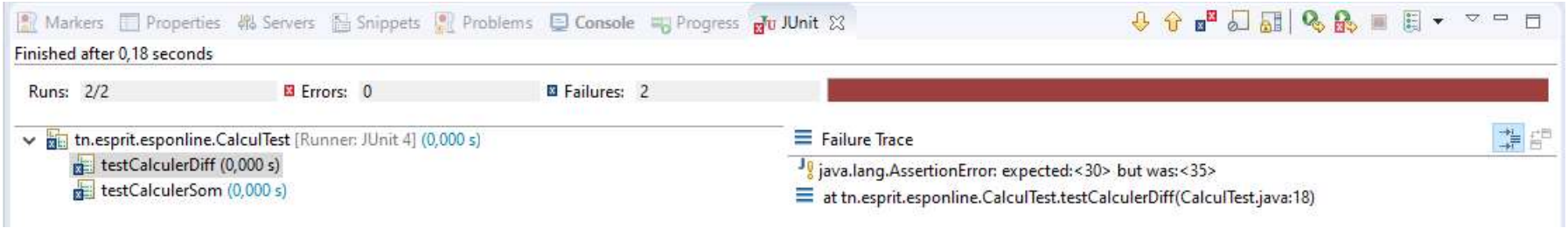
- Un test unitaire peux renvoyer 3 résultats différents :
  - **Success** : test réussi



  - **Error** : erreur inattendue a l'exécution



  - **Failure** : au moins une assertion est fausse



UTILISATION DE JUNIT

- 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** (a partir de JUnit4).

TP – Projet Spring Boot avec JUnit

- Utiliser le projet Spring Boot déjà créé dans les cours précédents.

- Aller dans le dossier de test src/test/java.
- Créer le package tn.esprit.rh.achat.service et la classe de test correspondante au service que vous avez choisi.

- Inspirez vous du contenu de la classe de test déjà fournie ci-dessous pour créer le contenu de la classe de test du service XyzServiceImpl (injecter le service correspondant dans cette classe de test et ajouter les tests nécessaires pour tester le **CRUD**).

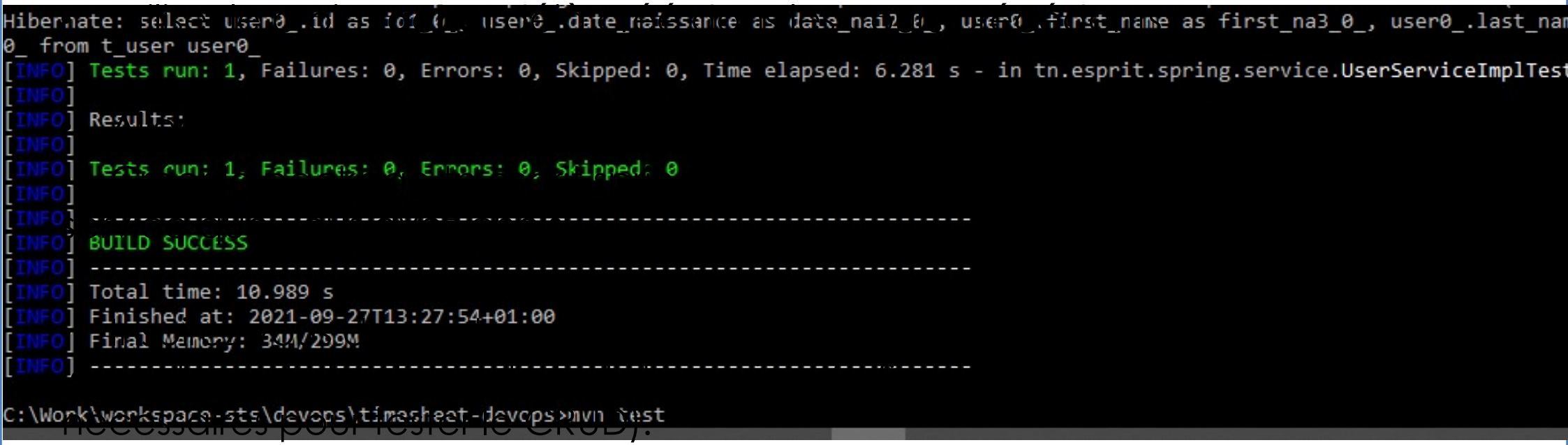
TP – Projet Spring Boot avec JUnit

- **Tester les méthodes dans votre IDE IntelliJ**
- **Tester les méthodes en ligne de commande (commande mvn test)**
- **Dans un pipeline Jenkins (stage à ajouter qui contient la commande Maven qui lance les tests).**

