Tests java

(JUnit , Mockito, ...)

Table des matières

I - Tests de composants logiciels

1. Types de tests et doublures

1.1. Différents types de tests

Classification des tests par niveaux

Niveau de Tests	Caractéristiques
Test unitaire de composant	Tester de façon isolée un seul composant logiciel
Test d'intégration technique	Tester un assemblage de composants au sein d'une application et vérifier "bonnes communications , pas d'incompatibilités, pas d'effets de bords , …"
Test "système fonctionnel" *(test d'intégration fonctionnel , VABF)*	Tester la validité fonctionnelle de tout un sous système informatique (Database + ESB + applications + ...)
Test d'acceptation (UAT = User Acceptance Test) alias "*recette*"	Acceptation du logiciel dans le contexte "client" ou "moa" .

Classification selon de niveau d'accessibilité :

- Test "**boîte noire**" (sans connaître la structure interne du composant à tester)
- Test "**boîte blanche**" (en connaissant la **structure** interne et en vérifiant certaines caractéristiques internes : intégrité , ...)

Classification selon une caractéristique (attribut de qualité , ...) :

- tests de **montée en charge** et de **performance** : temps de réponse corrects ? Combien de clients simultanés au maximum ? …
- test **fonctionnel** : fonctionnalités demandées sont bien supportées, ...
- test de **robustesse** : valider la stabilité et la fiabilité du logiciel dans le temps.
- test de **vulnérabilité** : vérification de sécurité du logiciel.
- ….

Autres catégories de tests ou synonymes :

- **Revue de code** (souvent automatisée) et rapport qualimétrique (code bien écrit ? respectant certaines normes/conventions ? , pas trop de copier/coller ? , …)
- **Tests de non-régression** : pour vérifier que des modifications n'ont pas altérées le fonctionnent de l'application.
- **Tests IHM** (charte graphique respectée ? Navigations ? …)
- **Tests d'adaptation à différentes configurations ou contextes** selon système hôte (linux, windows , …) , résolution écran, ...
- **Tests fonctionnels de bout en bout** (des processus métier au sein d'un sous système informatique).
  VABF = (Vérification d'aptitude au Bon Fonctionnement)
- **Tests d'exploitabilité** (dans un contexte de (pré-)production) Qualité de services (performances , sécurité, …) correctes ?
- **VSR** = **V**érification en **S**ervice **R**égulier (surveillance/pilotage).

1.2. Tests unitaires

4 phases d'un test unitaire

1) Initialisation (méthode *setUp* ou *init*): définition d'un environnement de test complètement reproductible (une "*fixture*" avec par exemple une instance de composant bien initialisée et un éventuel jeux de données)

2) **Exécution du code à tester**: appel d'une méthode avec certains paramètres d'entrée bien choisis (valides ou invalides, ...).

3) **Vérification (assertions)**: comparaison du résultat obtenu avec la valeur de réponse attendue. Ces assertions définissent le résultat du test: SUCCÈS ou ÉCHEC .

4) **Désactivation/terminaison** (méthode tearDown ou ...): désinstallation des "fixtures" pour retrouver l'état initial du système, dans le but de na pas polluer/perturber les tests suivants. Tous les tests doivent idéalement être indépendants et reproductibles.

NB: *Selon le degré de sophistication/complexité du test unitaire, les phases 1 et 4 pourront être triviales ou très évoluées (ex : avec dbUnit).*

Utilités/objectifs des tests unitaires

- **Bien appréhender/formaliser le contrat technico/fonctionnel d'un composant** logiciel (en lisant le code d'un test bien écrit , on comprend le comportement attendu des opérations/méthodes du composant à tester).

- **Trouver les erreurs rapidement et simplifier la maintenance** :
  Une fois le test unitaire écrit, on peut le relancer automatiquement sans effort des milliers de fois pour ***vérifier l'absence de régression*** et pour ***localiser rapidement une erreur*** en cas de problème .

- **Développer consciencieusement** (en testant naturellement l'absence de bug et le bon fonctionnement) au fur et a mesure de la programmation.

- S'assurer que tout le code écrit (et prévu pour être appelé/invoqué) soit **couvert** par un nombre suffisant de tests unitaires.

- Ne pas se contenter de la boutade "***tester c'est douter***" !

Stratégie habituelle de test (Test Driven Development)

1) **définir la structure du composant à tester** (exemple: *modèle UML* , ***interface java*** , …) et un éventuel embryon d'implémentation.

2) **écrire la classe de test** (en s'appuyant sur une structure connue et définie du composant à tester mais sur une implémentation qui n'est pas encore opérationnelle) .

3) **lancer une première fois les tests unitaires** et vérifier normalement que "sans code d'implémentation finalisé" les tests remontent bien des "échecs" .

4) **écrire le code d'implémentation du composant à tester**. Relancer les tests qui devraient normalement réussir.

5) coder et tester des **variantes** au niveau des tests (paramètres d'entrées volontairement erronés , vérification de remontée d'exception , …)

6) poursuivre de manière incrémentale et itérative (nouvelle méthode,...)

Bonnes pratiques sur tests unitaires

- Tester essentiellement les méthodes publiques (pas les méthodes privées)

- Factoriser (lorsque c'est possible) le code de quelques tests (en appelant des sous méthodes de la classe de test , en héritant de classes utilitaires sur les Tests , ..)

- En cas de test qui ne passe plus , d'abord remettre en question le code du composant à tester , puis ensuite le code du test lui même (qui peut également être bogué ou bien trop simpliste).

- Utiliser éventuellement des "mocks" (composants en arrière plan simulés) pour de multiples bonnes raisons qui doivent cependant se justifier selon le contexte . Trop de "mocks" ralentissent quelquefois le développement et rendre le code des tests moins lisible .

1.3. Tests d'intégration

Portées et contraintes des tests d'intégration

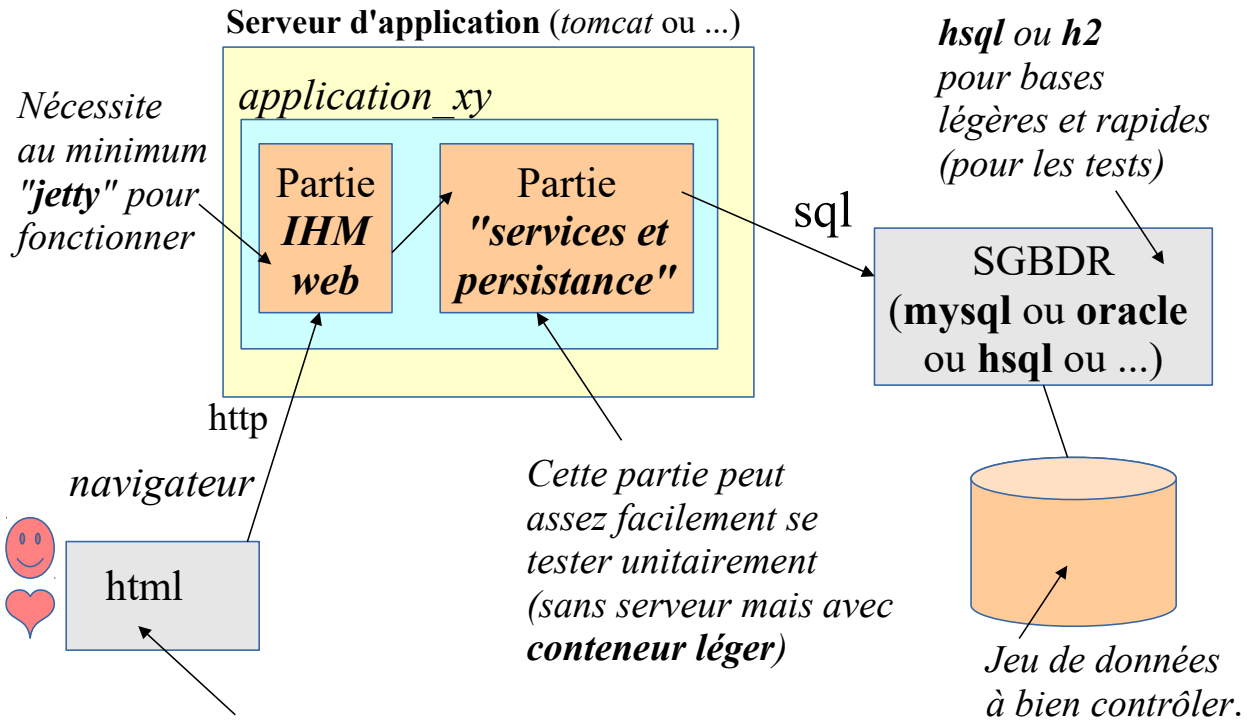
Un **test d'intégration technique porte** généralement **sur une application entière** qui **est quelquefois elle même prise en charge par un serveur d'application** *(ex: serveur JEE tomcat , jboss , websphere , ...)*.

Une application informatique est généralement composée de plusieurs sous modules complémentaires (partie "*IHM/web*" , partie "*services en arrière plan*" , ….) et chaque module correspond souvent à un sous projet séparé (géré par "maven" ou autre) .

Un **test d'intégration** nécessite souvent d'**automatiser** les points suivants :

- construction/compilation et assemblage/packaging des modules de l'application
- déploiement et démarrage de l'application au sein d'un serveur d'application (ex : tomcat ou …)
- Tester la partie web (qui elle même appelle un service qui lui même envoie des requêtes à la base de données)
- Arrêter l'application (et éventuellement le serveur)

Illustrations de certains contextes récurrents



1.4. Doublures et Mocks

Intérêts de "doublures" pour tests (ou ...)

- Le terme ***doublure*** se comprend bien dans le cadre d'un "*crash test automobile*" : le passager n'est pas une vraie personne humaine mais un mannequin ressemblant *(morphologie proche , ...)* et bourré de "*capteurs*" pour mesurer l'impact du choc .

- dans le cadre d'un test logiciel , certaines parties peuvent être simulées via des doublures lorsqu'elles sont pas encore prêtes , inaccessibles ou trop lentes.

- on peut simplifier énormément de code d'une doublure par rapport à celui d'un vrai composant fonctionnel tout en rendant son comportement identique : on peut ainsi simplifier l'initialisation des jeux de données et coder le lest plus rapidement.

- on peut simuler/forcer des événements exceptionnels (déconnexions , service inaccessible, …) pour vérifier les remontées d'exceptions .

Principaux types de "doublures"

- **dummy** (*fantôme, bouffon*) : objets avec implémentations "vides" .
- **stub** (*bouchon*) : classe alternative (codée très rapidement) qui renvoie en dur une valeur pour chaque méthode invoquée
- **fake** (*substitut, simulateur*) : sorte de "stub" assez réaliste qui renvoie des valeurs de retour dépendant des paramètres fournis
- **spy** (*espion*) : classe qui espionne les appels entrants pour vérifier (en fin de test) l'utilisation qui en est faite après l'exécution du code.
- **mock** (*simulacre*) : classes qui agissent comme un *stub* et un *spy*

NB: Beaucoup d'aspects (fonctionnalités) des "mocks" sont en général pris en charge dynamiquement par un framework spécialisé (ex : ***Mockito***) . ce qui évite de devoir écrire manuellement tout le comportement simulé.

Un bon paramétrage de mock (dans un test unitaire) permet de bien séparer le code réel d'un composant (toujours utile et sans trace) , du code supplémentaire qui n'est temporairement utile que lors d'une phase de test.

Idée du "mock" pour des tests les plus unitaires possibles

Un composant logiciel peut être de granularité plus ou moins fine :
- un sous service "DAO" (Data Access Object avec méthodes CRUD) codé par exemple avec JPA ou JDBC.
- un service métier (utilisant en interne ou en arrière plan des composants "DAO" et "DataSource" pour la persistance et la connexion à la base)

- Un test global/externe (de type boite "noire") du gros composant "service métier" permettra de tester d'un seul coup "le service , les "DAO" et le "dataSource" avec une vraie base de données en arrière plan)

- Dans certains cas complexes, il peut être difficile de localiser un bug dans tout cet assemblage de sous composants et on sera amené à tester plus finement/unitairement certaines méthodes de la classe principale du service en simulant le comportement des composants "dao" en arrière plan).

- Autrement dit, en simulant certains éléments périphériques, on concentre un test unitaire sur une portion de code pourtant dépendante des autres.

Test comportemental (d'acceptation fonctionnelle)

TDD = Test Driven Development
du coté "***développement***" : JUnit + ...
du coté "***acceptation fonctionnelle***" : *JBehave* ou *easyb* ou *cucumber* ou ....

ATDD = Acceptance Test Driven Development
BDD = Behavior Driven Development (synonyme de ATDD)

BDD (ou ATDD):

users_stories (fichiers ***".story"*** idéalement ***accrochés aux Uses Cases UML***)
= liste de scénarios rédigés de la façon suivante:

> **scénario xyz:**
>
> **Given** ***contexte***
> **When** ***événement ou condition***
> **Then** ***comportement attendu***

*Au départ: simple ***partie des spécifications fonctionnelles***
*Au final (***avec technologie annexe telle que jBehave***): ***réel test (exécutable) d'acceptation fonctionnelle***

Mutation testing (test de mutation) :

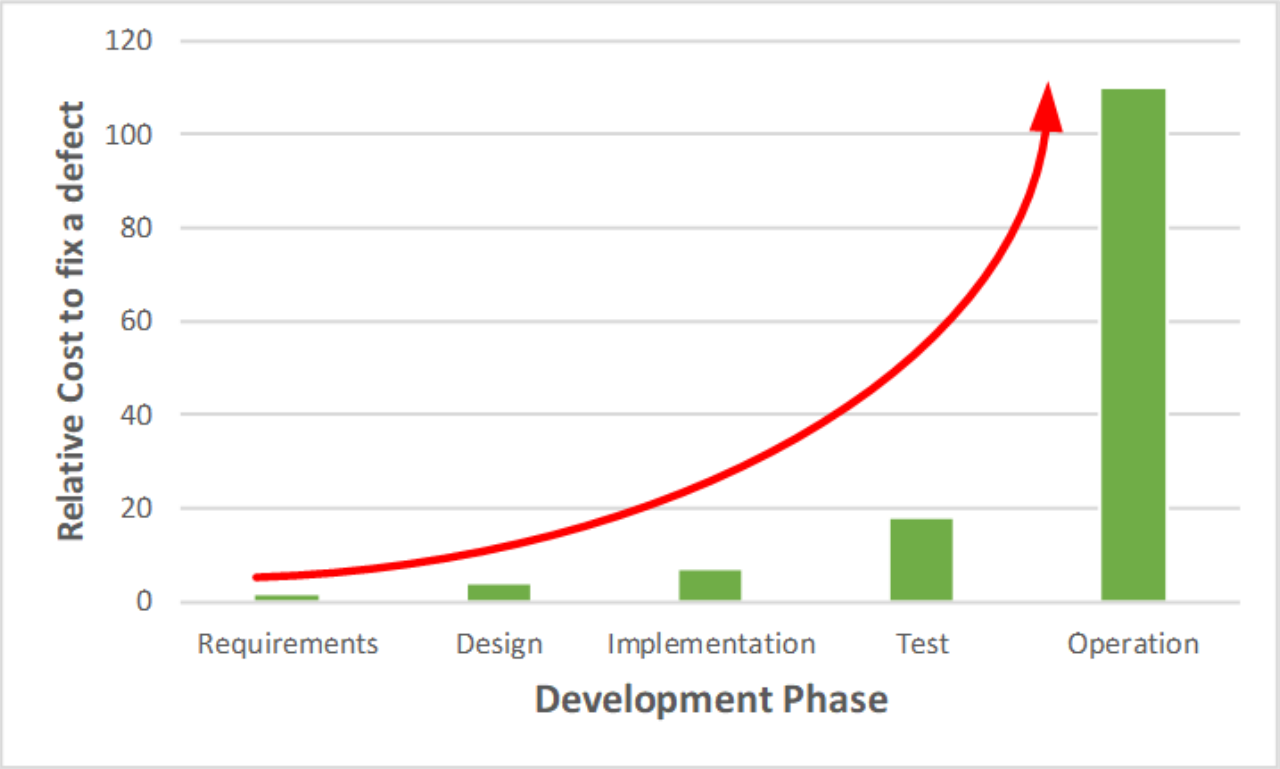
Les tests de mutation consistent à :

1. **créer des copies de votre code rendues volontairement défectueuses.**
2. **analyser les résultats de l'exécution de la suite de tests par rapport à ces copies.**

- **Si un test échoue, on dit que le mutant est tué (c'est bon signe : erreurs détectées)**
- **Si aucun test n'échoue, on dit que le mutant a survécu (c'est mauvais signe : erreurs pas détectées)**

C'est un moyen (parmi d'autres) de s'assurer d'une bonne couverture des tests .

