



Mise en pratique des tests avec JUnit

David THIBAU - 2021
david.thibau@gmail.com



Agenda

- Introduction

- TDD, Typologie des tests
- Les tests dans une pratique DevOps
- Mise en place, recommandations
- Les outils du monde Java

- Tests d'intégration

- Introduction, Particularités
- Couche HTTP
- Couche de persistance
- Spring et les tests d'intégration

- Tests unitaires avec JUnit

- Concepts JUnit
- *Matcher*
- Isolation avec Mockito
- Tests paramétrés, modèle d'extension

- Automatisation

- Les tests avec Maven ou Gradle
- Les tests dans une pipeline CI/CD



Introduction

TDD et typologie des tests

Les tests dans la pratique DevOps

Mise en place, recommandations

Les outils Java



Introduction

Les test sont devenus primordiaux dans les méthodologies de développement informatique et dans l'ingénierie logicielle en général

La méthodologie pionnière :

XP Programming

- Mais présents dans *RUP*, *Scrum*, *DevOps*



eXtreme Programming

Apparu vers 1998

- Centré sur les tests : ***Test Driven Development***
- Cycles de développement plus courts/nombreux et dévpt. agile
- Intègre pleinement le développeur dans un projet en constatant:
 - qu'il a des limitations (40-42h / semaine).
 - qu'il a des aspirations (pas faire du travail répétitif, vision sur le projet)...
 - qu'il a des défauts (il n'aime pas tester/documenter).



Kent Beck



eXtreme Programming

Philosophie de XP

Communication

- Avec le client et les autres développeurs

Simplicité plutôt que performance

- Design simple et propre coûte moins cher

Retour sur le travail effectué

- Tests dès le premier jour de développement.
- Livraison au + tôt au client et acceptation de changement.

Courage

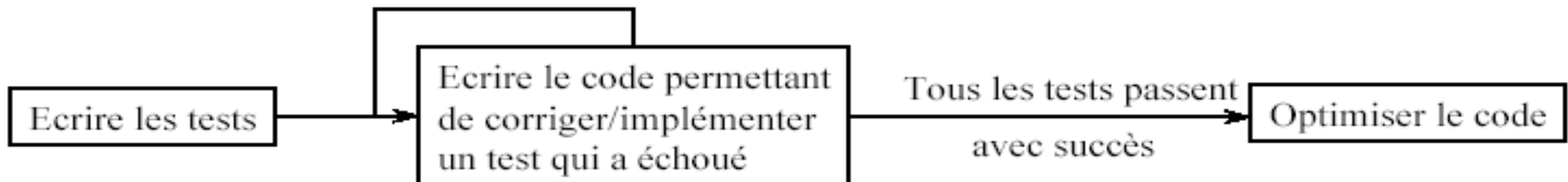
- Remise en cause du travail effectué



Test Driven Development

Le développement est guidé par le test :

- Le développeur écrit le résultat attendu de la fonction à tester au sein de la classe de test
- Il écrit la classe fonctionnelle en provoquant une erreur pour vérifier que la classe de test détecte les erreurs
- Il finalise l'algorithme de la classe à tester
- Il vérifie que le test ne détecte plus d'erreur.
- Si nécessaire, il ajoute les tests aux cas limites.

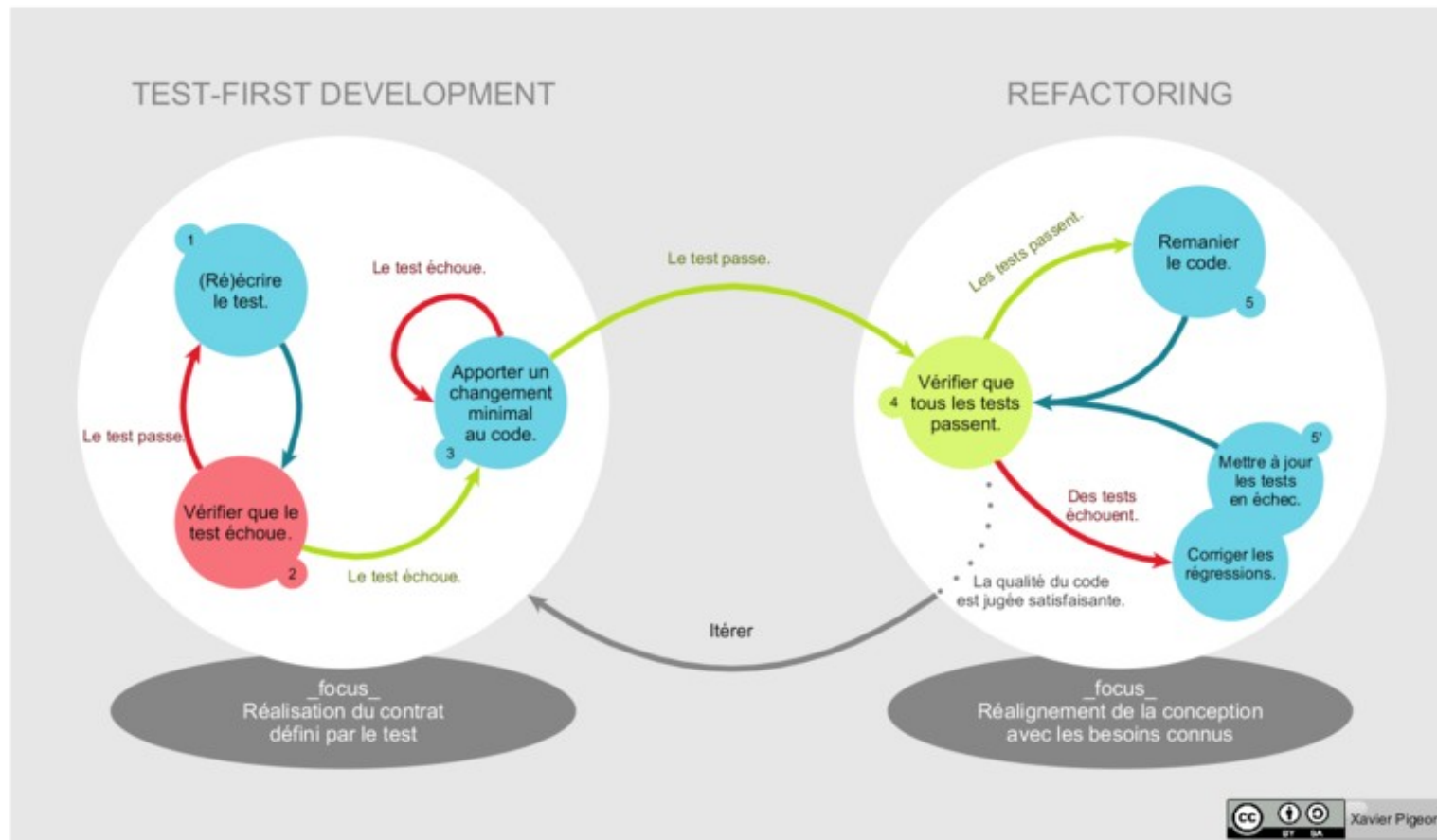




TDD : Apports

- ✓ Tester et développer deviennent une seule et même activité
- ✓ Écrire le test avant le code permet aux dév. de bien comprendre ce qu'il doit implémenter et d'affiner la spécification.
- ✓ Produit du code plus robuste
- ✓ Permet d'augmenter la confiance dans l'implémentation
- ✓ Facilite le refactoring
- ✓ L'automatisation permet d'avoir un retour en continu de l'état des tests

Cycle TDD et refactoring





Acteurs du test

La TDD concerne principalement les développeurs.

Mais certains tests impliquent d'autres acteurs :

- Expert métier, *Product Owner*
- Intégrateurs, Testeurs spécialisés
- Utilisateurs finaux

=> La spécification, l'exécution, l'automatisation des tests doivent être alors être compris par des profils non techniques



Activité du test

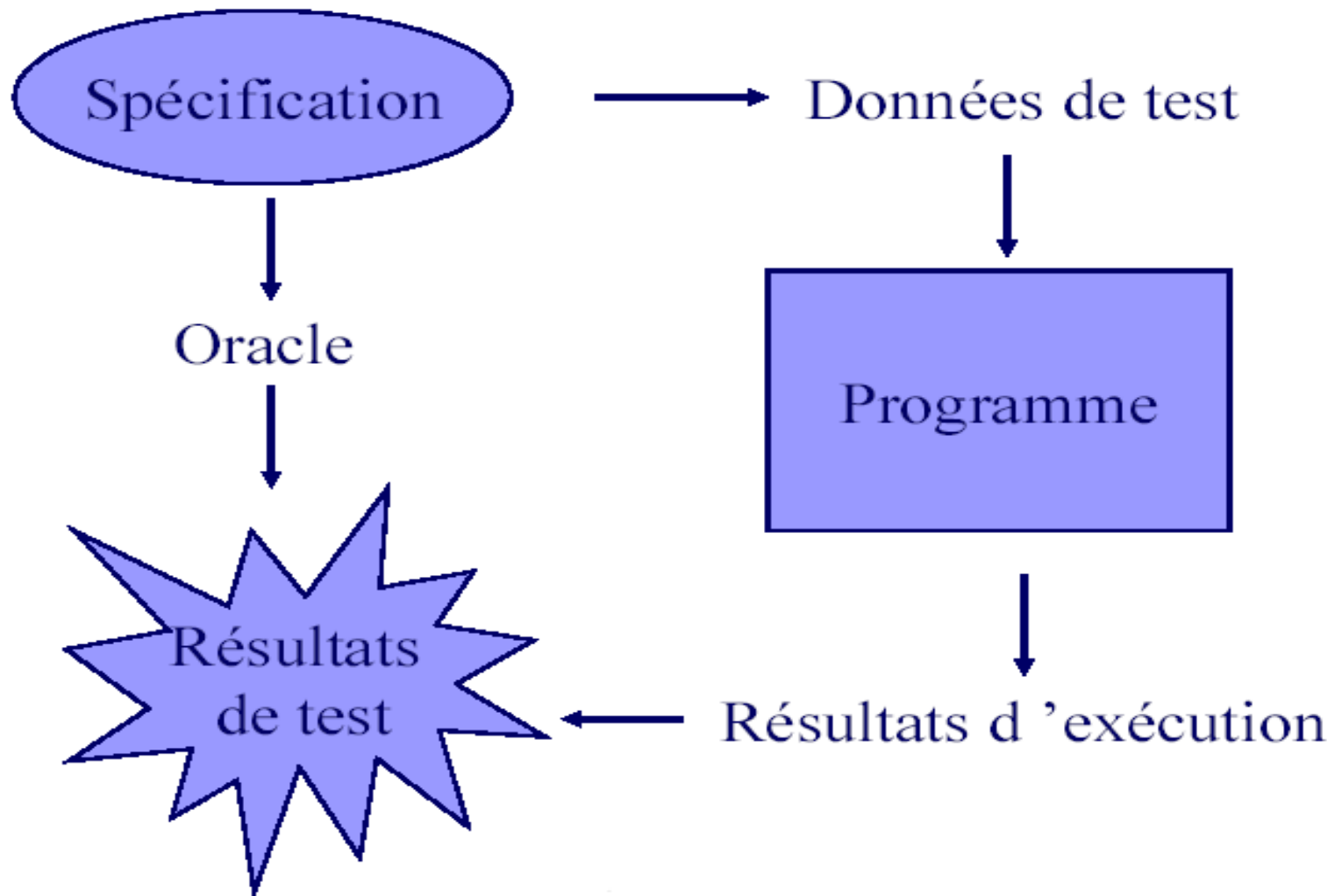
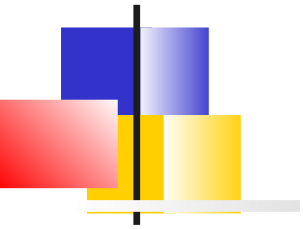
« Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus »

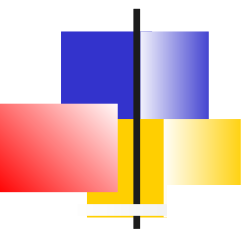
IEEE (Standard Glossary of Software Engineering Terminology)

« Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts »

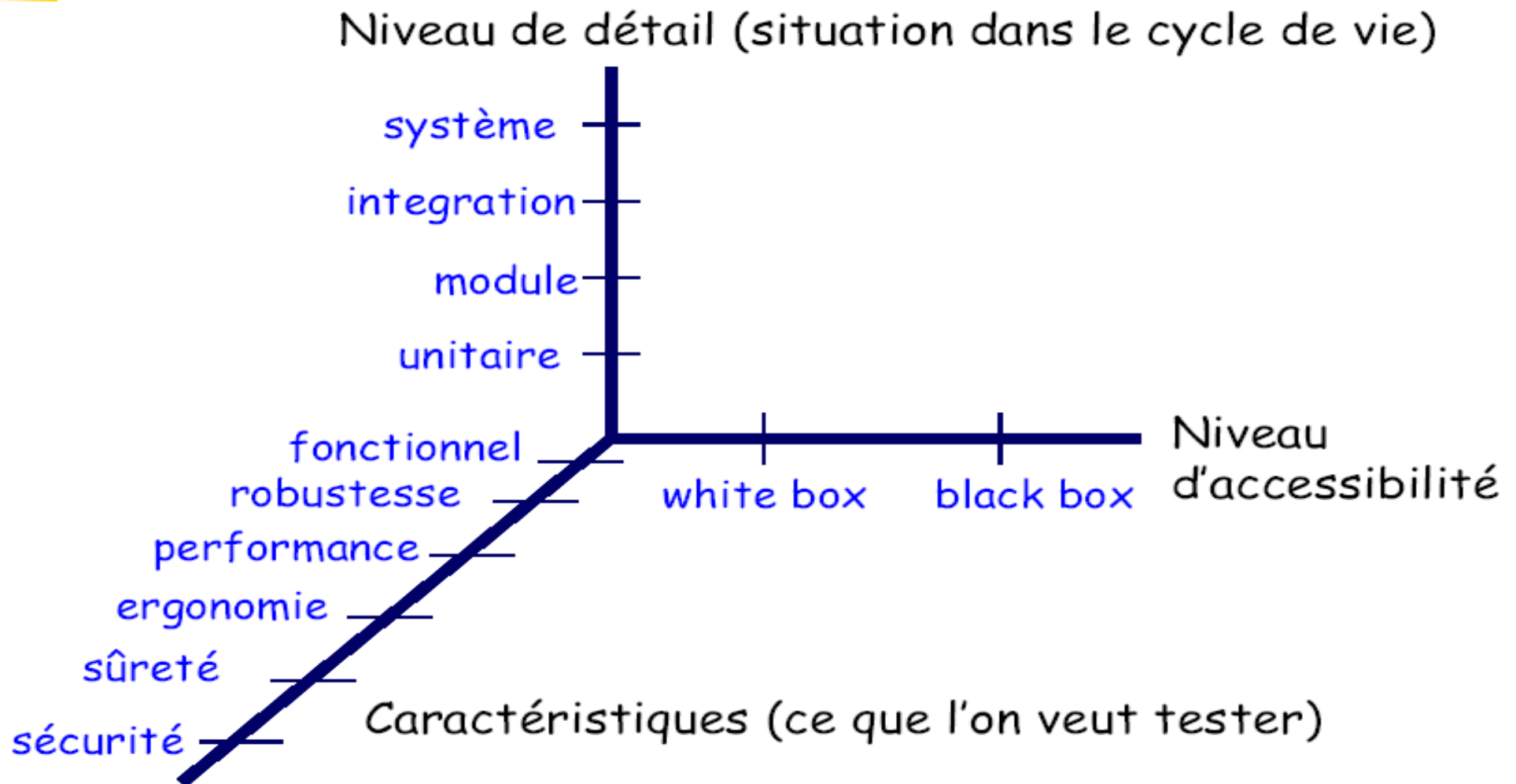
G. Myers (The Art of Software testing)

Vérification vis à vis d'une spécification





Typologie de test





Types de test

Test Unitaire :

Est-ce qu'une simple classe/méthode fonctionne correctement ?
JUnit, TestNG, Mockito

Test d'intégration :

Est-ce que plusieurs classes/couches fonctionnent ensemble ?
JUnit, BD/Serveurs embarqués, Mock serveur

Test fonctionnel :

Est-ce que mon application fonctionne ?
Selenium, HttpUnit, Protractor

Test de performance :

Est-ce que mon application fonctionne bien ?
JMeter, Gatling

Test d'acceptation :

Est-ce que mon client aime mon application ?
JUnit, Cucumber



White Box / Black Box

Les tests White Box concernent les développeurs

- Tests unitaires
- Tests d'intégration

Les tests Black Box concernent les testeurs, les fonctionnels, les intégrateurs

- Tests de performance
- Tests fonctionnels
- Tests d'acceptation



Introduction

TDD et typologie des tests

Les tests dans la pratique DevOps

Mise en place TDD, recommandations

Les outils Java



Objectifs DevOps et CI/CD

Les objectifs de l'approche DevOps et des pipeline CI/CD :

- Déployer le plus souvent possible dans les différents environnements (Intégration, QA, Production)

Chaque environnement requiert un niveau de confiance dans l'artefact déployé.

