TDD (Test Driven Development)

Qu'est-ce que le TDD ?

Le TDD est une technique de développement logiciel qui consiste à écrire des tests unitaires avant d'écrire le code source.

Le TDD est une technique de développement logiciel qui permet de développer des applications de qualité.

Il utilise les tests unitaires pour s'assurer que le code est de qualité.

Principe fondamental

Les etapes du TDD sont les suivantes :

- Ecrire un test unitaire
- Faire échouer le test
- Ecrire le code pour faire passer le test
- Refactoriser le code

Principe fondamental

Nous utilisons le TDD pour :

- Définir les exigences du projet
- Définir les cas d'utilisation
- Assurer la qualité du code
- Assurer une bonne maintenance du code

Le TDD est un processus itératif, qui permet de développer des applications de qualité.

Avantages et bénéfices

Les avantages du TDD sont les suivants :

- Développement de code de qualité
- Développement de code maintenable
- Développement de code testé
- Développement de code évolutif
- Développement de code fiable

Comment implémenter le TDD dans un projet Java

Pour implémenter le TDD dans un projet Java, il faut utiliser une librairie de tests unitaires.

Dans ce cours, nous allons utiliser JUnit.

Les librairies de tests unitaires en Java

Les librairies de tests unitaires en Java sont les suivantes :

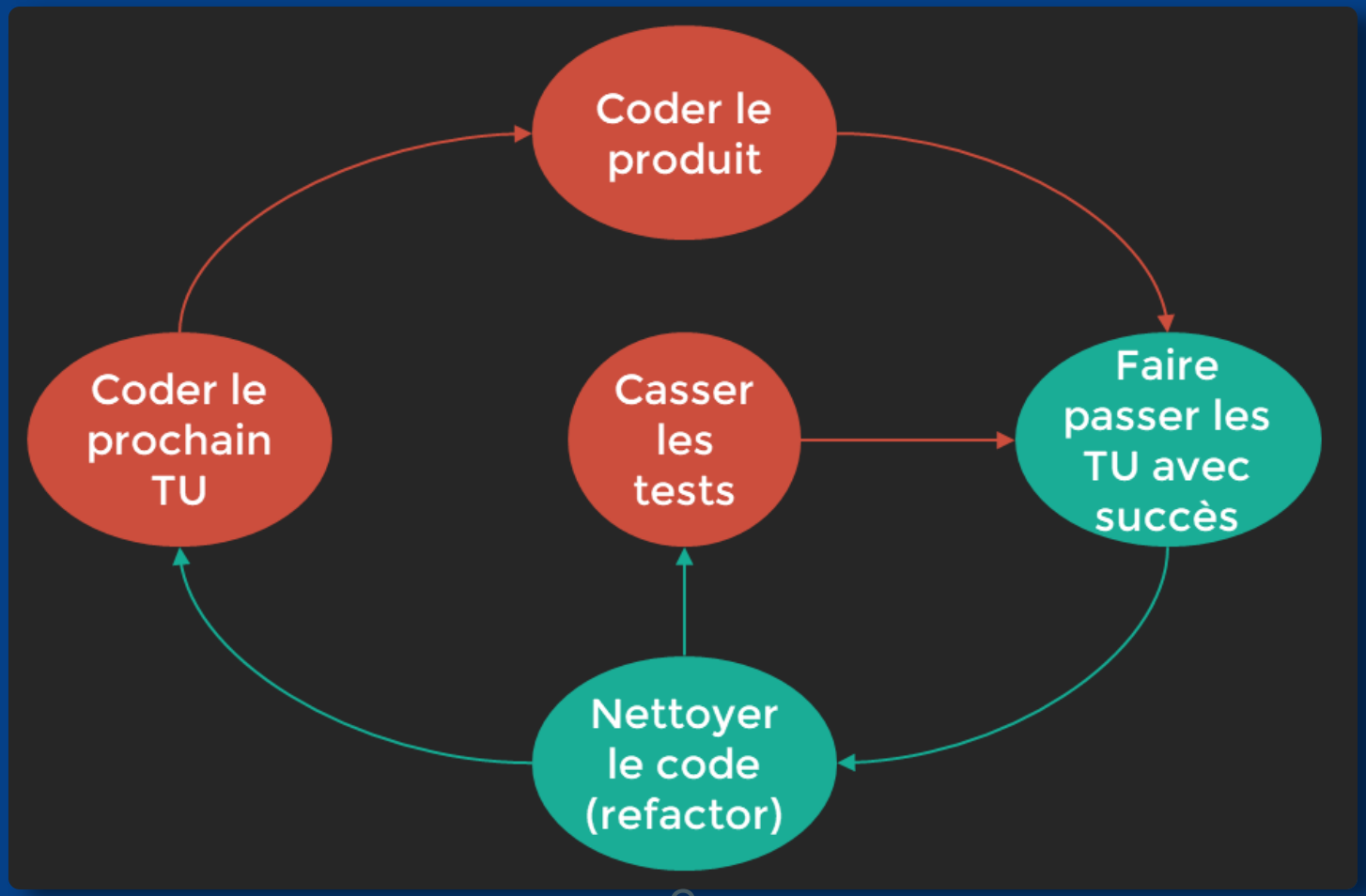
- JUnit
- TestNG
- Mockito
- AssertJ

Processus TDD

Le processus de TDD est le suivant :