Coûts des bugs pas assez identifiés



II - JUnit (tests unitaires java)

1. Evolutions de JUnit (3,4,5)

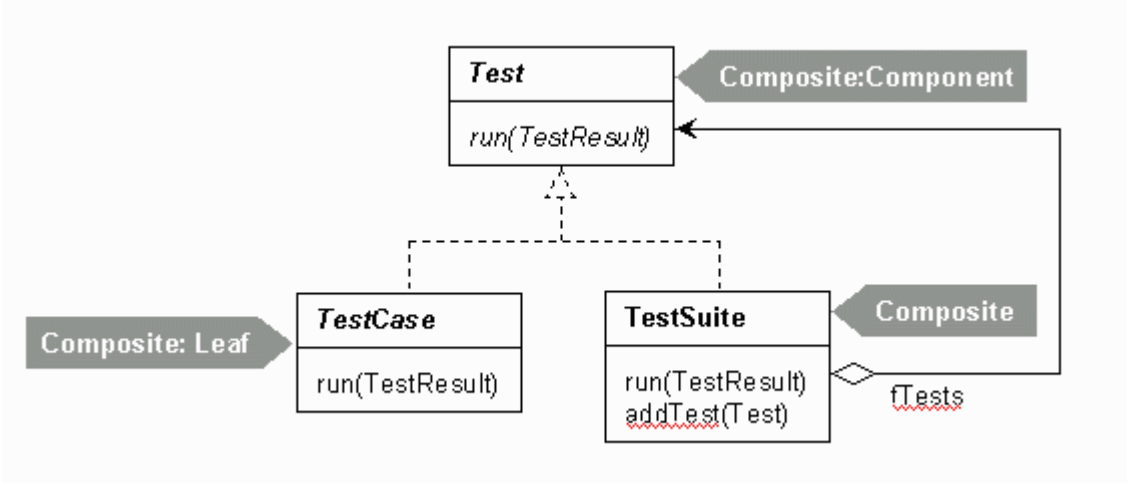
1.1. Versions importantes de JUnit

versions de JUnit	caractéristiques
JUnit3 (avant 2007)	• Sans annotation • basé sur des conventions de nom (setUp() , testXyz() , …) • basé sur quelques designs patterns ("composite" , "templateMethod" , …) • héritage obligatoire de TestCase
JUnit4 (après 2007)	• Compatible JUnit3 (utilisable sans annotation) • Avec annotations (héritage facultatif) • Quelques fonctionnalités spécifiques à la V4 (Rules, ...)
JUnit5 en mode "vintage"	*Transition entre JUnit4 et JUnit5 (compatibilité ascendante)*
JUnit5 (après 2017)	• Entièrement basé sur des configurations à base d'annotations • Nécessite java ≥ 8 (lambda expression) • Pas compatible avec Junit_3_et_4 • architecture entièrement restructurée • nouvelles fonctionnalités (intégrées ou bien sous forme d'extensions)

1.2. Principales extensions pour JUnit

Extensions pour JUnit	caractéristiques/utilités
dbUnit	Test de valeurs en base
EasyMock ou Mockito ou ...	Mocks (doublures)
Cucumber ou JBehave ou ...	BDD (Behavior , Given When Then)
Hamcrest ou JAssert ou ...	Assertions complémentaires
SpringExtension	Pour tests "Spring"

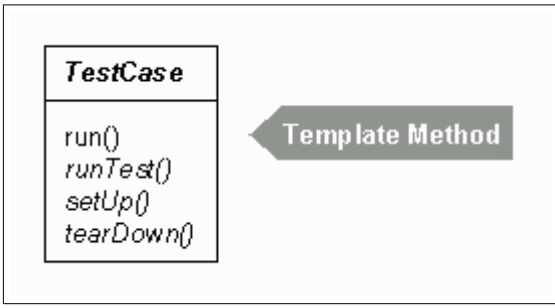
1.3. Structure composite de JUnit 3



L'interface junit.framework.**Test** de JUnit3 est implémentée par la classe ***TestCase*** et par la classe ***TestSuite*** .

Chaque classe de Test doit hériter (en V3) de TestCase

Une instance de TestSuite correspond à une suite ordonnée de Tests quelconques (Test(s) ou sous ensemble(s))




```
public void run() {
   setUp();
   runTest(); //lance toutes les méthodes testXyz()
   tearDown();
}
```

1.4. Lancement des tests unitaires via maven

mvn test **-> lance tous tests dont les noms de classes commencent ou se terminent par "Test"**

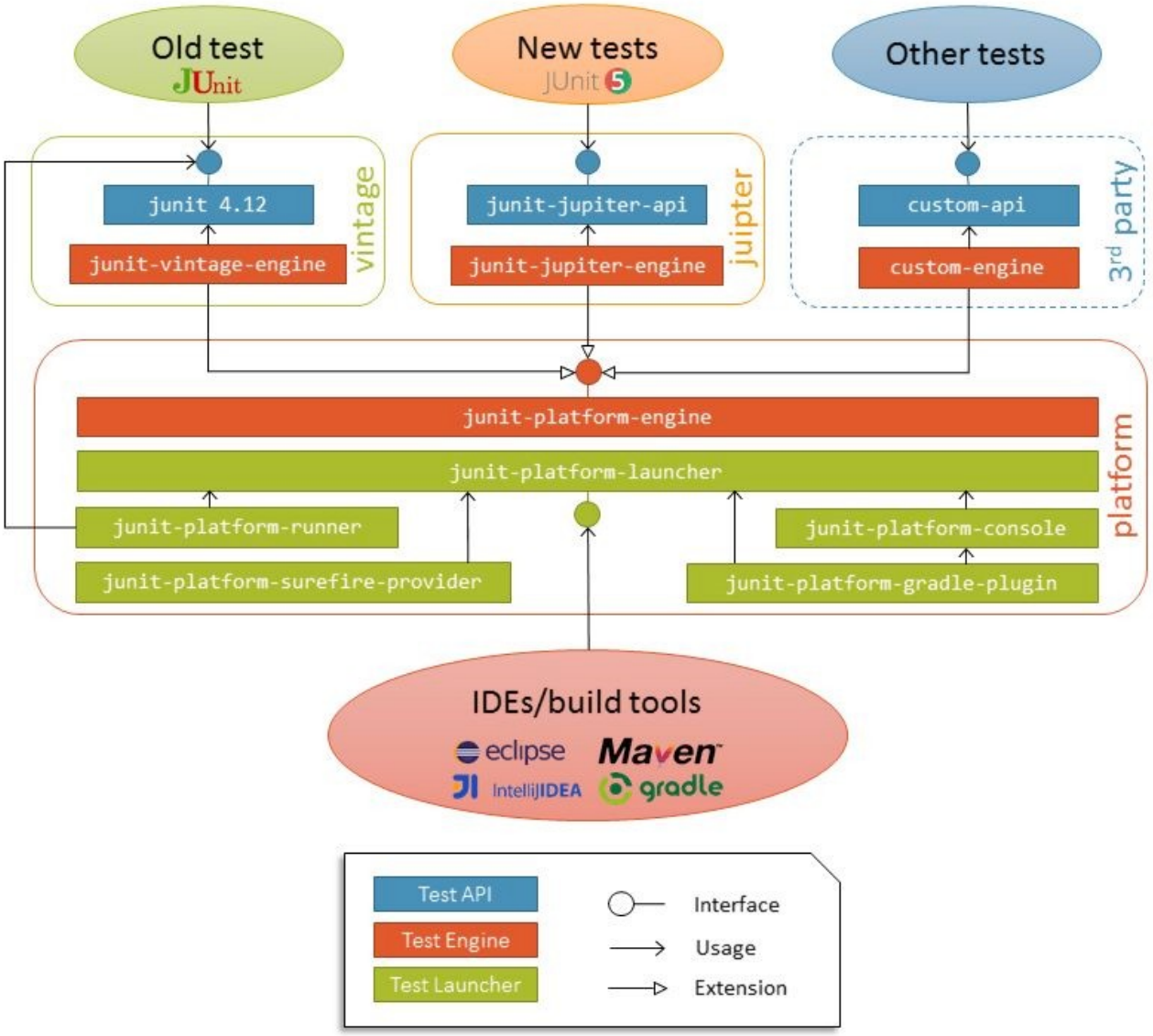
mvn test -Dtest=XxxTest -> lance que le test "XxxTest"
mvn test -Dtest=XyzSuite → lance la suite de tests *XyzSuite*

mvn test -Dtest=*xxTest -> lance tous les tests finissants par "xxTest"

NB:

- **par défaut , les tests unitaires sont systématiquement déclenchés lors d'un build ordinaire (ex: mvn package)**
- L'option "`-DskipTests=true`" de mvn permet d'annuler/sauter l'exécution des tests.

1.5. Structure de JUnit5



2. Tests unitaires avec JUnit (3 ou 4)

2.1. Présentation de JUnit

JUnit est un ***framework*** simple permettant d'effectuer des **tests** (unitaires , de non régression, ...) au cours d'un développement java . [**Projet Open source ---> http://junit.sourceforge.net/ , http://junit.org]** . ***JUnit est intégré au sein de l'IDE Eclipse*** . **JUnit** existe en **versions 3 et 4**.

La **version 4** utilise des **annotations** pour son paramétrage (**@Test** , **@Before** , ...)

2.2. Structure d'une classe de test (ancienne version JUnit3)

(Ancien) JUnit 3

héritage

Conventions de noms** sur méthodes **setUp()**, **testXy()** et **tearDown()

```
import junit.framework.TestCase;

public class CalculateurTest   extends TestCase {
   private Calculateur c; //objet/composant à tester

   protected void setUp(){  //initialisation du composant à tester
     c = new Calculateur();  //setUp() appelée avant chaque testXy()
   }

   public void testAdd()  {
      assertEquals( c.add(5,6) , 11 , 0.000001 );
      // ou assertTrue(condition_a_vérifier) .
   }

   public void testMult() {
      assertEquals( c.mult(5,6) , 30 , 0.000001 );
   }
   protected void tearDown(){
  // éventuel code de terminaison réinitialisant certaines valeurs
  // d'un jeu de données (méthode facultative) }
}
```

JUnit 3 est basée sur des conventions de nommage:

- La méthode ***setUp***() sera appelée automatiquement pour initialiser les valeurs de certains objets qui seront ultérieurement utilisés au sein des tests.
- Chaque test correspond à une méthode de type "***testXxx()***" ne retournant rien (**void**) mais effectuant quelques assertions (***Assert.assertXxxx(....)***)
- On peut éventuellement programmer une méthode ***tearDown***() qui sera alors appelée après chaque terminé (ex: pour ré-initialiser le contenu d'une base après).

2.3. Structure d'une classe de test (version JUnit4)

JUnit4 *(avec annotations)*

Plus besoin d'hériter de TestCase mais **@Before** **@After** et **@Test** attendus

```
import org.junit.Assert;
import org.junit.Test;   import org.junit.Before;

public class CalculateurTest
{  private Calculateur c;

   @Before   /* comportement proche d'un constructeur par défaut*/
   public void initialisation(){
     c = new Calculateur();  // déclenché avant chaque @Test .
   }

   @Test
   public void testerAdd()  {
      Assert.assertEquals( c.add(5,6) , 11 , 0.000001 );
       //ou Assert.assertTrue(5+6==11);
   }
   @Test
   public void testerMult() {
      Assert.assertEquals( c.mult(5,6) , 30 , 0.000001 );
   }
}
```

Méthode statique

Il existe @Before , @After (potentiellement déclenchés plusieurs fois [avant/après chaque test]) et @BeforeClass , @AfterClass (pour initialiser des choses "static")

NB: il faut que **JUnit-4...jar** soit dans le classpath /

La démarche conseillée consiste à

*** coder un embryon des classes à programmer (code incomplet)**

*** coder les tests (voir précédemment) et les déclencher une première fois (==> échecs normaux)**

*** programmer les traitements prévus**

*** ré-effectuer les tests (==> réussite ???)**

*** améliorer (peaufiner) le code**

- **ré-effectuer les tests (==> non régression ???) * ...**

2.4. Fonctionnement de JUnit

Une instance de "Test JUnit" pour chaque test unitaire !!!

La technologie JUnit (en version 3 ou 4) créer automatiquement une instance de la classe de test pour chaque méthode de test à déclencher.

→ constructeurs , setUp() et méthodes préfixées par @Before seront donc potentiellement appelés plusieurs fois !!!!
→ la notion d'ordre d'appel des méthodes est inexistante (non applicable) sur une classe de test JUnit.

Ceci permet d'obtenir des tests unitaires complètement indépendants mais ceci peut quelquefois engendrer certaines lenteurs ou lourdeurs.

Certains contextes (plutôt "stateless") peuvent se prêter à **des optimisations "static" (@BeforeClass , @AfterClass)** .

En combinant JUnit4 avec d'autres technologies (ex : SpringTest) , on peut également effectuer quelques optimisations (initialisations spéciales) au cas par cas selon le(s) framework(s) utilisé(s).

Assert.*assertNotNull*() et Assert.*fail*("*message*") peuvent être pratiques dans certains cas (exceptions non remontées, ...)

Ordre des méthodes de test sur une classe JUnit4

Une classe de test **JUnit4** peut comporter plusieurs méthodes de test.
Chacune de ces méthodes correspond à un **test unitaire** (censé être indépendant des autres)
et donc par défaut , pour garantir une bonne isolation entre les différents tests unitaires :

- l'ordre des méthodes de tests déclenchées est non déterministe
- chaque méthode de test est exécutée avec une instance différente de la classe de test

Dans certains cas rares et pointus, on peut préférer **contrôler l'ordre des méthodes qui seront appelées**. Les tests seront alors un peu **moins "unitaires"** et un peu plus "liés".

Ceci peut s'effectuer de deux manières :

* avec l'annotation **@FixMethodOrder** que l'on trouve sur les versions récentes de JUnit4 .
* En écrivant une classe contrôlant le déclenchement d'une **série ordonnée de tests unitaires** (avec **@Suite**) .

2.5. @BeforeClass et "static" (optimisations)

JUnit4 *(avec "static" et "@BeforeClass")*

@BeforeClass
@AfterClass
attendus pour gérer des éléments "static"

```
import org.junit.Assert;
import org.junit.Test;   import org.junit.Before;

public class CalculateurTest
{  private static Calculateur c;

   @BeforeClass   /* appelée une seule fois */
   public static void  initialisation(){
     c = new Calculateur();  // initialisation "static" .
   }

   @Test
   public void testerAdd()  {
      Assert.assertEquals( c.add(5,6) , 11 , 0.000001 );
      //ou Assert.assertTrue(5+6==11);
   }
   @Test
   public void testerMult() {
      Assert.assertEquals( c.mult(5,6) , 30 , 0.000001 );
   }
}
```

Méthode statique

2.6. Suite de tests (@Suite)

Suite ordonnées de tests (JUnit4)

Quelques usages potentiels (tests ordonnés):

séquence Create/insert , select , update , select , delete , …

variantes au niveau intégration continue :

- Suite_tests_essentiel (pour build rapides)
- Suite_complete_tests (pour build de nuit)

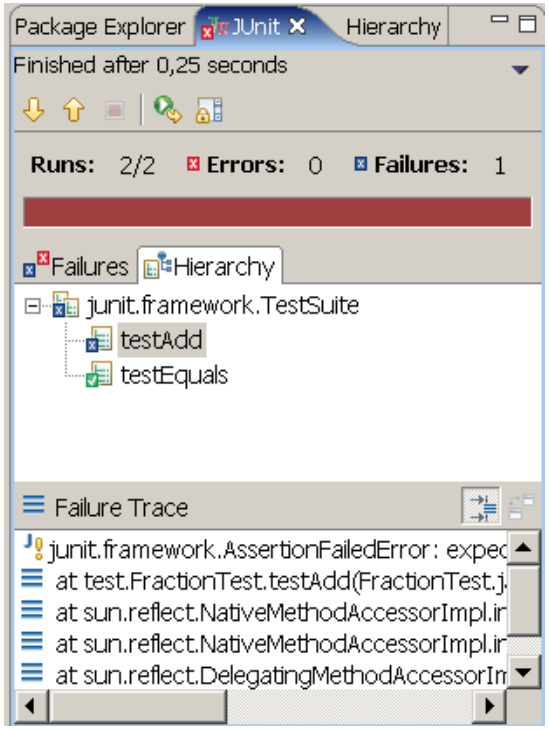
exemple :

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
  JunitTest1.class,
  JunitTest2.class
})
public class JunitTestSuite {
}
```

2.7. Lancement des tests unitaires

Lancement des tests unitaires

Depuis l'IDE eclipse:
Run as … / JUnit test



TestSuite en JUnit3
et lancement après une sélection de **package** en JUnit4 pour lancer d'un coup toute une série de tests.

Comptabilisations :
Error(s) : exceptions java non rattrapées.
Failure(s) : assertions non vérifiées.
VERT si aucune erreur.

Depuis "maven" :
coder des classes nommées "***Test****Xy*" ou "*XY****Test***" dans **src/test/java**
et lancement via **mvn test** ou autre.

Combinaisons de frameworks (de tests) via @RunWith

```
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
...
// nécessite spring-test.jar et junit4.11.jar dans le classpath
@RunWith(SpringJUnit4ClassRunner.class)
// Chargement automatique de "/mySpringConf.xml" pour la configuration
@ContextConfiguration(locations={"/mySpringConf.xml"})
public class TestGestionComptes {

// injection/initialisation du composant à tester contrôlée par @Autowired (de Spring)
@Autowired //ou bien @Inject
private InterfaceServiceXY serviceXy = null;

 @Test
public void testTransferer(){
        serviceXy.transferer(...) ;  Assert.assertTrue( ... );
        }
}
```

Il existe aussi @RunWith(MockitoJUnitRunner.class) et autres.

3. Très anciens aspects avancés de JUnit 3

3.1. Exemple de configuration maven pour JUnit3

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>tp</groupId>
  <artifactId>very_old_junit3</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.release>8</maven.compiler.release>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
      <dependency>
```

```
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>3.8.2</version>
                        <scope>test</scope>
        </dependency>
</dependencies>

</project>
```

3.2. Suite de tests (JUnit3)

CalculSuite.java (suite de tests)

```
package tp.calculs;
import junit.framework.Test;
import junit.framework.TestSuite;

public class CalculSuite extends TestSuite {
   public static Test suite() {
      final TestSuite s = new TestSuite();
      s.addTest(TestCalculsSimples.suite());
      s.addTest(TestEmptySerie.class);
      s.addTest(TestSerie.class);
      return s;
   }
}
```

AllTestsSuite.java (suite de sous-suites)

```
package tp;

import junit.framework.Test;
import junit.framework.TestSuite;
import tp.calculs.CalculSuite;
import tp.converter.ConvertSuite;

public class AllTestsSuite extends TestSuite {
   public static Test suite() {
      final TestSuite s = new TestSuite();
      s.addTest(CalculSuite.suite());
      s.addTest(ConvertSuite.suite());
      return s;
   }
}
```

3.3. Test de levée d'exception

```
...
public class TestConverter extends TestCase {

public void testBadDoubleFromStringConversion() {
```

```
        try {
                String sx = "12a.34";
                double x = BasicConverter.doubleFromString(sx);
                fail("NumberFormatException should be raised");
        } catch (NumberFormatException e) {
                System.err.println("NumberFormatException " +  e.getMessage());
                //e.printStackTrace();
                assertTrue(e.getClass().getSimpleName().equals("NumberFormatException"));
        }

}
}
```

3.4. Limitations de JUnit3

- **Manque de souplesse , de paramétrages , d'extensions**
- **pas d'optimisation simple pour static en JUnit3** (Tout de même la possibilité suivante) :

