

JUnit



Introduction

Quels sont les besoins d'un développeur ?

- ▼ **Tester son code**
- ▼ **Documenter son code**
- ▼ **Maintenir son code**

Le TDD (Test Driven Development) est une méthode de développement qui permet de répondre à ces besoins, en plus de permettre de développer plus rapidement.

Qu'est ce que le TDD ?

TDD : Test Driven Development

Les principes du TDD:

- ▼ **Ecrire un test**
- ▼ **Ecrire le code pour que le test passe**
- ▼ **Refactoriser le code**

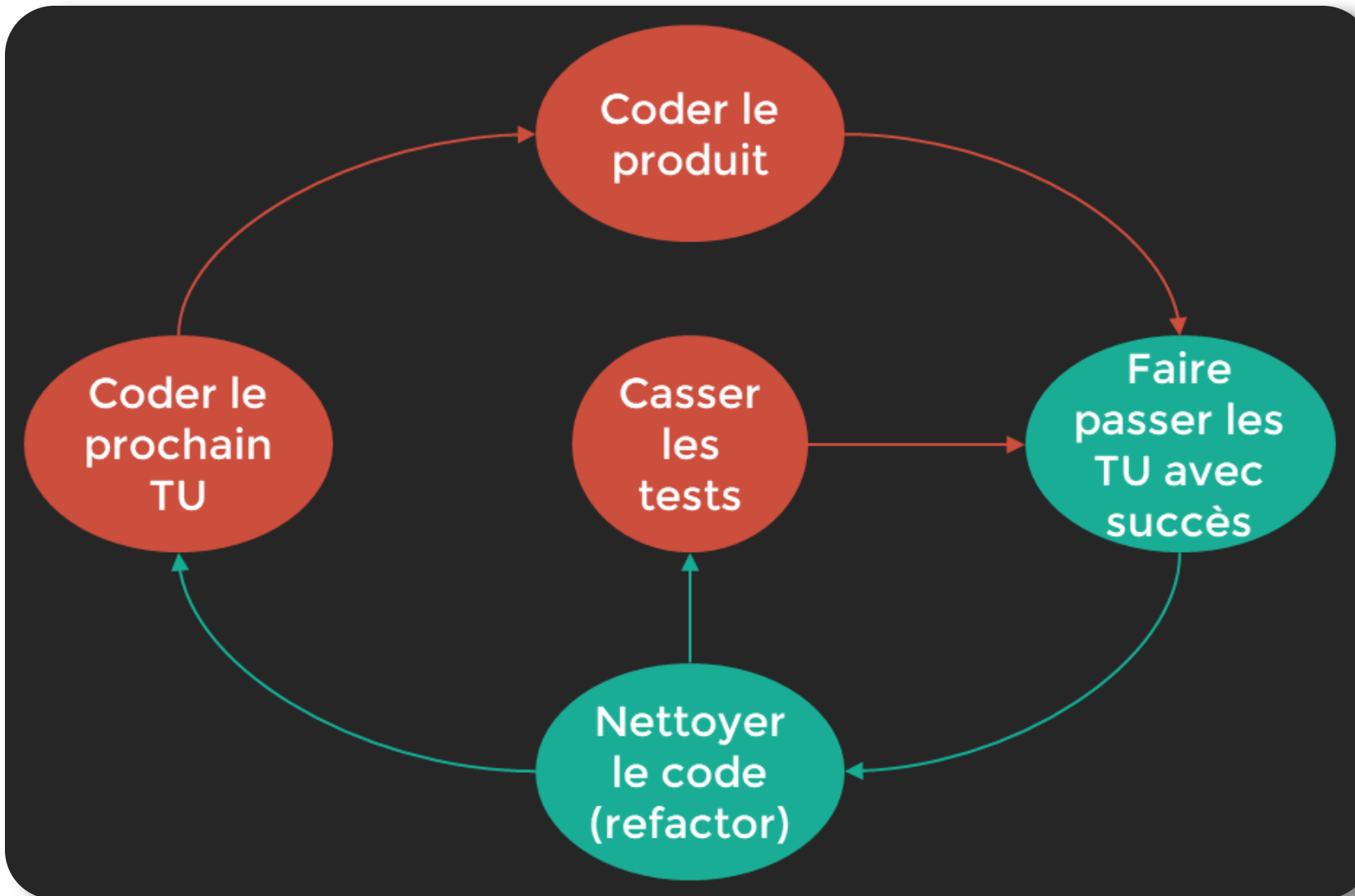
Le cycle red-green-refactor

Le cycle red-green-refactor vous guidera dans l'écriture de vos tests.

red : Vous écrivez un test qui échoue, car le code n'est pas encore implémenté. (Le resultat sera rouge)

green : Vous écrivez le code pour que le test passe. (Le resultat sera vert)

Refactor : Vous refactorisez votre code pour le rendre plus lisible et plus maintenable. (Le resultat doit rester vert)



Test unitaire

Soumettre une partie de code à un test

Il permet la mise en place d'un contrat entre le développeur et le client.

Ou le client peut :

- ▼ **Tester le code**
- ▼ **Quantifier la qualité du code**
- ▼ **Maintenir le code**
- ▼ **Sécuriser le code**

Exemple de test

Demande du client : "Je veux que la fonction `add` retourne la somme de deux nombres"

```
public class Test {  
    public void testAdd() {  
        Calculatrice calculatrice = new Calculatrice();  
        int resultat = calculatrice.add(1, 2);  
        System.out.println("resultat = " + resultat);  
        System.out.println("resultat attendu = 3");  
    }  
}
```

Problématique de tests

- ▼ La lisibilité des résultats
- ▼ La complexité des tests
- ▼ La maintenance des tests
- ▼ La répétition des tests

Une solution : JUnit

- ▼ Gain de temps
- ▼ Gain de productivité
- ▼ Gain de qualité
- ▼ Gain de maintenabilité

Qu'est-ce que JUnit ?

- ▼ JUnit est un framework de test unitaire
- ▼ Développé par Erich Gamma et Kent Beck en 2002
- ▼ JUnit est un framework open source
- ▼ JUnit5 est la dernière version de JUnit sortie en 2017

Pourquoi utiliser Junit ?