TP – Projet Spring Boot avec JUnit

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
@TestMethodOrder(OrderAnnotation.class)
class UserServiceImplTest {

    @Autowired
    IUserService us;

    @Test
    @Order(1)
    public void testRetrieveAllUsers() {
      List<User> listUsers = us.retrieveAllUsers();
      Assertions.assertEquals(0, listUsers.size());
    }

    ….
}
```

TP – Projet Spring Boot avec JUnit



nécessaires pour tester le CRUD).

- Le TP ci-dessus pour expliquer l'utilisation basique de JUnit. C'est aussi pour faciliter le TP ci-dessous (Mockito) :

- **Le Projet final utilisera MOCKITO de préférence et non JUNIT (voir détails ci-dessous)** :

Tests Unitaires : Mockito

- Quelques fois un test a besoin d'un composant donné pour s'exécuter.
- Par exemple pour tester une fonctionnalité, nous avons besoin du retour d'un Web Service, qui n'a pas été développé pour l'instant, ou la ase de données est H.S (Hors Service),
- Il est alors utile d'utiliser des bouchons (**MOCK**) pour isoler le test.
- De plus un bouchon permet de tester tout les cas (valeur correcte, valeur erronée, …), sans rien modifier au niveau du web service appelé.

Tests Unitaires : Mockito

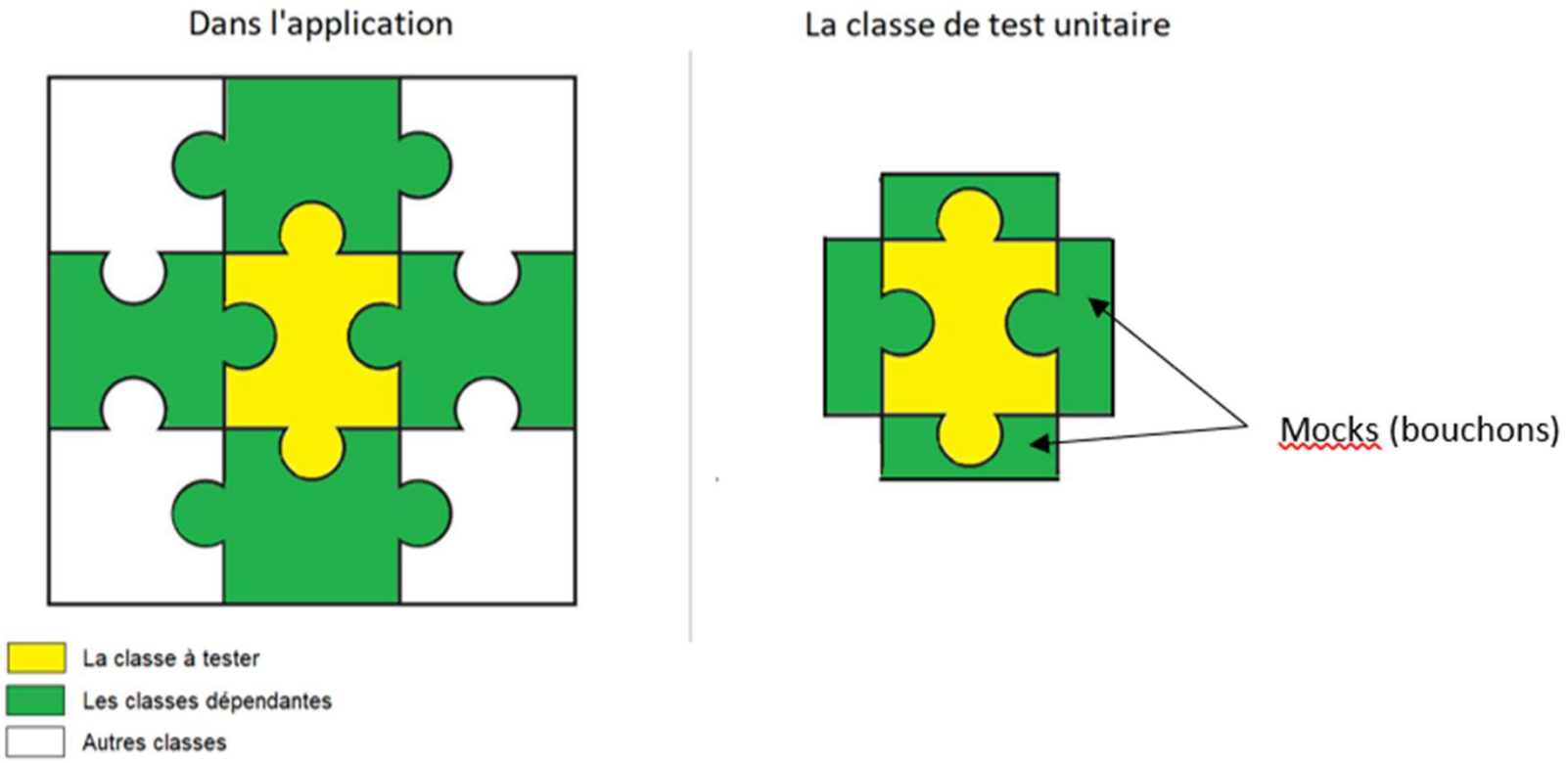
Tout cela va à l'encontre de l'objectif des tests unitaires qui sont atomiques, légers et rapides.

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.

Tests Unitaires : Mockito

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 :



Tests Unitaires : Mockito

Classe de Tests avec Mockito : Pour pouvoir utiliser le Mock, il suffit d'ajouter au niveau de la classe de test (exemple dans UserServiceImplMock) :