TestCalculsSimples.java

```
package tp.calculs;

import junit.extensions.TestSetup;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class TestCalculsSimples extends TestCase {

        public static CalculsSimples calculsSimples;

        //pas d'optimisation simple pour static en JUnit3
        //Tout de même cette possibilité:
        public static Test suite() {
           return new TestSetup(new TestSuite(TestCalculsSimples.class)) {

              protected void setUp() throws Exception {
                 System.out.println("Global setUp ");
                 myGlobalSetUp();
              }
              protected void tearDown() throws Exception {
                 System.out.println("Global tearDown ");
                 myGlobalTearDown();
              }
           };
        }

        public static void myGlobalSetUp() {
            System.out.println("myGlobalSetUp() in JUnit3 TestSetup/TestSuite called on TestCalculsSimples");
```

```
            calculsSimples= new CalculsSimples();
        }

        public static void myGlobalTearDown() {
            System.out.println("myGlobalTearDown() in JUnit3 TestSetup/TestSuite called on TestCalculsSimples");
        }

        public void testAddInt() {
                int res = calculsSimples.addInt(5, 6);
                System.out.println("testAddInt() , res="+res);
                //assertEquals(11, res);
                assertTrue(11==res);
        }

        public void testAddDouble() {
                double res = calculsSimples.addDouble(5.0, 6.0);
                System.out.println("testAddDouble() , res="+res);
                assertEquals(11, res,0.000001);
        }
}
```

4. Aspects avancés de JUnit 3 et/ou 4

4.1. Factoriser les fixtures

MyTranslationFixtureHelper.java

```
package tp.converter;

import java.util.Arrays;
import java.util.List;

public class MyTranslationFixtureHelper {

	public static List<String> englishTextList(){
		/*
		List<String> txtList = new ArrayList<String>();
		txtList.add("red"); txtList.add("green");
		return txtList;
		*/
		return Arrays.asList("red","green");
	}
}
```

TestEnglishToFrenchTranslator.java

```
public class TestEnglishToFrenchTranslator /* extends TestCase */ {
	private Translator translator;
	private List<String> textList;

	@Before
	public void setUp(){
		translator = new EnglishToFrenchTranslator();
		textList = MyTranslationFixtureHelper.englishTextList();
	}

	@Test
	public void testGoodTranslations() {
		Map<String,String> translatedMap = this.translator.translate(this.textList);
		System.out.println("translatedMap="+translatedMap);
		assertTrue(translatedMap.size()==textList.size());
		assertEquals("rouge", translatedMap.get("red"));
		assertEquals("vert", translatedMap.get("green"));
	}
}
```

TestEnglishToSpanishTranslator.java

```
public class TestEnglishToSpanishTranslator /* extends TestCase  */{
	private Translator translator;
	private List<String> textList;
```

```
	@Before
	public void setUp(){
		translator = new EnglishToSpanishTranslator();
		textList = MyTranslationFixtureHelper.englishTextList();
	}

	@Test
	public void testGoodTranslations() {
		Map<String,String> translatedMap = this.translator.translate(this.textList);
		System.out.println("translatedMap="+translatedMap);
		assertTrue(translatedMap.size()==textList.size());
		assertEquals("rojo", translatedMap.get("red"));
		assertEquals("verde", translatedMap.get("green"));
	}
}
```

Autres idées/possibilités :

- héritage (d'une classe abstraite ou concrète commune)
- utilisation de singleton+factory
- ...

4.2. Lancement des tests depuis main() avec résultats

```
package tp.runner;

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
//...

public class MyTestRunner {

	public static void main(String[] args) {
		runThisTest(TestParameterizedMyChecker.class);
		runThisTest(WrongTests.class);
	}

	public static void runThisTest(Class<?> testClass) {
	   System.out.println("runThisTest --> " + testClass.getSimpleName());
	   Result result = JUnitCore.runClasses(testClass);
	   for (Failure failure : result.getFailures()) {
	      System.err.println("\t" +failure.toString());
	    }
	   System.out.println("\t wasSuccessful=" + result.wasSuccessful());
	}

}
```

5. Aspects avancés de JUnit 4

5.1. Exemple de configuration maven pour JUnit4

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>tp</groupId>
  <artifactId>very_old_junit3</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.release>8</maven.compiler.release>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
      <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
     </dependency>

          <!-- hamcrest is a anagram for "matcher" , it's a old java assertion library -->
     <dependency>
          <groupId>org.hamcrest</groupId>
          <artifactId>hamcrest</artifactId>
          <version>2.2</version>
          <scope>test</scope>
    </dependency>

    <dependency>
          <groupId>org.apache.logging.log4j</groupId>
          <artifactId>log4j-slf4j-impl</artifactId>
          <version>2.15.0</version>
     </dependency>
</dependencies>

</project>
```

5.2. Suite de tests (JUnit4)

CalculSuite.java (suite de tests)

```
package tp.calculs;

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

import junit.framework.TestSuite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
        TestCalculsSimples.class ,
        TestEmptySerie.class,
        TestSerie.class
})
public class CalculSuite  {
}
```

AllTestsSuite.java

```
package tp;

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

import junit.framework.TestSuite;
//...

@RunWith(Suite.class)
@Suite.SuiteClasses({
        CalculSuite.class ,
        ConvertSuite.class
})
public class AllTestsSuite  {
}
```

NB : Les annotations pour les suites de tests seront changées en V5 de JUnit

5.3. Catégorie de Tests (v4 seulement)

NB : **@Categorie** est considéré comme experimental en JUnit 4 et sera remplacé par @Tag en v5

Interfaces de marquages pour catégories de tests (junit4) , à coder par exemple dans le package tp.categories :

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

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

```
public interface ImportantTests {
        /* category marker */
}
```

```
public interface SecondaryTests {
        /* category marker */
}
```

TestWithCategory.java

```
package tp.converter;

import org.junit.Test;
import org.junit.experimental.categories.Category;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import tp.converter.categories.FastTests;
import tp.converter.categories.ImportantTests;
import tp.converter.categories.SecondaryTests;
import tp.converter.categories.SlowTests;

public class TestWithCategory {
        private static Logger logger = LoggerFactory.getLogger(TestWithCategory.class);

        @Test
        @Category(SlowTests.class)
        public void testA() {
                logger.trace("slow testA");
        }

        @Test
        @Category(FastTests.class)
        public void testB() {
                logger.trace("fast testB");
        }
```

```
	@Test
	@Category({FastTests.class , ImportantTests.class})
	public void testC() {
		logger.trace("important fast testC");
	}

	@Test
	@Category({SlowTests.class , SecondaryTests.class})
	public void testD() {
		logger.trace("secondary slow testD");
	}
}
```

Suite de tests (V4) avec filtrage selon catégories :

FastTestsSuite.java

```
package tp.converter;

import org.junit.experimental.categories.Categories;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

import tp.converter.categories.SecondaryTests;
import tp.converter.categories.SlowTests;

@RunWith(Categories.class)  //à la place de @RunWith(Suite.class)
//@Categories.IncludeCategory(FastTests.class)
//@Categories.ExcludeCategory(SlowTests.class)
@Categories.ExcludeCategory({SlowTests.class , SecondaryTests.class})
@Suite.SuiteClasses({
	TestWithCategory.class ,
	TestWithAssumptions.class
})
public class FastTestsSuite  {
}
```

NB : ces paramétrages ne sont utilisables qu'en version 4 de JUnit !!!

En V5 de JUnit:

@Categorie(Xyz.class) sera remplacé par @Tag("xyz")

@Categories.IncludeCategory() sera remplacé par @IncludeTags()

@Categories.ExcludeCategory() sera remplacé par @ExcludeTags()

5.4. Désactivation d'un test

@Ignore en **V4** (org.junit.Ignore) , @Disabled en V5(org.junit.jupiter.api.Disabled)

WrongTests.java

```
package tp.converter;
import static org.junit.Assert.assertTrue;
import org.junit.Ignore;
import org.junit.Test;

public class WrongTests {

	@Test
	@Ignore //test désactivé/ignoré
	public void testWithError() {
		int a=3;
		int b=0;
		assertTrue(a/b==0);//div by zero exception/error !!!
	}

	@Test
	@Ignore //test désactivé/ignoré
	public void testWithBadAssertion() {
		assertTrue(2+2==5); //2+2==4 !!!! not 5 !!!
	}

	@Test
	public void goodTrivialTest() {
		assertTrue(2+2==4);
	}
}
```

5.5. Test de durée (JUnit 4)

```
@Test(timeout = 200) //200ms
	public void testWithTimeout() {
		try {
			Thread.sleep(100);//ok
			//Thread.sleep(300);//not_ok
		} catch (InterruptedException e) {
			System.err.println(e.getMessage());
		}
	}
```

Effet : si une méthode de Test **met trop de temps à s'exécuter** alors elle est **interrompue** et son exécution est considérée comme ***en erreur*** (org.junit.runners.model.***TestTimedOutException***: test timed out after 200 milliseconds)

NB :

- **org.junit.Test** de **JUnit4** comporte le paramètre optionnel **timeout** *(à exprimer en ms).*
- org.junit.jupiter.api.Test de JUnit5 ne comporte plus le paramètre timeout mais peut être compagnée de @Timeout()

Autre possibilité en JUnit4 (qui sera développée dans le paragraphe sur les règles):

```
@Rule
public Timeout globalTimeout = new Timeout(40, TimeUnit.MILLISECONDS);
```

5.6. Suppositions (assumptions)

TestWithAssumptions.java

```
package tp.converter;

import static org.junit.Assume.assumeTrue;

import org.junit.Before;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class TestWithAssumptions {
	private static Logger logger = LoggerFactory.getLogger(TestWithAssumptions.class);
	private String javaVersion;//ex: 17.0.6
	private int javaMajorVersion; //ex: 17

	@Before() //or @BeforeClass
	public void initBefore() {
		this.javaVersion = System.getProperty("java.version");
		this.javaMajorVersion=Integer.parseInt( javaVersion.split("\\.")[0] );
		logger.trace("javaVersion="+javaVersion + " javaMajorVersion="
				+ javaMajorVersion);
	}

	@Test()
	public void test1() {
		logger.trace("first step of test1");
		assumeTrue(this.javaMajorVersion>=11);
		logger.trace("other steps of test1 if javaVersion >= 11");
		assumeTrue(this.javaMajorVersion>=17);
		logger.trace("other steps of test1 if javaVersion >= 17");
		assumeTrue(this.javaMajorVersion>=22);
		logger.trace("other steps of test1 if javaVersion >= 22");
	}
}
```

NB : dès qu'une supposition/assumption n'est pas vérifiée , la fin de la méthode de test n'est pas exécutée (elle est ignorée et considérée sans erreur) .

Résultat (traces) de l'exécution de ce test lancé depuis un **jdk 17** en 2023 :

*2023-06-30 18:30:21 TRACE TestWithAssumptions:21 - **javaVersion=17.0.6 javaMajorVersion=17***

2023-06-30 18:30:21 TRACE TestWithAssumptions:27 - first step of test1

2023-06-30 18:30:21 TRACE TestWithAssumptions:29 - other steps of test1 if javaVersion >= 11

2023-06-30 18:30:21 TRACE TestWithAssumptions:31 - other steps of test1 if javaVersion >= 17

et pas de suite car pas encore java22 en 2023

5.7. Tests paramétrés (JUnit 4)

Exemple de code à tester:

```
package tp.calculs;

public class MyChecker {
        public static boolean isPrimeNumber(final int primeNumber) {
            for (int i = 2; i <= (primeNumber / 2); i++) {
              if (primeNumber % i == 0) {
                return false;
              }
            }
            return true;
          }

        public static boolean isEven(final int number) {
                return (number % 2)==0 ;//pair
        }

        public static boolean isOdd(final int number) {
                return (number % 2)!=0 ;//impair
        }
}
```

Test paramétré (en version JUnit4) :

```
package tp.calculs;
import static org.junit.Assert.assertTrue;
import java.util.Arrays;
import java.util.Collection;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.slf4j.Logger;   import org.slf4j.LoggerFactory;

@RunWith(Parameterized.class)
public class TestParameterizedMyChecker {
        private static Logger logger = LoggerFactory.getLogger(TestParameterizedMyChecker.class);

         private int inputNumber;
         private boolean shouldBeEven;
         private boolean shouldBePrime;

        //constructor params should be the sames as parameters set
        public TestParameterizedMyChecker(Integer inputNumber, Boolean shouldBeEven ,
                                                Boolean shouldBePrime){
                this.inputNumber = inputNumber;
                this.shouldBeEven = shouldBeEven;
                this.shouldBePrime = shouldBePrime;
```

```
            //éventuelle variante au constructeur récupérateur de paramètres:
            //des annotations @Parameter(0), @Parameter(1) , @Parameter(2)
             //placées sur les attributs inputNumber , shouldBeEven et shouldBePrime
    }

    //parameters set :
     @Parameterized.Parameters
      public static Collection<?> evenOrPrimeNumbers() {
            //return collection of { inputNumber , shouldBeEven , shouldBePrime}
        return Arrays.asList(new Object[][] {
          { 1, false, true },
           ...
          { 22, true , false },
          { 23, false, true }
        });
      }

      // This test will run n times since we have n values of parameters_set defined
      @Test
      public void testEvenOddPrimeNumberChecker() {
        logger.trace("Parameterized values: [inputNumber=" + this.inputNumber
                    + ",shouldBeEven=" + this.shouldBeEven
                     + ",shouldBePrime=" + this.shouldBePrime + "]");
        logger.trace("for inputNumber=" + this.inputNumber
              + " MyChecker.isEven=" + MyChecker.isEven(inputNumber)
              + " MyChecker.isPrimeNumber=" + MyChecker.isPrimeNumber(inputNumber) );
        assertTrue(MyChecker.isEven(inputNumber)==this.shouldBeEven);
        assertTrue(MyChecker.isOdd(inputNumber)!=this.shouldBeEven);
        assertTrue(MyChecker.isPrimeNumber(inputNumber)==this.shouldBePrime);
      }
}
```

Un **test paramétré** est un test qui est **automatiquement lancé plusieurs fois** avec des **paramètres variables** dont les valeurs sont issues d'un jeu de données (**@Parameterized.Parameters**)

Résultats :

2023-07-03 09:30:57 TRACE TestParameterizedMyChecker:54 - Parameterized values: [inputNumber=1,shouldBeEven=false,shouldBePrime=true]
2023-07-03 09:30:57 TRACE TestParameterizedMyChecker:57 - for inputNumber=1 MyChecker.isEven=false MyChecker.isPrimeNumber=true
...

2023-07-03 09:30:57 TRACE TestParameterizedMyChecker:54 - Parameterized values: [inputNumber=23,shouldBeEven=false,shouldBePrime=true]
2023-07-03 09:30:57 TRACE TestParameterizedMyChecker:57 - for inputNumber=23 MyChecker.isEven=false MyChecker.isPrimeNumber=true

5.8. Règles (@Rule) de JUnit 4

Le **concept de règle (@Rule) de JUnit4** (correspondant à peu près au concept *d'extension de JUnit5*) peut être vu comme une implémentation très spéciale du concept **d'intercepteur** avec code supplémentaire automatique.

Autrement dit, en version 4 de JUnit, la mise en place d'une règle revient à peu près au même que d'écrire un supplément de code dans des méthodes préfixées par @Before et @After.

L'intérêt des règles (@Rule) de Junit4 tient
dans leurs compacités et leurs potentielles **ré-utilisabilités** .

Le code et l'application d'une règle personnalisée permettra de mieux comprendre le principe.

5.8.a. Régle personnalisée (Junit4)

```
package tp.other.rules;

import org.junit.rules.TestRule;
import org.junit.runner.Description;
import org.junit.runners.model.Statement;
import org.slf4j.Logger;      import org.slf4j.LoggerFactory;

public class MyPerfCustomRule implements TestRule {
  private static Logger logger = LoggerFactory.getLogger(MyPerfCustomRule.class);

 @Override
 public Statement apply(Statement base, Description description) {
   return new Statement() {
     @Override
     public void evaluate() throws Throwable {
        //BEFORE
        logger.info("MyPerfCustomRule, before execution of " +description.getMethodName());
       long startTime = System.nanoTime();

       try {
             base.evaluate(); //TEST EXECUTION
         } catch(Exception ex){
             logger.error("MyPerfCustomRule, exception after execution of "
                         +description.getMethodName() + " " + ex.getMessage());
         }finally {
             //AFTER (bad or good test)
             long endTime = System.nanoTime();
             logger.info("MyPerfCustomRule, after execution of "
                     +description.getMethodName()
                     + " execDuration (ms)= " + ((double)(endTime - startTime)/1000000));
           }
       }
   };
 }
}
```

5.8.b. Application d'une règle (Junit4)

```
package tp.other;
import org.junit.Rule;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import tp.other.rules.MyPerfCustomRule;

public class TestWithCustomRule {
	private static Logger logger = LoggerFactory.getLogger(TestWithTimeoutRule.class);

	@Rule
	public MyPerfCustomRule myPerfCustomRule = new MyPerfCustomRule();

	//NB: il existe également @ClassRule en JUnit4

	@Test
	public void fastTest() {
		try {
			Thread.sleep(20);//ok
		} catch (InterruptedException e) {
			System.err.println(e.getMessage());
		}
	}

	@Test
	public void slowTest() {
		try {
			Thread.sleep(60);//ok
		} catch (InterruptedException e) {
			System.err.println(e.getMessage());
		}
	}
}
```

Résultats (règle appliquée automatiquement à l'éxécution de chacun des tests) :

INFO MyPerfCustomRule:20 - MyPerfCustomRule, before execution of **slowTest**

INFO … MyPerfCustomRule, after execution of **slowTest** execDuration (ms)= **61.5344**

INFO MyPerfCustomRule:20 - MyPerfCustomRule, before execution of **fastTest**

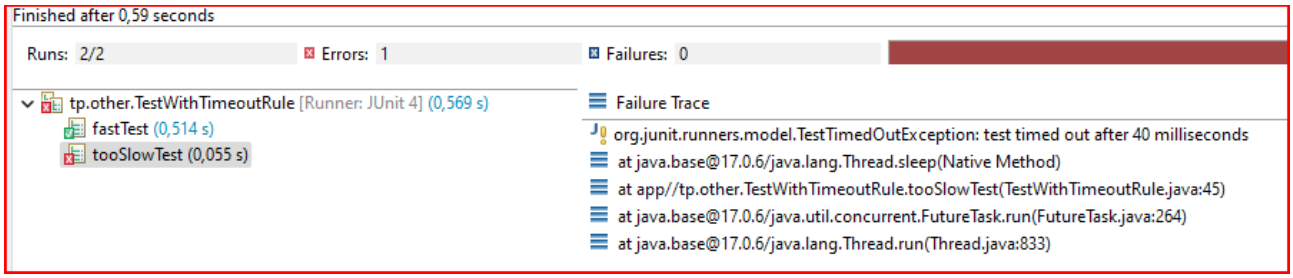
INFO ... - MyPerfCustomRule, after execution of **fastTest** execDuration (ms)= **25.1034**

5.8.c. Application d'une règle prédéfinie (Junit4)

Exemple d'application de la règle Timeout() :