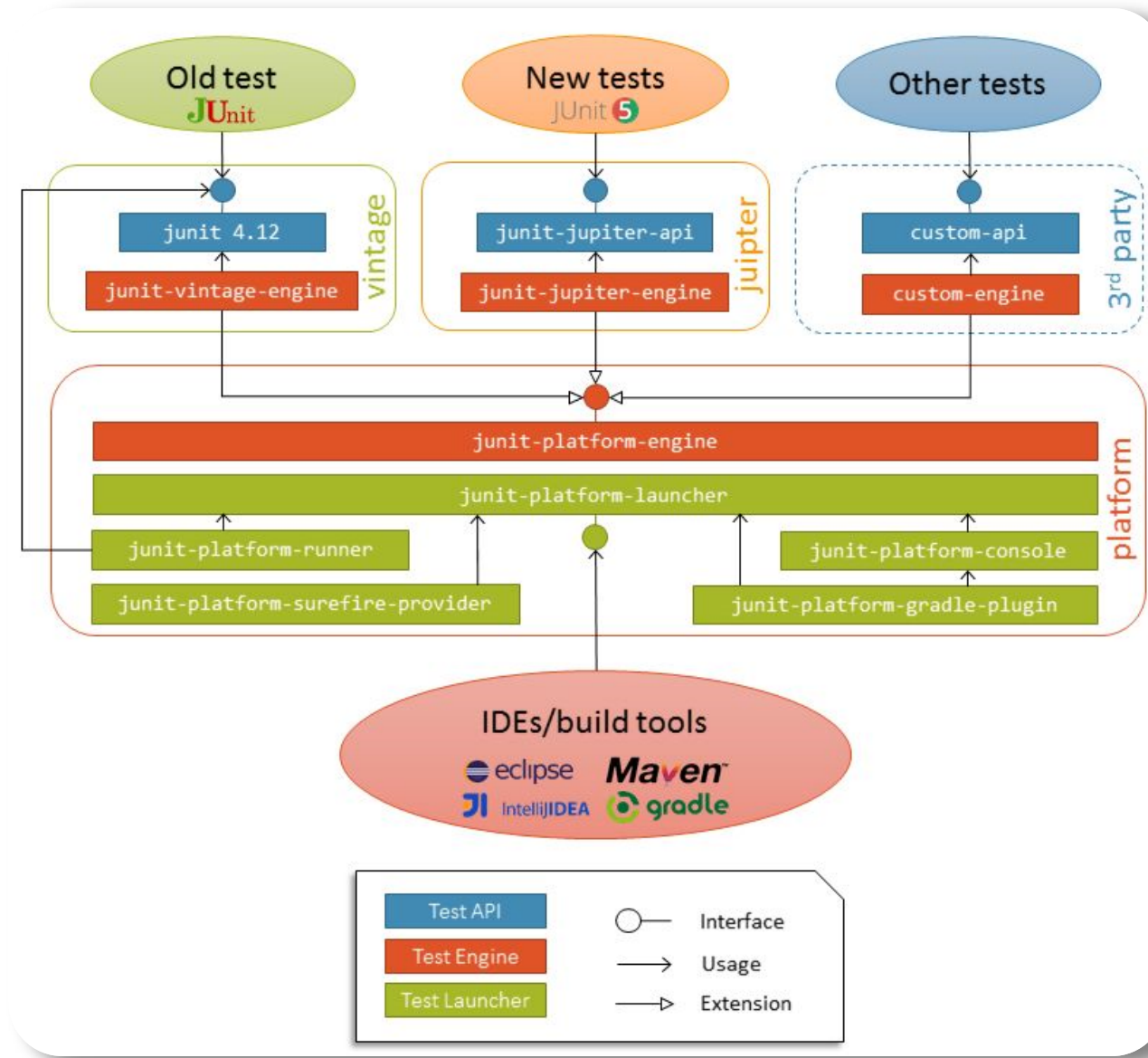
Pourquoi Junit ?

- ▼ Junit est un framework simple et facile à utiliser
- ▼ Junit est un framework très populaire
- ▼ Junit est un framework très complet
- ▼ Disponible sur Maven Central

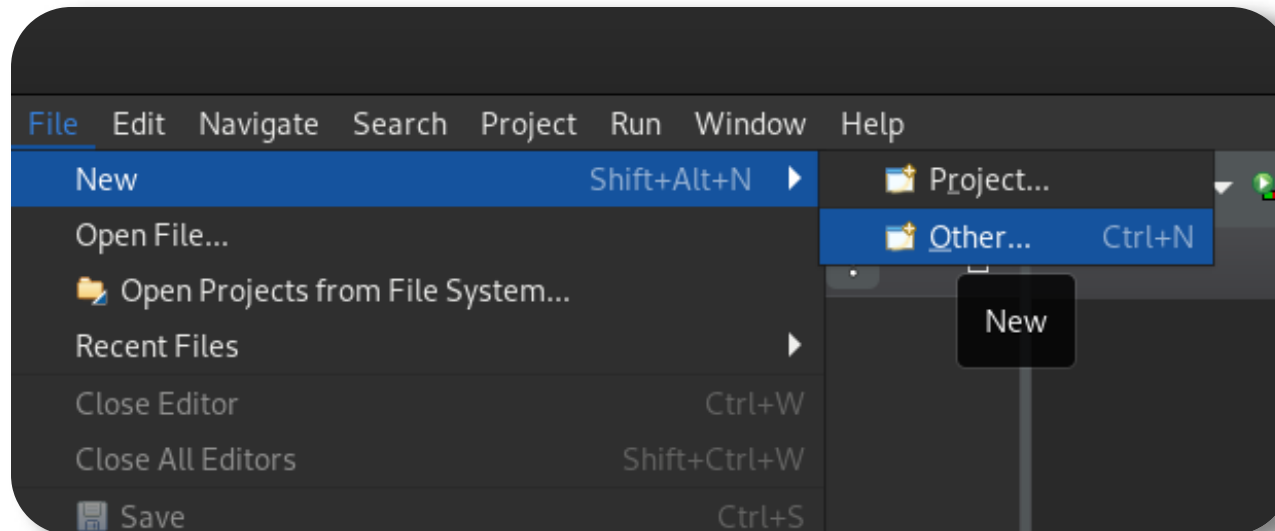
Quel est l'architecture de Junit ?

Architecture de Junit

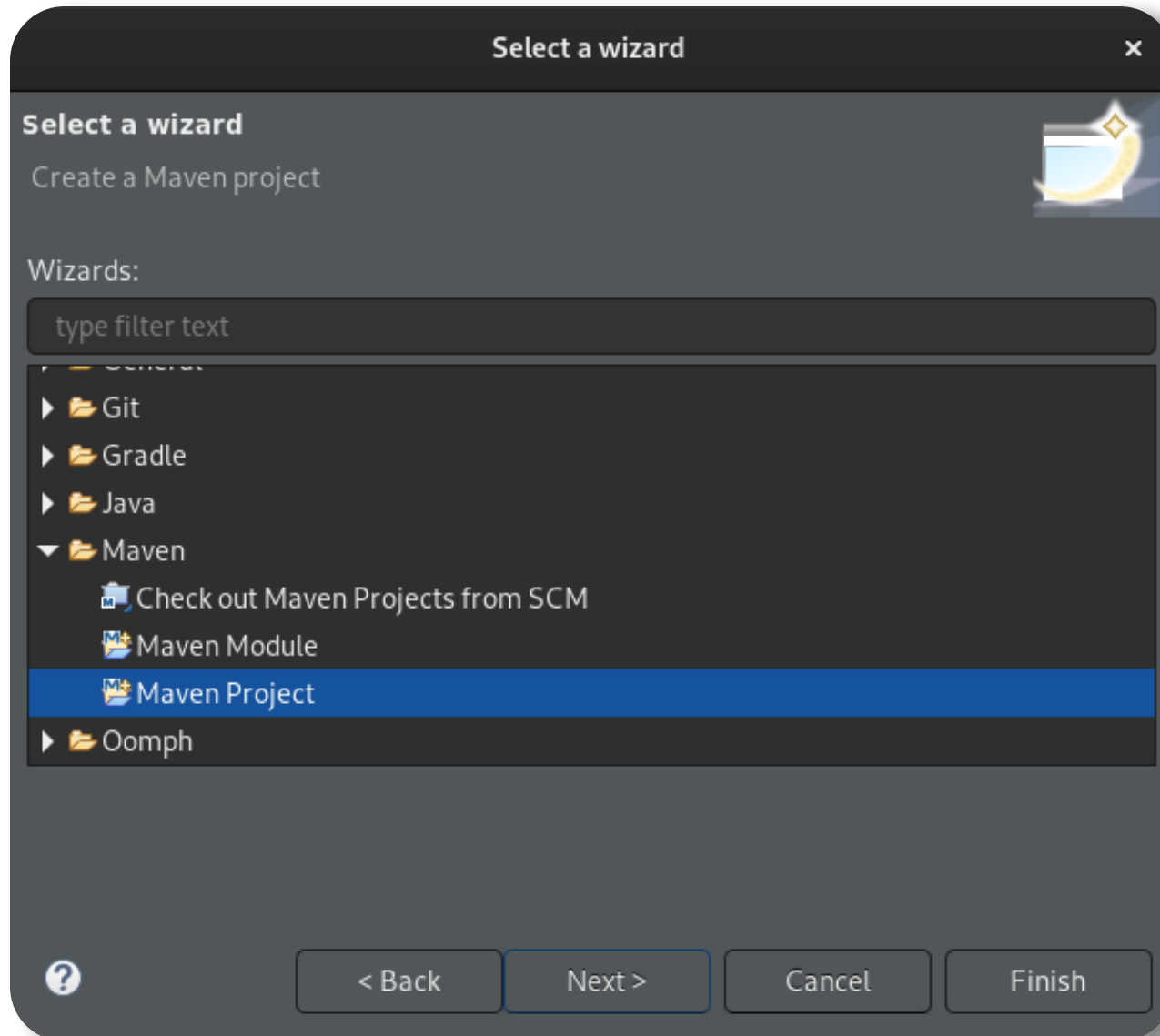
- ▼ Junit est composé de 4 modules
 - ▼ Junit Platform : Module de base de Junit
 - ▼ 3rd party extensions : Module qui permet d'ajouter des extensions à Junit
 - ▼ Junit Jupiter : Module pour les tests unitaires
 - ▼ Junit Vintage : Module pour les tests Junit 3 et 4