- Seuls les tests permettent d'atteindre le niveau de confiance requis
- Si ils sont automatisés, les déploiements peuvent être automatisés



Build is tests !

Chaque modification poussée par un développeur dans le SCM est construite et va être automatiquement testée.

En fonction de l'intensité des tests effectués, l'artefact construit sera éligible pour un environnement donné

L'activité de build intègre alors **tous les types de tests** que peut subir un logiciel :

- unitaires, intégration, fonctionnel, performance, sécurité, acceptation, analyse statique qualité, ...



Publication des résultats

L'exécution des tests produit des résultats.

Ces résultats sont publiés en permanence par la pipeline DevOps, permettant d'avoir une vision objective de la qualité du projet ou de son état d'avancement.

Les métriques visibles sont alors :

- Le nombre de tests échoués/réussis
- La couverture de code
- Les User/Strory implémentés ou non
- L'évolution des tests dans l'historique projet
- ...



Classification DevOps

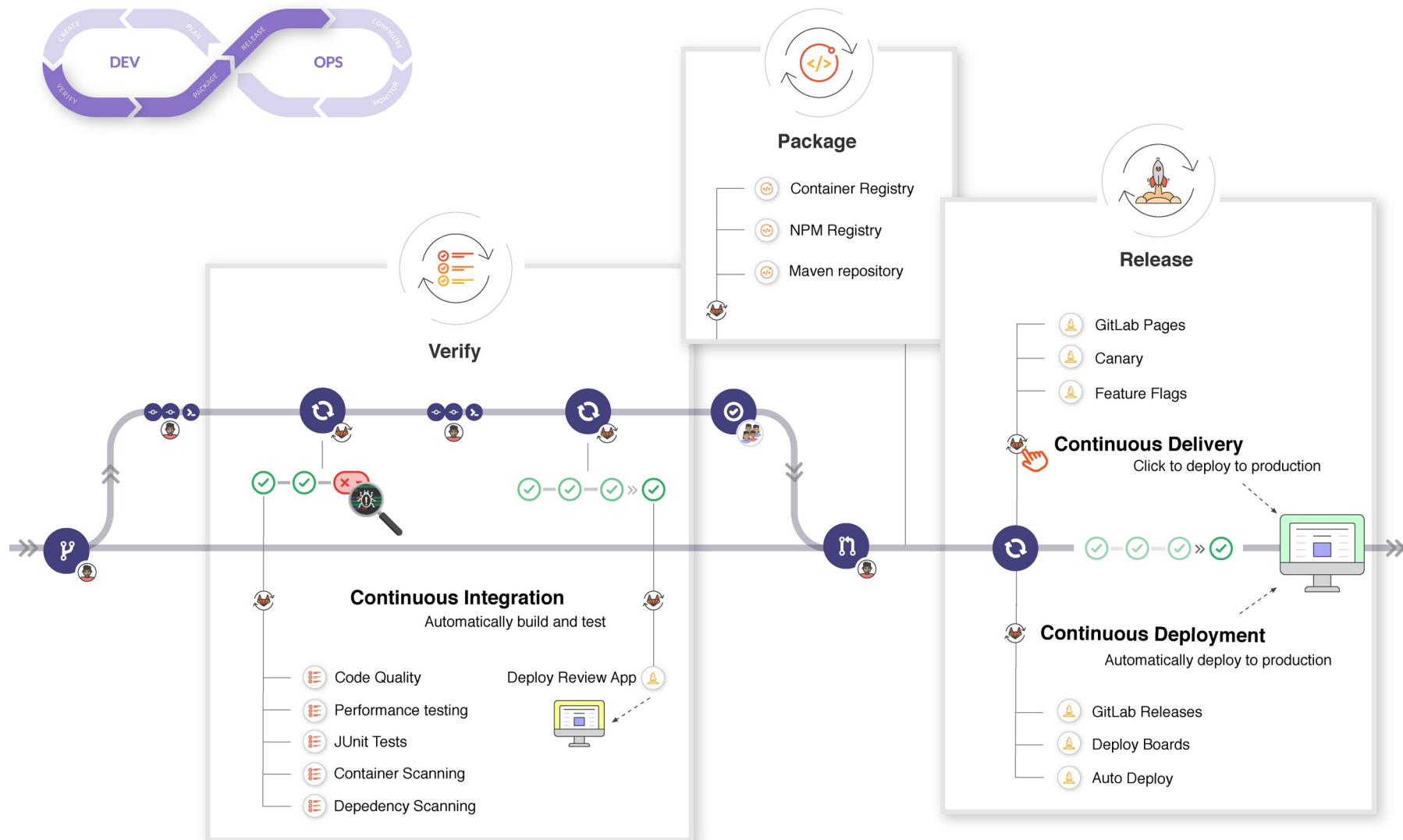
Dans l'optique d'une pipeline DevOps, on classe les tests en fonction de :

- Nécessitent-ils le provisionnement d'infrastructure ?
- Durée d'exécution

=> Conditionne la position des tests dans la pipeline

Typiquement : Tests unitaires et d'intégration en premier, tests fonctionnel, d'acceptation et performance en dernier sur les infrastructures de pré-production

Pipeline typique





Introduction

TDD et typologie des tests
Les tests dans la pratique DevOps
**Mise en place TDD,
recommandations**
Les outils du monde Java



Au niveau individuel

- Exécuter vos tests fréquemment
- Ne pas écrire trop de tests en une seule fois
- Ne pas écrire de trop grands tests ou des tests qui testent plusieurs choses à la fois
- Ne pas oublier les assertions
- Ne pas écrire des tests pour les codes triviaux, les accesseurs par exemple



Niveaux

Débutant :

- capable d'écrire un test unitaire avant le vrai code
- capable d'écrire le code suffisant afin qu'un test passe

Intermédiaire :

- Lorsqu'un bug arrive, capable d'écrire le test avant la correction
- Capable de décomposer une fonctionnalité d'un programme en une séquence de tests unitaires à écrire
- Capable de factoriser des éléments réutilisables à partir de tests unitaires existants

Senior

- Capable de formuler une road map de tests unitaires pour une partie d'un logiciel
- Capable d'appliquer la TDD pour différentes classes de langage : Orienté objet, fonctionnel, événementiel
- Capable d'appliquer la TDD pour différents domaines techniques : Calcul, Interface utilisateur, Persistance,



Au niveau de l'équipe

- Toute l'équipe doit adopter la TDD.
Pas seulement quelques développeurs à l'intérieur de l'équipe
- Ne pas abandonner des tests, les maintenir
- S'assurer que l'exécution de tous les tests unitaires restent rapides pour le projet
- Effectuer du refactoring afin que le design soit testable



Indices de réussite

La couverture de code permet de donner une indication sur la réussite de l'approche.

- Un taux de couverture de 70 % est minimal ... mais pas suffisant, les tests doivent être pertinents

Le réel métrique important : Les coûts de maintenance corrective

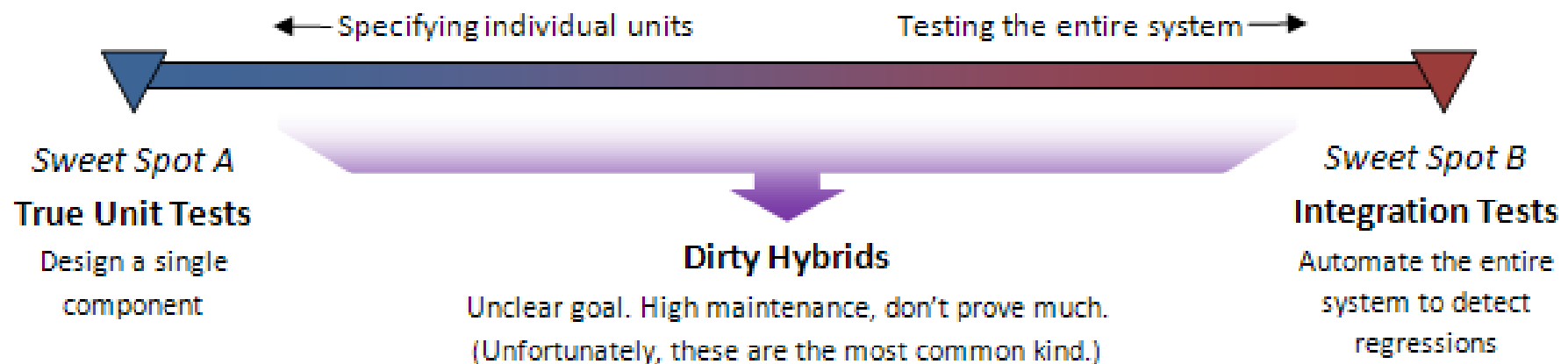


Bénéfices attendus

- Réduction très significative du nombre de bugs, en contrepartie d'un surcoût modéré du développement initial
- Précision de la spécification, documentation
- Effort de mise au point en fin de projet minimisé
- Coût des tests *black box* minimisé.
- Amélioration de la conception objet, meilleure cohésion et plus faible couplage

Attention !

Les tests peuvent avoir un impact négatif sur le projet





Introduction

TDD et typologie des tests
Les tests dans la pratique DevOps
Mise en place TDD, recommandations
Les outils du monde Java



Tests unitaires

Les 2 plus connus :

- **JUnit**, à l'origine de la TDD. Inclut directement dans les IDEs.
- **TestNG** : Inspiré de JUnit et de NUnit, il apporte quelques améliorations (à JUnit4) et essaient d'étendre son application aux tests d'intégration, fonctionnels, d'acceptation.



JUnit4 vs TestNG

Functionality - JUnit 4 vs TestNG

	Annotation Support	Exception Test	Ignore Test	Timeout Test	Suite Test	Group Test	Parameterized (primitive value)	Parameterized (object)	Dependency Test
TestNG	✓	✓	✓	✓	✓	✓	✓	✓	✓
JUnit 4	✓	✓	✓	✓	✓	✗	✓	✗	✗

<http://www.mkyong.com/unittest/junit-4-vs-testng-comparison/> (Mai 2013)

Comparaison avec JUnit5

<https://www.baeldung.com/junit-vs-testng>



Outils annexes pour le test unitaire

Mockito : Création de mock/stub objects.
Permet de facilement isoler les classes à tester

PowerMock, PowerMockito : Étend les fonctionnalités de Mockito

Hamcrest, AssertJ, Truth : Assertions plus lisibles, Notion de Matcher

JTest : Solution commerciale pour réduire les coûts des tests unitaires : génération de tests unitaires, d'assertions, analyse et optimisation du code de test, etc..



Tests d'intégration

Les **tests d'intégration** ont pour objectifs de tester la collaboration entre plusieurs composants logiciels

Ils sont plus complexes et plus long à l'exécution : démarrer un serveur, initialiser une base de données

Ils sont en général plus coûteux à maintenir

Ils sont également gérés par les développeurs



Outils

Arquillian permet de déployer un sous-ensemble du projet dans un serveur JavaEE

SpringTest permet d'initialiser un sous-ensemble du contexte applicatif Spring

JUnit, TestNG permet de distinguer tests unitaires/d'intégration

Citrus orienté protocole de transport et format de message (HTTP REST, JMS, TCP/IP, SOAP, FTP, SSH, XML, JSON)

RestAssured, Karate : Dédié aux APIs REST

DBUnit, Base de données embarquées, Hibernate : Permet d'initialiser une BD dans un état connu



Tests fonctionnels

Les **tests fonctionnels** sont des tests en boîte noire qui exécutent des scénarios d'usage de l'application et vérifient leur conformité

Ils sont en général fortement dépendants de l'interface utilisateur et de ce fait sont difficiles à automatiser et maintenir

Dans le cas d'une application web, ils simulent ou pilotent un navigateur et vérifient les réponses fournies par le serveur



Outils

HtmlUnit : Écriture des tests en Java, assertion sur les réponses du serveur

Selenium : Pilote le navigateur. Famille d'outils permettant de tester une interface. Plugins navigateur, recorder, API Java

JMeter : Simule le navigateur. Assertions sur réponse

Squash : Solution de gestion des campagnes de tests



Exemple : Selenium Web Driver

```
public class MonTest {
    public static void main(String[] args) {
        // Créer une nouvelle instance de Firefox driver
        WebDriver driver = new FirefoxDriver(); // Utiliser ca pour visiter Google
        driver.get("http://www.google.com");
        // Déterminer le champ dont le name est q
        WebElement element = driver.findElement(By.name("q"));
        element.sendKeys("Selenium"); // Taper le mot à chercher
        element.submit(); // Envoyer la formulaire
        System.out.println("Page title is: " + driver.getTitle()); // Verifier le titre de la page
        // Google fait la recherche dynamique avec JavaScript.
        // Attendre le chargement de la page de 10 secondes
        // Verifiez le titre "Selenium - Recherche Google"
        (new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>() {
            public Boolean apply(WebDriver d)
                {return d.getTitle().toLowerCase().startsWith("selenium");}
        });
        System.out.println("Page title is: " + driver.getTitle());
        // Fermer le navigateur
        driver.quit();
    }
}
```



Tests d'acceptation

Les **tests d'acceptation** ont pour but de valider que la spécification initiale est bien respectée

Les tests sont mis au point avec le métier, le testeur et le développeur (Les « 3 amigos »)

Dans les méthodes agiles, ils complètent et valident une « User Story »

En utilisant l'approche BDD (*Behaviour Driven Development*), les tests sont formalisés en langage naturel.



Tests d'acceptation et Spécifications via User Story

Pour un projet donné, chaque user story doit être écrit selon la même structure (Langage Gherkin):

Par exemple :

- **Titre**
- **Récit** : Court texte qui spécifie l'acteur et la valeur métier
- **Critère d'acceptation** ou scénarios : Une description pour chaque cas spécifique du récit dont la structure est :
 - Les conditions initiales
 - L'événement
 - L'issue



Exemple

#language : fr

Fonctionnalité: Retour lors de la saisie de détails de carte de crédit invalides
Dans les tests utilisateurs, nous avons vu beaucoup de gens qui ont fait des erreurs en entrant leur n° de carte de crédit. Nous devons être aussi utiles que possible ici pour éviter de perdre des utilisateurs à ce stade crucial de la transaction.

Contexte:

Étant donné que j'ai choisi certains articles à acheter

Et que je suis sur le point d'entrer les détails de ma carte de crédit

Scénario: numéro de carte de crédit trop court

Lorsque j'entre un numéro de carte de 15 chiffres seulement

Et tous les autres détails sont corrects

Et je soumetts le formulaire

Alors, le formulaire doit être affiché de nouveau

Et je devrais voir un message m'informant du nombre correct de chiffres

Scénario: la date d'expiration ne doit pas être antérieure

Lorsque j'entre une date d'expiration de carte qui est dans le passé

Et tous les autres détails sont corrects

Et je soumetts le formulaire

Alors, le formulaire doit être affiché de nouveau

Et je devrais voir un message me disant que la date d'expiration doit être fausse



Outils

Cucumber, JBehave : Capables de comprendre le Langage Gherkin de spécification des user stories et de les exécuter

S'intègre avec JUnit

La « plomberie » nécessaire doit être réalisé par les profils techniques :

- Code permettant d'interagir avec le système
- Association avec le langage naturel via Annotations et regexp



Exemple « *Code plomberie* »

```
private Game game;
private StringRenderer renderer;

@Given("a $width by $height game")
public void theGameIsRunning(int width, int height) {
    game = new Game(width, height);
    renderer = new StringRenderer();
    game.setObserver(renderer);
}

@When("I toggle the cell at ($column, $row)")
public void iToggleTheCellAt(int column, int row) {
    game.toggleCellAt(column, row);
}

@Then("the grid should look like $grid")
public void theGridShouldLookLike(String grid) {
    assertThat(renderer.asString(), equalTo(grid));
}
```



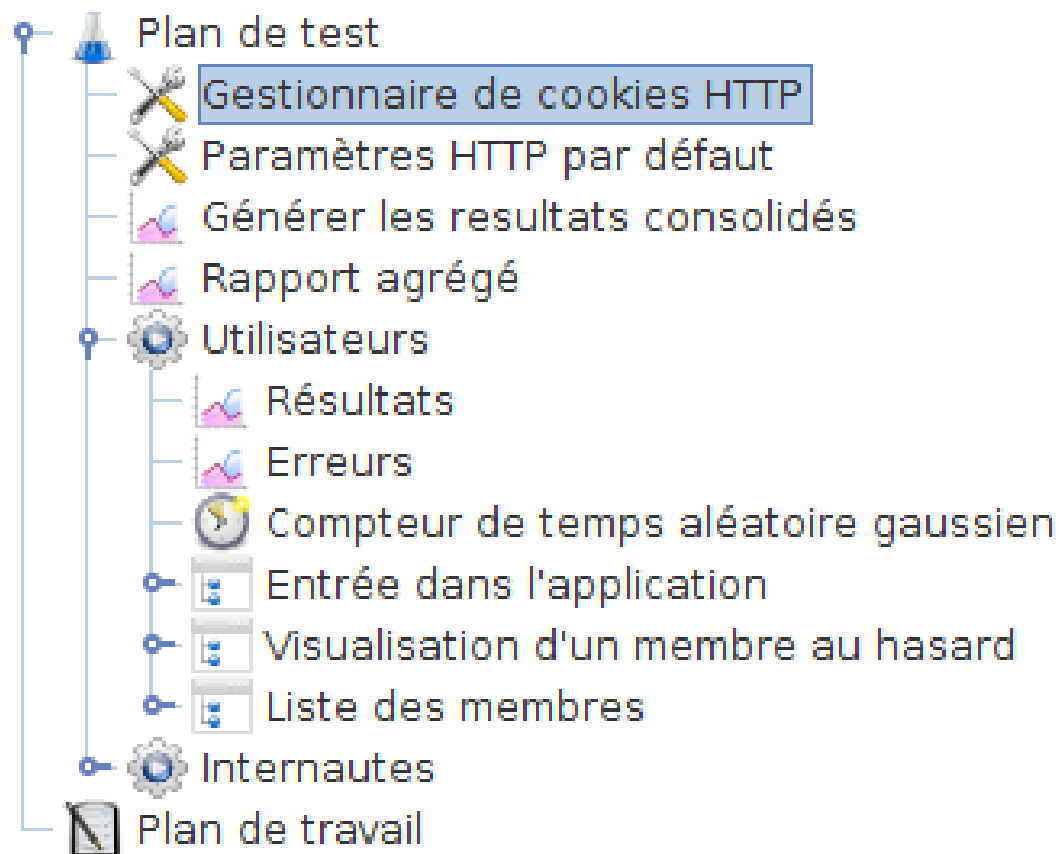
Tests de performance

JMeter : Simulation d'une charge utilisateur et mesure des temps de réponse, du taux d'erreur. Interface graphique pour la mise au point des tests, tests distribués. Reporting standard automatique

