Quid du Test dans un développement logiciel ?

Isabelle BLASQUEZ
@iblasquez

2022

Isabelle BLASQUEZ

@iblasquez

Enseignement : Génie Logiciel

Recherche : Développement logiciel agile

ICSTUG #IUTAgile

CodeWeek.

#Software Craftsmanship

MUSEOMIX LIMOUSIN

Extrait : https://twitter.com/Lilobase/status/952504781515509761

Les tests dans le développement logiciel, du cycle en V aux méthodes agiles

Isabelle Blasquez [1], **Hervé Leblanc** [2], **Christian Percebois** [2]

1. *Université de Limoges*
   isabelle.blasquez@unilim.fr
2. *Université de Toulouse, laboratoire IRIT*
   herve.leblanc,christian.percebois@irit.fr

RÉSUMÉ. *Le test logiciel est une méthode empirique utilisée pour la vérification et la validation de systèmes complexes. Il est notamment déployé lors de la phase ascendante du cycle en V au travers des tests unitaires, d'intégration et d'acceptation. Ces différents tests, dits classiques, s'appliquent a posteriori à un code déjà développé. Le développement agile, promouvant à l'extrême certaines bonnes pratiques du génie logiciel, fait jouer un rôle de première importance aux tests. En particulier, les cycles de développement dirigés par les tests utilisent les tests pour spécifier en sus de vérifier et forcent à leur automatisation. Dans cet article, nous montrons que les tests classiques et les tests agiles ne sont pas antinomiques ; bien au contraire ces deux approches peuvent s'enrichir l'une de l'autre.*

ABSTRACT. *Software testing is an empirical approach increasingly used for verification and validation of complex systems. It is especially deployed on the upward-sloping branch of the V-model through unit testing, integration testing and acceptance testing. Usually, these tests are performed after the development phase on an already written production code. Agile software development pushes some best traditional software engineering practices at extreme levels. In this context, testing is considered as a first and major element of a developement process. Test driven development cycles not only use test cases to check errors but also to specify requirements and lead to test automation. In this paper, we show that usual and agile testing are not opposite, but rather can mutually enhance one another.*

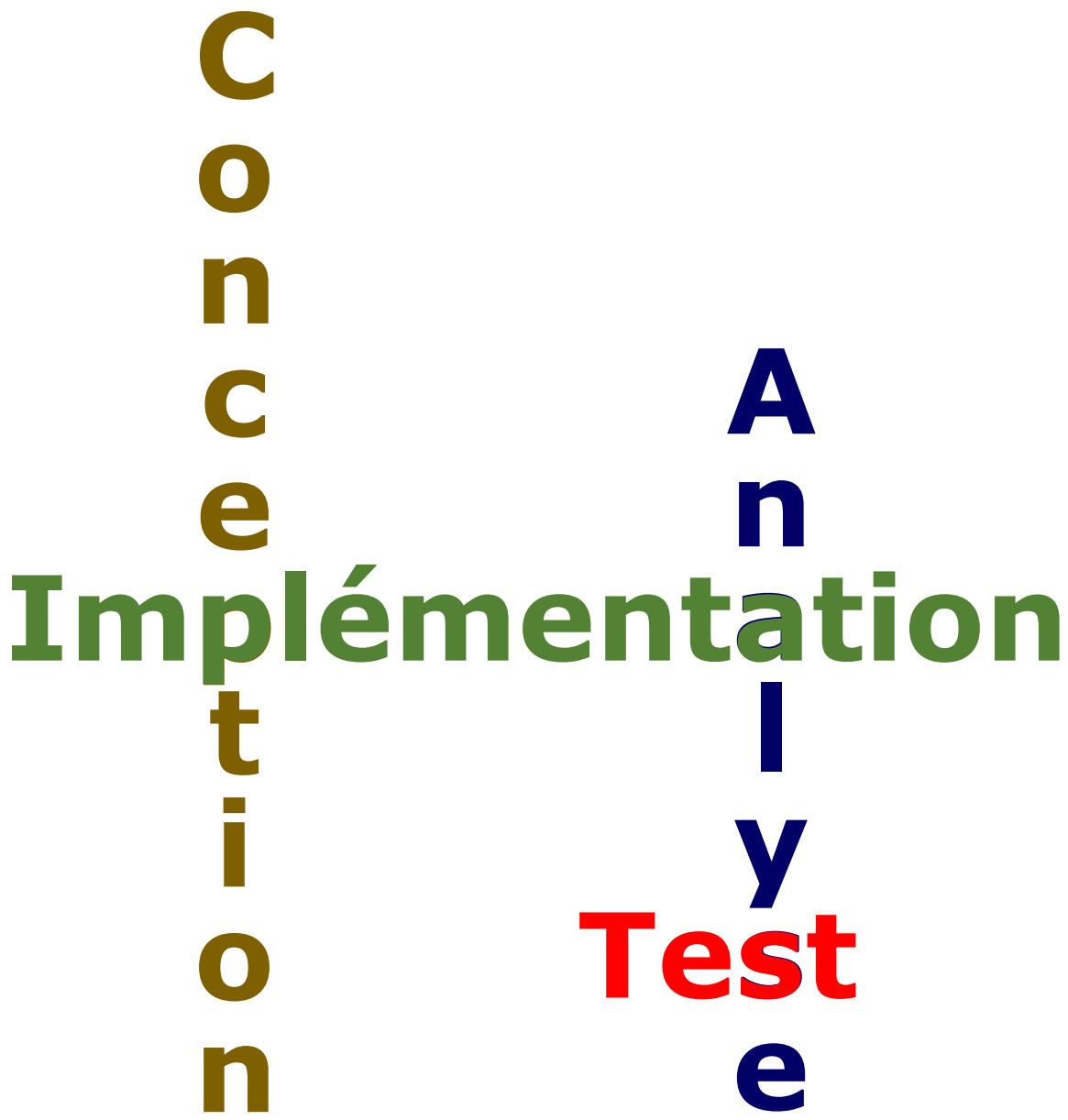
MOTS-CLÉS : *test logiciel, cycle en V, méthodes agiles*