```
package tp.other;
import java.util.concurrent.TimeUnit;
import org.junit.Rule;  import org.junit.Test;
import org.junit.rules.Timeout;
import org.slf4j.Logger;  import org.slf4j.LoggerFactory;

public class TestWithTimeoutRule {
   private static Logger logger = LoggerFactory.getLogger(TestWithTimeoutRule.class);

   @Rule
   public Timeout globalTimeout = new Timeout(40, TimeUnit.MILLISECONDS);

        @Test
        public void fastTest() {
                try {
                        Thread.sleep(20);//ok
                } catch (InterruptedException e) {
                        System.err.println(e.getMessage());
                }
        }

        @Test
        //@Ignore
        public void tooSlowTest() {
                try {
                        Thread.sleep(60);//not ok
                } catch (InterruptedException e) {
                        System.err.println(e.getMessage());
                }
        }
}
```

Résultats :



NB : il existe aussi @ClassRule de niveau "classe" pour éléments "static" :

```
@ClassRule
public static TemporaryFolder globalFolder = new TemporaryFolder();
```

5.8.d. Principales règles préfédinies (JUnit4)

```
import org.junit.rules.*
```

Principales régles prédéfinies de JUnit4	caractéristiques
Timeout	**Timeout de niveau global (pour toutes les méthodes de tests d'une classe)**
TemporyFolder	**Destruction automatique (en fin de test) de tous les fichiers et répertoires temporaires créés durant le test.** *Pourra être remplacé par @TempDir en version JUnit5*
`ExpectedException`	Pour vérifier que les méthodes remontent une exception (ou pas si ExpectedException.none())
TestName	Juste pour connaître et afficher (via logger) le nom de la méthode de test **@Rule** public TestName ***name = new TestName();*** et logger.info("Executing: {}", ***name.getMethodName()***); dans méthode de test
`ErrorCollector`	Pour ne pas s'arrêter à la première erreur/exception mais toutes les collecter/afficher. Le test échoue à la fin
Verifier	Vérification suppémentaire (à coder brièvement) en fin de test , Sorte de assert complémentaire global à toutes les méthodes de test. Exemple : *@Rule* *public Verifier verifier = new Verifier() {* *@Override* *public void verify() { assertTrue("....", ...);* *}};*
DisableOnDebug	Désactiver une règle (ex : Timeout.seconds(30)) en mode debug
`RuleChain`	Enchainement paramétré de règles
ExternalResource	Classe abstraite dont on peut hériter pour programmer une régle proche de TemporyFolder mais opérant sur une ressource de type "connexion à une base de données" ou autre (qui sera automiquement libérée en fin de test)

Exemple partiel :

```
@Rule
public TemporaryFolder tmpFolder = new TemporaryFolder();

@Test
public void givenTempFolderRule_whenNewFile_thenFileIsCreated() throws IOException {
   File testFile = tmpFolder.newFile("test-file.txt");

   assertTrue("The file should have been created: ", testFile.isFile());
   assertEquals("Temp folder and test file should match: ",
     tmpFolder.getRoot(), testFile.getParentFile());
```

```
}
```

6. Assertions complémentaires/spécifiques

6.1. Historique et évolutions

Les premières versions de Junit4 ont introduit org.junit.Assert.**assertThat**() permettant de passer en paramètre des fonctions d'assertions spécifiques (du paquet hamcrest 1.x à l'origine).

Au sein des versions récentes de JUnit4 , il est déconseillé d'utiliser ~~org.junit.Assert.assertThat()~~ car ça ne peut utiliser que des matchers de l'ancienne version 1.x de hamcrest .
Il est conseillé d'utiliser **org.hamcrest.MatcherAssert.assertThat()** au sein des versions récentes de Junit 4 ou JUnit5.

A coté de cela , **AssertJ** est un concurrent de **hamcrest** qui est plus lisible et qui gère mieux l'auto-complétion du code au sein des IDEs (eclipse, intelliJ, ..)

NB : la version 5/jupiter de JUnit n'est plus du tout liée à hamcrest .
org.junit.jupiter.api.Assertions ne comporte plus .assertThat() .
On peut donc librement utiliser AssertJ à la place de hamcrest en V5 .

6.2. Hamcrest

```
<!-- hamcrest is a anagram for "matcher" , it's a old java assertion library -->
        <dependency>
                <groupId>org.hamcrest</groupId>
                <artifactId>hamcrest</artifactId>
                <version>2.2</version>
                <scope>test</scope>
        </dependency>
```

Exemple :

```
import org.hamcrest.MatcherAssert;
import org.hamcrest.Matchers;

...

@Test
public void testPersonneWithHamcrestAssertThat() {
        Personne originalPers = new Personne(2L,"Jean" , "Bon" , 172.4);
        Personne p = (Personne) BasicConverter.allPartsInLowercase(originalPers);
        logger.trace("p with allPartsInLowercase = " +  p);
        MatcherAssert.assertThat(p.getNom(),Matchers.is("bon"));
}
```

Quelques exemples de "matcher" hamcrest :

```
assertThat("myString",  allOf( startsWith("my") , containsString("Str") ) )

 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)));
```

- **allOf** - matches if all matchers match (short circuits)
- **anyOf** - matches if any matchers match (short circuits)
- **not** - matches if the wrapped matcher doesn't match and vice
- **equalTo** - test object equality using the equals method
- **is** - decorator for equalTo to improve readability
- **hasToString** - test Object.toString
- **instanceOf**, isCompatibleType - test type
- **notNullValue**, **nullValue** - test for null
- **sameInstance** - test object identity
- **hasEntry**, **hasKey**, **hasValue** - test a map contains an entry, key or value
- **hasItem**, **hasItems** - test a collection contains elements
- **hasItemInArray** - test an array contains an element
- **hasProperty** - checks if a Java Bean has a certain property can also check the value of this property
- **closeTo** - test floating point values are close to a given value
- **greaterThan**, **greaterThanOrEqualTo**, **lessThan**, **lessThanOrEqualTo**
- **equalToIgnoringCase** - test string equality ignoring case
- **equalToIgnoringWhiteSpace** - test string equality ignoring differences in runs of whitespace
- **containsString**, **endsWith**, **startsWith** - test string matching

6.3. AssertJ

```
<!-- assertj is a more recent assertion library than hamcrest
        better auto-completion with java IDE , more fluent , ... -->
    <dependency>
        <groupId>org.assertj</groupId>
        <artifactId>assertj-core</artifactId>
        <version>3.24.2</version>
        <scope>test</scope>
    </dependency>
```

Exemple :

```
import static org.assertj.core.api.Assertions.assertThat;
...

@Test
public void testPersonneWithAssertJAssertThat() {
    Personne originalPers = new Personne(2L,"Jean" , "Bon" , 172.4);
    Personne p = (Personne) BasicConverter.allPartsInLowercase(originalPers);
    logger.trace("p with allPartsInLowercase = " +  p);
    assertThat(p.getNom()).isEqualTo("bon"); //.assertThat() of AssertJ
}
```

NB : **les assertions "AssertJ" peuvent être chainées .**

Exemple :

```
String s = "Mercredi" ;
assertThat(s)
  .startsWith("Me")
  .endsWith("di")
  .isEqualToIgnoringCase("mercredi");
```

Quelques exemples de "matcher" AssertJ:

```
assertThat(obj1).isEqualTo(obj2);
assertThat(obj1).isEqualToComparingFieldByFieldRecursively(obj2);
assertThat("".isEmpty()).isTrue();
```

List<String> list = Arrays.asList("1", "2", "3");

assertThat(list).contains("1");

```
assertThat(list).isNotEmpty();
assertThat(list).startsWith("1");
```

```
assertThat(Runnable.class).isInterface();
assertThat(Exception.class).isAssignableFrom(NoSuchElementException.class);


assertThat(someFile)
    .exists()
    .isFile()
    .canRead()
    .canWrite();

assertThat(5.1).isEqualTo(5, withPrecision(1d));


assertThat(map)
  .isNotEmpty()
  .containsKey(2)
  .doesNotContainKeys(10)
  .contains(entry(2, "a"));

assertThat(listeP)
   .filteredOn(p -> p.getTag().equals("manager"))
   .containsOnly(p1,p2,p3);
```

7. Tests unitaires avec JUnit 5

7.1. Présentation de JUnit 3,4,5

JUnit est un ***framework*** simple permettant d'effectuer des **tests** (unitaires , de non régression, ...) au cours d'un développement java . [**Projet Open source ---> http://junit.sourceforge.net/ , http://junit.org]** . ***JUnit est intégré au sein des IDE Eclipse*** et ***IntelliJ***.

JUnit existe en **versions 3 , 4 et 5 avec des différences significatives d'une version à l'autre .**

7.2. Présentation des anciennes versions 3 et 4

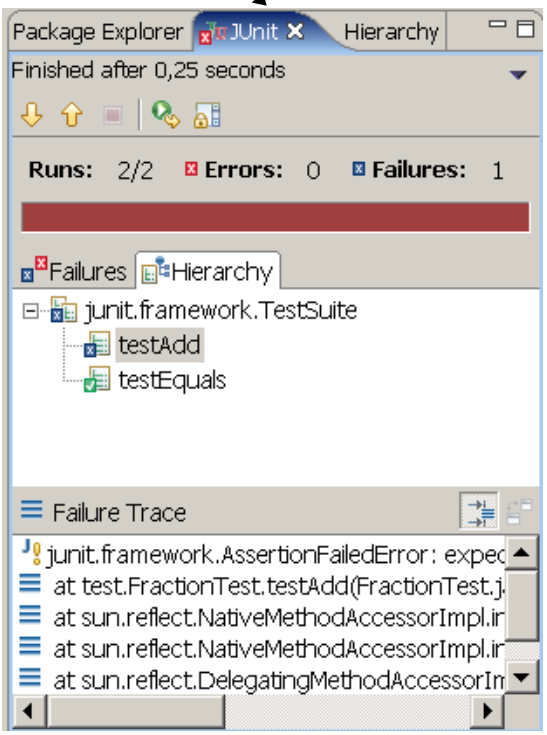
La très ancienne version 3 de JUnit n'utilisait pas d'annotations . Tout était basé sur un héritage (TestCase) et sur des conventions de nom sur les méthodes (setUp() , tearDown() , testXy()) . Depuis environ 2006/2007 , cette historique version 3 est devenue petit à petit obsolète (utilisée seulement dans les anciens projets).

La version 4 a été massivement utilisée dans la majorité des projets java de 2007 à 2019 environ . La version 4 était basée sur le package org.junit (Test, Assert, ...) et les principales annotations étaient @Test , @Before , @After , @BeforeClass , @AfterClass .

7.3. Lancement des tests unitaires

Lancement des tests unitaires

Depuis l'IDE eclipse:
Run as … / JUnit test



TestSuite en JUnit3
et lancement après une sélection de **package** en JUnit4 pour lancer d'un coup toute une série de tests.

Comptabilisations :
Error(s) : exceptions java non rattrapées.
Failure(s) : assertions non vérifiées.
VERT si aucune erreur.

Depuis "maven" :
coder des classes nommées "***Test****Xy*" ou "*XY****Test***" dans **src/test/java** et lancement via **mvn test** ou autre.

7.4. Présentation de JUnit 5 ("jupiter")

La version 5 de JUnit a été entièrement restructurée et s'appuie sur certaines nouvelles fonctionnalités apportées par la **version 8 du langage java** (lambda expressions, ...) .

Principales différences entre JUnit4 et JUnit5 :

Junit 4	Junit 5
package org.junit	package **org.junit.jupiter.api**
Assert.assertTrue() , Assert.assertEquals(...)	**Assertions**.assertTrue() , **Assertions**.assertEquals(...)
@Before , @After	**@BeforeEach , @AfterEach**
@BeforeClass , @AfterClass (avec static)	**@BeforeAll , @AfterAll(avec static)**
@Test(timeout="20")	**@Timeout()** *ou bien* **Assertions.assertTimeout(Duration.ofMillis(200), () -> { }) ;**
@Ignore	**@Disabled**
@Rule	**@ExtendWith**
@Category	**@Tag**
...	...

7.5. Configuration maven pour JUnit 5 ("jupiter")

```
...
<properties>
  <junit.jupiter.version>5.4.2</junit.jupiter.version>
</properties>
<dependencies>
<dependency>
        <groupId>org.junit.jupiter</groupId> <artifactId>junit-jupiter-api</artifactId>
        <version>${junit.jupiter.version}</version>    <scope>test</scope>
</dependency>
<dependency>
        <groupId>org.junit.jupiter</groupId>  <artifactId>junit-jupiter-engine</artifactId>
        <version>${junit.jupiter.version}</version>          <scope>test</scope>
</dependency>
</dependencies>
<build>   <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.1.2</version>  <!-- Need at least 2.22.0 to support JUnit 5 -->
     </plugin>
 </plugins> </build>  ...
```

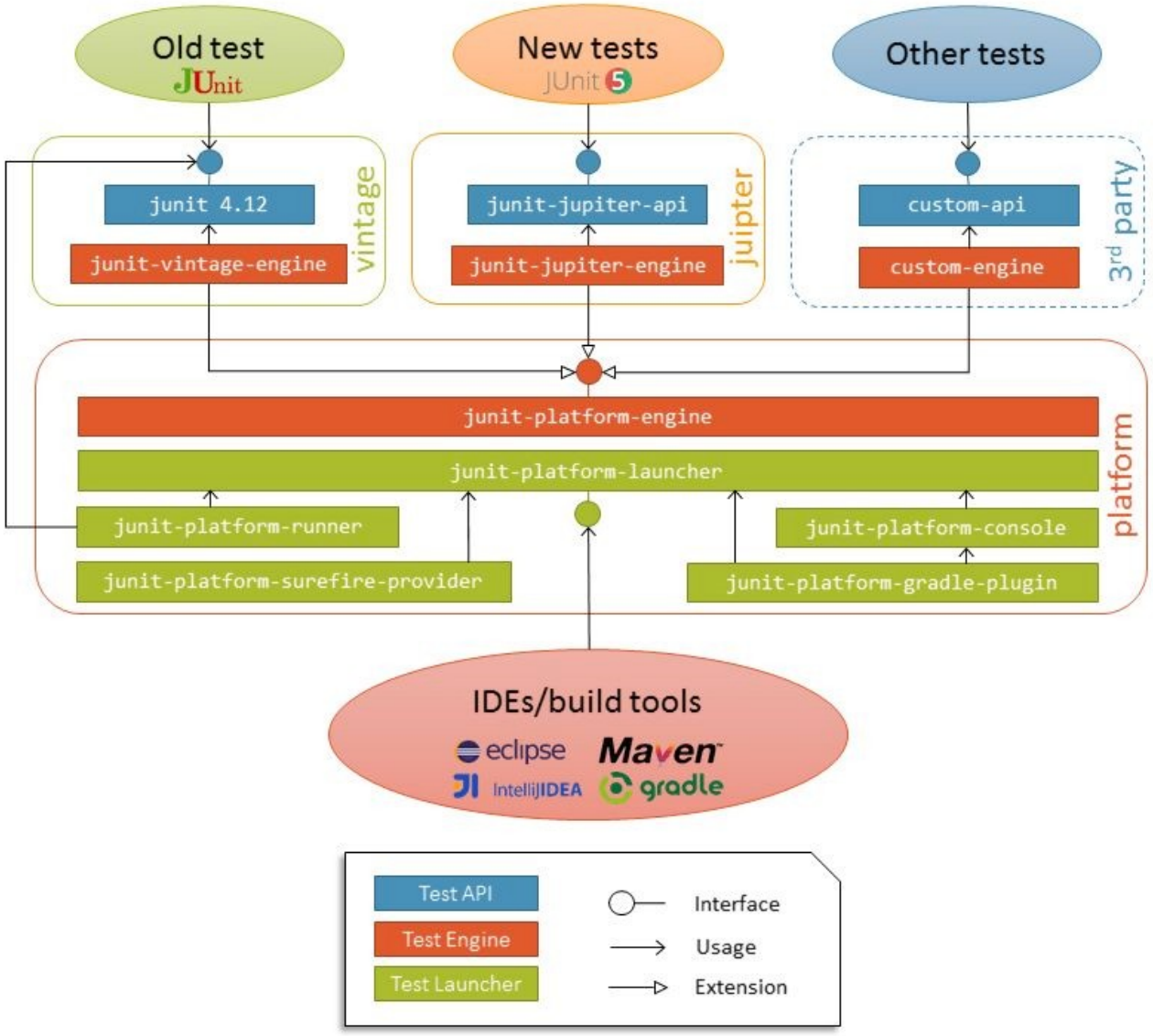
Eventuels compléments :

junit-jupiter-params *support des tests paramétrés avec JUnit Jupiter.*

junit-vintage-engine *pour interpréter et exécuter des anciens tests codés en JUnit 3 ou 4*

junit-platform-.... pour intégration et lancement (console , maven , gradle , ….)

7.6. Structure de JUnit5



7.7. Test Unitaire avec JUnit 5 (@Test, @BeforeEach, @AfterEach)

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...
public class TestCalculatrice {
	private static Logger logger = LoggerFactory.getLogger(TestCalculatrice.class);
	private CalculatriceEx calculatriceEx; //à tester

	@BeforeEach //@BeforeEach in jUnit5 , @Before in jUnit4
	public void initCalculatriceEx() {
		this.calculatriceEx = new CalculatriceEx();
	}

	@Test
	public void testSum() {
		calculatriceEx.pushVal(1.1);
		calculatriceEx.pushVal(2.2);
		calculatriceEx.pushVal(3.3);
		double somme = calculatriceEx.sum();
		logger.trace("somme(1.1, 2.2, 3.3)="+somme);
		//Assert.assert...() avec jUNit 4 et Assertions.assert...() avec jUnit5/jupiter
		Assertions.assertEquals( 6.6 , somme, 0.00000001);
	}

	@Test
	public void testAverage() {
		calculatriceEx.pushVal(1.1);
		calculatriceEx.pushVal(1.5);
		double moyenne = calculatriceEx.average();
		logger.trace("moyenne(1.1, 1.5)="+moyenne);
		Assertions.assertEquals( 1.3 , moyenne, 0.00000001);
	}
}
```

NB: Comme avec JUnit 4, **par défaut JUnit 5 créé une nouvelle instance de la classe de test pour exécuter chaque méthode de test**. (***@TestInstance(Lifecycle.PER_METHOD) par défaut***)

Assertions.**assertEquals**(***expectedValue, effectiveValue , delta***) ;
Assertions.**assertTrue**(**booleanExpression**) ; Assertions.**notNull**(...) ...