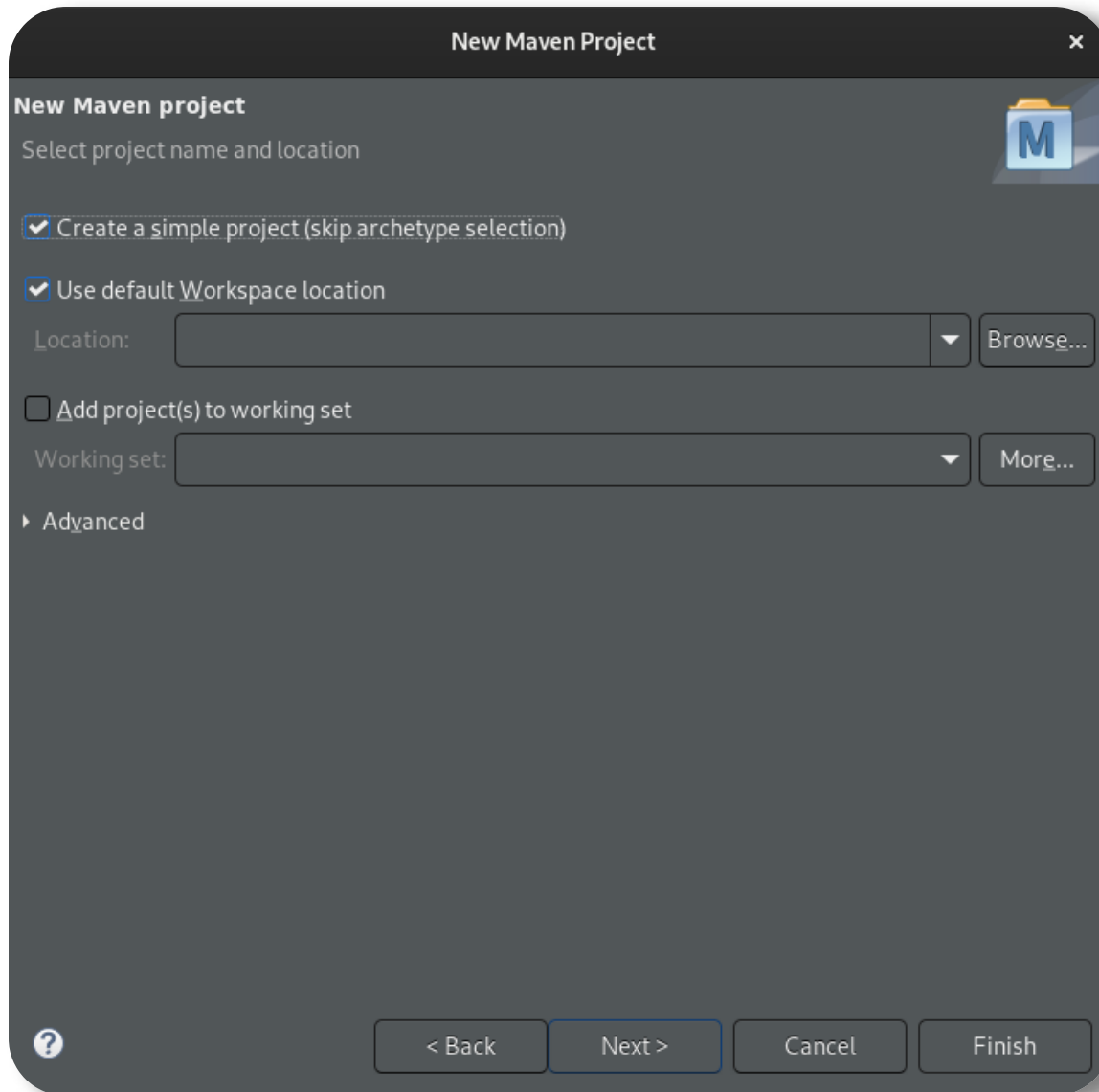
Installation



Installation



Installation



New Maven Project

New Maven project

Select project name and location

☒ Create a simple project (skip archetype selection)

☒ Use default Workspace location

Location: Browse...

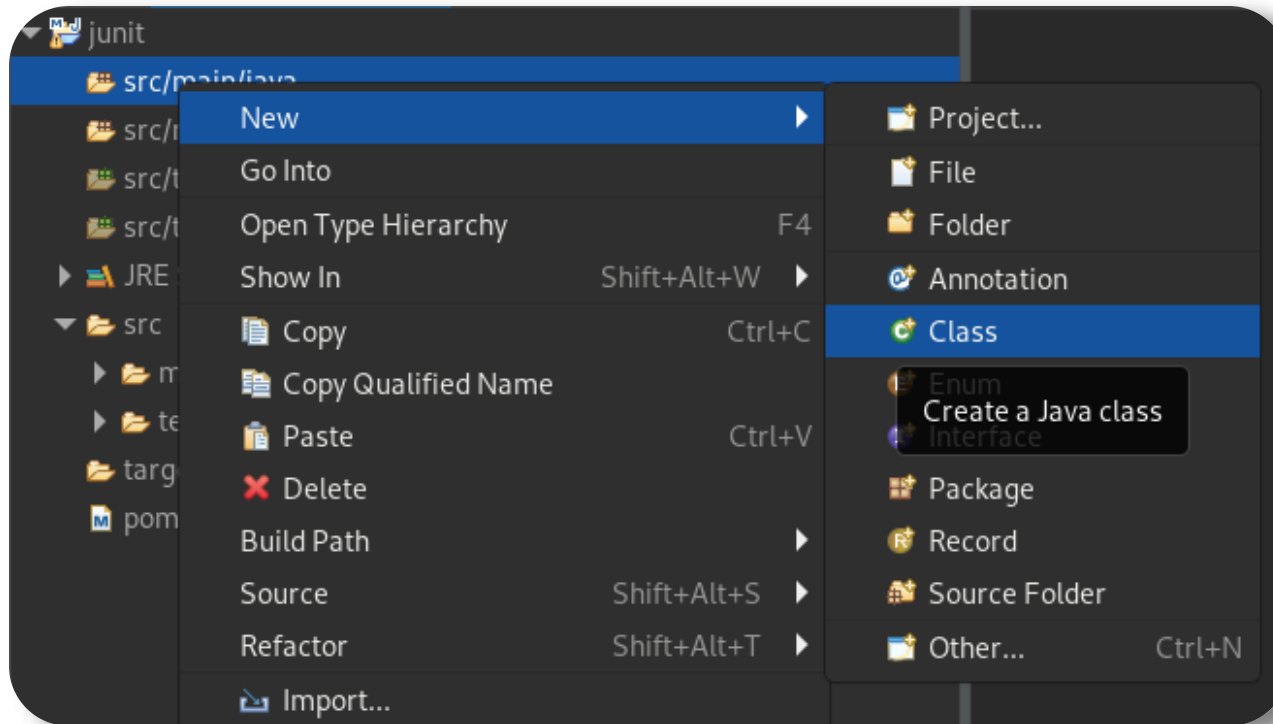
☐ Add project(s) to working set

Working set: More...