Gatling : Script de charge avec son propre DSL. Reporting généré automatiquement

LoadRunner : Solution commerciale la plus répandue

Exemple JMeter



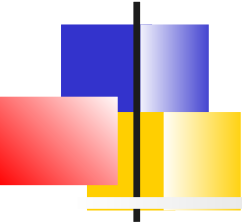


Analyse qualité

SonarQube : Outil collectant tous les métriques standard quelque soit la technologies

Autres outils qualité :

- ***Jalopy, checkstyle*** : respect des normes de codage
- ***JDepend*** : Métriques sur les dépendances
- ***Cobertura, jacocco*** : Couverture de tests
- ***FindBug, PMD/CMP*** : Détection des anti-patterns



Exemples de métriques *SonarQube*

Complexité : Nombre de chemins possibles du code source

Commentaires : Nombre de lignes de commentaire

Duplication : Nombre de lignes dupliquées

Maintenabilité : Nombre de « code smells » (i.e mauvaises pratiques de conception : anti-pattern)

Fiabilité : Nombre de bugs

Sécurité : Nombre de vulnérabilités

Couverture de test : Pourcentage de lignes et de conditions testées par les test-unitaires



Le framework JUnit

Concepts JUnit

Matchers

Isolation avec Mockito

Compléments



xUnit

xUnit est le nom collectif de tous les frameworks orientés tests unitaires

Ils dérivent de par leur structure et leur fonctionnalité du projet Smalltalk's *Sunit* développé par Kent Beck en 1998

Il a été porté facilement dans les langages orientés objet .



Vocabulaire *xUNit* (1)

Exécuteur (Test runner) : Programme exécutant les tests Exécutable et écrivant les résultats

Cas de test (Test case) : La classe dont tous les classes de tests unitaires héritent.

Contexte de test (Test fixtures) : Ensemble des préconditions pour exécuter un test. État initial du test.

Suite de tests (Test suites) : Un ensemble de tests partageant le même contexte. L'ordre des tests n'a alors pas d'importance.



Vocabulaire *xUNit* (2)

Exécution : L'exécution individuelle d'un test unitaire consiste de 3 phases :

- **setUp** : Initialise le contexte de test
- **test** ; Le test en lui même
- **tearDown** : Nettoyage du test pour revenir dans un état initial

Formatteur de résultat : Un exécuteur produit des résultats en 1 ou plusieurs formats (text, XML). Le format (créé par Junit) est souvent compris par d'autres outils (*Jenkins, Atlassian Bamboo*).

Assertions : Une assertion est une fonction ou macro qui vérifie le comportement (ou l'état) du code (unité) à tester.

Elle exprime généralement une condition logique sur le résultat attendu.

L'échec d'une assertion interrompt le test en cours



Versions

3 modèles de programmation des classes de tests en fonction des versions de *JUnit* :

- ***JUnit3*** : Basé sur le nommage des classes et des méthodes de test ainsi que l'héritage
- ***JUnit4*** : Utilisation des annotations
- ***JUnit5*** : Compatible Java8 et lambda expressions



JUnit5

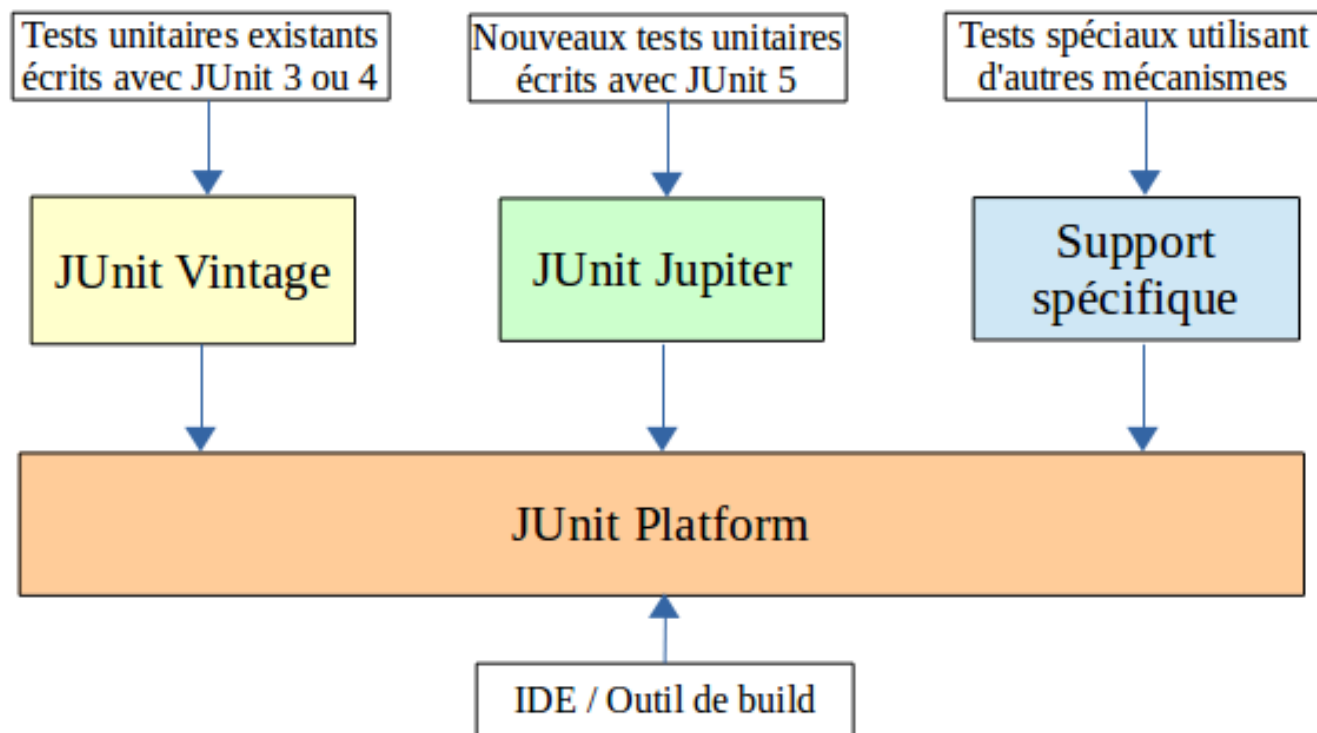
JUnit 5 est composé de 3 sous-projets :

- **JUnit Platform** : API cœur pour l'intégration avec les frameworks de tests et les outils de build. Fournit un exécuteur JUnit4
- **JUnit Jupiter** : Fournit un exécuteur JUnit5
- **JUnit Vintage** : Permet l'exécution de tests JUnit3 et 4

Projet exemples :

<https://github.com/junit-team/junit5-samples>

Composants JUnit5





JUnit : API

Librairie apportant les notions de :

- **TestCase** : Test d'une classe
- **TestSuite** : Enchaînement de *TestCase*
- **Assertion** : Comparaison entre résultat attendu et résultat obtenu

TestRunner permettant l'exécution de test

- Ligne de commande
- Outil de build
- IDE

Collecte des résultats en particulier les tests non passés (Distinction entre les failures et les erreurs d'assertion).

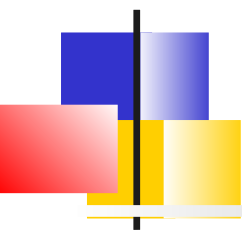


TestCase JUnit4+

A partir de JUnit4, il n'est plus nécessaire que le *TestCase* hérite d'une classe du framework.

Un cas de test est une classe classique dont une de ses méthodes est annotée par :

- **@Test**
- @RepeatedTest, @ParameterizedTest, @TestFactory, ou @TestTemplate



JUnit : Méthodes de test

Les méthodes de test interagissent avec la classe à tester et utilisent des assertions permettant de valider/invalidier le test d'une méthode dans des conditions particulières

@Test

```
public void addWithSameCurrency() {  
    Money m12CHF= new Money(12, "CHF");  
    Money m14CHF= new Money(14, "CHF");  
    Money expected= new Money(26, "CHF");  
    Money result=m12CHF.add(m14CHF);  
    assertTrue(expected.equals(result));  
}
```




JUnit : Fixture

Les **fixtures** désignent les données de test qui fixent l'environnement d'exécution des tests.

Elles permettent la reproductibilité du test.

- Définir une variable pour chaque donnée de la fixture
- Utiliser des annotations pour initialiser les variables.
- Utiliser des annotations pour relâcher toutes les ressources allouées le code d'initialisation



Annotations pour les fixtures

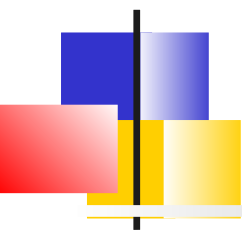
@BeforeAll, @AfterAll : Méthodes exécutées une fois avant/après toutes les méthodes de Test.

Équivalent à *@BeforeClass*, *@AfterClass* de JUnit4

@BeforeEach, @AfterEach :

Méthodes exécutées avant/après chaque méthode de test.

Équivalent à *@Before* et *@After* de JUnit4



JUnit : Exemple

```
public class MathTest {
    protected double fValue1, fValue2;

    @BeforeEach
    protected void setUp() {
        fValue1= 2.0; fValue2= 3.0;
    }

    @Test
    public void testAdd() {
        double result= add(fValue1,fValue2);
        assertTrue(result == 5.0);
    }

    @Test
    public void testSubstract() {
        double result= sub(fValue2,fValue1);
        assertTrue(result == 1.0);
    }
}
```



Instanciación de clases

Afin que les méthodes de test soient exécutées en isolation, *JUnit* crée une nouvelle instance de la classe de test avant l'exécution de chaque méthode de test.

- Jusqu'à *JUnit* 5, les constructeurs de classe de test ne pouvaient pas prendre d'arguments
- Avec *JUnit* 5, on peut profiter d'injection de dépendance via des constructeurs et méthodes (voir +loin)



Assertions

Les assertions sont des méthodes utilitaires permettant de vérifier les résultats du test.

Ce sont des méthodes statiques accessibles via ***Assert*** dans JUnit4 et ***Assertions*** dans JUnit5

Par soucis de lisibilité, il est recommandé d'importer de manière statique la classe respective.



JUnit4 : Méthodes statiques de *Assert*

Avec *JUnit4*, L'ordre des paramètres des méthodes de la classe *Assert* est :

- Optionnellement, un message d'erreur si l'assertion échoue
- La valeur attendue
- La valeur réelle



JUnit4 : Méthodes statiques de *Assert*

```
assertEquals("failure - strings are not equal", "text", "text");

assertArrayEquals("failure - byte arrays not same", expected, actual);
assertTrue("failure - should be true", true);
assertFalse("failure - should be false", false);
assertNull("should be null", null);
assertNotNull("should not be null", new Object());
assertNotSame("should not be same Object", new Object(), new Object());
assertSame("should be same", aNumber, aNumber);

// Faire échouer
fail("Exception not thrown") ;
// Dernier paramètre : objet Matcher. Attention déprécié
assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));
```



JUnit5 : *Assertions*

Assert est remplacée ***Assertions*** qui ajoute 4 nouvelles méthodes :

- ***assertAll*** qui regroupe en argument des lambdas exécutant d'autres assertions
- ***assertThrows*** pour indiquer qu'on s'attend à voir survenir une exception
- ***assertTimeout*** ou ***assertTimeoutPreemptively*** selon que l'on souhaite attendre ou non la fin d'exécution d'un traitement testé par rapport à une contrainte de temps

Les messages optionnels d'échec sont en dernier paramètre (lazy initialisation)

assertThat et les *Matcher* ne sont plus inclus.



Exemples JUnit5

```
assertAll("address",  
    () -> assertEquals("John", address.getFirstName()),  
    () -> assertEquals("User", address.getLastName())  
);
```

```
Throwable exception =  
assertThrows(IllegalArgumentException.class, () -> {  
    throw new IllegalArgumentException("a message");  
});
```

```
assertTimeoutPreemptively(ofMillis(10), () -> {  
    Thread.sleep(100); });
```



Tests des exceptions

Il est souvent nécessaire de vérifier que le code lance bien une exception sous certaines conditions

On peut utiliser la méthode *fail()*

```
public void testExceptionMessage() {  
    try {  
        new ArrayList<Object>().get(0);  
        fail("Expected an IndexOutOfBoundsException to be thrown");  
    } catch (IndexOutOfBoundsException anIndexOutOfBoundsException) {  
        assertThat(anIndexOutOfBoundsException.getMessage(),  
is("Index: 0, Size: 0"));  
    }  
}
```



Tests des exceptions

Le paramètre ***expected*** de l'annotation *@Test* permet de spécifier les exceptions attendues

```
@Test(expected = IndexOutOfBoundsException.class)  
public void empty() {  
    new ArrayList<Object>().get(0);  
}
```



JUnit5 : *assertThrow*

La méthode ***assertThrow*** est utilisée pour tester le lancement d'un type d'exception.

La valeur de retour peut également être utilisée pour tester les détails.

```
@Test
void shouldThrowException() {
    Throwable exception =
        assertThrows(UnsupportedOperationException.class,
        () > {
            throw new UnsupportedOperationException("Not supported");
        });
    assertEquals(exception.getMessage(), "Not supported");
}
```



Timeout

Avec JUnit4, Un timeout en millisecons peut être spécifié pour l'exécution d'une méthode de test.

- Si le timeout est dépassé une failure est déclenchée via une Exception :

```
@Test(timeout=1000)  
public void testWithTimeout() {  
    ...  
}
```

JUnit5 propose des assertions de timeout :

assertTimeout ou ***assertTimeoutPreemptively***

```
assertTimeoutPreemptively(ofMillis(10),  
    () -> { Thread.sleep(100); });
```



Librairies tierces et Matcher

Des librairies tierces comme *Hamcrest*, *AssertJ*, *Truth*, *JsonAssertion* ajoutent des fonctionnalités supplémentaires en particulier les **Matchers**.

Les objets *Matcher* permettent de spécifier plusieurs contraintes sur un objet

- La méthode *assertThat* de JUnit4 utilisant des *Matcher Hamcrest* a disparu de JUnit5, il faut alors utiliser les mécanismes propres à la librairie tierce.



Examples

```
assertThat("albumen",  
both(containsString("a")).and(containsString("b")));
```

```
assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }),  
everyItem(containsString("n")));
```

```
@Test  
public void testNonAuthenticatedUser() throws Exception {  
    restUserMockMvc.perform(get("/api/authenticate")  
        .accept(MediaType.APPLICATION_JSON))  
        .andExpect(status().isOk())  
        .andExpect(content().string(""));  
}
```



Exécution des tests

Plusieurs alternatives pour exécuter les tests :

- L'IDE. Support pour IntelliJ IDEA, Eclipse, Netbeans, VSCode
- Les outils de build.
 - Maven avec le plugin *SureFire* et *FailSafe*
 - Gradle support natif
- L'exécuteur *JUnit* (Console Launcher)

```
java -jar junit-platform-console-standalone-1.7.0.jar <Options>
```




Intégration IDE

Les IDEs embarquent en général une version de JUnit.

Cependant, il est préférable d'indiquer explicitement la dépendance JUnit que l'on veut utiliser dans le *pom.xml* ou *build.gradle*



Options de l'exécuteur *JUnit*

De nombreuses options sont disponibles.
Elles permettent principalement :

- De spécifier un point d'entrée pour la découverte des classes et méthodes de tests :
Packages, Classes, Ressources, Répertoires, fichiers, URIs
- De spécifier des filtres, pour exclure/inclure les tests à exécuter:
Via des expressions régulières, des tags, des moteurs d'exécution



JUnit4 : @RunWith

Si une classe est annotée avec **@RunWith** (ou étend une classe annotée avec *@RunWith*), JUnit invoquera la classe référencée plutôt que le runner prédéfini de *JUnit*.
(*BlockJUnit4ClassRunner*)

Exemple :

```
@RunWith(SpringJUnit4ClassRunner.class)
```

On peut exécuter des tests JUnit5 avec un environnement JUnit4 :

```
@RunWith(JUnitPlatform.class)
```



Rapports de test

Le Runner est également responsable de l'écriture des rapports de test.