KEYWORDS: *software testing, V-model, agile software development*

DOI:10.3166/TSI.x.1-44

Principales phases du développement logiciel

Principales phases du développement logiciel



A propos de la phase d'Analyse des exigences (Requirements phase)

La phase d'analyse des exigences *est la période du cycle de vie pendant laquelle les exigences, fonctionnelles et non fonctionnelles du produit logiciel, sont définies et documentées.* **(IEEE, 1990[1])**.

Ce que doit faire le logiciel

Cette phase donne lieu à l'écriture d'un document de ***spécifications*** *qui précise les missions du logiciel. Ce document est une trace des besoins utilisateurs et sera utilisé dans les autres phases du cycle de développement.*

[1] **IEEE Standard Glossary of Software Engineering Terminology**

Abstract: IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*, identifies terms currently in use in the field of Software Engineering. Standard definitions for those terms are established.
Keywords: Software engineering; glossary; terminology; definitions; dictionary

A propos de la phase de Conception (Design phase)

COMMENT
faire ce logiciel

La phase de conception *est la période du cycle de vie pendant laquelle l'architecture logicielle, les composants logiciels, les données et les interfaces sont conçus et documentés afin de satisfaire aux exigences.* **(IEEE, 1990)**.

A propos de la phase d'Implementation (Implementation phase/Coding)

Ecriture du code

La phase d'implémentation *est la période du cycle de vie pendant laquelle le logiciel est créé et débuggé à partir des spécifications de conception.* **(IEEE, 1990)**.

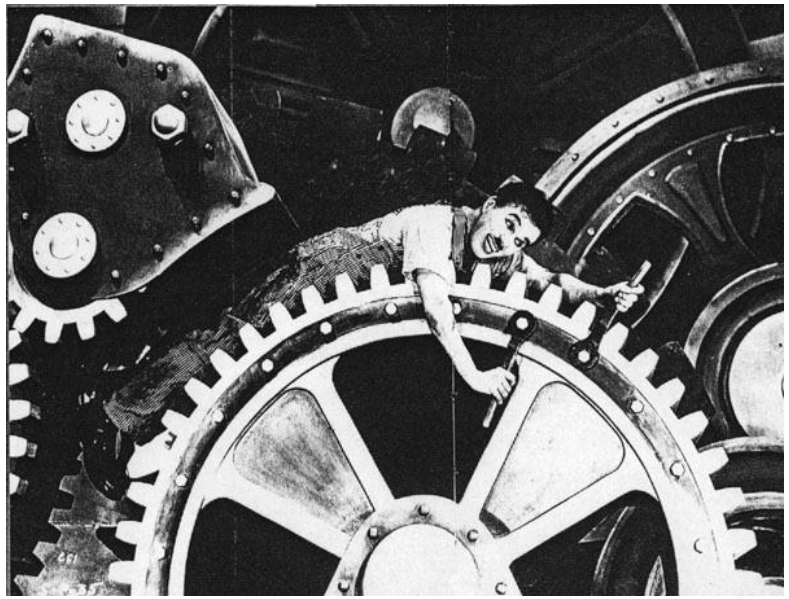
Les tâches de cette phase se concentrent autour du ***code*** *où les composants sont implémentés et testés individuellement* ***dans un langage de programmation*** *donné afin de mettre en œuvre la conception*

A propos de la phase de Test (Test phase)

Qualité logicielle

La phase de Test *est la période du cycle de vie consacrée à l'intégration et à l'évaluation des composants et du logiciel afin de vérifier les exigences aussi bien au niveau système qu'utilisateur* **(IEEE, 1990)**.