▸ Advanced

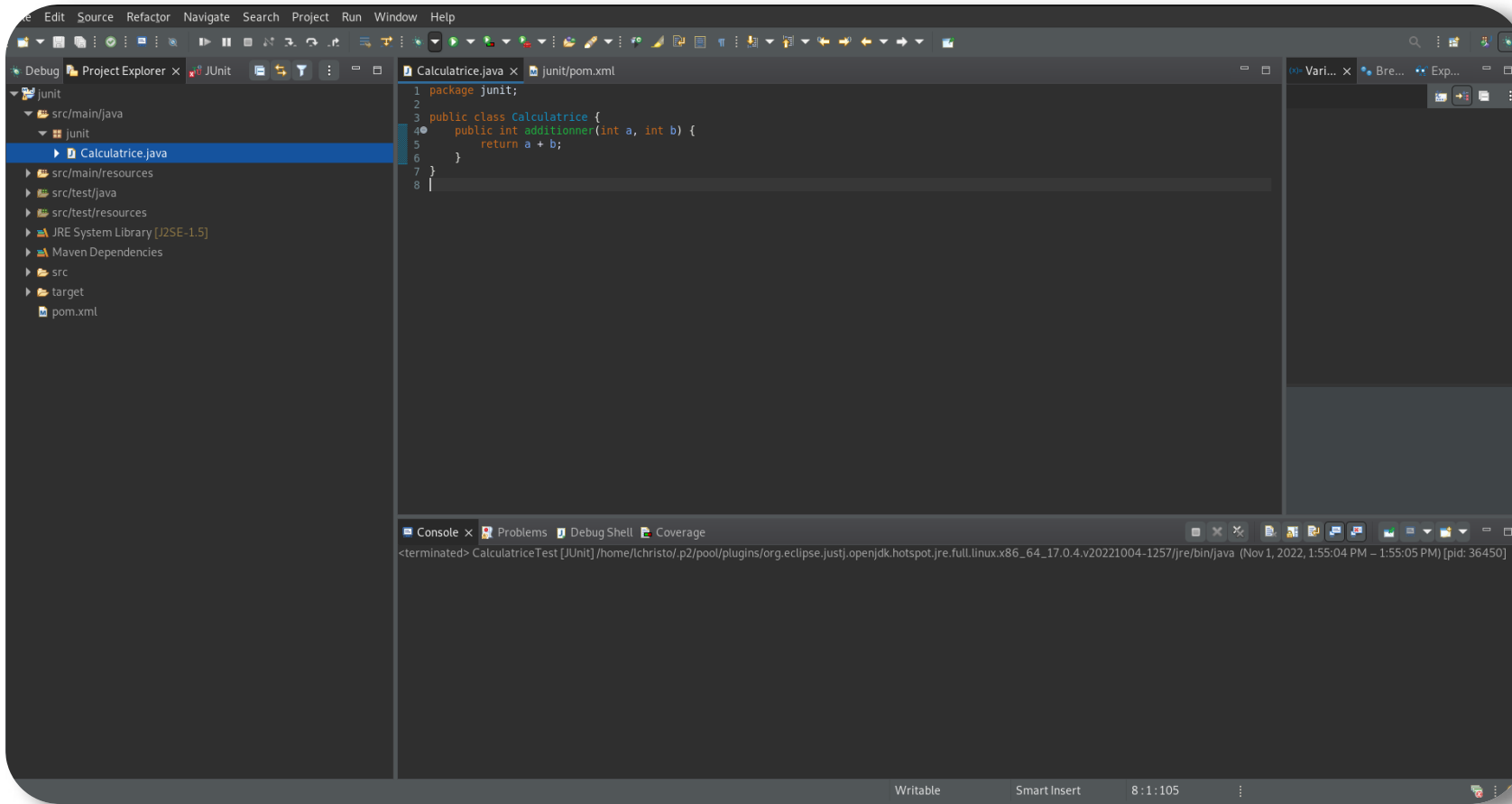
? < Back Next > Cancel Finish

Installation



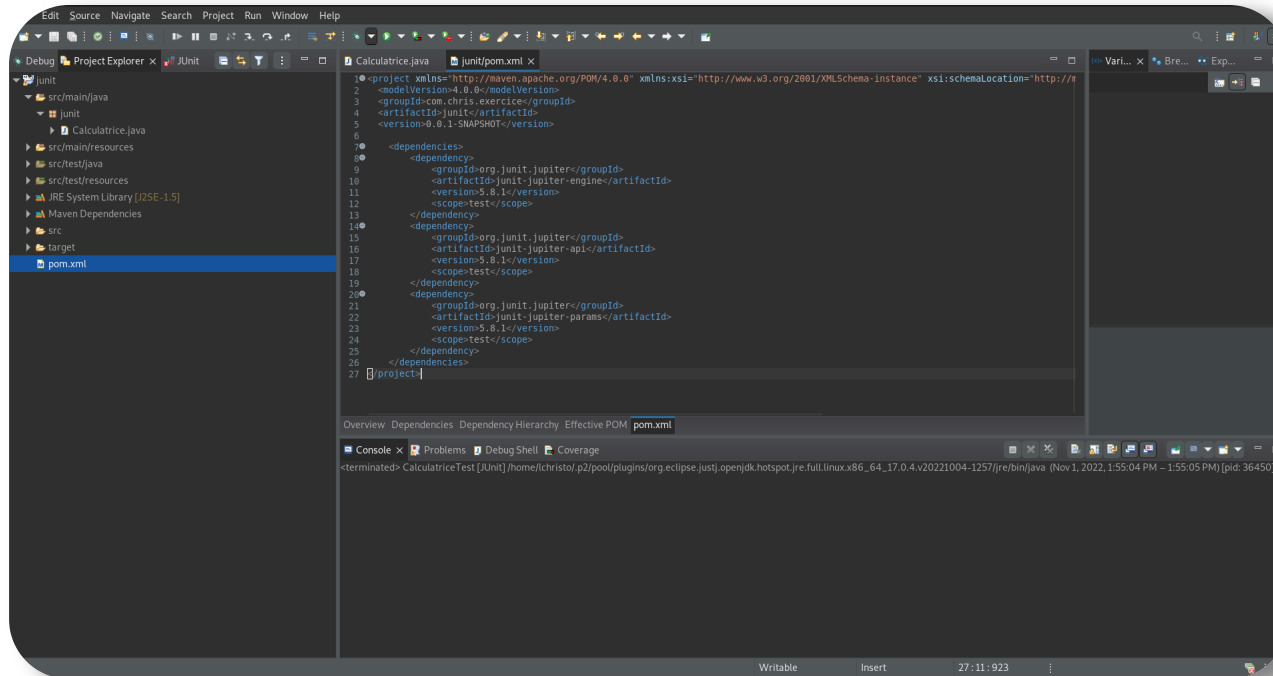
Installation

JUnit 5



Installation

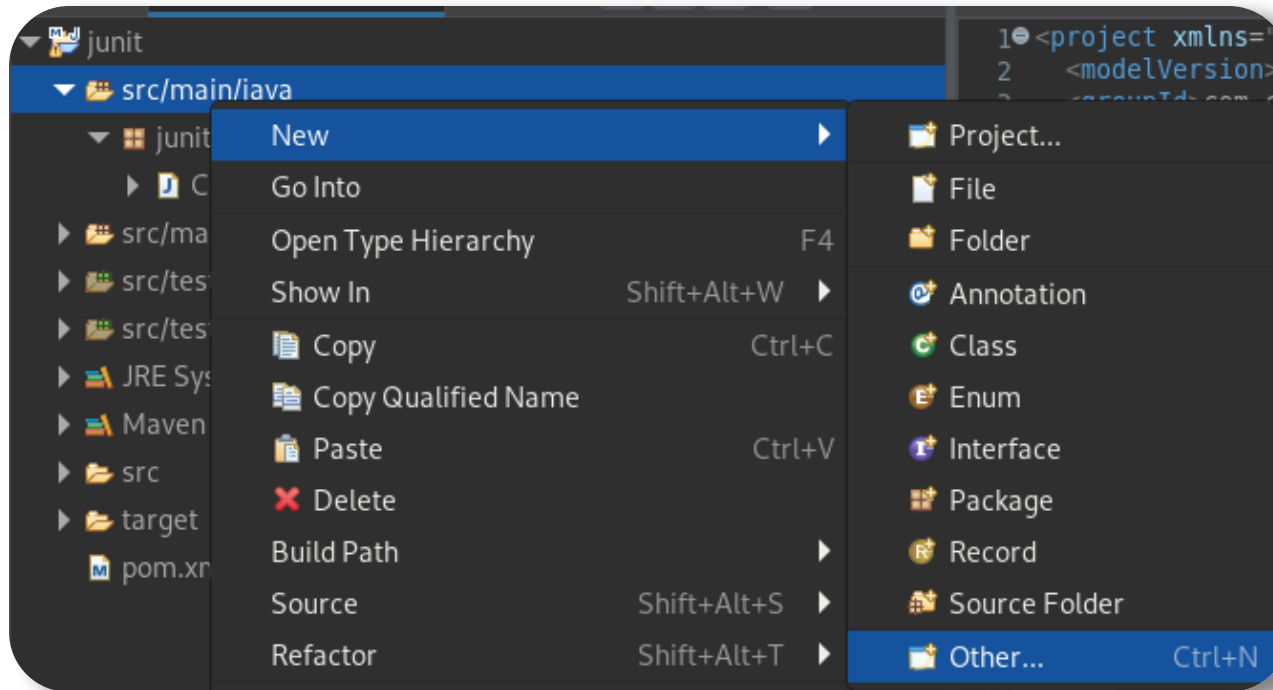
JUnit 5



Pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Installation



Installation

New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. Specify the class under test to select methods to be tested on the next page.

☐ New JUnit 3 test

☐ New JUnit 4 test