Ceux-ci peuvent être contrôlés via :

- Les ***TestSuite*** : Regroupements de tests
- Le nommage des méthodes !!
- Ou mieux avec l'annotation ***@DisplayName*** (JUnit5)



TestSuite

Les ***TestSuite*** permettent d'exécuter plusieurs cas de test et d'agréger leurs résultats

Ils utilisent des runners spécialisés

Avec *JUnit4*, On annote la suite avec :

`@RunWith(Suite.class)` et
`@SuiteClasses(TestClass1.class, ...)`

Avec *JUnit5*, on utilise `@SuiteDisplayName`
et `@SelectClasses` ou `@SelectPackages`



Example *JUnit* 4.x

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
```

```
@RunWith(Suite.class)
```

```
@Suite.SuiteClasses({
    TestFeatureLogin.class,
    TestFeatureLogout.class,
    TestFeatureNavigate.class,
    TestFeatureUpdate.class
})
```

```
public class FeatureTestSuite {
    // the class remains empty,
    // used only as a holder for the above annotations
}
```



Exemple *JUnit5*

```
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.platform.suite.api.SuiteDisplayName;
import org.junit.runner.RunWith;
```

```
@RunWith(JUnitPlatform.class)
@SuiteDisplayName("JUnit 5 Suite Demo")
@SelectPackages("example")
public class JUnit5SuiteDemo {
}
```



Préconisations (1)

1 classe écrite => 1 classe de test

1 méthode *public* ou *package*

- Au moins une méthode de test
- Plusieurs appels de la méthode, cas aux limites et levées d'exceptions...

Écrire la classe de test dans le même package que la classe testée

- Plus facile de retrouver la classe testée à partir du rapport
- Permet de tester les méthodes à visibilité *package*



Préconisations (2)

Les tests unitaires doivent toujours créer leur propre données de tests.

- => Indépendance vis à vis d'un environnement
- => Les tests sont reproductibles

Si de nombreux tests unitaires nécessitent les mêmes données de test, utiliser les techniques OO (héritage) pour ne pas se répéter.



Préconisations (3)

Chaque test doit être orthogonal (indépendant) par rapport aux autres

Chaque comportement doit être spécifié dans un unique test (pas de redondance)

Pas de redondance dans les assertions

Pas de dépendance sur la séquence des tests

Éviter les fixtures non nécessaires (attention lors de la mutualisation des codes de *setUp()*)

Ne pas tester les variables de configuration

Nommer clairement les tests et de façon cohérente. Le nom doit comporter :

- Le sujet (la méthode testée)
- Le scénario (les données de test)
- Le résultat attendu



Le framework JUnit

Concepts *JUnit*

Matchers

Isolation avec Mockito

Compléments



Hamcrest

Hamcrest a pour objectif de rendre le plus lisible possible les assertions.

Il est basé sur la notion de matchers qui encapsulent toutes les conditions à tester sur un objet.

Les matchers sont expressifs et peuvent être combinés.

Les messages d'erreur générés par Hamcrest sont plus facile à lire.



Dépendances

```
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-all</artifactId>
  <version>1.3</version>
  <scope>test</scope>
</dependency>
```

Il est ensuite conseillé d'ajouter les directives d'import de tous les *Matchers* possible cela facilite le travail des outils d'aide à la complétion.

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;
```



Principaux matcher

not : Applique la négation au matcher encapsulé

equalTo : Test avec la méthode equals

is : décorateur pour la lisibilité

hasToString : Test le retour de la méthode
toString

isA, instanceof, isCompatibleType : test sur le
type

notNullValue, nullValue : Test pour null

sameInstance : Test l'identité



Principaux matcher (2)

hasProperty : Test la propriété d'un JavaBeans

closeTo : Test si une valeur flottante est proche d'une valeur donnée

greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo

equalToIgnoringCase : Test l'égalité de String sans prendre en compte la casse

equalToIgnoringWhiteSpace : Sans prendre en compte les espaces

containsString, endsWith, startsWith : Test sur les String



Exemples *is()* et *isA()*

Le matcher ***is()*** permet les tests d'égalité, ***isA()*** les tests de type

Exemples :

// Tests identiques

```
assertThat(a, equalTo(b));  
assertThat(a, is(equalTo(b)));  
assertThat(a, is(b));
```

// Not Equal

```
assertThat(actual, is(not(equalTo(expected))));
```

// Test sur le type

```
assertThat(Long.valueOf(1),  
instanceOf(Integer.class));  
assertThat(Long.valueOf(1), isA(Integer.class));
```




Matcher - Collection

everyItem : Applique la contrainte sur l'ensemble des éléments d'une collection

contains, containsInAnyOrder : Test si une liste contient une sous-liste en tenant en compte ou non l'ordre

hasItem, hasItems, hasItemInArray : Test si une collection/array contient des éléments

hasSize : Taille d'une collection

hasEntry, hasKey, hasValue : Test si une map contient une entrée, une clé, une valeur



Examples

```
public class HamcrestListMatcherExamples {  
    @Test  
    public void listShouldInitiallyBeEmpty() {  
        List<Integer> list = Arrays.asList(5, 2, 4);  
        assertThat(list, hasSize(3));  
        // ensure the order is correct  
        assertThat(list, contains(5, 2, 4));  
        assertThat(list, containsInAnyOrder(2, 4, 5));  
        assertThat(list, everyItem(greaterThan(1)));  
    }  
}
```



Combiner les Matchers

Les matchers peuvent être combinés.

Les méthodes ***both*** et ***either*** retournent des ***CombinerMatcher*** supportant la méthode ***and()*** ou ***or()*** :

```
assertThat(list, both(hasSize(1)).and(contains(42)));  
assertThat(list, either(hasSize(1)).or(hasSize(2)));
```

Les méthodes ***allOf*** et ***anyOf*** offrent des raccourcis :

```
assertThat(list, allOf(hasSize(1), contains(42)));  
assertThat("test", anyOf(hasSize(1), hasSize(2)));
```



Matcher custom

Lorsque l'on a besoin de tester plusieurs fois les mêmes combinaison de matcher, il peut être intéressant d'implémenter ses propres Matchers

Hamcrest fournit 2 classes de base :

- ***FeatureMatcher*** : Pour les matchers testant une caractéristique d'un objet. Il suffit d'implémenter alors ***featureValueOf()***
- ***TypeSafeMatcher*** : La classe de base effectue les tests d'une valeur non *null* et d'un type spécifique. Il suffit de la compléter avec une implémentation de ***matchesSafely()***



Exemple *FeatureMatcher*

```
@Test
public void fellowShipOfTheRingShouldContainer7() {
    assertThat("Gandalf", length(is(8)));
}

public static Matcher<String> length(Matcher<? super
Integer> matcher) {
    return new FeatureMatcher<String, Integer>(matcher,
"a String of length that", "length") {
        @Override
        protected Integer featureValueOf(String actual) {
            return actual.length();
        }
    };
}
```



Exemple *TypeSafeMatcher*

```
public class RegexMatcher extends TypeSafeMatcher<String> {
    private final String regex;
    public RegexMatcher(final String regex) {
        this.regex = regex;
    }
    @Override
    public void describeTo(final Description description) {
        description.appendText("matches regular expression=" + regex + "`");
    }
    @Override
    public boolean matchesSafely(final String string) {
        return string.matches(regex);
    }
    // matcher method you can call on this matcher class
    public static RegexMatcher matchesRegex(final String regex) {
        return new RegexMatcher(regex);
    }
}
```



Le framework JUnit

Concepts *JUnit*
Matchers

Isolation avec *Mockito*

Compléments

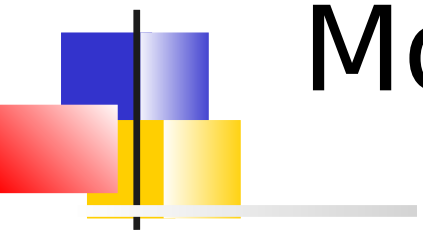


Isolation

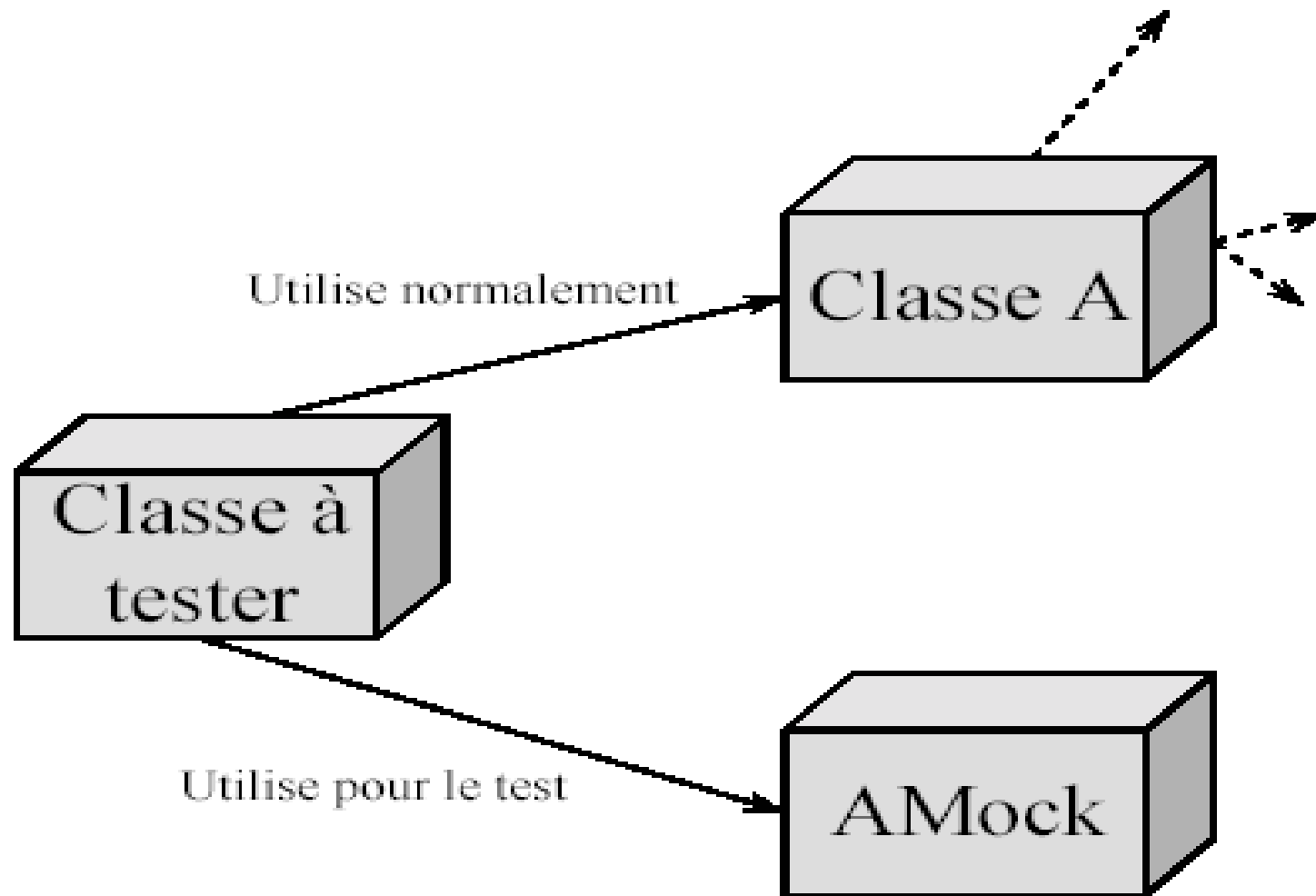
Les parties à tester doivent être isolées.

L'isolation apporte plusieurs bénéfices :

- Tests peuvent être réalisés même si les parties dont dépend le code ne sont pas encore développées
- Permet d'éviter les effets de bord
- Permet de tester le code lorsque les parties dont il dépend ont des erreurs



Mock objects





Classification des « bouchons »

On peut distinguer plusieurs types de bouchon :

- Les objets ***dummy*** sont passés en argument mais jamais utilisés. Ses méthodes ne sont jamais appelées. Ils peuvent être utilisés pour remplir les paramètres d'une méthode.
- Les objets ***fake*** ont des implémentations fonctionnelles mais simplifiées. Par exemple, une base de données mémoire.
- Les classes ***stub*** est une implémentation partielle d'une classe ou d'une interface. Elles ne répondent qu'à ce qui est programmé pour les tests, et peuvent également tracer les appels.
- Un objet ***mock*** est une implémentation d'une interface ou d'une classe dans laquelle sont définis les résultats de certains appels. Ils sont configurés pour un comportement spécifique durant le test. Ils sont plus facile à mettre en place



Mockito

Mockito permet de créer et configurer les des mock et également des stubs.

L'exécution d'un test consiste généralement à :

- Spécifier les résultats des objets mockés en fonction de paramètres d'entrée.
- Les injecter dans le code à tester
- Exécuter le code à tester
- Optionnellement, vérifier que l'objet Mock a été correctement appelé
- Vérifier le résultat attendu



Dépendances

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>3.6.0</version>
  <scope>test</scope>
</dependency>
```

```
<!-- Extension JUnit5 -->
```

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-junit-jupiter</artifactId>
  <version>3.6.0</version>
  <scope>test</scope>
</dependency>
```

```
import static org.mockito.Mockito.*;
```



Usage

Pour commencer à créer des mock avec mockito, plusieurs alternatives :

- Manuellement
Utiliser la méthode statique ***Mockito.mock***
- Utilisé l'annotation ***@Mock*** et initialiser Mockito avec :
 - ***MockitoAnnotations::initMocks***
 - Si JUnit4
@RunWith(MockitoJUnitRunner.class)
 - Si extension JUnit5
@ExtendWith(MockitoExtension.class)



Exemple JUnit5

@ExtendWith(MockitoExtension.class)

```
public class ExampleTest {
```

```
    @Mock
```

```
    private List list;
```

```
    @Test
```

```
    public void shouldDoSomething() {
```

```
        list.add(100);
```

```
    }
```

```
}
```



Configuration des mocks

Mockito permet de configurer les valeurs de retour des méthodes des Mock pour des paramètres prédéfinis

Cela est effectué via la chaîne de méthodes ***when(...).thenReturn(...)***

Ex:

```
// Définir la valeur de retour pour  
getUniqueId()
```

```
MyClass test = mock(MyClass.class);
```

```
when(test.getUniqueId()).thenReturn(43);
```



Compléments

Un appel avec des paramètres non prédéfinis retourne *null*, 0, *false*, *empty*, ...

Les méthodes *anyString*, *anyInt*,... permettent de définir un retour en fonction du type d'un paramètre

```
Comparable<Integer> c= mock(Comparable.class);  
when(c.compareTo(anyInt())).thenReturn(-1);
```

Il est possible de spécifier plusieurs valeurs de retour. Elles sont alors retournées dans l'ordre des appels

```
Iterator<String> i= mock(Iterator.class);  
when(i.next()).thenReturn("Mockito").thenReturn("rocks");
```

La chaîne peut spécifier le lancement d'une Exception

```
Properties properties = mock(Properties.class);  
when(properties.get("Anddroid")).thenThrow(new IllegalArgumentException(...));
```




Exemple (JUnit4)

```
@RunWith(MockitoJUnitRunner.class)
public class MyControllerTest {

    @Mock
    MyService1 myServiceMock;

    @Test
    public void testMyMethod() {
        // Conditions and injections
        when(myServiceMock.callService(member)).thenReturn(result);
        MyController myController = new MyController();
        myController.setMytService(myServiceMock);

        // Méthode à tester
        String ret = loginController.login();
        // Assertion
        assertTrue(ret.equals(expected));
        // Vérification
        verify(myServiceMock).callService(member);
    }
}
```



UnnecessaryStubbingException

Depuis la version 2.x, Mockito vérifie que les mocks/stubs configurés sont réellement utilisés, si ce n'est pas le cas il lance l'exception

UnnecessaryStubbingException

Si l'on veut être plus permissif, on peut utiliser la méthode ***lenient()***

Par ex :

```
lenient().when(mockList.add("one")).thenReturn(true);
```



Vérification explicite des appels

Mockito garde une trace de tous les appels et de leurs paramètres.

La méthode ***verify()*** permet de vérifier explicitement que les appels ont eu lieu

```
// Vérification de l'appel de la méthode size()
verify(mockedList).size();
// Vérification du nombre d'appels
verify(mockedList, times(1)).size();
// Vérification qu'il n'y a pas eu d'interaction
verifyNoInteractions(mockedList);
verify(mockedList, times(0)).size();
verify(mockedList, never()).clear();
// Vérifier l'ordre des interactions
InOrder inOrder = Mockito.inOrder(mockedList);
inOrder.verify(mockedList).size();
inOrder.verify(mockedList).add("a parameter");
// Vérifier les interactions avec les arguments (Voir également ArgumentCaptor)
verify(mockedList).add("test");
verify(mockedList).add(anyString());
```



Encapsulation d'objet avec Spy

Avec **@Spy** ou **spy()**, il est possible de se créer des *stubs* à partir d'objets réels. Seule une partie de l'objet est mockée.