Le test : pour quoi ? … Un outil de qualité logicielle



Vérification

Fonctionnement correct du produit

(Product Right)

Validation

Respect des exigences utilisateurs

(Right Product)

Images : http://www.allocine.fr https://stocksnap.io/photo/206B1A2E4C

Zoom sur le Test

Définition sur l'outil de Test (Testing)

Le test *consiste à exécuter et évaluer un système ou un composant sous des conditions spécifiques, pour vérifier qu'il répond à ses spécifications ou pour identifier des différences entre les résultats spécifiés et attendus et les résultats effectivement obtenus* **(IEEE, 1990[1])**.

[1] **IEEE Standard Glossary of Software Engineering Terminology**

Abstract: IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*, identifies terms currently in use in the field of Software Engineering. Standard definitions for those terms are established.
Keywords: Software engineering; glossary; terminology; definitions; dictionary

Le test : une méthode de Vérification & Validation (conforme à la définition IEEE)

Le processus de test consiste à exécuter un programme dans l'intention de détecter des erreurs.
(Myers, Sandler, 2004)

Tester peut seulement montrer la présence d'erreur, mais pas leur absence.
(Dijkstra, 1972)

Quand tester ?

Analyse

Conception

Implémentation

Test

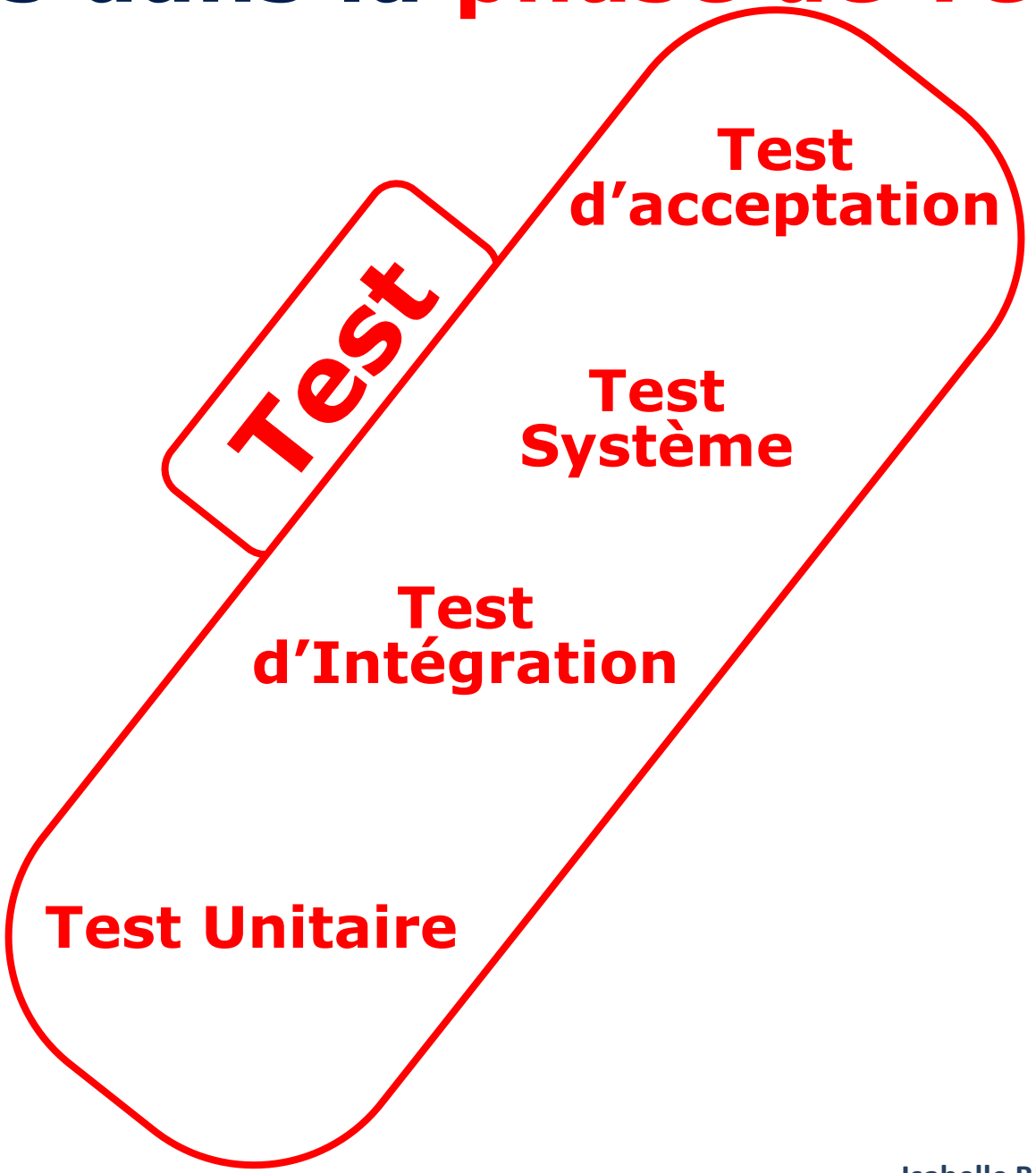
Image : http://www.99hdwallpaper.com/waterfall/waterfall-wallpapers-for-desktop.html