☒ New JUnit Jupiter test

Source folder:

junit/src/test/java

Browse...

Package:

junit

Browse...

Name:

CalculatriceTest

Superclass:

java.lang.Object

Browse...

Which method stubs would you like to create?

☐ @BeforeAll setUpBeforeClass()

☐ @AfterAll tearDownAfterClass()

☐ @BeforeEach setUp()

☐ @AfterEach tearDown()

☐ constructor

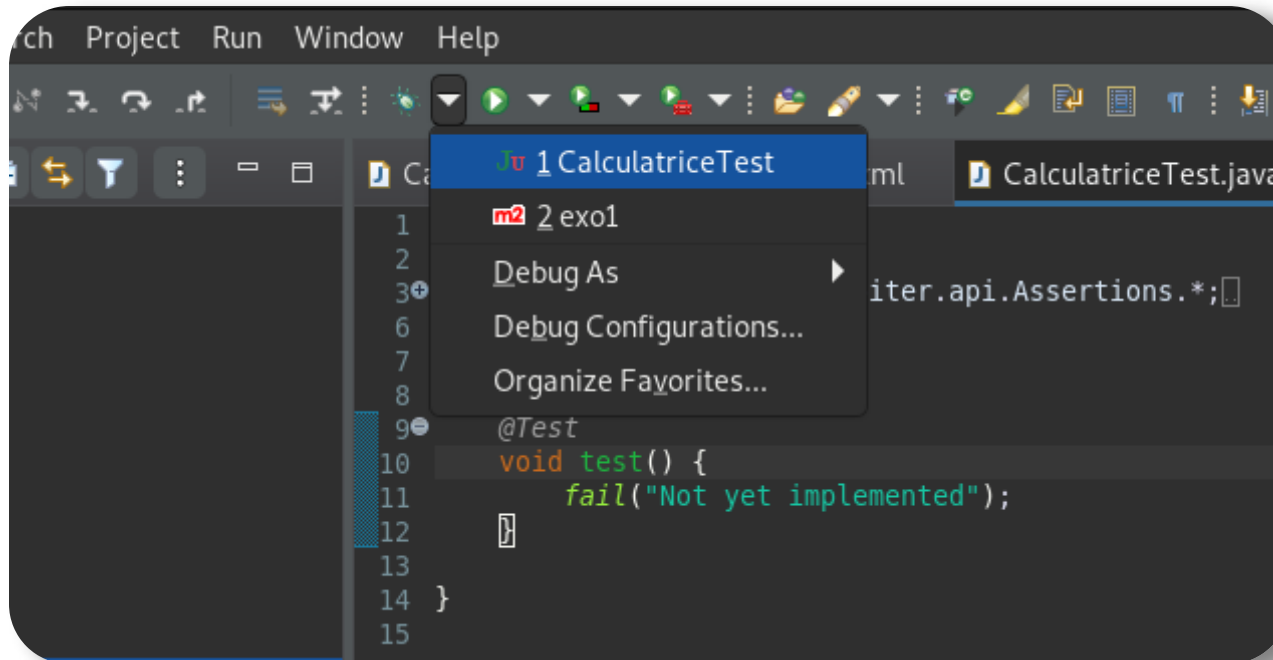
Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Class under test:

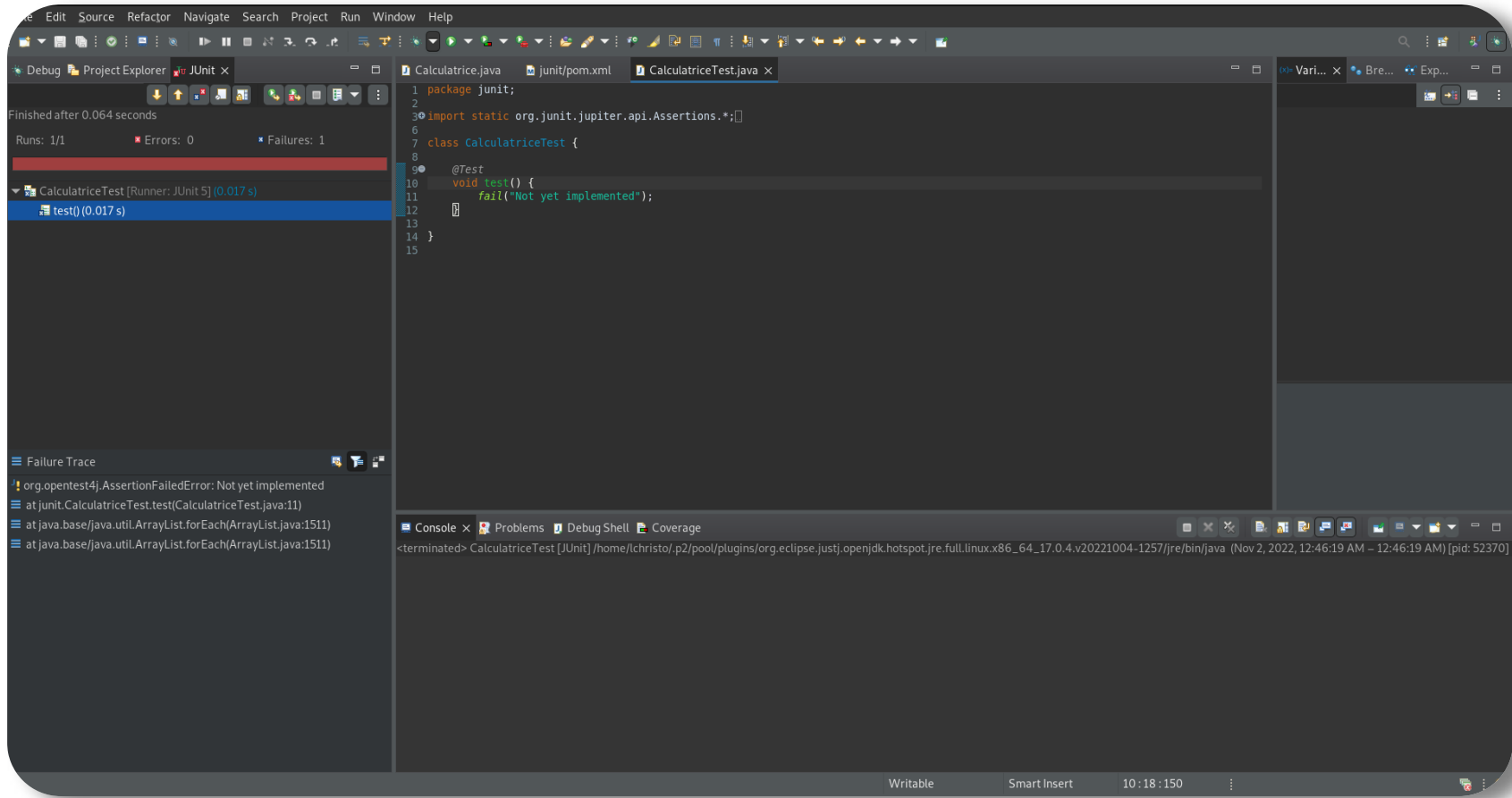
Browse...