Les appels de méthodes peuvent être délégués soit à la classe réelle soit aux méthodes *stubbed* et Mockito permet de traquer et vérifier tous les appels de méthodes

```
@Spy
List<String> spyList = new ArrayList<String>();
@Test
public void whenUsingTheSpyAnnotation_thenObjectIsSpied() {
    spyList.add("one");
    spyList.add("two");
    Mockito.verify(spyList).add("one");
    Mockito.verify(spyList).add("two");
    assertEquals(2, spyList.size());
    when(spyList.size()).thenReturn(100) ;
    assertEquals(100, spyList.size());
}
```



Autres annotations

@Captor : Permet de capturer les valeurs des arguments passés aux méthodes.

@InjectMocks : Permet d'injecter des mock objects dans les champs de l'objet à tester



Exemple *@Captor*

```
@Mock
List mockedList;
@Captor
ArgumentCaptor argCaptor;
@Test
public void whenUseCaptorAnnotation_thenTheSam() {
    mockedList.add("one");
    Mockito.verify(mockedList).add(argCaptor.capture());
    assertEquals("one", argCaptor.getValue());
}
```



Example *@InjectMocks*

```
@Mock
Map<String, String> wordMap;
@InjectMocks
MyDictionary dic = new MyDictionary();
@Test
public void whenUseInjectMocksAnnotation_thenCorrect() {
    Mockito.when(wordMap.get("aWord")).thenReturn("aMeaning");
    assertEquals("aMeaning", dic.getMeaning("aWord"));
}

public class MyDictionary {
    Map<String, String> wordMap;
    public MyDictionary() {
        wordMap = new HashMap<String, String>();
    }
    public void add(final String word, final String meaning) {
        wordMap.put(word, meaning);
    }
    public String getMeaning(final String word) {
        return wordMap.get(word);
    }
}
```



Compléments JUnit



Injection de dépendances

Avec JUnit5, il est possible d'injecter des arguments aux constructeurs et aux méthodes de test.

Si un constructeur, une méthode de test ou une méthode de Callback accepte un paramètre, le paramètre doit être résolu au moment de l'exécution par un ***ParameterResolver***.

3 implémentations sont fournies par JUnit5 :

- *TestInfoParameterResolver* permet d'injecter des paramètres de type ***TestInfo*** (avoir des informations sur le contexte d'exécution du test)
- *RepetitionInfoParameterResolver* permet d'injecter des paramètres de type ***RepeatedTestInfo*** (Voir + loin)
- *TestReporterParameterResolver* permet d'injecter des paramètres de type ***TestReporter*** (permettant de publier des résultats de tests additionnels)

Il est possible d'implémenter ses propres *ParameterResolver*



Tests paramétrés – JUnit 4

Le runner spécialisé ***Parameterized*** exécute la même méthode de tests pour toutes les valeurs possibles d'un jeu de données.

- Le jeu de données est indiqué via l'annotation ***@Parameters***
- Il est injecté via le constructeur ou par l'annotation ***@Parameter***



Example

@RunWith(Parameterized.class)

```
public class FibonacciTest {  
    @Parameters  
    public static Collection<Object[]> data() {  
        return Arrays.asList(new Object[][] {  
            { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }  
        });  
    }  
  
    private int fInput;  
    private int fExpected;  
  
    public FibonacciTest(int input, int expected) {  
        fInput= input;  
        fExpected= expected;  
    }  
  
    @Test  
    public void test() {  
        assertEquals(fExpected, Fibonacci.compute(fInput));  
    }  
}
```



JUnit5 *@ParameterizedTest*

JUnit5 permet d'annoter des méthodes avec
@ParameterizedTest

- => A la différence de *JUnit4*, une classe de test peut donc contenir des tests paramétrés et non paramétrés

L'annotation s'utilise avec un ensemble d'annotations permettant de définir le jeu de données :

- *@ValueSource*, *@EnumSource*, *@MethodSource*,
@CsvSource, *@CsvFileSource* et *@ArgumentsSource*

Il offre également ***@RepeatedTest*** qui permet de simplement répéter une méthode de test.



Source des paramètres

Les annotations définissant la source des paramètres sont :

- **@ValueSource** permet de fournir un tableau de type primitif à une méthode ne prenant qu'1 seul paramètre
- **@EnumSource** permet de fournir toutes les valeurs d'une *Enumeration*
- **@MethodSource** permet de fournir un flux de tableau d'objets via une méthode *static*
- **@CsvSource** et **@CsvFileSource** part d'un tableau de donnée en String
- **@ArgumentSource** permet de spécifier une classe externe fournissant un flux de tableau d'objets



Exemple *MethodSource*

```
public class FibonacciTest {  
    private static Stream<Arguments> data() {  
        return Stream.of(  
            arguments(0, 0), arguments(1, 1),  
            arguments(1, 1), arguments(3, 2),  
            arguments(4, 3), arguments(5, 5)  
        );  
    }  
    private int fInput;  
    private int fExpected;  
  
    @ParamterizedTest @MethodSource(names = "data")  
    public test(int input, int expected) {  
        assertEquals(fExpected, Fibonacci.compute(fInput));  
    }  
}
```



Exemple *ArgumentSource*

```
@ParameterizedTest
@ArgumentsSource(MyArgumentsProvider.class)
void testWithArgumentsSource(String argument) {
    assertNotNull(argument);
}

public class MyArgumentsProvider implements ArgumentsProvider {

    @Override
    public Stream<? extends Arguments>
    provideArguments(ExtensionContext context) {
        return Stream.of("apple", "banana").map(Arguments::of);
    }
}
```



Example *@RepeatedTest*

```
@BeforeEach
void beforeEachTest() {
    System.out.println("Before Each Test");
}
@AfterEach
void afterEachTest() {
    System.out.println("After Each Test");
    System.out.println("=====");
}
@RepeatedTest(3)
void repeatedTest(TestInfo testInfo) {
    System.out.println("Executing repeated test");
    assertEquals(2, Math.addExact(1, 1), "1 + 1 should equal 2");
}
```

```
Before Each Test
Executing repeated test
After Each Test
=====
Before Each Test
Executing repeated test
After Each Test
=====
Before Each Test
Executing repeated test
After Each Test
=====
```




TestTemplate

Les ***TestTemplate*** sont des gabarits de génération de cas de test. Ils sont une généralisation des tests paramétrés.

Le gabarit est appelé pour chaque contexte d'appel fournit par une autre classe (Invocation Context Provider).

Avec ce mécanisme, on peut configurer différemment chaque invocation :

- Modification des paramètres des méthodes
- Préparation différente de la classe de test, injection différentes
- Exécuter le tests sous différentes hypothèses (Voir Assumption)
- Avoir des méthodes de call-back différentes



Tests Dynamiques

JUnit5 propose également les tests dynamiques ou les cas de tests sont créés à partir de fabriques annotées par ***@TestFactory***

```
class DynamicTestsDemo {  
  
    @TestFactory  
    Collection<DynamicTest> dynamicTestsFromCollection() {  
        return Arrays.asList(  
            dynamicTest("1st dynamic test", () > assertTrue(true)),  
            dynamicTest("2nd dynamic test", () > assertEquals(4, 2 * 2))  
        );  
    }  
}
```



Assumption (JUnit4)

JUnit4 permet de faire des suppositions sur l'environnement

Si ces suppositions ne sont pas vérifiées au runtime, le test s'arrête mais n'échoue pas

```
import static org.junit.Assume.*  
@Test public void filenameIncludesUsername() {  
    assumeThat(File.separatorChar, is('/'));  
    assertThat(new User("optimus").configFileName(),  
is("configfiles/optimus.cfg"));  
}
```



Assumptions (JUnit5)

Assumptions fournit un ensemble de méthodes utilitaires qui conditionnent l'exécution des tests à des hypothèses.

Si les hypothèses ne sont pas remplies, le test est interrompu.

```
assumeTrue("DEV".equals(System.getenv("ENV")),  
    () -> "Aborting test: not on developer workstation");
```

```
assumingThat("CI".equals(System.getenv("ENV")),  
    () -> {  
        // perform these assertions only on the CI server  
        assertEquals(2, calculator.divide(4, 2));  
    });
```



Activation / Désactivation des tests

L'API d'extension ***ExecutionCondition*** permet de fixer des conditions d'exécution des tests.

Par exemple :

- En fonction de l'OS : *@EnabledOnOs*, *@DisabledOnOs*
- En fonction de la JRE : *@EnabledOnJre*, *@DisabledOnJre*
- En fonction d'une propriété de la JVM :
@EnabledIfSystemProperty, *@DisabledIfSystemProperty*
- D'une variable d'environnement :
@EnabledIfEnvironmentVariable et
@DisabledIfEnvironmentVariable
- D'un script : *@EnabledIf*, *@DisabledIf*



Examples

```
@Test
@EnabledOnOs(MAC)
void onlyOnMacOs() {
}
@Test
@EnabledOnJre({ JAVA_9, JAVA_10 })
void onJava9Or10() {
}
@Test
@EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
void onlyOn64BitArchitectures() {
}
@Test
@EnabledIfEnvironmentVariable(named = "ENV", matches = "stagingserver")
void onlyOnStagingServer() {
}
@Test // Regular expression testing bound system property.
@DisabledIf("/32/.test(systemProperty.get('os.arch'))")
void disabledOn32BitArchitectures() {
    assertFalse(System.getProperty("os.arch").contains("32"));
}
```



Catégories - JUnit4

Les **catégories** permettent de classifier les tests. (Annotations **@Category**)

Ensuite, une exécution particulière permet de limiter les méthodes de tests à exécuter à un ensemble de catégories

C'est une méthode efficace pour différencier les tests unitaires et les tests d'intégration par exemple

Avec JUnit5, l'annotation **@Tag** peut être utilisée



Examples

```
public interface FastTests { /* category marker */ }
public interface SlowTests { /* category marker */ }
```

```
public class A {
    @Test
    public void a() { fail(); }
```

```
    @Category(SlowTests.class)
```

```
    @Test
    public void b() { }
}
```

```
@Category({SlowTests.class, FastTests.class})
```

```
public class B {
    @Test
    public void c() { }
}
```

```
@RunWith(Categories.class)
```

```
@IncludeCategory(SlowTests.class)
```

```
@SuiteClasses( { A.class, B.class } ) // Note that Categories is a kind of Suite
```

```
public class SlowTestSuite {
    // Will run A.b and B.c, but not A.a
}
```

```
@RunWith(Categories.class)
```

```
@IncludeCategory(SlowTests.class)
```

```
@ExcludeCategory(FastTests.class)
```

```
@SuiteClasses( { A.class, B.class } ) // Note that Categories is a kind of Suite
```

```
public class SlowTestSuite {
    // Will run A.b, but not A.a or B.c
}
```




Exemple *@Tag*

```
import org.junit.jupiter.api.Tag;  
import org.junit.jupiter.api.Test;
```

```
@Tag("fast")
```

```
@Tag("model")
```

```
class TaggingDemo {
```

```
    @Test
```

```
    @Tag("taxes")
```

```
    void testingTaxCalculation() {  
    }
```

```
}
```

Les options de la console launcher **--include-tag** et **--exclude-tag** permettent de filtrer les tests à exécuter en fournissant une expression de tag permettant de combiner les tags.
Ex : *(micro | integration) & (product | shipping)*



Règles - JUnit4

Les **règles** (***Rule***) permettent une redéfinition du comportement de chaque méthode de tests.

Les testeurs peuvent utiliser ou surcharger les règles fournies par *JUnit*

Les règles sont remplacées par l'annotation *@ExtendedWith* en JUnit5



Modèle d'extension de JUnit5

Le modèle d'extension de JUnit5 repose dorénavant sur l'interface ***Extension*** avec l'annotation ***@ExtendWith***

- Le moteur *JUnit* enregistre les *Extensions* présentes dans le classpath et les applique lorsqu'il voit une annotation *@ExtendWith* sur une classe ou sur une méthode.

5 principaux types de points d'extension peuvent être utilisés:

- post-traitement de l'instance de test
- exécution de test conditionnel
- Méthodes de Call-back
- résolution des paramètres des méthodes
- gestion des exceptions



Sous-classes d'Extension

L'interface `Extension` est une classe marqueur. Ses principales sous-classes sont :

- ***TestInstancePostProcessor*** : Post-processing après la création de l'instance de test
- ***ExecutionCondition*** : Étendre les conditions d'exécution du test
- ***BeforeAllCallback, BeforeEachCallback, BeforeTestExecutionCallback, AfterTestExecutionCallback, AfterEachCallback, AfterAllCallback*** : Méthodes de call-back du cycle de vie
- ***ParameterResolver*** : Résolution des arguments des méthodes
- ***TestExecutionExceptionHandler*** : Étendre le comportement d'un test lors d'une Exception

Toutes les méthodes définies reçoivent en argument une classe de contexte (*ExtensionContext*) englobant les informations nécessaires.



Exemple

TestInstanceProcessor

```
public class LoggingExtension implements TestInstancePostProcessor {
```

```
    @Override
    public void postProcessTestInstance(Object testInstance,
        ExtensionContext context) throws Exception {
        Logger logger = LogManager.getLogger(testInstance.getClass());
        testInstance.getClass()
            .getMethod("setLogger", Logger.class)
            .invoke(testInstance, logger);
    }
}
```



Méthodes de Callback

```
BeforeAllCallback (1)
  @BeforeAll (2)
    BeforeEachCallback (3)
      @BeforeEach (4)
        BeforeTestExecutionCallback (5)
          @Test (6)
            TestExecutionExceptionHandler (7)
              AfterTestExecutionCallback (8)
                @AfterEach (9)
                  AfterEachCallback (10)
                    @AfterAll (11)
                      AfterAllCallback (12)
```

Lifecycle Callbacks (@ExtendWith(Extension))

User code: methods of the test class



Extension

```
public class TimingExtension implements BeforeTestExecutionCallback,  
AfterTestExecutionCallback {  
  
    private static final Logger logger = Logger.getLogger(TimingExtension.class.getName());  
    private static final String START_TIME = "start time";  
  
    @Override  
    public void beforeTestExecution(ExtensionContext context) throws Exception {  
        getStore(context).put(START_TIME, System.currentTimeMillis());  
    }  
    @Override  
    public void afterTestExecution(ExtensionContext context) throws Exception {  
        Method testMethod = context.getRequiredTestMethod();  
        long startTime = getStore(context).remove(START_TIME, long.class);  
        long duration = System.currentTimeMillis() - startTime;  
        logger.info(() -> String.format("Method [%s] took %s ms.", testMethod.getName(),  
duration));  
    }  
    private Store getStore(ExtensionContext context) {  
        return context.getStore(Namespace.create(getClass(), context.getRequiredTestMethod()));  
    }  
}
```



Exemple *ParameterResolver*

```
public class InvalidPersonParameterResolver implements ParameterResolver {  
    public static Person[] INVALID_PERSONS = {  
        new Person().setId(1L).setLastName("Ad_ams").setFirstName("Jill,"),  
        new Person().setId(2L).setLastName(",Baker").setFirstName(""),  
        new Person().setId(3L).setLastName(null).setFirstName(null),  
    };  
  
    @Override  
    public Object resolveParameter(ParameterContext parameterContext,  
        ExtensionContext extensionContext) throws ParameterResolutionException {  
        return INVALID_PERSONS[new Random().nextInt(INVALID_PERSONS.length)];  
    }  
  
    @Override  
    public boolean supportsParameter(ParameterContext parameterContext,  
        ExtensionContext extensionContext) throws ParameterResolutionException {  
        return parameterContext.getParameter().getType() == Person.class  
    }  
}
```




Usage *ParameterResolver*

```
@DisplayName("Testing Validator When using Invalid data")
@ExtendWith(InvalidPersonParameterResolver.class)
public class InvalidData {

    @RepeatedTest(value = 10)
    @DisplayName("All first names are invalid")
    public void validateFirstName(Person person) {
        assertThrows(
            PersonValidator.ValidationException.class,
            () -> PersonValidator.validateFirstName(person));
    }
}
```



Tests d'intégration

Particularités

Couche HTTP

Couche de persistance

Spring et les tests



Introduction

Les tests unitaires sont rapides et à granularité fine.