- Écrire un test unitaire
- Faire échouer le test
- Écrire le code pour faire passer le test
- Refactoriser le code

Cycle TDD (Red, Green, Refactor)



Les bonnes pratiques en TDD

Les bonnes pratiques en TDD sont les suivantes :

- Structurer les tests pour une meilleure lisibilité
- Organisation des tests par catégories
- Nommage des tests pour une meilleure compréhension
- Documenter les tests pour une meilleure maintenance
- Éviter les tests redondants
- Éviter les tests inutiles

Écriture de tests pertinents

Les tests doivent être pertinents et répondre aux exigences du projet.

Chaque projet a ses propres exigences, et les tests doivent être adaptés en fonction de ces exigences.

Ainsi, les tests doivent être écrits en fonction des attentes du client.

Couverture complète des cas d'utilisation

Le principe de couverture est de tester tous les chemins possibles dans le code, afin de s'assurer que le code fonctionne correctement.

Le TDD permet de couvrir complètement les cas d'utilisation du projet.

Cela permet de s'assurer que le projet est bien robuste et qu'il répond aux exigences du client.

Utilisation de l'assurance qualité

L'assurance qualité est une technique de développement logiciel qui consiste à vérifier la qualité du code.

L'assurance qualité est une technique utilisée dans le TDD, afin de s'assurer que le code est de qualité.

Présentation de l'assurance qualité

L'assurance qualité consiste à vérifier la qualité du code en utilisant des outils d'analyse statique.

Les outils d'analyse statique sont les suivants :

- SonarQube
- Checkstyle
- PMD

JUnit

Présentation de JUnit

JUnit est une librairie de tests unitaires en Java.

Son utilisation permet de tester le code source d'une application Java.

Cette librairie est très utilisée dans le TDD, elle met à disposition des annotations et des assertions pour écrire des tests unitaires.

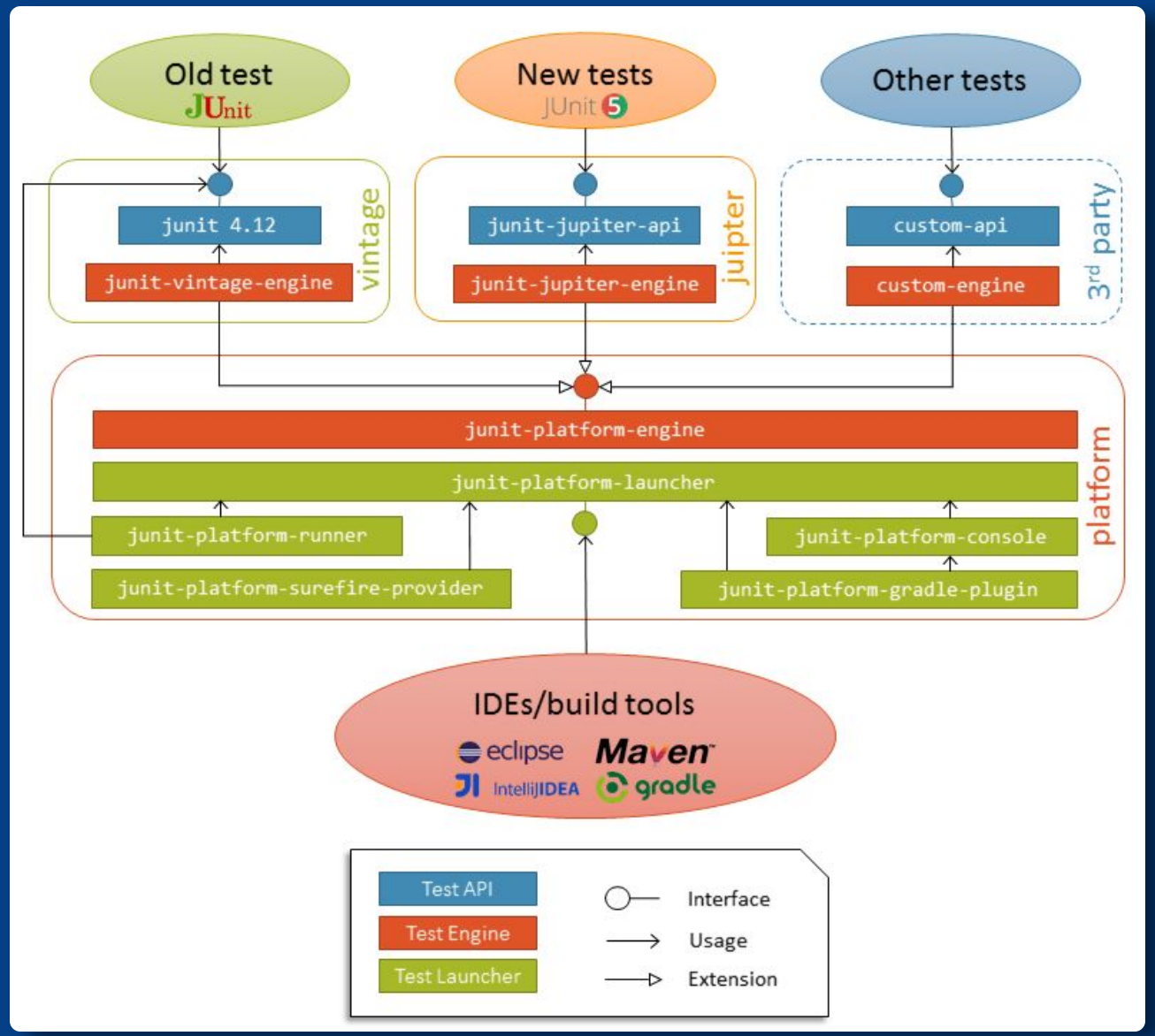
Qu'est-ce que JUnit ?