Installation



Installation

JUnit 5



Terminologie JUnit

Test unitaire(unit test)

- ▼ **Test d'un seul cas**

Cas de test(test case)

- ▼ **Fichier de test qui contient un ou plusieurs tests**

Suite de test(test suite)

- ▼ **Fichier de test qui contient plusieurs cas de test**

Utilisation de Junit

Quoi tester ?

- ▼ Les méthodes de la classe
- ▼ Les exceptions levées par la classe
- ▼ Les conditions de la classe
- ▼ Les comportements de la classe
- ▼ L'etat de la classe
- ▼ Tous les chemins possibles du code

A éviter :

- ▼ Les méthodes getter/setter des classes
- ▼ Les méthodes triviales des classes

Les annotations de JUnit

- ▼ `@Test` : Annotation pour créer un test
- ▼ `@BeforeAll` : Annotation pour exécuter une méthode avant tous les tests
- ▼ `@AfterAll` : Annotation pour exécuter une méthode après tous les tests
- ▼ `@BeforeEach` : Annotation pour exécuter une méthode avant chaque test
- ▼ `@AfterEach` : Annotation pour exécuter une méthode après chaque test
- ▼ `@Disabled` : Annotation pour désactiver un test
- ▼ `@DisplayName` : Annotation pour donner un nom à un test

Annotation pour créer un test

```
@Test
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    int resultat = calculatrice.add(1, 2);
    System.out.println("resultat : " + resultat);
}
```

@BeforeAll

Annotation pour exécuter une méthode avant tous les tests

```
@BeforeAll
public static void beforeAll() {
    System.out.println("beforeAll");
}

@Test
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    int resultat = calculatrice.add(1, 2);
    System.out.println("resultat : " + resultat);
}
```

@AfterAll

Annotation pour exécuter une méthode après tous les tests

```
@AfterAll
public static void afterAll() {
    System.out.println("afterAll");
}

@Test
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    int resultat = calculatrice.add(1, 2);
    System.out.println("resultat : " + resultat);
}
```

@BeforeEach

Annotation pour exécuter une méthode avant chaque test

```
@BeforeEach
public void beforeEach() {
    System.out.println("beforeEach");
}

@Test
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    int resultat = calculatrice.add(1, 2);
    System.out.println("resultat : " + resultat);
}
```

@AfterEach

Annotation pour exécuter une méthode après chaque test

```
@AfterEach
public void afterEach() {
    System.out.println("afterEach");
}

@Test
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    int resultat = calculatrice.add(1, 2);
    System.out.println("resultat : " + resultat);
}
```


Annotation pour désactiver un test

```
@Disabled
@Test
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    int resultat = calculatrice.add(1, 2);
    System.out.println("resultat : " + resultat);
}
```

@DisplayName

Annotation pour donner un nom à un test

```
@DisplayName("test d'addition")
@Test
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    int resultat = calculatrice.add(1, 2);
    System.out.println("resultat : " + resultat);
}
```

TP : Ordre d'exécution des annotations

- ▼ Créer un projet maven
- ▼ Créer une classe de test
- ▼ Créer des methodes avec les annotations suivantes :
 - ▼ @BeforeAll : affichera "beforeAll 0"
 - ▼ @AfterAll : affichera "afterAll 2"
 - ▼ @Test : affichera "test 1"
 - ▼ @BeforeEach : affichera "beforeEach 3"
 - ▼ @AfterEach : affichera "afterEach 4"
- ▼ Exécuter le test :

quel est l'ordre d'exécution des annotations avec un seul test ?
Que se passe-t-il si on ajoute un deuxième test ?

Gerer l'exécution des tests

Annotation pour définir le cycle de vie des tests

- ▼ **@TestInstance**

Annotation pour donner un ordre d'exécution aux tests

- ▼ **@TestMethodOrder**

- ▼ **@Order**

@TestInstance

Annotation pour définir le cycle de vie des tests

- ▼ Lifecycle.PER_CLASS : cycle de vie par classe
- ▼ Lifecycle.PER_METHOD : cycle de vie par méthode

@TestInstance

Annotation pour donner un ordre d'exécution aux tests

```
@TestInstance(Lifecycle.PER_CLASS)
class CalculatriceTest {

    public int max = 0;

    @Test
    public void testadd() {
        System.out.println("premiere fonction max :" + max);
        Calculatrice calculatrice = new Calculatrice();
        max = calculatrice.add(1, 2);
        System.out.println("premiere fonction max :" + max);
    }

    @Test
    public void testadd2() {
        System.out.println("Seconde fonction max :" + max);
    }
}
```

```
@TestInstance(Lifecycle.PER_METHOD)
class CalculatriceTest {

    public int max = 0;

    @Test
    public void testadd() {
        System.out.println("premiere fonction max :" + max);
        Calculatrice calculatrice = new Calculatrice();
        max = calculatrice.add(1, 2);
        System.out.println("premiere fonction max :" + max);
    }

    @Test
    public void testadd2() {
        System.out.println("Seconde fonction max :" + max);
    }
}
```

@TestMethodOrder

Annotation pour donner un ordre d'exécution aux tests

- ▼ **MethodOrderer.OrderAnnotation.class** : ordre d'exécution par annotation
- ▼ **MethodOrderer.Random.class** : ordre d'exécution aléatoire
- ▼ **MethodOrderer.Alphanumeric.class** : ordre d'exécution alphanumérique

@Order et @TestMethodOrder

```
@TestInstance(Lifecycle.PER_CLASS)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class CalculatriceTest {

    public int max = 0;

    @Test
    @Order(1)
    public void testadd() {
        System.out.println("premiere fonction max :" + max);
        Calculatrice calculatrice = new Calculatrice();
        max = calculatrice.add(1, 2);
        System.out.println("premiere fonction max :" + max);
    }

    @Test
    @Order(2)
    public void testadd2() {
        System.out.println("Seconde fonction max :" + max);
    }
}
```

```
@TestInstance(Lifecycle.PER_CLASS)
@TestMethodOrder(MethodOrderer.Random.class)
class CalculatriceTest {

    @Test
    public void testadd() {
        System.out.println("premiere fonction");
    }

    @Test
    public void testadd2() {
        System.out.println("Seconde fonction");
    }
}
```

```
@TestInstance(Lifecycle.PER_CLASS)
@TestMethodOrder(MethodOrderer.Alphanumeric.class)
class CalculatriceTest {
    @Test
    public void testadd1() {
        System.out.println("premiere fonction");
    }

    @Test
    public void testadd2() {
        System.out.println("Seconde fonction");
    }
}
```

!!! Cette methode a été dépréciée depuis JUnit 5 !!!