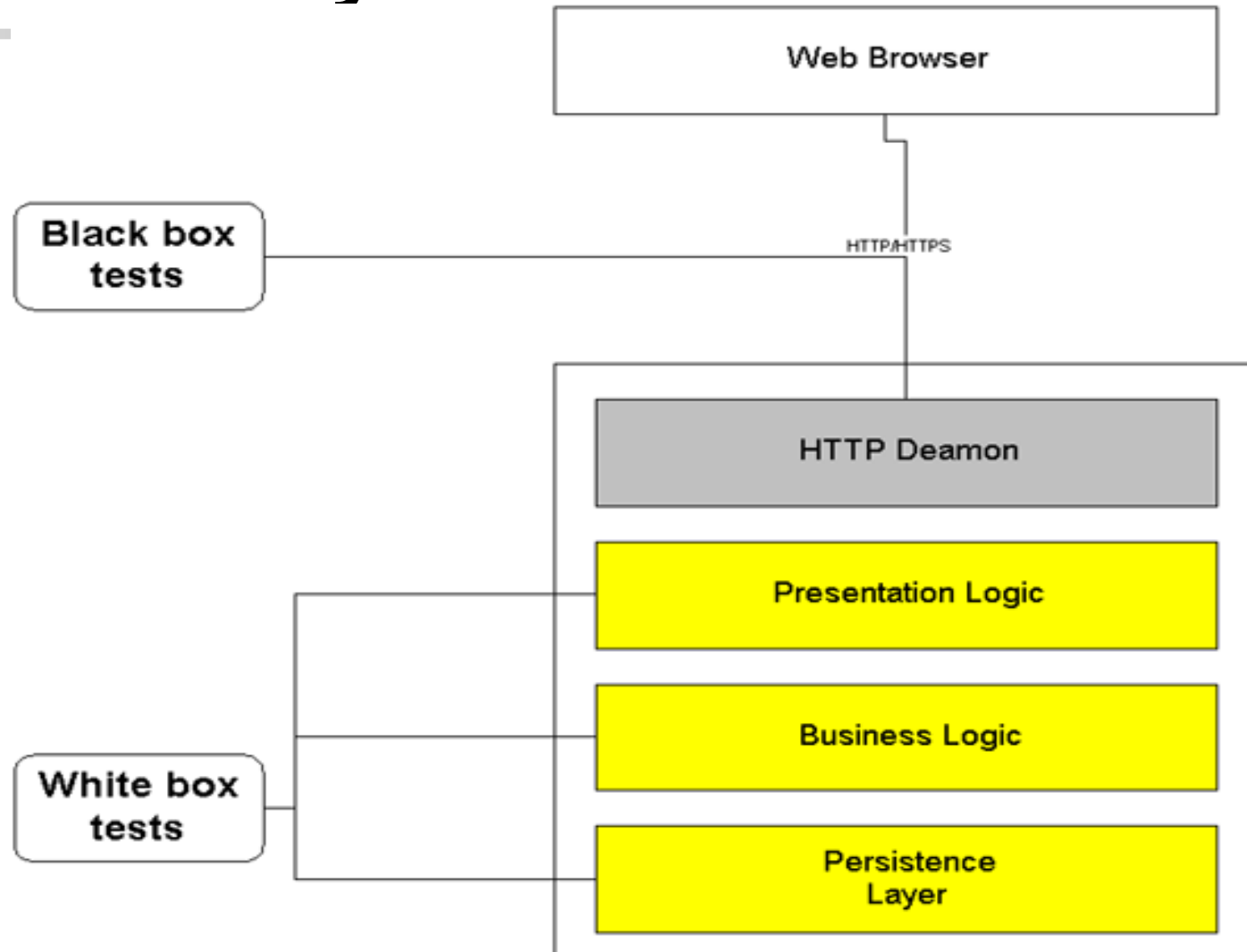
Les tests d'intégration sont plus longs et à plus grosse granularité.

- Ils consistent à tester un sous-ensemble du système (~ une couche)

Ils sont généralement exécutés après une exécution réussie des tests unitaires, donc moins fréquemment.

Les techniques pour raccourcir les temps d'exécution permettent d'améliorer les pipelines CI.

Applications Web Java et API Rest





Particularités

- Les applications sont accédées par des clients Web (Navigateur, Librairies)
- La couche contrôleur nécessite un serveur web :
Serveur JavaEE, Tomcat, Netty
- Les couches basses utilisent :
 - des supports de persistance (JDBC, NoSQL)
 - des Messages Brokers
 - des serveurs annexes (SMTP, LDAP)
 - des APIs Rest tierces (Micro-services)
- Les applications sont développées à l'aide de frameworks (Spring, Quarkus, Containers JavaEE)



Problématiques

Les problématiques classiques des tests d'intégration sont donc :

- Disposer de client permettant d'exécuter des requêtes HTTP
- Disposer de Mock de serveurs (Http, LDAP, API Rest tierce)
- Disposer de support de persistance rapides et facilement initialisables
- Être capable d'initialiser un sous-ensemble des objets du framework (*DispatcherServlet* dans architecture MVC, Pool de connexions pour JDBC, ...)



Tests d'intégration

Particularités

Couche HTTP

Couche de persistance

Spring et les tests



Clients d'APIs Rest

On peut utiliser les APIs de bas niveau pour exécuter ses requêtes HTTP et vérifier les réponses

- *HttpURLConnection* standard.
- Bibliothèque Apache *HttpClient*.
- *RestTemplate*, *TestRestTemplate* Spring

Quelques outils se sont spécialisés sur cette fonctionnalité

- *RestAssured*
- *Karate*



Exemple *RestAssured*

```
@Test public void  
lotto_resource_returns_200_with_expected_id_and_winners() {  
  
    when().  
        get("/lotto/{id}", 5).  
    then().  
        statusCode(200).  
        body("lotto.lottoId", equalTo(5),  
            "lotto.winners.winnerId", hasItems(23, 54));  
}
```



Client Web

Pour tester une interface Web,
différents outils s'intégrant avec *JUnit*

- Selenium :

- Intégration JUnit5 :

- <https://bonigarcia.github.io/selenium-jupiter/>

- HtmlUnit :

- <https://htmlunit.sourceforge.io/>

- Geb

- <https://gebish.org/>



Exemple jupiter-Selenium

@ExtendWith(SeleniumJupiter.class)

```
class ChromeAndFirefoxJupiterTest {
```

```
    @Test
```

```
    void testWithOneChrome(ChromeDriver chromeDriver) { // Use Chrome in this test
    }
```

```
    @Test
```

```
    void testWithFirefox(FirefoxDriver firefoxDriver) { // Use Firefox in this test
```

```
        driver.get("http://www.google.com");
```

```
        WebElement element = driver.findElement(By.name("q"));
```

```
        element.sendKeys("Selenium");
```

```
        element.submit();
```

```
        (new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>() {
```

```
            public Boolean apply(WebDriver d)
```

```
            {return d.getTitle().toLowerCase().startsWith("selenium");}
```

```
        });
```

```
    }
```

```
}
```



Conteneurs Web

Afin de réduire le temps d'exécution et faciliter l'exécution des tests d'intégration, différentes alternatives sont possibles :

- Conteneur Web embarqué
- Démarrage, Déploiement (partiel) de l'application, Arrêt du conteneur lors de l'exécution du test.
- Mock complet du Conteneur



Conteneurs embarqués

Un conteneur embarqué est un serveur Java disponible sous forme de jar qui s'exécute à l'intérieur du runner des tests d'intégrations

La librairie offre les mêmes services qu'un serveur indépendant

Les conteneurs embarqués sont même utilisés en dehors des tests d'intégration (Exemple Spring)

Les plus répandus sont :

- Jetty
- Tomat
- Undertow
- Netty



Autres alternatives

Utilisation de serveurs JavaEE :

- **Cargo** : API de manipulation des serveurs JavaEE
- **Arquillian** : Déploiement partiel sur de vrais serveur (JBoss)

Mock Complet du conteneur :

- **MockMvc** (Spring) : Permet de tester la couche contrôleur sans démarrage de serveur

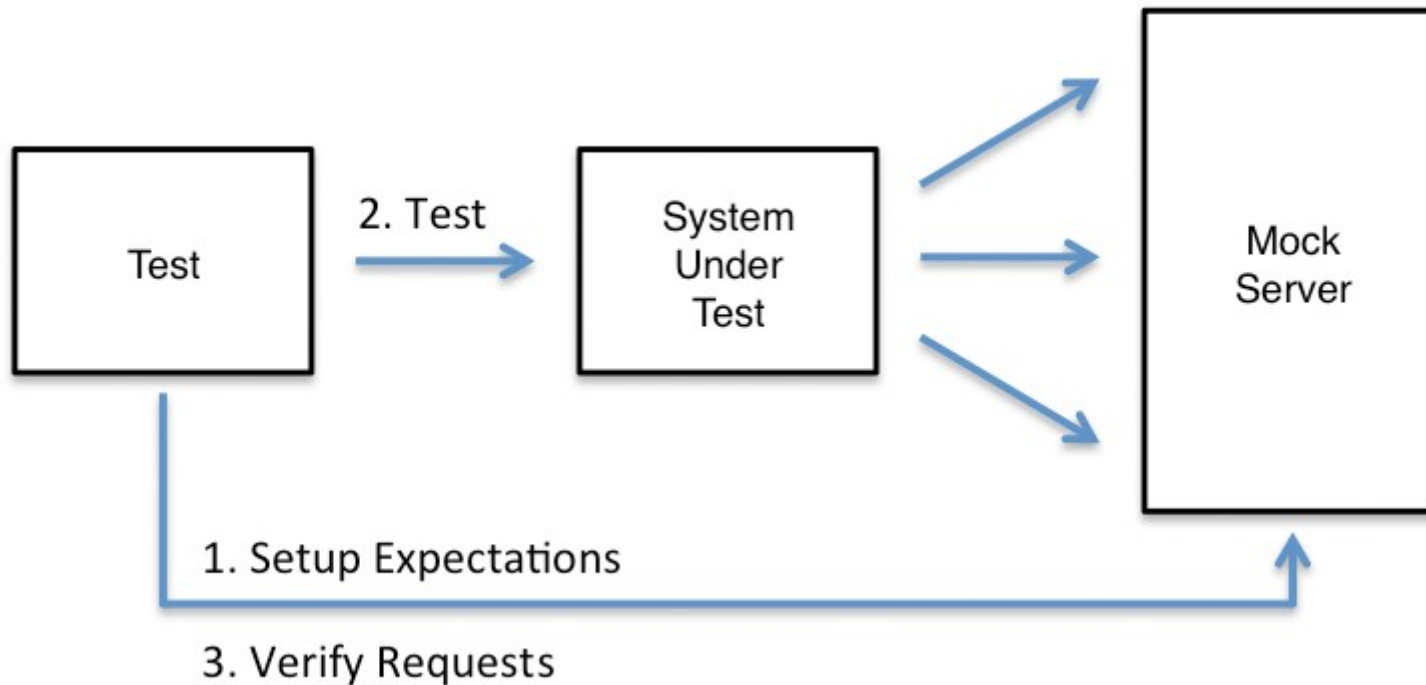


Mock d'API tierces

De nombreuses solutions existent :

- MockServer :
<https://www.mock-server.com/>
- WireMock :
<http://wiremock.org/>
- Solutions en ligne,
Exemple : <https://designer.mocky.io/>

Scénario de test





Exemple *WireMock*

```
@Test
public void exampleTest() {
    stubFor(get(urlEqualTo("/my/resource"))
        .withHeader("Accept", equalTo("text/xml"))
        .willReturn(aResponse()
            .withStatus(200)
            .withHeader("Content-Type", "text/xml")
            .withBody("<response>Some content</response>")));

    Result result = myHttpServiceCallingObject.doSomething();

    assertTrue(result.wasSuccessful());

    verify(postRequestedFor(urlMatching("/my/resource/[a-z0-9]+"))
        .withRequestBody(matching(".*<message>1234</message>.*"))
        .withHeader("Content-Type", notMatching("application/json")));
}
```



Consumer Driven Contract

Spring Cloud Contract est un projet qui permet d'adopter une approche *Consumer Driven Contract*

A partir d'une spécification d'interaction entre un producteur/serveur et consommateur/client, cela permet

- De générer des tests d'acceptation côté producteur
- De créer des mocks serveur pour le client



Example Groovy

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    description "should return even when number input is even"
    request{
        method GET()
        url("/validate/prime-number") {
            queryParameters {
                parameter("number", "2")
            }
        }
    }
    response {
        body("Even")
        status 200
    }
}
```



Tests Clients

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureMockMvc
@AutoConfigureJsonTesters
@AutoConfigureStubRunner(
    stubsMode = StubRunnerProperties.StubsMode.LOCAL,
    ids = "com.baeldung.spring.cloud:spring-cloud-contract-producer::stubs:8090")
public class BasicMathControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void given_WhenPassEvenNumberInQueryParam_ThenReturnEven()
        throws Exception {

        mockMvc.perform(MockMvcRequestBuilders.get("/calculate?number=2")
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().string("Even"));
    }
}
```



Tests d'intégration

Particularités
Couche HTTP
Couche de persistance
Spring et les tests



Introduction

Pour tester la couche de persistance, il est nécessaire de disposer d'une base de données

Les bases de données embarquées peuvent être démarrées par le runner du tests d'intégration et sont rapides (HSQLDB, H2, Derby)

Les objets *Connection*, *DataSource* (JDBC) ou *EntityManager* et *EntityManagerFactory* (JPA) peuvent être initialisés dans des méthodes *@BeforeAll* ou *@BeforeEach*

La problématique principale est de s'assurer que la base est dans un état connu au démarrage du test

- Framework DBUnit, Database Rider
- Framework d'ORM



Exemple JPA (*JUnit4*)

```
public class PredictionAuditIT {
    private PredictionAudit cut;
    private EntityTransaction transaction;

    @Before
    public void initializeDependencies(){
        cut = new PredictionAudit();
        cut.em = Persistence.createEntityManagerFactory("integration").createEntityManager();
        this.transaction = cut.em.getTransaction();
    }
    @Test
    public void savingSuccessfulPrediction(){

        transaction.begin();
        // execute test
        transaction.commit();
        // make assertions

    }
    @Test
    public void savingRolledBackPrediction(){
        final Result expectedResult = Result.BRIGHT;
        Prediction expected = new Prediction(expectedResult, false);
        this.cut.onFailedPrediction(expectedResult);
    }
}
```



DBUnit

DbUnit est une extension *JUnit* qui initialise une base de données dans un état connu entre les exécutions de test

- Il permet d'importer/exporter des données à partir de ressources XML
- Il permet de vérifier que la base est dans l'état attendu

Dépendance :

```
<dependency>
  <groupId>org.dbunit</groupId>
  <artifactId>dbunit</artifactId>
  <version>2.7.0</version>
  <scope>test</scope>
</dependency>
```




Etape 1

Mettre en place des jeux de données en mettant au point manuellement un XML ou en le générant à partir d'une BD.

```
<dataset>
  <TEST_TABLE COL0="row 0 col 0"
              COL1="row 0 col 1"
              COL2="row 0 col 2"/>
  <TEST_TABLE COL1="row 1 col 1"/>
  <SECOND_TABLE COL0="row 0 col 0"
                COL1="row 0 col 1" />
  <EMPTY_TABLE/>
</dataset>
```



Exemple

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Initialisation de 2 tables : CLIENTS et ITEMS -->
<dataset>
  <CLIENTS id='1' first_name='Charles' last_name='Xavier' />
  <ITEMS id='1' title='Grey T-Shirt' price='17.99' produced='2019-03-20' />
  <ITEMS id='2' title='Fitted Hat' price='29.99' produced='2019-03-21' />
  <ITEMS id='3' title='Backpack' price='54.99' produced='2019-03-22' />
  <ITEMS id='4' title='Earrings' price='14.99' produced='2019-03-23' />
  <ITEMS id='5' title='Socks' price='9.99' />
</dataset>
```



Etape 2 : Initialisation de la base

DBUnit inclut *JUnit4*

Pour initialiser la base, il faut écrire un cas de test qui hérite de ***DBTestCase*** ou une de ses sous-classes

- *JdbcBasedDBTestCase*,
 DataSourceBasedDBTestCase,
 JndiBasedDBTestCase, ...



Exemple h2 avec script de création

```
public class DataSourceDBUnitTest extends DataSourceBasedDBTestCase {
    @Override
    protected DataSource getDataSource() {
        JdbcDataSource dataSource = new JdbcDataSource();
        dataSource.setURL(
            "jdbc:h2:mem:default;DB_CLOSE_DELAY=-1;init=runscript from
'classpath:schema.sql'");
        dataSource.setUser("sa");
        dataSource.setPassword("sa");
        return dataSource;
    }

    @Override
    protected IDataset getDataSet() throws Exception {
        return new FlatXmlDataSetBuilder().build(getClass().getClassLoader()
            .getResourceAsStream("data.xml"));
    }
}
```



Options de *setUp* et *tearDown*

Par défaut, DBUnit réinitialisera la base avec les tests avant chaque exécution de méthode de test.

Ce comportement peut être modifié en surchargeant les méthodes ***getSetUpOperation*** et ***getTearDownOperation***

```
@Override
protected DatabaseOperation getSetUpOperation() {
    // Réinitialiser les données
    return DatabaseOperation.REFRESH;
}

@Override
protected DatabaseOperation getTearDownOperation() {
    // Supprimer toutes les données du DataSet
    return DatabaseOperation.DELETE_ALL;
}
```



Etape3 : Méthode de test

Les méthodes de tests peuvent récupérer l'état de la base après l'interaction et le vérifier

```
@Test
public void
givenDataSetEmptySchema_whenDataSetCreated_thenTablesAreEqual()
throws Exception {
    IDataset expectedDataSet = getDataSet();
    ITable expectedTable = expectedDataSet.getTable("CLIENTS");
    IDataset databaseDataSet = getConnection().createDataSet();
    ITable actualTable = databaseDataSet.getTable("CLIENTS");
    new DBUnitAssert().assertEquals(expectedTable, actualTable);
}
```



Assertion avec requête SQL

Les états attendus de la base peuvent être stockés dans d'autres ressources XML et des requêtes peuvent être utilisées pour récupérer l'état réel de la base



Assertion avec requête SQL - Exemple