NB :
```
import static org.junit.jupiter.api.Assertions.assertTrue;
```

permet d'écrire **assertTrue** (res==5) au lieu de ***Assertions.assertTrue***(res==5)
et ça peut aider à masquer la différence de préfixe entre JUnit 4 et JUnit5 .

7.8. Test Unitaire JUnit5 avec static (@BeforeAll , @BeforeAll)

```
import org.junit.jupiter.api.BeforeAll;
...
public class TestCalculatrice {
	private static SimpleCalculatrice calculatrice; //à tester

	@BeforeAll //@BeforeAll in jUnit5 , @BeforeClass in jUnit4
	public static void initCalculatrice() {
		calculatrice = new SimpleCalculatrice();
	}

	@Test
	public void testAddition() {
		double resAdd=calculatrice.addition(5.5, 6.6);
		logger.trace("addition(5.5,6.6)="+resAdd);
		Assertions.assertTrue(resAdd>= (12.1 - 0.00000001) &&
					resAdd <= (12.1 + 0.00000001));
	}

	@Test
	public void testMultiplication() {
		double resMult=calculatrice.multiplication(2.0, 3.3);
		logger.trace("multiplication(2.0, 3.3)="+resMult);
		Assertions.assertEquals( 6.6 , resMult, 0.00000001);
	}
}
```

Variante (sans static) :

```
import org.junit.jupiter.api.* ; ...
import org.junit.jupiter.api.TestInstance.Lifecycle;

@TestInstance(Lifecycle.PER_CLASS)
//a single reused and reinitialized instance of this test class for all test methods (ex:TestPerClassInstance@aeab9a1)
public class TestPerClassInstance {
	private static Logger logger = LoggerFactory.getLogger(TestPerClassInstance.class);

	@BeforeAll
	public void initBeforeAll() {
	   logger.trace("TestPerMethodInstance.initBeforeAll() called (no static way) on this="+this );
	}

	@Test
	public void test1() {
		logger.trace("TestPerClassInstance.test1() called on this="+this );
	}
}
```

NB : En plaçant **@TestInstance(Lifecycle.PER_CLASS)** au dessus d'une classe de test Junit5 (imbriquée ou pas) , on a comme comportement le fait qu'une seule instance de la classe de test sera utilisée pour exécuter successivement toutes les méthodes de tests de la classe de test (contrairement au comportement par défaut PER_METHOD) .

7.9. Désactivation d'un test

@Ignore en V4 (org.junit.Ignore) , **@Disabled** en **V5**(org.junit.jupiter.api.Disabled)

WrongTests.java

```
package tp.converter;
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.Disabled;    import org.junit.jupiter.api.Test;

public class WrongTests {
        @Test
        @Disabled //test désactivé/ignoré
        public void testWithError() {
                int a=3;  int b=0;
                assertTrue(a/b==0);//div by zero exception/error !!!
        }

        @Test
        @Disabled  //test désactivé/ignoré
        public void testWithBadAssertion() {
                assertTrue(2+2==5); //2+2==4 !!!! not 5 !!!
        }

        @Test
        public void goodTrivialTest() { assertTrue(2+2==4);
        }
}
```

```
@Disabled("test temporairement désactivé")
sur classe ou méthode de test
```

7.10. Spécificités de JUnit5

```
@DisplayName("Ma classe de test JUnit5 que j'aime")
public class MyTest {
        @Test
        @DisplayName("Mon cas de test Xy qui va bien")
        void Testxy() {  // ... }
}
```

NB: Les valeurs des @DisplayName seront affichées au sein des rapports d'exécution des tests

NB : contrairement à JUnit4, les **méthodes** préfixées par @BeforeEach , @Test , ... **n'ont plus absolument besoin d'être public** (protected ou bien ***la visibilité implicite par défaut de niveau package suffit***) .

```
Iterable<Integer> attendu = new ArrayList<>(Arrays.asList(1, 2, 3));
Iterable<Integer> actuel = new ArrayList<>(Arrays.asList(1, 2)); //manque 3
Assertions.assertIterableEquals(attendu, actuel); //echec
```

vérifie que les collections ont mêmes tailles et mêmes valeurs.

@RepeatedTest(value = 3) est une variante de @Test permettant de lancer n fois un même test

7.11. Test avec timeout

```
@Test()
@Timeout(value = 200 , unit = TimeUnit.MILLISECONDS)
public void testWithTimeout() {
        try {
                Thread.sleep(100);//ok
                //Thread.sleep(300);//not_ok
        } catch (InterruptedException e) { System.err.println(e.getMessage());
        }
}
```

ou bien

```
@Test()
public void testWithTimeoutV2() {
        Assertions.assertTimeout(Duration.ofMillis(200),
                        () -> {     try {
                                        Thread.sleep(100);//ok
                                        //Thread.sleep(300);//not_ok
                                } catch (InterruptedException e) {
                                        System.err.println(e.getMessage());
                                }
                        });
}
```

7.12. Assertions avec lambdas :

AssertAll() avec lambdas :

```
Assertions.assertAll("wrong Dimension",
    () -> Assertions.assertTrue(elt.getWidth() == 400, "wrong width"),
    () -> Assertions.assertTrue(elt.getHeight() == 500, "wrong height"));
```

vérifie que chacune des lambdas est ok (sans exception) .

AssertThrows avec lambda :

```
@Test
 void verifierException() {
   String valeur = "123a";
   NumberFormatException  nfe  =
      Assertions.assertThrows(NumberFormatException.class,
                                () -> {
                                        Integer.valueOf(valeur);
                                });
 }
```

vérifie qu'une exception est bien levée (suite à erreur volontaire) et du bon type .

AssertTimeout avec lambda :

```
@Test
 void verifierTimeout() {
  Assertions.assertTimeout(Duration.ofMillis(200),
                                () -> {   * traitement potentiellement trop lent *    return "...";
                                });
}
```

7.13. Suppositions (pour exécuter ou pas un test selon le contexte) :

```
assumeTrue(  System.getenv("OS").startsWith("Windows") );
//la suite du test ne sera exécutée que si os courant est Windows
//si le test n'est pas exécuté --> même comportement que si test vide --> ok/réussi/vert
```

```
//variante avec lambda exécutée qui si supposition ok :
assumingThat(System.getenv("OS").startsWith("Windows"), () -> {
    assertTrue(new File("C:/Windows").exists(), "Repertoire Windows inexistant");
  });
```

7.14. Tests imbriqués de JUnit5

```
public class MyTest {

@BeforeEach
   void mainInit() {
     System.out.println("BeforeEach / first level");
   }

 @Nested
  class MonTestImbrique {
    @BeforeEach
    void subInit() {
      System.out.println("BeforeEach imbrique");
      valeur = 5;
    }

    @Test
    void simpleTestImbrique() {
      System.out.println("SimpleTest imbrique valeur=" + valeur);
      Assertions.assertEquals(5, valeur);
    }
```

```
 }

}
```

NB : On peut éventuellement utiliser le mode PER_CLASS au niveau de la classe comportant des sous classes annotées avec @Nested pour contrôller le nombre d'appels effectués (au niveau des méthodes préfixées par @BeforeAll et @AfterAll).

7.15. Tests paramétrés via source/série de valeurs

Après avoir ajouté la dépendance nécessaire ***junit-jupiter-params*** , on peut écrire des méthodes de tests avec un paramètre qui sera alimenté avec une source/série de valeurs .

Un test paramétré sera ainsi lancé plusieurs fois (avec des arguments différents pour obtenir une certaine variété)

```
@ParameterizedTest
 @ValueSource(ints = { 1, 2, 3 })
 void testParametreAvecValueSource(int valeur) {
  assertEquals(valeur + valeur, valeur * 2);
 }
```

@ValueSource(ints = { 1, 2, 3 }) ou @ValueSource(strings = { "un", "deux" }) ou ...

import java.util.stream.Stream;

```
@ParameterizedTest
 @MethodSource("fournirDonneesParametres")
 void testTraiterSelonMethodSource(String element) {
  assertTrue(element.startsWith("elt"));
 }


 static Stream<String> fournirDonneesParametres() {
  return Stream.of( "elt1" , "elt2" );
 }
```

```
public enum Direction  { NORD , SUD, EST , OUEST  }
@ParameterizedTest
	@EnumSource(TestParameterizedWithSources.Direction.class)
	void testTraiterSelonEnumValues(Direction direction) {
		assertTrue(direction.toString().length()>=3);
		logger.trace("testTraiterSelonEnumValues direction="+direction);

	}
```

```
@ParameterizedTest
	@CsvSource(delimiter = ';',
			value= { "1;1;un" ,
				 "2;4;quatre" ,
				 "3;9;neuf"
				}
	)
	void testTraiterSelonCsvSource(int n,int expectedCarre,String comment) {
		assertTrue(n*n == expectedCarre);
		logger.trace("testTraiterSelonCsvSource n="+n
				+ " expectedCarre="+expectedCarre
				+ " comment="+comment);

	}
```

```
@ParameterizedTest
	@CsvFileSource( resources= "/carres.csv", delimiter = ';', numLinesToSkip = 1)
	//files= "carres.csv" found at eclipse project root
	//resources= "/carres.csv" found in src/test/resources
	void testTraiterSelonCsvFileSource(int n,int expectedCarre,String comment) {
		assertTrue(n*n == expectedCarre);
		logger.trace("testTraiterSelonCsvFileSource n="+n
				+ " expectedCarre="+expectedCarre
				+ " comment="+comment);

	}
```

src/test/resources/carres.csv

```
n;carre;comment
1;1;un
2;4;quatre
3;9;neuf
```

7.16. Tests dynamiques / @TestFactory

```
...
import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.ArrayList;
import java.util.Collection;
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

class MonTestSimple {

  @TestFactory
  Collection<DynamicTest> dynamicTestsWithCollection() {
    Collection<DynamicTest> myDynamicTests = new ArrayList<>();
    for (int i = 1; i <= 5; i++) {
      int val = i;
      myDynamicTests.add(DynamicTest.dynamicTest("Ajout " + val + "+" + val,
          () -> assertEquals(val * 2, val + val)));
    }
    return myDynamicTests;
  }
}
```

--> **à priori intéressant que si couplé avec une introspection** dynamique (framework de tests).

7.17. Suite de Tests (JUnit 5)

Durant la période de transition JUnit4 vers JUnit5 , une suite de Tests pouvait être codée de cette manière :

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


//NB: JUnitPlatform nécessite org.junit.platform:junit-platform-runner:1.9.3 dans pom.xml
//    @RunWith() nécessite org.junit.vintage:junit-vintage-engine:5.9.3 dans pom.xml

@RunWith(JUnitPlatform.class)
@SelectClasses({
        TestCalculsSimples.class,
        TestEmptySerie.class,
        TestSerie.class
})
public class CalculsSuiteV1 {
}
```

Au sein d'un projet récent (prise en charge au sein d'un IDE récent) , une suite de tests moderne junit5 sera plutôt codée de cette manière :

```
...
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;

//@RunWith(JUnitPlatform.class) will be deprecated , use @Suite instead
//but @Suite not already supported by some old IDE (ex: ok with eclipse 2023-03)


@Suite
@SelectClasses({
	TestCalculsSimples.class,
	TestEmptySerie.class,
	TestSerie.class
})
public class CalculsSuite {
}
```

```
@Suite
@SelectPackages({"tp.calculs", "tp.converter" })
public class AllTestsSuite {
}
```

7.18. Suite de Tests avec Tags

@Tag("nomDeTag") de JUnit5 peut être placé sur une classe ou bien une méthode de test

Ces noms de tags (préalablement appelés catégories en Junit4) permettront d'effectuer ultérieurement des filtrages sur les tests à lancer ou pas .

Il est éventuellement possible de placer plusieurs tags complémentaires au dessus d'un même test

(ex : @Tag("**prod**") @Tag("***dev***")) .

Exemple :

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

public class TestWithCategoryTag {
	private static Logger logger = LoggerFactory.getLogger(TestWithCategoryTag.class);

	@Test
	@Tag("SlowTests")
	public void testA() {
		logger.trace("slow testA");
	}
```

```
	@Test
	@Tag("FastTests")
	public void testB() {
		logger.trace("fast testB");
	}

	@Test
	@Tag("FastTests")   @Tag("ImportantTests")
	public void testC() {
		logger.trace("important fast testC");
	}

	@Test
	@Tag("SlowTests")   @Tag("SecondaryTests")
	public void testD() {
		logger.trace("secondary slow testD");
	}
}
```

```
...
import org.junit.platform.suite.api.ExcludeTags;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;

@Suite
//@IncludeTags({"FastTests"})
@ExcludeTags({"SlowTests", "SecondaryTests"})
@SelectClasses({
	TestWithCategoryTag.class ,
	TestWithAssumptions.class
})
public class FastTestsSuite {
}
```

NB :

- par défaut , @SelectPackages(...) sélectionne tous les sous-packages également .
- On peut éventuellement ajouter @IncludePackages(...) ou bien @ExcludePackages(...) pour filtrer les sous-packages à sélectionner.
- On peut également ajouter @IncludeClassNamePatterns({ "^.*Simple$" }) pour filtrer les noms des classes de Tests à lancer .
- Possibilité de filter les category/tags au sein de la config maven:

```
<plugin>
	<artifactId>maven-surefire-plugin</artifactId>
	<configuration>
	  <properties>
	    <includeTags>FastTests</includeTags>
	  </properties>
	</configuration>
 </plugin>
```

7.19. Extensions JUnit5

L'ancien concept de règle (@Rule) JUnit4 a été remplacé par la notion d'extension (plus clair) en JUnit5 .

Exemple d'extension personnalisée :

```
package tp.extensions;
import org.junit.jupiter.api.extension.AfterEachCallback;
import org.junit.jupiter.api.extension.BeforeEachCallback;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.api.extension.ExtensionContext.Namespace;
import org.slf4j.Logger;   import org.slf4j.LoggerFactory;

//NB1: just one instance of this class
//(don't use attributes but context.getStore(Namespace.GLOBAL or SPECIFIC_NAMESPACE))
//NB2: JUnit5 Extensions seems to be rather slow (may be aop/weaver) ...
public class MyPerfCustomExtension implements AfterEachCallback, BeforeEachCallback {
        private static Logger logger = LoggerFactory.getLogger(MyPerfCustomExtension.class);
        private static final Namespace NAMESPACE =
                Namespace.create("tp", "extensions", "MyPerfCustomExtension");


        @Override
        public void beforeEach(ExtensionContext context) throws Exception {
                long startTime =System.nanoTime();
                logger.info("MyPerfCustomExtension, before execution of "
                        +context.getDisplayName()
                        + " at startTime=" +startTime );
                Namespace methodNamespace =
                        NAMESPACE.append(context.getDisplayName());
                context.getStore(methodNamespace).put("startTime", startTime);
        }

        @Override
        public void afterEach(ExtensionContext context) throws Exception {
                long endTime = System.nanoTime();
                Namespace methodNamespace =
                        NAMESPACE.append(context.getDisplayName());
                long startTime = (Long) context.getStore(methodNamespace).get("startTime");
                logger.info("MyPerfCustomExtension, after execution of "
                                +context.getDisplayName()
                                + " execDuration (ms) without good precision = "
                                + ((double)(endTime - startTime)/1000000)
                                + " startTime=" +startTime  + " endTime=" +endTime );
        }
}
```

Utilisation d'une extension JUnit5 :

```
package tp.converter.other;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.slf4j.Logger; import org.slf4j.LoggerFactory;
import tp.extensions.MyPerfCustomExtension;

@ExtendWith(MyPerfCustomExtension.class)
public class TestWithExtension {
	private static Logger logger = LoggerFactory.getLogger(TestWithExtension.class);


	@Test
	public void fastTest() {
		try {
			Thread.sleep(20);//ok
		} catch (InterruptedException e) {
			System.err.println(e.getMessage());
		}
	}

	@Test
	public void slowTest() {
		try {
			Thread.sleep(60);//ok
		} catch (InterruptedException e) {
			System.err.println(e.getMessage());
		}
	}
}
```

Résultats :

MyPerfCustomExtension, before execution of slowTest() at startTime=510085534119300

*MyPerfCustomExtension, after execution of slowTest() execDuration (ms) without good precision = **92.514** startTime=510085534119300 endTime=510085626633300*

MyPerfCustomExtension, before execution of fastTest() at startTime=510085636236500

*MyPerfCustomExtension, after execution of fastTest() execDuration (ms) without good precision = **35.5131** startTime=510085636236500 endTime=510085671749600*

Possibilité d'utiliser conjointement plusieurs extensions complémentaires :

```
@ExtendWith({ MyPerfCustomExtension.class , MyBasicLogExtension.class })
public class TestWithExtension {
…
}
```

Principales extensions prédéfinies de JUnit5

Extensions	Caractéristiques
TempDirectory et @TempDir	**Destruction automatique (en fin de tests) des fichiers et répertoires temporaires créés durant le test.** NB : @TempDir peut être placé sur un attribut (static ou pas) d'une classe ou bien peut être placé sur un paramètre d'entrée d'une méthode de test
...	