JUnit permet d'écrire des tests unitaires en Java.

Les tests unitaires sont des tests qui vérifient le bon fonctionnement des méthodes d'une classe.

Chaques méthodes d'une classe doivent être testées.

Nous appelons ces méthodes des unités de code.


Old test
JUnit
New tests
JUnit 5
Other tests
junit 4.12
junit-vintage-engine
vintage
junit-jupiter-api
junit-jupiter-engine
juipter
custom-api
custom-engine
3rd party
junit-platform-engine
junit-platform-launcher
junit-platform-runner
junit-platform-surefire-provider
junit-platform-console
junit-platform-gradle-plugin
platform
IDEs/build tools
eclipse
Maven
IntelliJIDEA
gradle
Test API
Test Engine
Test Launcher
Interface
Usage
Extension

Utilisation de JUnit

Pour utiliser JUnit, il faut ajouter la dépendance JUnit au projet Maven.

Dans le fichier pom.xml, il faut ajouter la dépendance suivante :

```
<dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.1</version>
        <scope>test</scope>
</dependency>
```

Les annotations JUnit

JUnit met à disposition des annotations pour écrire des tests unitaires.

Les annotations JUnit sont les suivantes :

- `@Test`
- `@DisplayName`
- `@Ignore`

`@Test`

L'annotation `@Test` permet d'indiquer que la méthode est un test.

Lorsque une méthode est considérée comme un test, elle doit respecter les règles suivantes :

- La méthode doit être publique
- La méthode ne doit pas retourner de valeur
- La méthode ne doit pas prendre de paramètres

Exemple de test

Voici un exemple de test :

```
public class CalculatorTest {

        `@Test`
        public void testAdd() {
                Calculator calculator = new Calculator();
                int result = calculator.add(1, 2);
                assertEquals(3, result);
        }
}
```

`@DisplayName`

L'annotation `@DisplayName` permet de donner un nom à un test.

Cette annotation prend en paramètre une chaîne de caractères qui représente le nom du test.

Cette annotation est optionnelle.

Elle changera le nom du test dans la console.

Les assertions JUnit

Quand on écrit un test, on doit vérifier que le résultat du test est le résultat attendu.

Pour vérifier que le résultat du test est le résultat attendu, on utilise des assertions.

Une assertions est une instruction qui permet de vérifier une condition.

Les assertions JUnit

JUnit met à disposition des assertions pour réaliser des assertions dans les tests.

Les assertions JUnit sont les suivantes :

- AssertEquals
- AssertTrue et assertFalse
- AssertArrayEquals
- AssertThrows

AssertEquals

L'assertion assertEquals permet de tester l'égalité entre deux objets.

Cette assertion prend deux paramètres :

- Le premier paramètre est l'objet attendu
- Le deuxième paramètre est l'objet testé

Exemple d'utilisation

Voici un exemple d'utilisation de l'assertion assertEquals :

```
public class CalculatorTest {

        private Calculator calculator;

        @Test
        public void testAdd() {
                int result = calculator.add(1, 2);
                assertEquals(3, result);
        }
}
```

Exemple avec des objets

```java
public class CalculatorTest {

        private Calculator calculator;

        @Test
        public void testAdd() {
                int result = calculator.add(1, 2);
                assertEquals(new Integer(3), new Integer(result));
        }
}
```

AssertTrue et assertFalse

L'assertion assertTrue permet de tester si un objet est vrai.

L'assertion assertFalse permet de tester si un objet est faux.

Ces assertions prennent un seul paramètre :

- Le paramètre est l'objet testé

Exemple d'utilisation

Voici un exemple d'utilisation de l'assertion assertTrue :

```
public class CalculatorTest {

        private Calculator calculator;

        @Test
        public void testIsEven() {
                boolean result = calculator.isEven(2);
                assertTrue(result);
        }
}
```