```
@Test
public void givenDataSet_whenInsert_thenTableHasNewClient() throws Exception {
    try (InputStream is =
        getClass().getClassLoader().getResourceAsStream("dbunit/expected-user.xml")) {
        IDataSet expectedDataSet = new FlatXmlDataSetBuilder().build(is);
        ITable expectedTable = expectedDataSet.getTable("CLIENTS");
        Connection conn = getDataSource().getConnection();

        conn.createStatement()
            .executeUpdate(
                "INSERT INTO CLIENTS (first_name, last_name) VALUES ('John', 'Jansen')");
        ITable actualData = getConnection()
            .createQueryTable(
                "result_name",
                "SELECT * FROM CLIENTS WHERE last_name='Jansen'";

        assertEqualsIgnoreCols(expectedTable, actualData, new String[] { "id" });
    }
}
```




Database Rider

Database Rider apporte des fonctionnalités supplémentaires à *DBUnit*. Citons :

- Extension JUnit5
- Dataset en JSON, CSV, XLS
- Scriptable Data Set : Groovy, Javascript
- Configuration via des annotations
- Expression régulières dans les datasets attendus
- ...

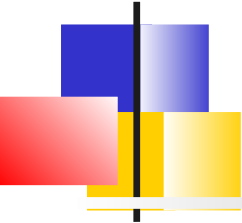


Exemple JUnit5

```
@ExtendWith(DBUnitExtension.class)
@RunWith(JUnitPlatform.class)
@DataSet(cleanBefore = true)
public class DBUnitJUnit5It {

    private ConnectionHolder connectionHolder = () ->
        EntityManagerProvider.instance("junit5-pu").clear().connection();

    @Test
    @DataSet(value = "usersWithTweet.yml")
    public void shouldListUsers() {
        List<User> users = EntityManagerProvider.em().
            createQuery("select u from User u").getResultList();
        assertThat(users).isNotNull().isNotEmpty().hasSize(2);
    }
}
```



ORM - JPA

Les outils d'ORM utilisés avec JPA peuvent généralement être configurés afin qu'ils initialisent la base de données.

Dans le cas d'Hibernate, il suffit de :

- Positionner la propriété ***hibernate.hbm2ddl.auto*** à la valeur ***create*** et ***create-drop***
- Mettre à disposition un fichier ***import.sql*** à la racine du classpath ou plusieurs fichiers indiqués par la propriété ***hibernate.hbm2ddl.import_files***



Exemple via *persistence.xml*

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

  <persistence-unit name="plbsi" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <properties>
        <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:default"/>
        <property name="javax.persistence.jdbc.user" value="sa"/>
        <property name="javax.persistence.jdbc.password" value=""/>
        <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
    </persistence-unit>

</persistence>
```



Tests d'intégration

Particularités
Couche HTTP
Couche de persistance
Spring et les tests



Versions

Spring/SpringBoot/JUnit

SpringBoot 1, Spring 4, JUnit4

Dernière version Septembre 2018

SpringBoot 2, Spring 5, JUnit5

Première version ~2018



Rappels *spring-test*

Spring Test apporte peu pour le test unitaire

- **Mocking** de l'environnement en particulier l'API servlet ou Reactive
- Package **d'utilitaires** : *org.springframework.test.util*

Et beaucoup pour les tests d'intégration (impliquant un *ApplicationContext* Spring) :

- **Cache** du conteneur Spring pour accélérer les tests
- **Injection** des données de test
- Gestion de la **transaction** (roll-back)
- Des classes **utilitaires**
- **Intégration JUnit4 et JUnit5**



Intégration JUnit

- Pour JUnit4 :

@RunWith(SpringJUnit4ClassRunner.class)

ou **@RunWith(SpringRunner.class)**

Permet de charger un contexte Spring, effectuer l'injection de dépendances, etc.

- Pour JUnit5 :

@ExtendWith(SpringExtension.class)

Permet aussi de charger un contexte Spring, effectuer l'injection de dépendances, etc.

Et en plus de l'injection de dépendance pour les méthodes de test, des conditions d'exécution en fonction de la configuration Spring, des annotations supplémentaires pour gérer les transactions



Exemple JUnit5

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = TestConfig.class)
class SimpleTests {

    @Test
    void testMethod() {
        // test logic...
    }
}
```



Spring Boot

L'ajout de ***spring-boot-starter-test*** (dans le scope test), ajoute les dépendances suivantes :

- *Spring Test : Utilitaires Spring pour le Test*
- ***Spring Boot Test*** : Utilitaire liant Spring Test à Spring Boot
- ***Spring Boot Test Autoconfigure*** : Tests auto-configurés
- *JUnit4, AssertJ, Hamcrest* (SB 1.x) ou *JUnit5* (SB 2.X):
- *Mockito* : Un framework pour générer des classes Mock
- *JSONassert* : Une librairie pour les assertions JSON
- *JsonPath* : XPath pour JSON.



Annotations apportées

De nouvelles annotations sont disponibles via le starter :

- **@SpringBootTest** permettant de définir le contexte Spring à utiliser
- Annotations permettant des tests auto-configurés.
Ex : Auto-configuration pour tester une couche en isolation
- Annotation permettant de créer des beans Mockito



@SpringBootTest

L'annotation **@SpringBootTest** remplace la configuration standard de *spring-test* (*@ContextConfiguration*)

Elle crée le contexte Spring utilisé lors des tests en :

- Recherchant la classe principale de l'application en remontant dans la hiérarchie de packages
 - Utilisant l'attribut **classes** qui lui indique la configuration de beans à utiliser
- ```
@SpringBootTest(classes = ForumApp.class)
```



# Attribut *WebEnvironment*

---

L'attribut ***WebEnvironment*** de *@SpringBootTest* permet de préciser l'environnement HTTP désiré pour les tests :

- ***MOCK*** : Fournit un environnement de serveur Mocké. Le conteneur embarqué n'est pas démarré
- ***RANDOM\_PORT*** : Le conteneur est démarré sur un port aléatoire
- ***DEFINED\_PORT*** : Le conteneur est démarré sur un port spécifié
- ***NONE*** : Pas d'environnement SpringMVC



# Mocking des beans

---

L'annotation **@MockBean** définit un bean Mockito

Cela permet de remplacer ou de créer de nouveaux beans

L'annotation peut être utilisée :

- Sur les classes de test
- Sur les champs de la classe de test, dans ce cas le bean mockito est injecté

Les beans Mockito sont automatiquement réinitialisés après chaque test



# Exemple *MockBean*

---

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

 @MockBean
 private RemoteService remoteService;

 @Autowired
 private Reverser reverser;

 @Test
 public void exampleTest() {
 // RemoteService has been injected into the reverser bean
 given(this.remoteService.someCall()).willReturn("mock");
 String reverse = reverser.reverseSomeCall();
 assertThat(reverse).isEqualTo("kcom");
 }
}
```



# Tests auto-configurés

---

Les capacités d'auto-configuration de Spring Boot peuvent ne pas être adaptées au test.

- Lorsque l'on teste la couche contrôleur, on n'a pas envie que SpringBoot nous démarre automatiquement une base de données

Le module *spring-boot-test-autoconfigure* incluent des annotations qui permettent de tester par couche les applications





# Tests JSON

---

Afin de tester si la sérialisation JSON fonctionne correctement, l'annotation **@JsonTest** peut être utilisée.

Elle configure automatiquement l'environnement *Jackson* ou *Gson*

Les classes utilitaires *JacksonTester*, *GsonTester* ou *BasicJsonTester* peuvent être injectées et utilisées, les assertions spécifiques à JSON peuvent être utilisées



# Example

---

```
@RunWith(SpringRunner.class)
@JsonTest
public class MyJsonTests {

 @Autowired
 private JacksonTester<VehicleDetails> json;

 @Test
 public void testSerialize() throws Exception {
 VehicleDetails details = new VehicleDetails("Honda", "Civic");
 // Assert against a `.json` file in the same package as the test
 assertThat(this.json.write(details)).isEqualToJson("expected.json");
 // Or use JSON path based assertions
 assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");
 assertThat(this.json.write(details).extractingJsonPathStringValue("@.make")
 .isEqualTo("Honda");
 }

 @Test
 public void testDeserialize() throws Exception {
 String content = "{\"make\":\"Ford\",\"model\":\"Focus\"}";
 assertThat(this.json.parse(content))
 .isEqualTo(new VehicleDetails("Ford", "Focus"));
 assertThat(this.json.parseObject(content).getMake()).isEqualTo("Ford");
 }
}
```



# Tests de Spring MVC

---

L'annotation **@WebMvcTest** configure l'infrastructure Spring MVC et limite le scan aux annotations de Spring MVC

Elle configure également *MockMvc* qui permet de se passer d'un serveur Http complet

Pour les tests *Selenium* ou *HtmlUnit*, un client Web est également fourni



# Example

---

```
@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyControllerTests {

 @Autowired
 private MockMvc mvc;

 @MockBean
 private UserVehicleService userVehicleService;

 @Test
 public void testExample() throws Exception {
 given(this.userVehicleService.getVehicleDetails("sboot"))
 .willReturn(new VehicleDetails("Honda", "Civic"));
 this.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
 .andExpect(status().isOk()).andExpect(content().string("Honda
Civic"));
 }
}
```



# Example (2)

---

```
@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyHtmlUnitTests {

 // WebClient is auto-configured thanks to HtmlUnit
 @Autowired
 private WebClient webClient;

 @MockBean
 private UserVehicleService userVehicleService;

 @Test
 public void testExample() throws Exception {
 given(this.userVehicleService.getVehicleDetails("sboot"))
 .willReturn(new VehicleDetails("Honda", "Civic"));
 HtmlPage page = this.webClient.getPage("/sboot/vehicle.html");
 assertThat(page.getBody().getTextContent()).isEqualTo("Honda Civic");
 }
}
```



# Tests JPA

---

**@DataJpaTest** configure une base de donnée mémoire, scanne les *@Entity* et configure les Repository JPA

Les tests sont transactionnels et un rollback est effectué à la fin du test

- Possibilité de changer ce comportement par *@Transactional*

Un *TestEntityManager* peut être injecté ainsi qu'un *JdbcTemplate*



# Exemple

---

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class ExampleRepositoryTests {

 @Autowired
 private TestEntityManager entityManager;

 @Autowired
 private UserRepository repository;

 @Test
 public void testExample() throws Exception {
 this.entityManager.persist(new User("sboot", "1234"));
 User user = this.repository.findByUsername("sboot");
 assertThat(user.getUsername()).isEqualTo("sboot");
 assertThat(user.getVin()).isEqualTo("1234");
 }
}
```



# Autres tests auto-configurés

---

**@WebFluxTest** : Test des contrôleurs Spring Webflux

**@JdbcTest** : Seulement la *datasource* et *jdbcTemplate*.

**@JooqTest** : Configure un *DSLContext*.

**@DataMongoTest** : Configure une base mémoire Mongo, *MongoTemplate*, scanne les classes *@Document* et configure les MongoDB repositories.

**@DataRedisTest** : Test des applications Redis applications.

**@DataLdapTest** : Serveur embarqué LDAP (if available), *LdapTemplate*, Classes *@Entry* et LDAP repositories

**@RestClientTest** : Test des clients REST. Jackson, GSON, ... + *RestTemplateBuilder*, et du support pour *MockRestServiceServer*.





# Test et sécurité

---

Spring propose plusieurs annotations pour exécuter les tests d'une application sécurisée par SpringSecurity.

```
<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-test</artifactId>
<scope>test</scope>
</dependency>
```

**@WithMockUser** : Le test est exécuté avec un utilisateur dont on peut préciser les détails (login, password, rôles)

**@WithAnonymousUser** : Annote une méthode

**@WithUserDetails("aLogin")** : Le test est exécuté avec l'utilisateur chargé par *UserDetailsService*

**@WithSecurityContext** : Qui permet de créer le SecurityContext que l'on veut



# Automatisation

---

## **Les tests avec Maven**

Les tests dans une pipeline CI/CD



# Introduction

---

Maven intègre les tests dans son cycle de vie

Avec la version 2.2, il fournit un support natif pour l'exécution des tests JUnit avec les plugins ***Surefire*** et ***FailSafe***

- *Surefire* est dédié à l'exécution des tests unitaires
- *FailSafe* aux tests d'intégration



# Configuration JUnit4

---

```
<build>
 <plugins><plugin>
 <artifactId>maven-surefire-plugin</artifactId>
 <version>2.22.2</version>
 </plugin><plugin>
 <artifactId>maven-failsafe-plugin</artifactId>
 <version>2.22.2</version>
 </plugin></plugins>
</build>
<!-- ... -->
<dependencies>
 <dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>4.13</version>
 <scope>test</scope>
 </dependency>
 <dependency>
 <groupId>org.junit.vintage</groupId>
 <artifactId>junit-vintage-engine</artifactId>
 <version>5.7.0</version>
 <scope>test</scope>
 </dependency>
</dependencies>
```



# Configuration JUnit5

---

```
<build>
 <plugins><plugin>
 <artifactId>maven-surefire-plugin</artifactId>
 <version>2.22.2</version>
 </plugin><plugin>
 <artifactId>maven-failsafe-plugin</artifactId>
 <version>2.22.2</version>
 </plugin></plugins>
</build>
<!-- ... -->
<dependencies>
 <dependency>
 <groupId>org.junit.jupiter</groupId>
 <artifactId>junit-jupiter-api</artifactId>
 <version>5.7.0</version>
 <scope>test</scope>
 </dependency>
 <dependency>
 <groupId>org.junit.jupiter</groupId>
 <artifactId>junit-jupiter-engine</artifactId>
 <version>5.7.0</version>
 <scope>test</scope>
 </dependency>
</dependencies>
```



# Cycle complet Maven

---

*validate* : Valider que le projet est correct

*initialize* : initialiser l'état du build.

*generate-sources* : Générer des sources

*process-sources* : Traite les sources du projet.

*generate-resources* : Génère des ressources .

*process-resources* : Traite ls ressources.

*compile* : compile les sources.

*process-classes* : Traite les classes.

***generate-test-sources*** : Génère le code source de test.

***process-test-sources*** : Traite les sources.

***generate-test-resources*** : Crée des ressources pour le test.

***process-test-resources*** : Traite les ressources de test.

***test-compile*** : Compilation du code de test

***process-test-classes*** : Traitement des .class de test

***test*** : Exécution des tests.

*prepare-package* : Préparation au packaging

*package* : Packaging

***pre-integration-test*** : Actions avant les tests d'intégration

***integration-test*** : Exécutions des tests d'intégration

***post-integration-test*** : Actions après les tests d'intégration

***verify*** : Phase terminant les tests d'intégration .

*install* : Installation dans le repository local

*deploy* : Déploiement ou release s.



# Plugin *Surefire*

---

Le plugin *Surefire* a un seul objectif : ***surefire:test*** qui est associé par défaut à la phase test de Maven

L'objectif génère des rapports de tests en 2 formats dans le répertoire *target/surefire-reports*

- Plain text (\*.txt)
- XML (\*.xml)



# Paramètres *Surefire*

---

Paramètres requis :

- ***testSourceDirectory*** : Répertoire des sources de test

Paramètres optionnels :

- ***disableXmlReport*** : Ne pas générer les rapports XML
- ***includes/excludes*** : Inclusion/Exclusion des classes de tests
- ***groups/excludedGroups*** : Groupes ou tags à inclure/exclure
- ***parallel*** : Utilisation de threads
- ***forkCount, forkMode*** : Nbre de threads à démarrer
- ***printSummary*** : Résumé pour les suites de test
- ***reportFormat, reportDirectory*** : Options pour le reporting
- ***testFailureIgnore*** : Continue le cycle Maven même si un test a échoué





# Distinction entre tests unitaires et d'intégration

---

Il n'y a pas d'approche standard pour différencier les tests unitaires des tests d'intégration. Différentes techniques peuvent être utilisées :

1. Convention de nommage: Tous les tests d'intégration peuvent se terminer par "*IntegrationTest*", ou être placés dans un package particulier.
2. Avec JUnit5, les tests peuvent être différenciés par des tags

Il faut en plus configurer FailSafe afin qu'il exécute les objectifs voulus aux bonnes phases



# Plugin FailSafe

---

*FailSafe* s'appuie sur *Surefire*.

- Cependant si un test échoue, à la différence de *Surefire* il permet n'arrête pas le cycle Maven et permet d'exécuter la phase *post-integration* dédié à l'arrêt des serveurs (bd, serveur web) utilisés durant les tests

Il comporte 2 objectifs :

- ***failsafe:integration-test*** : Exécution des tests d'intégration avec *Surefire*. Attaché par défaut à la phase *integration-test*
- ***failsafe:verify*** : Vérification de l'exécution des tests d'intégration. Attaché par défaut à la phase *verify*

Après configuration de *FailSafe*, les tests d'intégration doivent donc se lancer avec :