Exemples :

```
import org.junit.jupiter.api.io.TempDir;
...

@TempDir
File tempDir;

@Test
void testWithTempDir() throws IOException {
   assertTrue(this.tempDir.isDirectory() , "Should be a directory ");

   File f1 = new File(tempDir, "f1.txt");
   List<String> lines = Arrays.asList("aaa", "bbb", "ccc");

   Files.write(f1.toPath(), lines);

   assertAll(
     () -> assertTrue(Files.exists(f1.toPath()) , "File should exist",),
     () -> assertLinesMatch(lines, Files.readAllLines(f1.toPath())));
}
```

7.20. Test(s) dans interface (contrat d'interface)

Code à tester :

```
public interface IDecorate {
        //returned String must contains inputString
        //and may be decorated by any prefix or any suffix or both
        String decorate(String inputString);
}
```

```
public class DecorateWithPrefix implements IDecorate{
        @Override
```

```
	public String decorate(String inputString) {
		return ">>>"+inputString;
	}
}
//---------------------------------------------------------------
public class DecorateWithSuffix implements IDecorate{
	@Override
	public String decorate(String inputString) {
		return inputString+"<<<";
	}
}
```

```
package tp.other;
import static org.assertj.core.api.Assertions.assertThat;
import org.junit.jupiter.api.Test;
import org.slf4j.Logger;import org.slf4j.LoggerFactory;

public interface IDecorateContract {

   static final Logger logger = LoggerFactory.getLogger(IDecorateContract.class);
   IDecorate create();

   @Test
   default void postcondition() {
      String parameter = "abc"; //or random string
      String res = create().decorate(parameter);
      logger.trace("res="+res);
      assertThat(res).isNotBlank()
                     .contains(parameter);
   }

}
```

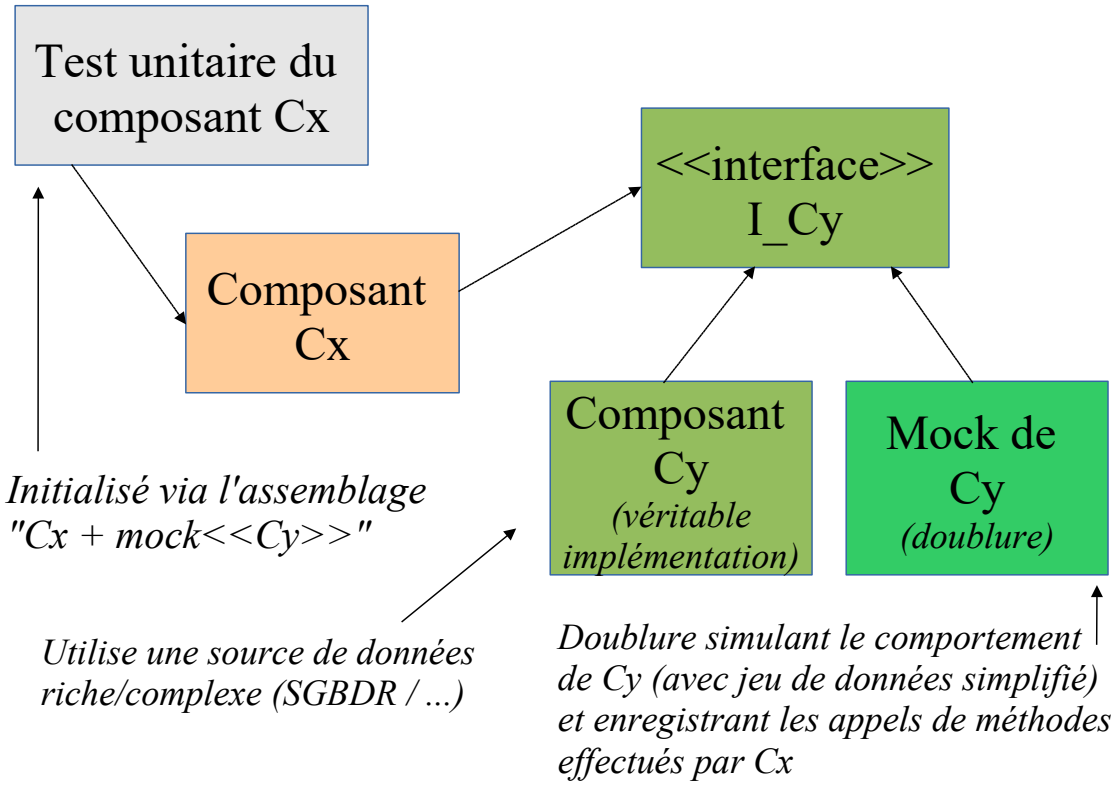
```
...
public class TestDecorateWithSuffix implements IDecorateContract{
	@Override
	public IDecorate create() {
		return new DecorateWithSuffix();
	}
	//AVEC DECLENCHEMENT AUTOMATIQUE
	 //du @Test postcondition() defini dans l'interface IDecorateContract
}
```

... TRACE IDecorateContract:20 - res=>>>abc

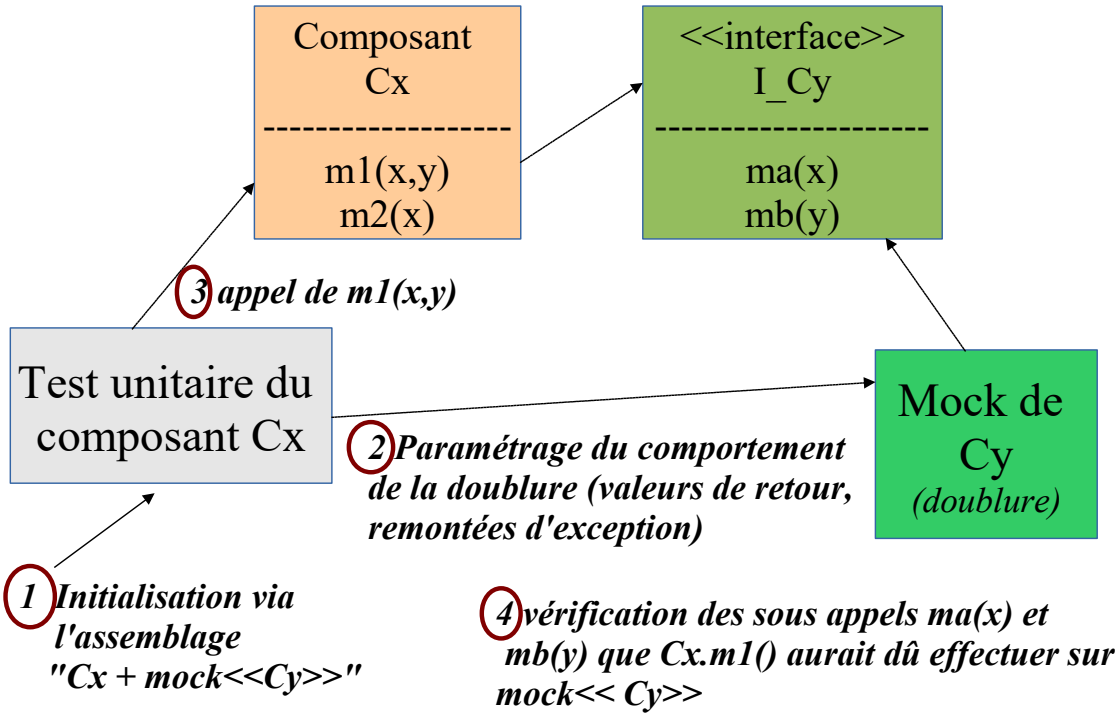
III - Mockito (un des frameworks "mock" java)

1. Positionnement et intérêts des "mocks"

Positionnement des "mocks"



intérêts des "mocks"



2. Mockito

2.1. Présentation de Mockito

Mockito est une technologie java assez populaire pour mettre en œuvre des "Mocks" (simulacres).

2.2. Dépendance maven nécessaire

```
<dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <!-- <artifactId>mockito-all</artifactId> -->
        <version>1.10.19</version>
</dependency>
```

2.3. Initialisation du Mock au niveau d'un test Junit:

Solution1 (par annotation ***"@Mock"*** interprétée du fait de ***@RunWith***(*MockitoJUnitRunner.class*)) :

```
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class UserLoginMockTest {

  @Mock
   private static UserLogin userLogin;
....
```

Solution2 (par annotation "***@Mock***" interprétée du fait de l'appel à *MockitoAnnotations.**initMocks**(this);* lors de l'initialisation) :

```
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

//rien ou @RunWith(SpringJUnit4ClassRunner.class)
public class UserLoginMockTest {

   @Mock
   private  UserLogin userLogin;

   @Before
        public void init() /* or setUp or ... */ {
        MockitoAnnotations.initMocks(this);
   }  …
```

Solution3 (***sans annotation*** Mockito , avec un appel explicite à ***Mockito.mock()***) :

```
import org.mockito.Mockito;
public class UserLoginMockTest {
   private static UserLogin userLogin;

   @BeforeClass
   public static void init(){
         userLogin = Mockito.mock(UserLogin.class);
   }
….
```

Soit l'interface fonctionnelle suivante :

```
public interface UserLogin {

   public boolean verifyLogin(String username,String password);
   public String goodPasswordForUser(String username);
   public void setSize(int size);
   //pour tester valeur de retour par défaut:
   public String getAuteur();
   public int getSize(); //return nbAccount ( >=0)
   public double getDoubleValue(double x); //return x * 2
}
```

et soit une classe d'implémentation basique suivante :

```
public class UserLoginImpl implements UserLogin {
        private int size=10;  //par défaut
        public boolean verifyLogin(String username, String password) {
                boolean res=false;
                if(password !=null && password.equals("pwd_"+username))
                        return true;
                return res;
        }
        public String goodPasswordForUser(String username) {
                        return "pwd_"+username;
        }
        public void setSize(int size) { this.size=size;
        }
        public String getAuteur() {    return "didier";
```

```
        }
        public int getSize() {  return size ;
        }
        public double getDoubleValue(double x) {   return 2 *x;
        }
}
```

Le comportement de Mockito est alors le suivant :

2.4. Mockito as stub

Comportement par défaut d'un mock (géré par Mockito)

Avec aussi bien
userLogin = Mockito.mock(UserLogin.class); **//interface**
que
userLogin = Mockito.mock(UserLoginImpl.class); **//classe d'implémentation**
tout appel de méthode sur l'objet "mock" géré par mockito retourne une valeur par défaut de type "null" , false , 0 ou 0.0 selon le type de retour :

```
public void displayReturnValues(){
    boolean pwdOk= userLogin.verifyLogin("toto", "pwd_toto");
    System.out.println("pwdOk="+pwdOk);

    String goodPwd= userLogin.goodPasswordForUser("toto");
    System.out.println("goodPwd="+goodPwd);

    String auteur = userLogin.getAuteur();
    System.out.println("auteur="+auteur);

    int taille = userLogin.getSize();
    System.out.println("taille="+taille);

    double  val = userLogin.getDoubleValue(3.2);
    System.out.println("val="+val);
  }
```

Comportement vraie classe:

pwdOk=*true*
goodPwd=*pwd_toto*
auteur=*didier*
taille=*10*
val=*6.4*

Comportement du mock:

pwdOk=**false**
goodPwd=**null**
auteur=**null**
taille=**0**
val=**0.0**

Préciser (forcer) une valeur de retour via Mockito :

```
Mockito.when(userLogin.getSize()).thenReturn(5);
```

int taille = userLogin.getSize();
System.out.println("taille="+taille); → affiche toujours taille=5
userLogin.setSize(20); taille = userLogin.getSize();
System.out.println("taille="+taille); → affiche toujours taille=5 (et pas 20 !)

On peut forcer un **retour d'exception** selon par exemple certaines valeurs en entrée :

```
Mockito.when(userLogin.setSize(Mockito.eq(-1)))
   .thenThrow(new IllegalArgumentException("message xy"));
```

Lorsque l'on "mock" une classe (et pas une interface), on peut explicitement demander à Mockito de rétablir le comportement de la véritable classe d'implémentation sur certaines méthodes :

```
Mockito.when(userLogin.getSize()).thenCallRealMethod();
Mockito.doCallRealMethod().when(userLogin).setSize(Mockito.anyInt());
// il existe aussi  Mockito.anyString() , ...
```

userLogin.setSize(20) ; taille = userLogin.getSize();
System.out.println("taille="+taille); → affiche taille=20

2.5. Mockito as "spy" et "mock = stub + spy"

Comportement par défaut d'un mock initialisé via Mockito.spy()

Fonctionnellement : stub = bouchon (comportement simulé) avec 0,null,0.0 par défaut
.spy() = espionnage (enregistrement des appels pour vérifications ultérieures).
.mock() = comportement "stub" + fonctionnalité de .spy()

---

userLogin = **Mockito.spy**(*UserLogin.class*); //**interface**
==> même comportement que via Mockito.mock() car pas de code par défaut

---

```
userLogin = Mockito.spy(new UserLoginImpl()); //classe d'implémentation
```

On obtient alors un **comportement normal** (identique à la classe d'origine) **sur toutes les méthodes sauf sur celles où l'on demande explicitement à redéfinir le comportement** :

```
Mockito.when(userLogin.getSize()).thenReturn(5);
```

Résultats (par défaut) depuis code précédent :
pwdOk=true
goodPwd=pwd_toto
auteur=didier
taille=**5** *(à la place de taille = 10)*
val=6.4

On peut désactiver le comportement d'un setter (ou d'une méthode en void) :

```
Mockito.doNothing().when(userLogin).setSize(Mockito.anyInt());
```

Autrement dit :

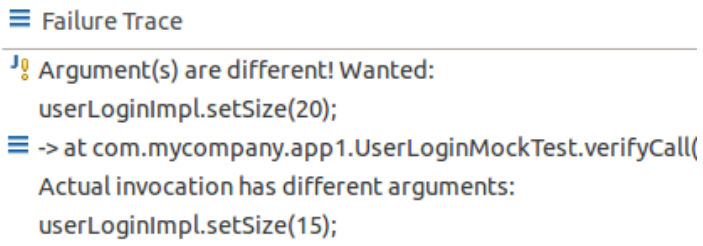
- sémantiquement ***mock = "stub + spy"***
- pragmatiquement pas de différence notable entre Mockito.mock() et Mockito.spy() en partant d'une interface car si pas de code d'implémentation ---> toujours comportement "mock=stub+spy" .
  Si par contre Mockito.spy() est appelé en partant d'une réelle instance de composant alors :
  - spy --> espionnage seulement pour ultérieur "verify"
    sans changer comportement de l'application
    et mock (= stub + spy)--> verify possible ET implémentation de départ écrasée par Mockito.when().then...

Vérification des appels effectués sur un mock (spy) :

Mockito.spy(...) porte bien son nom lorsque l'on sait que l'**on peut demander à Mockito d'espionner les appels effectués sur un "mock" et vérifier si certaines méthodes ont bien été appelées (avec certaines valeurs attendues de paramètres en entrée)** :

```
@Test
   public void verifyCall(){
      userLogin.setSize(15);  //userLogin.setSize(20);
       //appel habituellement indirect effectué depuis
      // le code caché d'un composant à tester
      // vers le "spy" ou "stub+spy=mock"  d'un sous composant
      Mockito.verify(userLogin).setSize(Mockito.eq(20));
   }
```

Avec setSize(15)
au lieu de setSize(20)



Quelques exemples de vérifications via Mockito

```
// vérifie que la méthode m1 a été appelée sur obj,
// avec une String strictement égale à "s1" :
Mockito.verify(obj).m1(Mockito.eq("s1"));
// note : ici, le matcher n'est pas indispensable, la ligne suivante est équivalente :
Mockito.verify(obj).m1("s1");

// vérifie que la méthode m2 n'a jamais été appelée sur l'objet obj :
Mockito.verify(obj, Mockito.never()).m2();

// vérifie que la méthode m3 a été appelée exactement 2 fois sur l'objet obj :
Mockito.verify(obj, Mockito.times(2)).m3();

// idem avec un nombre minimum et maximum d'appels :
Mockito.verify(obj, Mockito.atLeast(3)).m3();
Mockito.verify(obj, Mockito.atMost(10)).m3();

// vérifie que la méthode m4 a été appelée sur obj,
// avec un objet similaire à celui passé en argument :
Mockito.verify(obj).m4(Mockito.refEq(obj2));
```

Quelques "matchers" pour vérifier ou paramétrer les valeurs des paramètres :

Mockito.**eq**(...)	Ègal à ...
Mockito.**refEq**(obj2)	Égal à cet objet
Mockito.**anyString**() , **anyInt**() , **anyFloat**() , ….	Chaîne quelconque , entier quelconque, ..
Mockito.**anyObject**()	Objet quelconque
Mockito.**any**(Class<T> c)	Objet d'un certain type
Mockito.**anyList**()	Toute implémentation de List
Mockito.**argThat**(new MyMatcher())	Vérifiant matcher spécifique
...	

On peut définir de nouveaux "matcher" via des classes qui héritent de **ArgumentMatcher**<T>

Exemple de "matcher" personnalisé/spécifique:

```
import org.hamcrest.Description;    import org.hamcrest.Matcher;

public class MyIntegerBetween implements Matcher<Integer>{
     private double inclusiveMini;
     private double exclusiveMaxi;

     public MyIntegerBetween() { super();
          this.inclusiveMini = 0;    this.exclusiveMaxi = 100;
     }

     public MyIntegerBetween(double inclusiveMini, double exclusiveMaxi) {
          super();   this.inclusiveMini = inclusiveMini;
                     this.exclusiveMaxi = exclusiveMaxi;
     }

     @Override
     public boolean matches(Object arg0) {
          Integer x= (Integer) arg0;
          if(x>= inclusiveMini && x < exclusiveMaxi)
               return true;
          /*else*/
          return false;
     }
...}
```

Utilisation :

```
Mockito.when(serviceTauxCourants.tauxMensPctCourant(
        Mockito.intThat(new MyIntegerBetween(97,1000)))).thenReturn(0.1);
```

IV - Tests de comportements / fonctionnels

1. Tests comportementaux (fonctionnels)

Test comportemental (d'acceptation fonctionnelle)

TDD = Test Driven Development
du coté "***développement***" : JUnit + ...
du coté "***acceptation fonctionnelle***" : *JBehave* ou *easyb* ou *cucumber* ou ....