TP : Gérer l'exécution des tests

Dans un nouveau projet maven, créer une classe de test pour la classe Calculatrice.

Créer les methodes avec les annotations suivantes :

fonction	affichage	ordre d'exécution
-----	-----	-----
beforeAll	"methode beforeAll"	avant tout
beforeEach	"methode beforeEach"	avant chaque test
afterEach	"methode afterEach"	après chaque test
afterAll	"methode afterAll"	après tous
methode1	"methode1"	2
methode2	"methode2"	3
methode3	"methode3"	1

```
@TestInstance(Lifecycle.PER_CLASS)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class CalculatriceTest {
    public int max = 0;

    @BeforeAll
    public void testbefore() {
        System.out.println("before all");
    }

    @Test
    @Order(2)
    public void testadd() {
        System.out.println("premiere fonction");
    }

    @Test
    @Order(1)
    public void testadd2() {
        System.out.println("Seconde fonction");
    }
}
```

```
@TestInstance(Lifecycle.PER_CLASS)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class CalculatriceTest {
    public int max = 0;

    @Test
    @BeforeAll
    public void testbefore() {
        System.out.println("before all");
    }

    @Test
    @Order(2)
    public void testadd() {
        System.out.println("premiere fonction");
    }

    @Test
    @Order(1)
    public void testadd2() {
        System.out.println("Seconde fonction");
    }
}
```

Qualité d'un test

Un bon test est :

- ▼ Simple : il doit être simple à comprendre et à écrire
- ▼ Précis : il doit tester une seule chose
- ▼ Rapide : il doit être rapide à exécuter
- ▼ Indépendant : il ne doit pas dépendre d'un autre test
- ▼ Reproductible : il doit pouvoir être exécuté plusieurs fois sans échec

Pourquoi tester ?

- ▼ **Pour vous :**
 - ▼ Assurer que votre code fonctionne
 - ▼ Sécuriser votre code
- ▼ **Pour le client :**
 - ▼ Assurer la robustesse de votre code
 - ▼ Valider chaque demande du client

Le principe d'assertion

Une assertion est une vérification de la valeur d'une expression.
Elle renvoie une exception si l'expression est fausse.

```
@Test
public void testadd() {
    assertEquals(3, 3); // OK
    assertEquals(3, 4); // KO
}
```

Les assertions JUnit

- ▼ **AssertEquals** : tester l'égalité
- ▼ **AssertNotEquals** : tester la différence
- ▼ **AssertTrue** : tester la valeur True
- ▼ **AssertFalse** : tester la valeur False
- ▼ **AssertNull** : tester si une valeur est nulle
- ▼ **AssertNotNull** : tester si une valeur n'est pas nulle
- ▼ **AssertSame** : tester l'identité
- ▼ **AssertNotSame** : tester la différence d'identité

Exemple assert Equals

```
public class CalculatriceTest {  
  
    @Test  
    public void testadd() {  
        Calculatrice calculatrice = new Calculatrice();  
        int resultat = calculatrice.add(1, 2);  
        assertEquals(3, resultat);  
    }  
}
```

AssertEquals

Le client demande une fonction qui permet d'additionner deux nombres.

Quels tests pouvez-vous écrire ?

TP : Ecrire sa propre méthode add

Le client demande une fonction qui permet d'additionner deux nombres.

Si le résultat est supérieur à `Integer.MAX_VALUE`, on retourne `Integer.MAX_VALUE`.

Si le résultat est inférieur à `Integer.MIN_VALUE`, on retourne `Integer.MIN_VALUE`.

Si le résultat est égal à zéro, on retournera 1.

Ecrire les tests unitaires correspondant.

Puis écrire la méthode `add` qui validera les tests.

Exemple assert Not Equals

```
public class CalculatriceTest {  
  
    @Test  
    public void testadd() {  
        Calculatrice calculatrice = new Calculatrice();  
        int resultat = calculatrice.add(1, 2);  
        assertEquals(0, resultat);  
    }  
  
}
```

TP : Ecrire sa propre méthode add 2

Le client insatisfait de la méthode add précédente et demande une modification.

Si le résultat est égal à zéro, on retournera un nombre aléatoire entre 1 et 100.

Ecrire les tests unitaires correspondant.

Puis écrire la méthode add qui validera les tests.

Exemple assert False et assert Null

```
public class CalculatriceTest {  
  
    @Test  
    @DisplayName("Test assert False")  
    public void testadd() {  
        boolean resultat = false;  
        assertFalse(resultat);  
    }  
  
    @Test  
    @DisplayName("Test assert True")  
    public void testadd() {  
        boolean resultat = false;  
        assertTrue(resultat);  
    }  
  
}
```


TP : assert False et assert True

Le client demande maintenant une fonction qui permet de vérifier si un nombre est pair.

Ecrire les tests unitaires correspondant.

Puis écrire la méthode isPair qui validera les tests.

Exemple assert Null et assert Not Null

```
public class CalculatriceTest {  
  
    public Calculatrice calculatrice = null;  
  
    @Test  
    @DisplayName("Test assert Null")  
    public void testadd() {  
        assertNull(calculatrice);  
    }  
  
    @Test  
    @DisplayName("Test assert Not Null")  
    public void testadd() {  
        calculatrice = new Calculatrice();  
        assertNotNull(calculatrice);  
    }  
}
```

TP : assert Null et assert Not Null

Afin de garder une trace des calculs effectués, le client demande de stocker les résultats dans une liste de integer.

Notre calculatrice devra donc avoir une liste d'integer en attribut, initialisée à null.

Chaque methode add, sub, mult devra ajouter le résultat dans la liste.

Vous ecrirez les methodes addlist(int), et getlist(int) qui permettront l'utilisation de la liste.

Ecrire les tests unitaires correspondant.

Faites évoluer la classe Calculatrice pour valider les tests.

Exemple assert Same et assert Not Same

```
@Test
@DisplayName("Test assert Same")
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    assertEquals(calculatrice, calculatrice);
}

@Test
@DisplayName("Test assert Not Same")
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    Calculatrice calculatrice2 = new Calculatrice();
    assertEquals(calculatrice, calculatrice2);
}
```

TP : assert same et assert not same

Le client demande maintenant une fonction qui permet de créer une nouvelle calculatrice.

Cette calculatrice devra être une copie de la calculatrice courante.

- ▼ Les deux calculatrices devront avoir la même liste de résultat
- ▼ Elle ne sont pas égaux
- ▼ Elles ne sont pas la même instance
- ▼ Leurs listes, elles seront le même objet

Ecrire les tests unitaires correspondant.

Puis écrire la méthode copy qui validera les tests.

TP : suite

La methode copy devra maintenant créer la meme instance de calculatrice.

Elles devront donc être egales et la meme instance.

Elles devront donc avoir la meme liste de résultat.

Ecrire les tests unitaires correspondant.

Puis changer la méthode copy pour valider les tests.

La couverture de code

**La couverture de code est un indicateur de la qualité du code.
Elle se définit :**

- ▼ **Par le nombre de lignes de code testées**
- ▼ **Par le nombre de branches testées**

Une bonne couverture de code est comprise entre 80% et 90%.