```
mvn verify
```



# Convention de nommage

---

```
<plugin>
 <artifactId>maven-surefire-plugin</artifactId>
 <executions><execution>
 <goals><goal>test</goal></goals>
 <configuration>
 <excludes><exclude>/**/*.IntegrationTest.java</exclude></excludes>
 </configuration>
 </execution></executions>
</plugin>
<plugin>
 <artifactId>maven-failsafe-plugin</artifactId>
 <executions><execution>
 <goals> <goal>integration-test</goal> <goal>verify</goal> </goals>
 <configuration>
 <includes><include>/**/*.IntegrationTest.java</include></includes>
 </configuration>
 </execution></executions>
</plugin>
```



# Séparation selon les tags

---

```
<build>
 <plugins>
 <plugin>
 <artifactId>maven-surefire-plugin</artifactId>
 <version>2.22.2</version>
 <configuration>
 <groups>acceptance | !feature-a</groups>
 <excludedGroups>integration, regression</excludedGroups>
 </configuration>
 </plugin>
 </plugins>
</build>
```



# Automatisation

---

Les tests avec Maven ou Gradle  
**Les tests dans une pipeline CI/CD**



# Introduction

---

Les tests automatisés sont une obligation si l'on veut mettre en place l'intégration continue.

Un des principes de base de la CI est qu'un build doit être *vérifiable*.

- Cela veut dire être capable objectivement de déterminer si un build peut passer à l'étape suivante  
=> le seul moyen objectif est d'utiliser des tests automatisés



# Pipeline DevOps

---

Les pipelines DevOps sont décrits par des fichiers de script commité dans le SCM (*Jenkinsfile*, *.gitlab-ci.yml*, *travis.yml*, ...)

Les problématiques à résoudre pour y inclure des tests automatisés :

- Comment démarrer les tests ?  
En général en s'appuyant sur l'outil de build du projet
- Comment publier les résultats ?  
En s'équipant de plugin spécialisés (Jenkins), en archivant les résultats,
- Comment distinguer les tests d'intégration, unitaires, d'acceptation ? Tags et configuration du build
- Comment mettre en place les ressources nécessaires aux tests d'intégration ? Serveurs embarqués, Démarrage de serveurs externe : Side-car pattern, Utilisation de serveur d'intégration



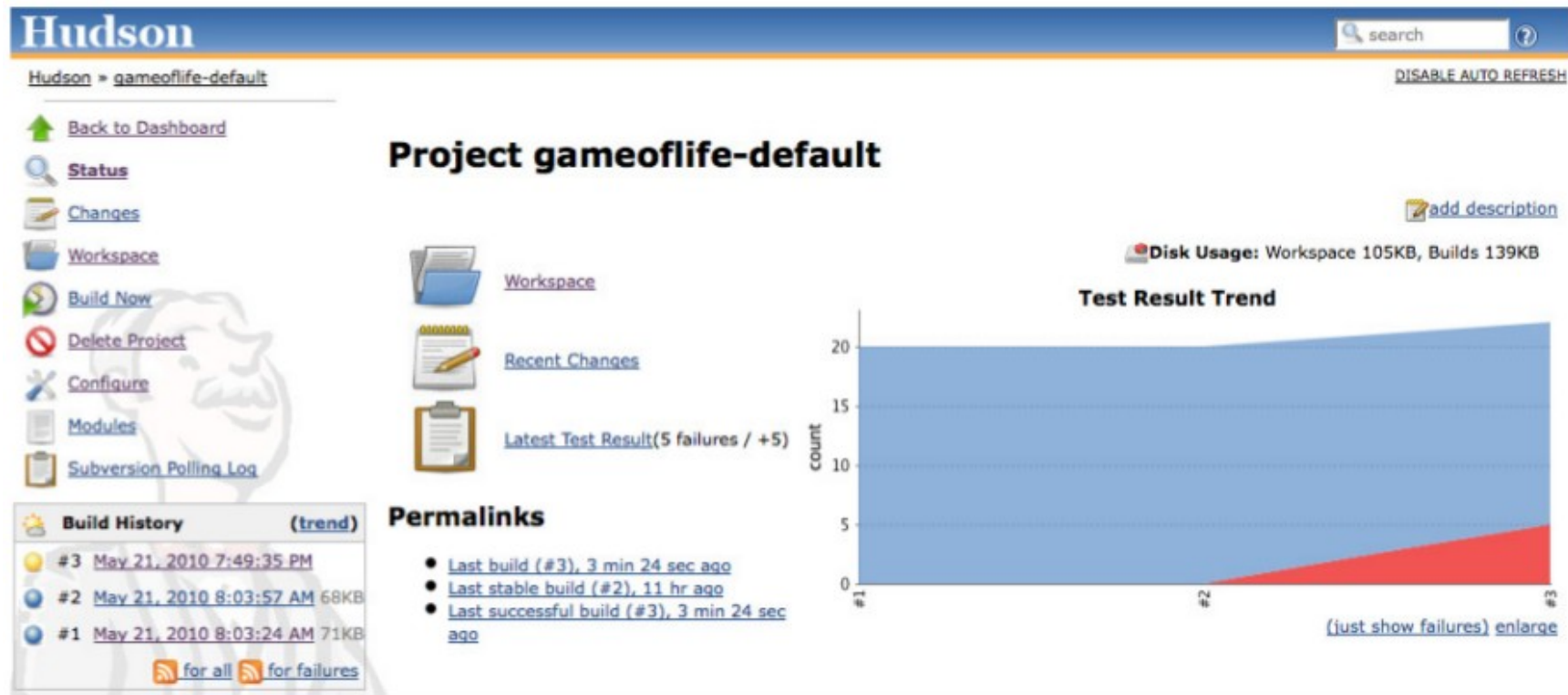
# Exemple Jenkinsfile

---

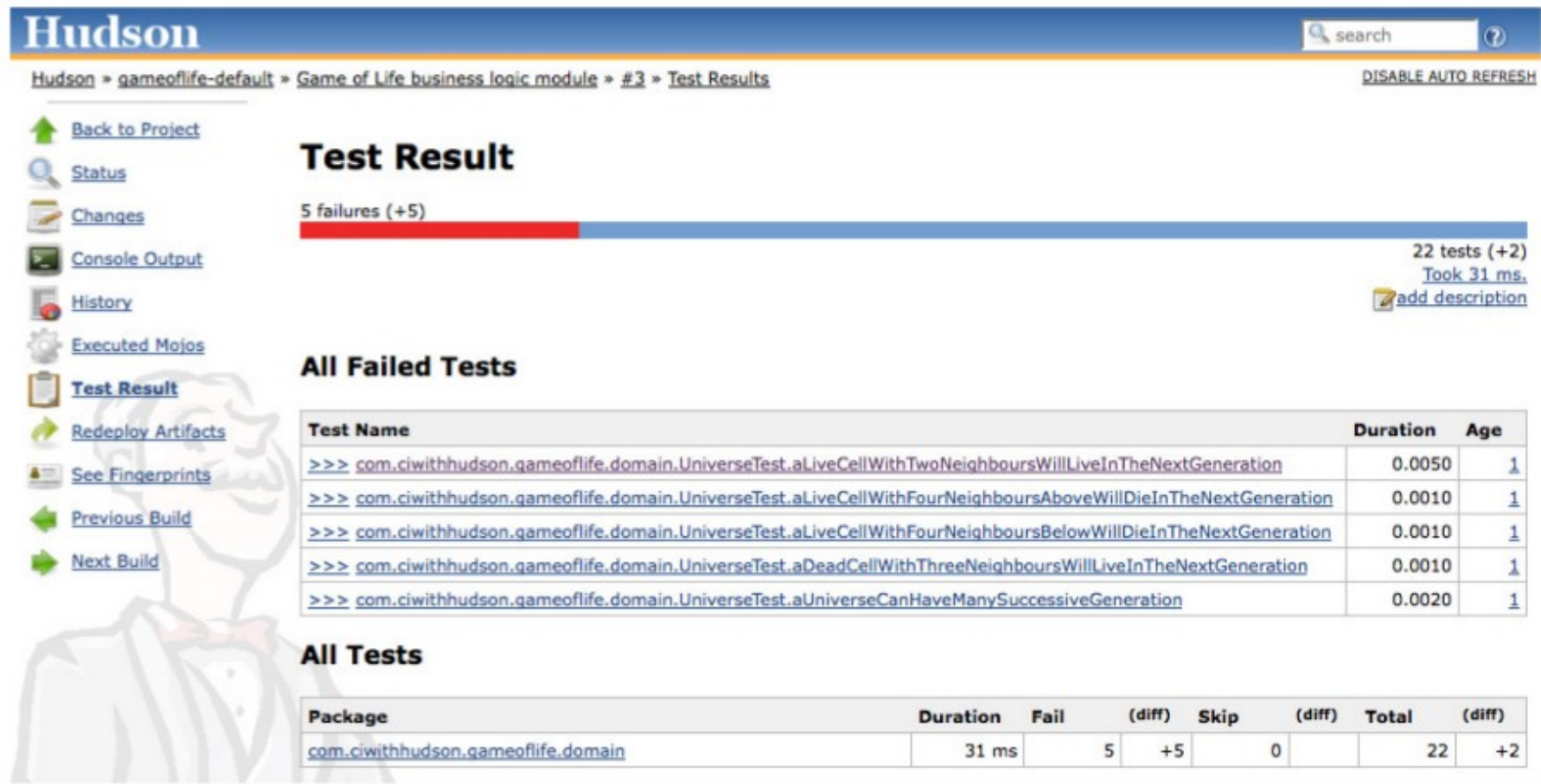
```
stage('Build et Tests unitaires') {
 agent any
 steps {
 echo 'Building and Unit tests'
 sh './mvnw -Dmaven.test.failure.ignore=true clean test'
 }
 post {
 always {
 junit '**/target/surefire-reports/*.xml'
 }
 failure {
 mail body: 'It is very bad', from: 'admin@jenkins.fr',
 subject: 'Compilation or Unit test broken',
 to: 'david.thibau@gmail.com'
 }
 }
}
```



# Graphe de Tendance



# Derniers tests



The screenshot shows the Hudson web interface. The breadcrumb trail is 'Hudson > gameoflife-default > Game of Life business logic module > #3 > Test Results'. The left sidebar contains links: 'Back to Project', 'Status', 'Changes', 'Console Output', 'History', 'Executed Mojos', 'Test Result' (highlighted), 'Redeploy Artifacts', 'See Fingerprints', 'Previous Build', and 'Next Build'. The main content area is titled 'Test Result' and shows '5 failures (+5)' with a red progress bar. To the right, it says '22 tests (+2)', 'Took 31 ms.', and has a link to 'add description'. Below this is the 'All Failed Tests' section with a table of failed tests. At the bottom is the 'All Tests' section with a summary table.

## Test Result

5 failures (+5)

22 tests (+2)  
Took 31 ms.  
[add description](#)

### All Failed Tests

Test Name	Duration	Age
>>> <a href="#">com.ciwithhudson.gameoflife.domain.UniverseTest.aLiveCellWithTwoNeighboursWillLiveInTheNextGeneration</a>	0.0050	<a href="#">1</a>
>>> <a href="#">com.ciwithhudson.gameoflife.domain.UniverseTest.aLiveCellWithFourNeighboursAboveWillDieInTheNextGeneration</a>	0.0010	<a href="#">1</a>
>>> <a href="#">com.ciwithhudson.gameoflife.domain.UniverseTest.aLiveCellWithFourNeighboursBelowWillDieInTheNextGeneration</a>	0.0010	<a href="#">1</a>
>>> <a href="#">com.ciwithhudson.gameoflife.domain.UniverseTest.aDeadCellWithThreeNeighboursWillLiveInTheNextGeneration</a>	0.0010	<a href="#">1</a>
>>> <a href="#">com.ciwithhudson.gameoflife.domain.UniverseTest.aUniverseCanHaveManySuccessiveGeneration</a>	0.0020	<a href="#">1</a>

### All Tests

Package	Duration	Fail	(diff)	Skip	(diff)	Total	(diff)
<a href="#">com.ciwithhudson.gameoflife.domain</a>	31 ms	5	+5	0		22	+2

# Détail

## Hudson

search


Hudson » gameoflife-default » gameoflife-core » #13 » Test Results » com.wakaleo.gameoflife.domain » GameOfLifeTest » aDeadCellWithNoNeighboursShouldRemainDeadInTheNextGeneration

[DISABLE AUTO REFRESH](#)

[Back to Project](#)  
[Status](#)  
[Changes](#)  
[Console Output](#)  
[History](#)  
[Executed Mojos](#)  
[Test Result](#)  
[Redeploy Artifacts](#)  
[See Fingerprints](#)  
[Previous Build](#)

### Regression

com.wakaleo.gameoflife.domain.GameOfLifeTest.aDeadCellWithNoNeighboursShouldRemainDeadInTheNextGeneration (from GameOfLifeTest)

Failing for the past 1 build (Since  #13 )  
[Took 1 ms.](#)  
[add description](#)

#### Error Message

Expected: is "...\\n...\\n...\\n" got: ".\*\\n\*.\*\\n\*.\*\\n"

#### Stacktrace

```
java.lang.AssertionError:
Expected: is "...\\n...\\n...\\n"
got: ".*\\n*.*\\n*.*\\n"

 at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:21)
 at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:8)
 at com.wakaleo.gameoflife.domain.GameOfLifeTest.aDeadCellWithNoNeighboursShouldRemainDeadInTheNextGeneration(GameOfLifeTest.java:28)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
 at java.lang.reflect.Method.invoke(Method.java:597)
 at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:44)
 at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:15)
 at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:41)
 at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:20)
 at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:76)
 at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:50)
 at org.junit.runners.ParentRunner$3.run(ParentRunner.java:193)
 at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:52)
 at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:191)
```



# Automatisation

---

Les tests avec Maven ou Gradle  
Les tests dans une pipeline CI/CD  
**Couverture des tests**



# Introduction

---

La **couverture de code** est une métrique logicielle utilisée pour mesurer le nombre de lignes de code exécutées lors de tests automatisés.

Dans le monde Java, l'outil le plus couramment utilisé est **JaCoCo**

*JaCoCo* fournit un plugin Maven

Il existe également un plugin Eclipse  
*EclEmma*



# Plugin Maven

---

Le plugin Maven fournit 2 principaux objectifs :

- ***prepare-agent*** : Initialisation de JaCoCo, attachement d'un agent à la JVM exécutant le build.  
S'effectue dans la phase initialize
- ***report*** : Génération du rapport de couverture.  
S'effectue après les tests, dans la phase prepare-package par exemple



# Pom.xml

---

```
<plugin>
 <groupId>org.jacoco</groupId>
 <artifactId>jacoco-maven-plugin</artifactId>
 <version>0.7.7.201606060606</version>
 <executions>
 <execution>
 <goals>
 <goal>prepare-agent</goal>
 </goals>
 </execution>
 <execution>
 <id>report</id>
 <phase>prepare-package</phase>
 <goals>
 <goal>report</goal>
 </goals>
 </execution>
 </executions>
</plugin>
```



# Mécanisme

---

Lors de l'exécution des avec JUnit JaCoCo crée un rapport de couverture :




***target/jacoco.exec***

- Ce format est compris par certains outils comme Sonar Qube.

L'objectif ***jacoco:report*** génère des rapports de couverture de code dans plusieurs formats lisibles - par ex. HTML, CSV et XML.



# Résultats

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
 <a href="#">Palindrome</a>		21%		17%	3	5	4	7	0	2	0	1
Total	30 of 38	21%	5 of 6	17%	3	5	4	7	0	2	0	1

```
1. package com.baeldung.testing.jacoco;
2.
3. public class Palindrome {
4.
5. public boolean isPalindrome(String inputString) {
6. if (inputString.length() == 0) {
7. return true;
8. } else {
9. char firstChar = inputString.charAt(0);
10. char lastChar = inputString.charAt(inputString.length() - 1);
11. String mid = inputString.substring(1, inputString.length() - 1);
12. return (firstChar == lastChar) && isPalindrome(mid);
13. }
14. }
15. }
```



# Analyse

---

Le rapport montre

- le pourcentage d'instructions (bytecode) couvert par les tests,
- le pourcentage de branches couvertes par les tests
- Le nombre de chemins linéairement indépendants couverts par les tests (la complexité cyclomatique)

Une aide visuelle à l'analyse est fournie par des diamants de couleurs pour les branches et des couleurs d'arrière-plan pour les lignes:

- Rouge : aucune branche/lignes n'a été couverte.
- Jaune : Couverture partielle .
- Vert : Couverture totale.



# Seuil minimal

---

*JaCoCo* via l'objectif ***check*** offre un moyen simple de déclarer les exigences minimales à respecter

Si ces exigences ne sont pas respectées, le build échoue

L'objectif *check* est associé par défaut à la phase *verify* de Maven.



# Configuration seuil

---

```
<execution>
 <id>jacoco-check</id>
 <goals>
 <goal>check</goal>
 </goals>
 <configuration>
 <rules>
 <rule>
 <element>PACKAGE</element>
 <limits>
 <limit>
 <counter>LINE</counter>
 <value>COVEREDRATIO</value>
 <minimum>0.50</minimum>
 </limit>
 </limits>
 </rule>
 </rules>
 </configuration>
</execution>
```