ATDD = Acceptance Test Driven Development
BDD = Behavior Driven Development (synonyme de ATDD)

BDD (ou ATDD):

users_stories (fichiers ***".story"*** idéalement ***accrochés aux Uses Cases UML***)
= liste de scénarios rédigés de la façon suivante:

> **scénario xyz:**
>
> **Given** ***contexte***
> **When** ***événement ou condition***
> **Then** ***comportement attendu***

*Au départ: simple ***partie des spécifications fonctionnelles***
*Au final (***avec technologie annexe telle que jBehave***): ***réel test (exécutable) d'acceptation fonctionnelle***

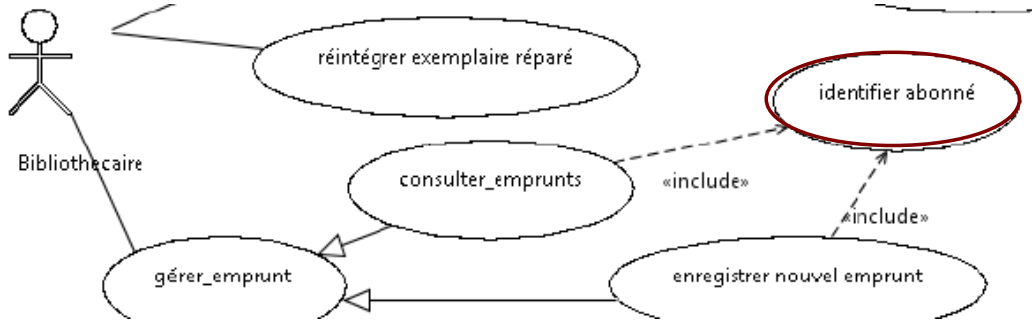
Test comportemental (d'acceptation fonctionnelle)

Certaines technologies (ex: JBehave en java , ... en ruby , ...) **sont prévues pour ré-exploiter les fichiers ".story" et déclencher tous les "test d'acceptation"**

Un rapport est généralement produit automatiquement.

Dans le cas particulier de la technologie java "JBehave" ,
un développeur code une ***classe java en arrière plan de chaque scénario*** avec des **annotations @Given , @When , @Then** de façon à **associer le scénario à de véritables étapes (steps) d'un test exécutable** (que l'on peut facilement re-déclencher plusieurs fois).

Exemple (dans spécifications fonctionnelles)



Exemple de "user-stories" pour UC "identifier abonné" :

```
scenario abonnement valide
given abonne déjà enregistré avec dernier réabonnement depuis
moins d'un an
when identifiant valide
then statut ABONNEMENT_VALIDE and message bien identifié

scenario abonnement expiré
given abonné déjà  enregistré mais dernier réabonnement depuis
plus d'un an
when identifiant valide
then statut ABONNEMENT_EXPIRE
  and message abonnement expiré (à  renouveler)
```

JBehave (utilisation)

stories/xy**.story**

```
Scenario xy :
Given ctxVal a=6
When evtMult b=5
Then resMultAttendu=30
```

steps/xySteps**.java**

```
public class XySteps {
private … contextData1 ;
private … statefulData2 ;
@Given("ctxVal a=$a")
public void initContextValue(...a)
   contextData1=a ; ...
}
@When("evtMult b=$b")
public void lorsqueMultPar(...b)
  statefulData2=b*contextData1 ;
}
@Then("resMultAttendu=$val")
public void verifResult(...val)
 Assert.assertTrue(statefulData2==val) ;
}...}
```

```
class MyAbstractJBehaveStories
  extends JunitStories  {
@Override
public Configuration configuration() {...}
@Override
protected List<String> storyPaths() {
        return … "**/*.story" ;
    }
@Override
public InjectableStepsFactory stepsFactory() {
   return …. new XySteps()...
        }
}
```

⇒ JUnit success or failure
+ *HTML report*

2. Tests comportementaux avec Cucumber

A coté de JBehave , Cucumber est l'une des technologies java qui permet de rendre exécutable un test fonctionnel (exprimé avec Given ,When , Then).

Cucumber est utilisable dans plein de langage de programmation (java , js, c++ , …) et est ainsi considéré comme la technologie BDD de référence.

2.1. Configuration maven pour "cucumber+junit"

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>tp</groupId>
  <artifactId>bdd_cucumber</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <maven.compiler.release>11</maven.compiler.release>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <junit.jupiter.version>5.9.3</junit.jupiter.version>
    <junit.platform.version>1.9.3</junit.platform.version>
    <cucumber.version>7.12.1</cucumber.version>
</properties>


    <dependencyManagement>
      <dependencies>
        <dependency>
          <groupId>io.cucumber</groupId>
          <artifactId>cucumber-bom</artifactId>
          <version>${cucumber.version}</version>
          <type>pom</type>
          <scope>import</scope>
        </dependency>
        <dependency>
          <groupId>org.junit</groupId>
          <artifactId>junit-bom</artifactId>
          <version>${junit.jupiter.version}</version>
          <type>pom</type>
          <scope>import</scope>
        </dependency>
      </dependencies>
```

```
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>io.cucumber</groupId>
      <artifactId>cucumber-java</artifactId>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>io.cucumber</groupId>
      <artifactId>cucumber-junit-platform-engine</artifactId>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-suite</artifactId>
      <scope>test</scope>
    </dependency>

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

      <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-slf4j-impl</artifactId>
      <version>2.15.0</version>
   </dependency>

  </dependencies>


<build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.1.2</version>
            </plugin>
        </plugins>
</build>

</project>
```

2.2. Configuration des scénarios de tests

Ces fichiers doivent idéalement être placés dans **src/test/resources** et dans des sous répertoires qui correspondent à certains packages java .

is_it_friday_yet**.feature**

```
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday

  Scenario Outline: Today is or is not Friday
    Given today is "<day>"
    When I ask whether it's Friday yet
    Then I should be told "<answer>"

  Examples:
    | day               | answer |
    | Friday            | Yes    |
    | Sunday            | No     |
    | anything else!    | No     |
```

compte**.feature**

```
Feature: CompteBancaire bien géré
  Gestion de compte bancaire (debiter, crediter)

  Scenario Outline: Decouvert autorisé ou pas
    Given soldeInitial=<soldeInitial>
    When debiter montant=<montant>
    Then nouveauSolde=<nouveauSolde>
    And statut="<statut>"
    And AvecException=<AvecException>

  Examples:
    | soldeInitial | montant | nouveauSolde | statut          |AvecException |
    | 100          | 200     | -100         | A_DECOUVERT |false         |
    | 400          | 300     | 100          | OK              |false         |
    | 100          | 500     | 100          | OK              |true          |
    | -100         | 400     | -100         | A_DECOUVERT |true          |
```

2.3. Exemple de code à tester :

tp.hello.IsItFriday.java

```
package tp.hello;
public class IsItFriday {
	static String isItFriday(String today) {
		return "Friday".equals(today)?"Yes":"No";
   }
}
```

tp.bank.Compte.java

```
package tp.bank;

public class Compte {
	public enum Status { OK, A_DECOUVERT }
	public static double DECOUVERT_AUTORISE = -300.0 ;
	private Long numero;
	private String label;
	private Double solde;

	public Compte(Long numero, String label, Double solde) {
		this.numero = numero ;  this.label = label; this.solde = solde;
	}
	public Compte() {	this(null,null,0.0);  }

	public void crediter(double montant) {
		this.solde += montant;
	}

	public void debiter(double montant) {
		double nouveauSolde = this.solde -montant;
		if(nouveauSolde >= DECOUVERT_AUTORISE)
			this.solde = nouveauSolde;
		else throw new RuntimeException("decouvert trop important pas accepté");
	}

	//+get/set, .toString() , ...
	public Status getStatut() {
		if(this.solde>=0)
		  return Status.OK;
		else
		  return Status.A_DECOUVERT;
	}

}
```

2.4. Définitions des "StepDefinitions cucumber"

src/test/java/tp.hello.IsItFridayStepDefinitions.java

```
package tp.hello;

import static org.junit.jupiter.api.Assertions.assertEquals;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import tp.hello.IsItFriday;

public class IsItFridayStepDefinitions {
    //before implementation:
    //throw new io.cucumber.java.PendingException();

    private String today;
    private String actualAnswer;

    @Given("today is {string}")
    public void today_is(String today) {
        this.today = today;
    }

    @When("I ask whether it's Friday yet")
    public void i_ask_whether_it_s_Friday_yet() {
        actualAnswer = IsItFriday.isItFriday(today);
    }

    @Then("I should be told {string}")
    public void i_should_be_told(String expectedAnswer) {
        assertEquals(expectedAnswer, actualAnswer);
    }
}
```

src/test/java/tp.bank.CompteStepDefinitions.java

```
package tp.bank;

import static org.junit.jupiter.api.Assertions.assertEquals;
import org.slf4j.Logger;  import org.slf4j.LoggerFactory;
import io.cucumber.java.ParameterType;
import io.cucumber.java.en.And;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class CompteStepDefinitions {

   private static Logger logger = LoggerFactory.getLogger(CompteStepDefinitions.class);
```

```
  private Compte compte=null;
  private Boolean avecException =false;

  @Given("soldeInitial={double}")
  public void given_soldeInitial(double soldeInitial) {
     this.compte = new Compte(1L,"compte_1" , soldeInitial);
     logger.trace("given_soldeInitial="+soldeInitial + " compte="+compte);
  }

  @When("debiter montant={double}")
  public void when_debiter(double montant) {
        logger.trace("when_debiter montant="+montant );
              try {
                    this.compte.debiter(montant);
                    this.avecException=false;
              } catch (RuntimeException e) {
                    logger.error(e.getMessage());
                    this.avecException=true;
                    //e.printStackTrace();
              }
  }

  @Then("nouveauSolde={double}")
  public void then_nouveauSolde(Double expectedNouveauSolde) {
     assertEquals(expectedNouveauSolde, compte.getSolde());
     logger.trace("then_nouveauSolde="+compte.getSolde() );
  }

  //@And or @Then
  @Then("statut={string}")
  public void and_status_is(String expectedStatus) {
     assertEquals(expectedStatus, compte.getStatut().toString());
  }


  @ParameterType(value = "true|True|TRUE|false|False|FALSE")
        public Boolean booleanValue(String value) {
           return Boolean.valueOf(value);
        }

  //@And or @Then
  @Then("AvecException={booleanValue}")
  public void and_AvecException_is(Boolean expectedAvecException) {
     assertEquals(expectedAvecException, avecException);
  }
}
```

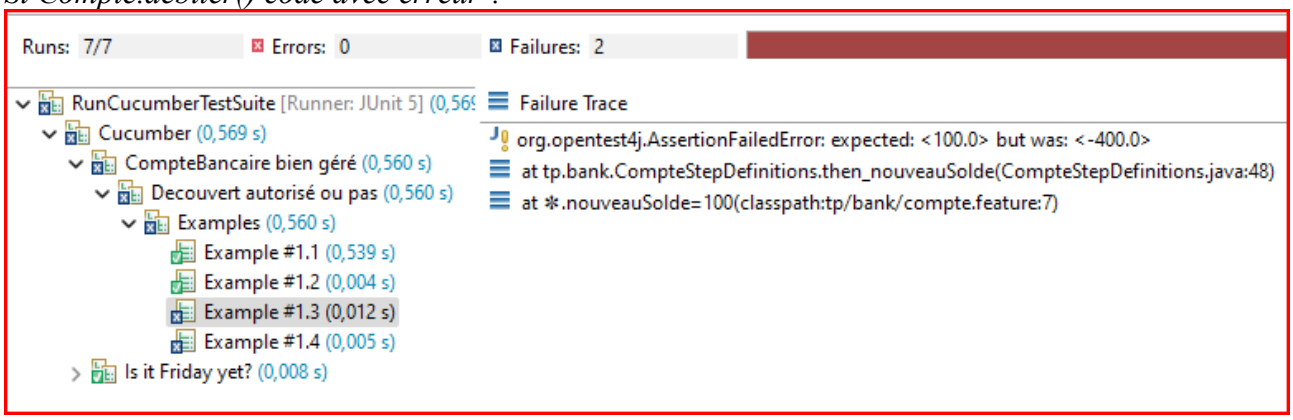
2.5. de tests "cucumber" à lancer

*tp. **RunCucumberTestSuite**.java* (à placer dans src/test/java)

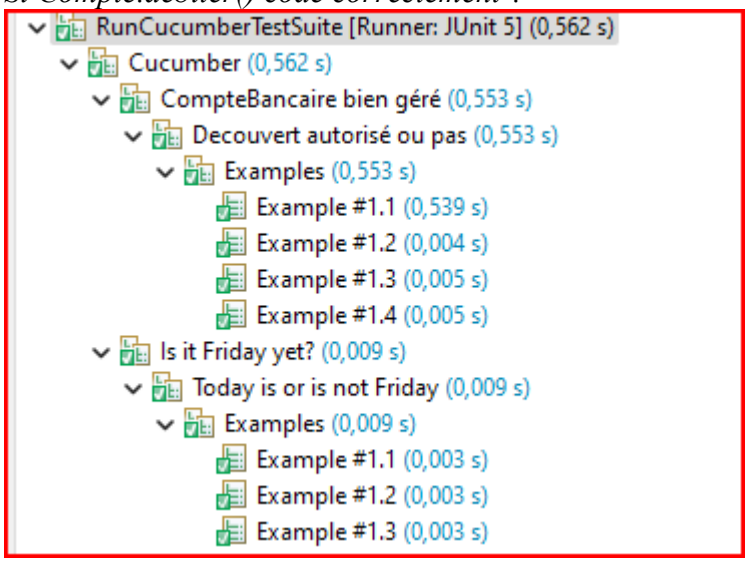
```
package tp;
import org.junit.platform.suite.api.ConfigurationParameter;
import org.junit.platform.suite.api.IncludeEngines;
import org.junit.platform.suite.api.SelectClasspathResource;
import org.junit.platform.suite.api.Suite;
import static io.cucumber.junit.platform.engine.Constants.GLUE_PROPERTY_NAME;