Les annotations d'executions

Les différentes annotations d'exécution :

- ▼ **@RepeatedTest** : répète le test
- ▼ **@ParameterizedTest** : exécute le test avec des paramètres
- ▼ **@Timeout** : limite le temps d'exécution du test
- ▼ **@Tag** : permet de taguer les tests

Les tests répétés

L'annotation `@RepeatedTest` permet de répéter le test

La methode répétée :

- ▼ Ne pas être private
- ▼ Ne pas être static
- ▼ Doit obligatoirement retourner void

```
@RepeatedTest(5)
@Test
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    int resultat = calculatrice.add(1, 2);
    assertEquals(3, resultat);
}
```

Les tests paramétrés

L'annotation `@ParameterizedTest` permet de paramétrer le test

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
public void testadd(int a) {
    Calculatrice calculatrice = new Calculatrice();
    int resultat = calculatrice.add(a, 2);
    assertEquals(a + 2, resultat);
}
```

TP : Les tests paramétrés

Le client demande maintenant une fonction qui permet de multiplier un nombre.

Comme précédemment, le résultat doit être compris entre Integer.MIN_VALUE et Integer.MAX_VALUE.

Et le résultat devra être ajouté à la liste de int.

Ecrire les tests unitaires correspondant en utilisant les test paramétrés.

Puis écrire la méthode mult qui validera les tests.

Vous ajouterez également un test paramétré pour chacune des méthodes précédentes.

Les tests paramétrés

Les différents types de paramètres :

- ▼ **@ValueSource** : permet de passer une liste de valeurs
- ▼ **@EnumSource** : permet de passer une liste d'énumérations
- ▼ **@CsvSource** : permet de passer une liste de valeurs séparées par des virgules
- ▼ **@CsvFileSource** : permet de passer une liste de valeurs séparées par des virgules dans un fichier

Les tests conditionnels

L'annotation `@EnabledOnOs` permet d'exécuter le test sur un système d'exploitation donné

- ▼ `@EnabledOnOs(OS.WINDOWS)`
- ▼ `@EnabledOnOs(OS.LINUX)`
- ▼ `@EnabledOnOs(OS.MAC)`
- ▼ `@EnabledOnOs(OS.SOLARIS)`

```
@EnabledOnOs(OS.WINDOWS)
@Test
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    int resultat = calculatrice.add(1, 2);
    assertEquals(3, resultat);
}
```

Les tests d'exceptions

Plusieurs méthodes permettent de tester les exceptions :

- ▼ **AssertThrows** : permet de tester une exception
- ▼ **AssertDoesNotThrow** : permet de tester qu'aucune exception n'est levée

```
@Test
public void testadd() {
    assertThrows(IllegalArgumentException.class, () -> {
        Calculatrice calculatrice = new Calculatrice();
        calculatrice.add(1, 2);
    });
    assertDoesNotThrow(() -> {
        Calculatrice calculatrice = new Calculatrice();
        calculatrice.add(1, 2);
    });
}
```

TP : Les tests d'exceptions

Le client demande maintenant une fonction qui permet de diviser un nombre.

Comme précédemment, le résultat doit être compris entre Double.MIN_VALUE et Double.MAX_VALUE.

Le résultat ne sera pas ajouté à la liste de int.

Ecrire les tests unitaires correspondant en utilisant les test d'exceptions.

Puis écrire la méthode div qui validera les tests.

Les tests de timeout

L'annotation `@Timeout` permet de limiter le temps d'exécution du test

```
@Timeout(1) // le test doit s'exécuter en moins d'une seconde
@Test
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    int resultat = calculatrice.add(1, 2);
    assertEquals(3, resultat);
}

@Timeout(value = 1, unit = TimeUnit.MILLISECONDS) // le test doit s'exécuter en moins d'une milliseconde
@Test
public void testadd() {
    Calculatrice calculatrice = new Calculatrice();
    int resultat = calculatrice.add(1, 2);
    assertEquals(3, resultat);
}
```


TP : Les tests de timeout

Le client demande maintenant une fonction qui permet de trouver si un nombre est premier.

Ecrire les tests unitaires correspondant en utilisant les test de timeout.

Puis écrire la méthode isPrime qui validera les tests.

Les tests avec des méthodes de test dynamiques

L'annotation `@TestFactory` permet d'exécuter des tests dynamiques

Les test dynamiques doivent obligatoirement :

- ▼ Retourner une collection de `DynamicTest`
- ▼ Etre en dehors du cycle de vie des tests
- ▼ Ne pas etre une statique

L'objet `DynamicTest`

L'objet `DynamicTest` permet de créer un test dynamique

Il prend en paramètre :

- ▼ Un nom de test
- ▼ Une fonction lambda qui retourne un objet `Executable`

Ses avantages :

- ▼ Permet de créer des tests paramétrés
- ▼ Permet de créer des tests conditionnels

```
@TestFactory
public Collection<DynamicTest> testaddfactory() {
    Calculatrice calculatrice = new Calculatrice();
    return Arrays.asList(
        DynamicTest.dynamicTest("1 + 2 = 3", () -> assertEquals(3, calculatrice.add(1, 2))),
        DynamicTest.dynamicTest("2 + 2 = 4", () -> assertEquals(4, calculatrice.add(2, 2))),
        DynamicTest.dynamicTest("3 + 2 = 5", () -> assertEquals(5, calculatrice.add(3, 2)));
}
```

Les bonnes pratiques

- ▼ Ne pas tester les getters et les setters
- ▼ Ne pas tester les méthodes privées
- ▼ Ne pas tester les méthodes statiques
- ▼ Ne pas tester les méthodes héritées
- ▼ Ne pas tester les méthodes de configuration

jmock est un framework de mock qui permet de créer des objets de test

```
<dependency>
  <groupId>org.jmock</groupId>
  <artifactId>jmock</artifactId>
  <version>2.12.0</version>
  <scope>test</scope>
</dependency>
```

Les mocks

Un mock est un objet qui simule le comportement d'un objet réel.

Le mock presente la même interface que l'objet réel.

exemple :

```
Public class Pilote {  
    private Voiture voiture;  
    public Pilote(Voiture voiture) {  
        this.voiture = voiture;  
    }  
    public void rouler() {  
        voiture.avancer();  
    }  
}
```

System in Production



- Component Under Test
- Depended on Components
- Additional Components

System in Unit Test



- Component Under Test
- Mocks for Components


```
public class PiloteTest {  
    @Test  
    public void testrouler() {  
        Voiture voiture = mock(Voiture.class);  
        Pilote pilote = new Pilote(voiture);  
        pilote.rouler();  
        verify(voiture).avancer();  
    }  
}
```

Test unitaires avancés

- ▼ Mockito : permet de mocker les dépendances
- ▼ PowerMock : permet de mocker les méthodes statiques
- ▼ MockMVC : permet de tester les contrôleurs
- ▼ TestContainers : permet de tester les applications avec des bases de données

TP : recapitulatif

**Faire une classe societe qui contient une liste de personne.
elle aura :**

Une méthode qui permet d'ajouter une personne

Une méthode qui permet de supprimer une personne

Une méthode qui permet de trouver une personne par son nom

Ecrire les tests unitaires correspondant.

Puis écrire les méthodes qui valideront les tests.

Conclusion

Ecrire des tests unitaires est un travail fastidieux mais nécessaire.

Il permet :

- ▼ **Vérifier le bon fonctionnement de l'application**
- ▼ **Eviter les régressions**
- ▼ **Documenter le code**

Qualité d'un test unitaire :

- ▼ **Simple : le test doit être simple à comprendre**
- ▼ **Reproductible : le test doit être reproductible**
- ▼ **Indépendant : le test doit être indépendant des autres tests**