```
@SpringBootTest
@ExtendWith(MockitoExtension.class)
class UserServiceImplMock {
```

Le service UserServiceImpl a besoin de méthode du repository UserRespository, dans la classe de test ajouter :

```
@Mock
UserRepository userRepository;
// Ou
UserRepository userRepository = Mockito.mock(UserRepository.class);
```

Tests Unitaires : Mockito

Classe de Tests avec Mockito : Créer une instance du Service et Injecter les Mocks qui sont créées avec l'annotation @Mock :

```
@InjectMocks
UserServiceImpl userService;
```

Il faut initialiser les objets à tester (dans la classe de test) :

```
User user = new User("f1", "l1", new Date(), Role.ADMINISTRATEUR);
List<User> listUsers = new ArrayList<User>() {
  {
    add(new User("f2", "l2", new Date(), Role.ADMINISTRATEUR));
    add(new User("f3", "l3", new Date(), Role.ADMINISTRATEUR));
  }
};
```

Tests Unitaires : Mockito

Implémentons maintenant la méthode de test « **retreiveUserTest** » en utilisant les paramètres « **when** » et « **thenReturn** » :

```
@Test
public void testRetrieveUser() {
  Mockito.when(userRepository.findById(Mockito.anyLong())).thenReturn(Optional.of(user));
  User user1 = userService.retrieveUser("2");
  Assertions.assertNotNull(user1);
}
```

Tests Unitaires : Mockito

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) « **UserRepository** », alors on retourne l'objet «**user**» 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 « **UserRepository** », car on le simule, et on indique comment simuler.

Tests Unitaires : Résumé

- La phase de tests unitaire c'est une phase primordiale dans la réalisation d'un projet.
- On peut tester nos services réels (tests de toutes les couches et connexion à la base : avec JUnit), ou en testant que le service lui-même sans un vrai accès à la couche d'accès aux données : avec Mockito).

- Mockito nécessite JUnit.

- 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).

TP – Projet Spring Boot avec JUnit / Mockito

Chaque étudiant utilisera **Mockito** pour tester le CRUD d'un service de son choix du projet « achat ».

L'appel aux tests unitaires se fera :

- **En ligne de commande (commande mvn test) ,**
- **Puis dans un pipeline Jenkins (stage à ajouter qui contient la commande Maven qui lance les tests).**