@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("tp")
@ConfigurationParameter(key = GLUE_PROPERTY_NAME, value = "tp")
public class RunCucumberTestSuite {
}
```

2.6. Résultats

Si Compte.debiter() codé avec erreur :



Si Compte.debiter() codé correctement :



ANNEXES

V - Annexe – Intégration Continue / Jenkins

1. Principaux objectifs de l'intégration continue

Sur un gros (ou moyen) projet, chaque développeur se concentre une une partie bien précise et génère des simples composants devant ultérieurement être assemblés entre eux pour produire l'application complète.
Sans automatisme, il faut alors manuellement:

- vérifier que les différents composants soient bien compatibles (mêmes versions, prévus pour s'interfacer entre eux)
- rassembler/packager les composants dans des modules exécutables (.exe, .dll , .jar , .ear, ...)
- déployer le tout sur un serveur d'application
- lancer des tests globaux

Ce qui peut prendre beaucoup trop de temps !!!!!!

Pour rester concurrentielle , une SSII/ESN ou une maîtrise d'œuvre interne doit s'appuyer sur un système automatisant la plupart des points précédents.

Un tel serveur système dit "d'intégration continue" va *(à peu près)*:

- récupérer le code source du composant dans un référentiel (SVN ou GIT ou …) dans une structure neutre (indépendante de l'ide)
- recompiler ce code source (pour bien contrôler la version du compilateur utilisé)
- relancer (dans un contexte contrôler) des jeux de tests unitaires
- packager le code compilé d'un composant ou d'un module dans une archive adéquate (.jar , .war , .ear)
- déployer éventuellement l'ensemble sur un environnement spécifique de tests (JVM , serveur d'application, ...)
- lancer éventuellement des tests d'intégrations (ou globaux) qui ont été préalablement préparés
- remonter des messages et des statistiques vers les développeurs
- générer et stoker une nouvelle version du logiciel si les tests ont réussis.

Dans le monde java, la plupart des environnements d'intégrations continues sont basés sur l'une et/ou l'autre des trois technologies fondamentales suivantes:

- **ANT** (sorte de makefile en XML et donc indépendants de la plate-forme) *10 % des projets*
- **MAVEN** (gestionnaire de projet indépendant de l'IDE) *70 % des projets java*
- **GRADLE** (plus flexible et moderne que maven , pas XML mais DSL) *20 % des projets*

1.1. Tester très régulièrement une application complète (fruit d'un assemblage de modules)

L'intégration continue est tout à fait dans l'esprit des **méthodes agiles (XP , Scrum, ….)** et **du développement piloté par les tests (TDD : Test Driven Developpement)**.

Le fait que le logiciel satisfasse les tests est un **indicateur concret** permettant de suivre l'évolution du projet.

Une grande partie de l'intérêt de l'intégration continue tient dans le fait de pouvoir relancer très régulièrement et fréquemment toute une série de tests même si le logiciel à tester est un assemblage de modules fabriqués à partir d'environnement de développement (IDE) très variés .

Un module peut être développé avec VsCode sous Windows ; un autre avec IntelliJ sous Macintosh et encore un autre avec Eclipse sous linux.

Si tous ces modules respectent au moins certaines conventions maven fondamentales et s'il sont récupérables depuis un gestionnaire de code source (tel que SVN ou GIT) , alors un logiciel d'intégration continue (tel que Hudson/jenkins) bien paramétré pourra sans problème créer et tester le logiciel complet (résultant d'un assemblage des modules).

1.2. Notifier les développeurs du résultats des tests (état du projet)

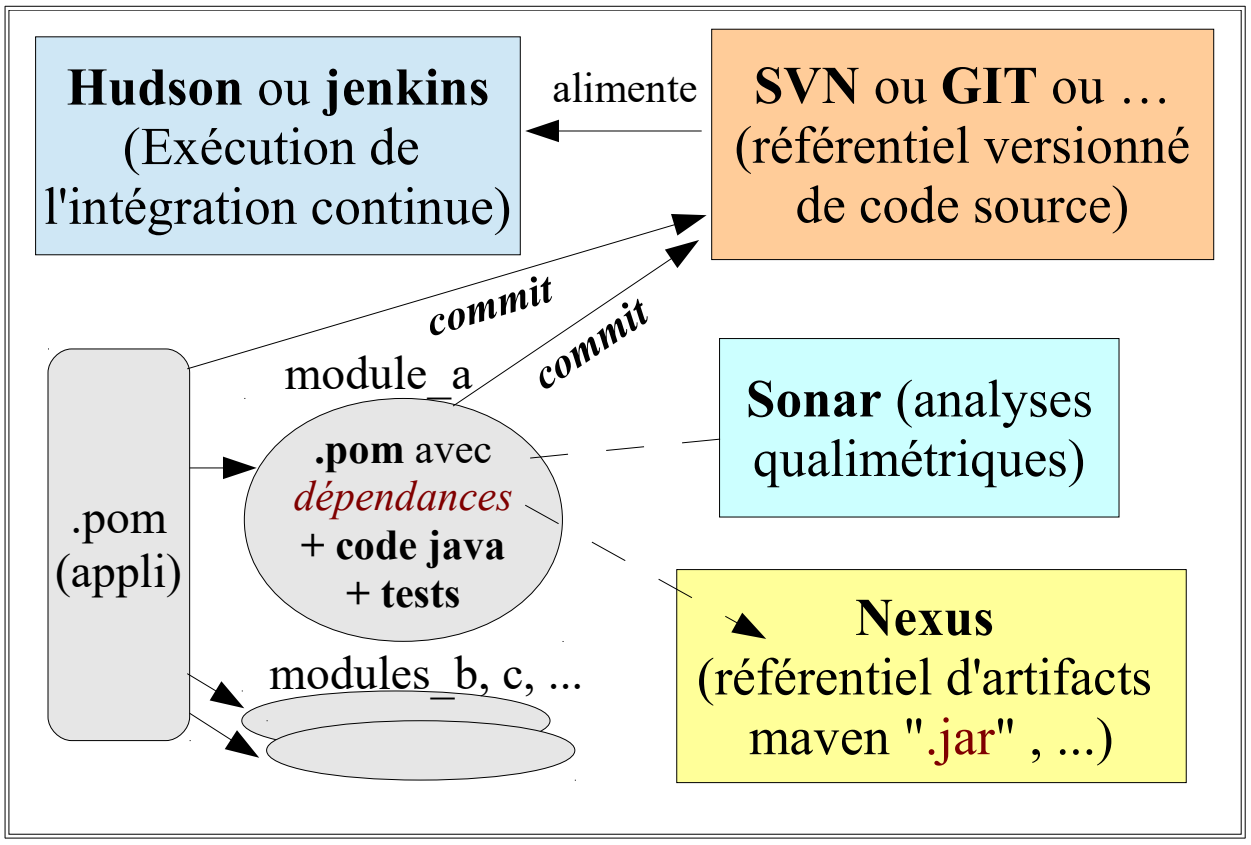
Au niveau d'un logiciel d'intégration continue, on peut toujours paramétrer une liste d'adresse email qui servira à avertir les développeurs :

- résultats des tests (statistiques , erreurs , …)
- état du projet (vert , rouge , …) et évolution
- ...

2. Chaine d'intégration continue

Combinaison de technologies selon le principal langage de programmation

2.1. Chaîne classique pour java



Nexus (ou archiva ou …) est un gestionnaire de référentiel maven qui permet essentiellement de stocker et récupérer sur demande (de façon versionnée) les librairies (".jar") utilisées par les modules du projet. (ex : hibernate-core-3.5.jar , …) .

Ceci permet d'éviter de stocker inutilement des tonnes de ".jar" dans le référentiel de code source (svn ou git ou …) car grâce aux dépendances exprimées dans les fichiers "pom.xml" des modules du projet, la technologie maven sera capable de récupérer automatiquement les dépendances (librairies, …) au sein de nexus pour reconstruire WEB-INF/lib/liste_des_jars ou un équivalent .

Sonar (contrôlé/piloté par maven ou ant) permet d'effectuer quasi automatiquement des analyses qualimétriques sur le code du logiciel (ex : couverture des tests , respects de certaines règles au sein de la structure orientée objet et modulaire du code , style et convention , …) .
les rapports effectués par Sonar sont facilement accessibles au bout d'une URL.

Le gestionnaire/référentiel de code source (**GIT** ou **SVN** ou ….) sert essentiellement à **stocker tout le code source des modules d'un projet** .
Ce code source (idéalement basé sur une structure maven) contiendra généralement :

- les packages et les classes java (code du module / de l'application)

- les ressources de configuration (ex : fichiers de config pour Spring+Hibernate+cxf)
- les ressources WEB (images , web.xml , faces-config.xml, pages HTML et JSP , …
- les classes java et les ressources (données/configurations) pour les tests unitaires
- des variantes dans la configuration exprimées sous forme de profils maven .
- liste des dépendances (librairies) utiles pour le module (expression dans pom.xml)

Le gestionnaire d'intégration continue (exemple : **Hudson / jenkins**) pourra ensuite récupérer régulièrement dans GIT ou SVN le code et la configuration maven des modules d'une application pour :

- reconstruire (recompiler) les modules
- relancer tous les tests (unitaires , intégration selon profil, …)
- notifier les développeurs des résultats des tests et des constructions
- ….

Autrement dit , mettre en œuvre une plate-forme d'intégration continue consiste à

- installer et configurer de façon cohérente les différents logiciels (Nexus , Sonar , SVN ou GIT , Hudson, …)
- bien configurer les fichiers maven (pom.xml) des modules de l'application.
  La configuration maven est essentielle car elle comporte sous forme de profils/variantes les paramétrages utiles pour les tests , les bonnes reconstructions et les analyses de sonar.

2.2. Evolution récente/moderne de l'intégration continue:

- de plus en plus intégré dans des conteneurs "**dockers**"
- configuration **jenkins** de plus en plus en mode **pipeline**
- multi-langages (java et javascript et autres)
- dans un cadre "***DevOps***" pas que de l'intégration continue mais aussi de la **livraison ou déploiement continu**.

En anglais : **CI** : **C**ontinuous **I**ntegration
CI/CD : CI et **C**ontinuous **D**elivery or **D**eployment

Métaphore classique: **usine logicielle**

3. Premiers pas avec Hudson/Jenkins

3.1. Présentation et installation de kenkins

Premiers pas avec Jenkins

Jenkins est actuellement un **logiciel d'intégration continue très en vogue** car il est **très simple à configurer et à utiliser**.

Installation de Jenkins :
Recopier **jenkins.war** dans **TOMCAT_HOME/webapps** (avec un éventuel Tomcat dédié à l'intégration continue configuré sur le port 8585 ou autre).

Etant donné que la configuration de jenkins ne nécessite pas de base de données relationnelle (mais de simples fichiers sur le disque dur) , il n'y a rien d'autre à configurer lors de l'installation .

Url de la console "jenkins" :

http://localhost:8585/jenkins

Premier menu à activer :

Administrer Jenkins / Configurer le système

3.2. Installation/démarrage de Jenkins sans tomcat

NB : il est également possible de démarrer une version récente de jenkins sans serveur tomcat via un script de ce genre :

startJenkins.bat

```
set JAVA_HOME=C:\Program Files\Java\jdk-17
set MVN_HOME=C:\Prog\apache-maven-3.8.4
set PATH="%JAVA_HOME%/bin";"%MVN_HOME%/bin";%PATH%
REM  java -jar jenkins.jar -D"hudson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT=true"
java -jar jenkins.war --httpPort=8585
```

et après un tel lancement l'url menant à la console jenkins sera simplement **http://localhost:8585**

3.3. Configuration nécessaire lors du premier démarrage

Lire le mot de passe temporaire à la console lors du premier démarrage (ex: *d1223a3ba2a44d079ecb7deec0625de8*)
et **reporter/recopier celui-ci dans la console de jenkins**

Installer quelques plugins fondamentaux (ceux qui sont suggérés)

Configurer un compte principal (administrateur) pour les futurs démarrages :
*par exemple username=**admin** password=**admin123***

En configuration de TP, choisir l'URL http://localhost:8585

3.4. Installation ou mise à jour de plugins pour Jenkins

Menu **"tableau de bord" / "Administrer Jenkins" / "Gestion des plugins"**

3.5. Configuration élémentaire d'une tâche "jenkins / freeStyle"

Menu **"tableau de bord" / "Nouveau item"**
puis **:**
- donner un nom (ex : *jobXy*)
- choisir souvent "projet free-style" pour les cas simples/ordinaires
- OK

Dans la partie "**gestion du code source**", choisir généralement :
- GIT
et préciser l'url du référentiel git (par exemple ***https://github.com/…./repoXy.git***)
Attention: les versions récentes de Jenkins n'acceptent des URLS de type file:///c:/xx/yy qu'avec l'option -D"hudson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT=true" à fixer au démarrage
et dans la partie "branch to build" on pourra par exemple choisir ***/master** ou ***/main** .

Dans la partie "**build**" , choisir généralement :
- Invoquer les cibles Maven de haut niveau
et préciser la "***cible (ou goal)***" maven à déclencher (*ex* : **clean package**)

*NB: si le projet maven à construire est dans un sous répertoire du référentiel git (cas pas très conseillé mais admis), alors au niveau du build on peut préciser un chemin menant au pom.xml de type **sous_rep1/pom.xml** ou bien sous_rep1/sous_sous_rep2/pom.xml dans **config avancée** .*

Sauvegarder assez rapidement ces configurations essentielles.
Les configurations secondaires annexes pourront être ajoutées ultérieurement

Lancement sur demande d'un "build" (associé à un job jenkins configuré)

Affichage des résultats via la console de jenkins

La logique de navigation/sélection de jenkins est la suivante :

Jenkins (server) > **Job (name/type/config)** > **number of instance (with status/results)**

Exemple: 

Après avoir sélectionné un des niveaux , on accède à un menu (coté gauche) pour :
* créer/activer de nouveaux éléments
* (re)configurer plus en détails l'élément sélectionné
* afficher des détails sur l'élément sélectionné
* ...

Concernant les résultats d'un build, la partie la plus intéressante est souvent "***sortie console***" :


```
------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------
[INFO] Total time: 24.003s
[INFO] Finished at: Tue Apr 21 14:51:42 CEST 2015
[INFO] Final Memory: 12M/32M
[INFO]
------------------------------------------------
```

3.6. job/item de type "pipeline"

Un job de type "**pipeline**" est assez conseillé au sein de jenkins car :
- il est grandement configurable/extensible
- il peut comporter plusieurs étapes (enchainement en pipeline)
- un script de type "pipeline_jenkins" peut être placé dans un référnentiel git et ainsi être versionné

Exemple de pipeline simple pour un projet java :

```
pipeline {
    agent any
    stages {
        stage('SCM') {
            steps {
                git url : 'https://github.com/didier-tp/test_junit.git' , branch : 'main'
            }
        }
        stage('Build') {
            steps {
                script {
                    dir('with_mockito') {
                        //sh "mvn -Dmaven.test.failure.ignore=true clean package"
                        bat "mvn -Dmaven.test.failure.ignore=true clean package"
                        }
                    }
                }
            post {
                // If Maven was able to run the tests, even if some of the test failed, .....
                success {
                    script {
                     dir('with_mockito') {
                            bat "mvn javadoc:javadoc"
                            echo "javadoc generated , ..."
                     }
                    }
                }
            }
        }
        stage('sonar scan or prepa_docker') {
            steps {
                echo "sonar scan ou construction container docker (souvent sous linux)"
            }
        }
    }
}
```

NB: **pas besoin de**

~~script {~~
~~dir('with_mockito') {~~
~~}~~
~~}~~

si pom.xml directement à la racine du référentiel git

4. Différents types de "builds" (avec Jenklins)

Différents types de "builds"

Types de "build"	Commentaires/considérations
Local / privé	**Lancé manuellement** (et idéalement fréquemment) par le développeur (depuis son IDE) . → *Permet de savoir si ses propres changements fonctionnent*
Intégration rapide (de jour)	**Tests d'intégration rapides déclenchés après chaque commit ou bien régulièrement (ex : toutes les 20 minutes)**. Seuls les tests rapides (unitaires + intégrations) sont lancés → *Permet de savoir si l'assemblage des changements de tous les développeur fonctionne .*
Intégration journalière poussée/sophistiquée à heure fixe **(nightly build)**	**Tests sophistiqués (longs)** , tests de performance , génération de documentation, de rapports , …. → *Permet de savoir si la dernière version produite du logiciel est en état de marche .*

Réglages de fréquence via une syntaxe "crontab" (par exemple dans Jenkins) :

Syntaxe "crontab" :

mm hh jj MM JJ *[tâche]*

`mm` représente les minutes (de 0 à 59)
`hh` représente l'heure (de 0 à 23)
`jj` représente le numéro du jour du mois (de 1 à 31)
`MM` représente le numéro du mois (de 1 à 12)
`JJ` représente le numéro du jour dans la semaine
(0 : dimanche , 1 : lundi , 6 : samedi , 7:dimanche)

Si, sur la même ligne, le *« numéro du jour du mois »* et le *« jour de la semaine »* sont renseignés, alors **cron** (ou ….) n'exécutera la `tâche` que quand ceux-ci coïncident .

…/...

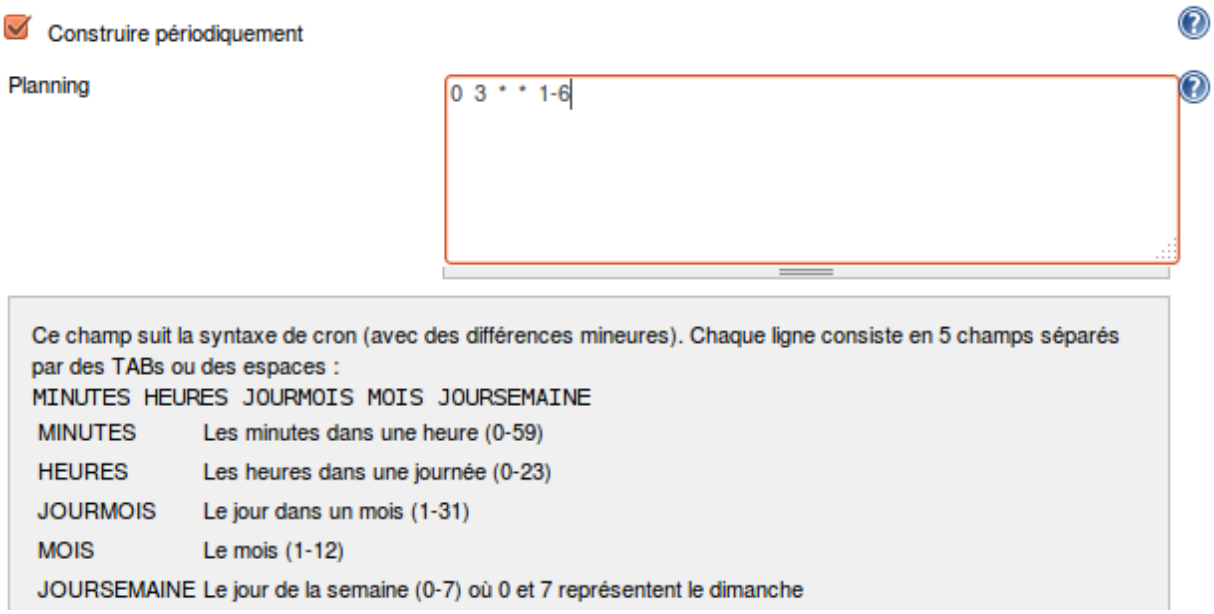
Réglage de la fréquence des "builds"

Pour chaque valeur numérique (`mm`, `hh`, `jj`, `MMM`, `JJJ`) les notations possibles sont :
* : à chaque unité (0, 1, 2, 3, 4...)
5,8 : les unités 5 et 8
2-5 : les unités de 2 à 5 (2, 3, 4, 5)
**/3* : toutes les 3 unités (0, 3, 6, 9...)
10-20/3 : toutes les 3 unités, entre la dixième et la vingtième (10, 13, 16, 19)

Et donc pour un nightly build d'intégration continue :
---> **0 3 * * 1-6** *(tous les jours à 3h du matin sauf les dimanches)*

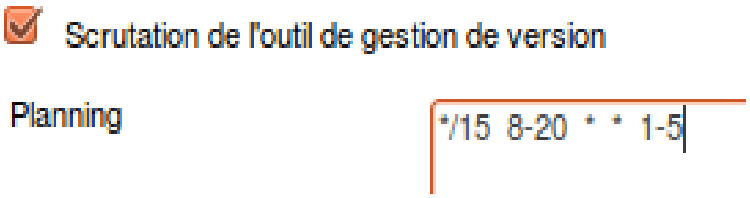
et pour un build rapide de jour :
→ ***/15 8-20 * * 1-5** *(tous les jours sauf les week-ends , toutes les 15 minutes de 8h à 20h)*

Lancement périodique (ex journalier) au niveau de Jenkins



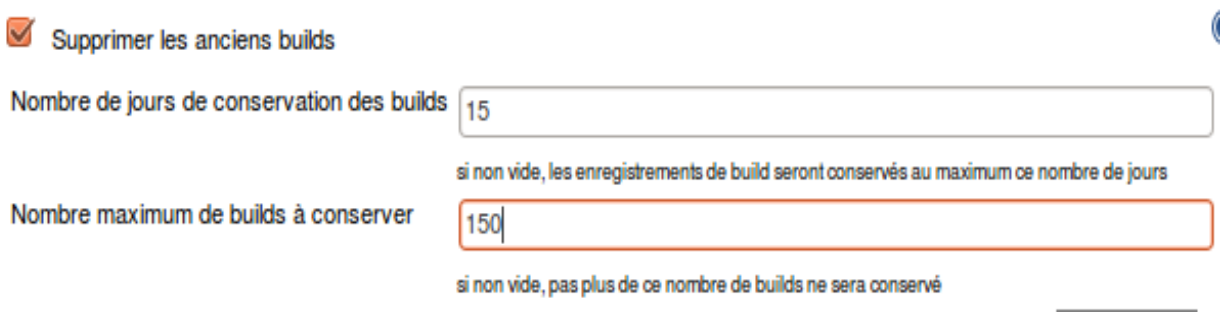
dans cet exemple : tous les jours (de lundi à samedi) à 3h du matin. "nightly build"

Lancement sur détection périodique des changements (SVN/GIT)



dans cet exemple : Jenkins vérifie toutes les 15 minutes si certains changements ont eu lieu au niveau du référentiel de code source (de lundi à vendredi et de 8h à 20h). En cas de changement détecté → lancement (potentiellement très fréquent) d'un build d'intégration.

Suppression des anciens builds (jenkins)



VI - Annexe – Propositions de TP

1. Chargement du point de départ des Tps

- Charger les projets exemples "**old_junit4**" et "**current_junit5**" du référentiel git
  https://github.com/didier-tp/test_junit
  au sein d'un IDE java (ex : eclipse ou intelliJ) .

2. TPs progressifs sur JUnit 4 ou 5

-

3. TPs sur JUnit 5

- Compléter le code de tp.converter.**TestsImbriques**

4. TPs sur mockito

- Charger le projet exemple "**with_mockito**" du référentiel git
  https://github.com/didier-tp/test_junit
  au sein d'un IDE java (ex : eclipse ou intelliJ) .

- Analyser et faire tourner l'exemple **tp.service.TestServiceDevise** .
- Au sein du package **tp.emprunt** s'inspirer de `TestServiceEmpruntSansMock` pour coder **`TestServiceEmpruntAvecMock`**