Différentes granularités dans la phase de Test

Analyse

Conception

Implémentation



Test Unitaire

(Vérification au plus proche du code)

Un test unitaire *(appelé aussi test de composant) est un test qui permet de tester de manière* ***isolée*** *une unité logicielle ou un groupe d'unités* **(IEEE, 1990)**.

Outil : ***Framework xUnit pour faciliter la création et l'exécution des tests***

Image : http://www.beilers.com/2010/03/unit-testing-not-required/

Mais ...

l'interopérabilité
n'est **pas garantie**
par les **tests unitaires**

2 tests unitaires : ✓

0 test d'integration : ✗

Un test d'intégration va s'attacher à vérifier
le bon comportement de l'assemblage de plusieurs composants (testés unitairement).

Extraits : http://keynote.hoa-project.net/ForumPHP12/
https://twitter.com/ThePracticalDev/status/687672086152753152

Test d'Intégration

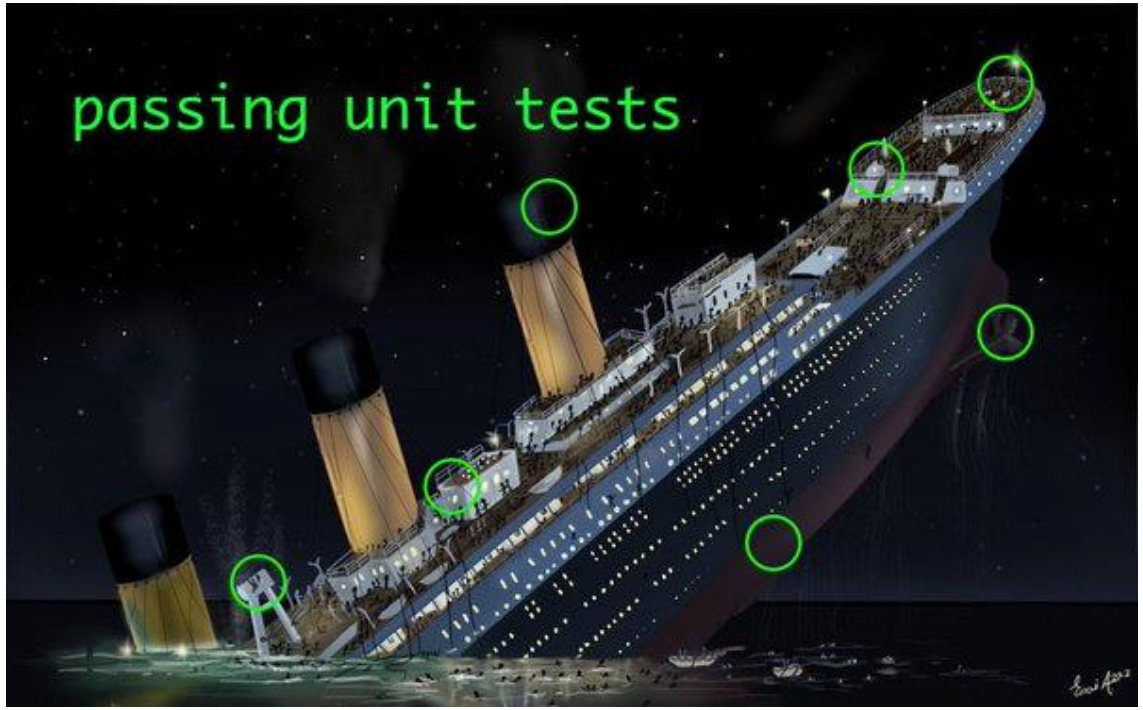
(Vérification des connexions d'interface des composants)

Un **test d'intégration** valide
l'interaction entre les
dépendances

Un test d'intégration *est un test dans lequel les composants logiciels, les composants matériels ou les deux sont combinés pour tester et évaluer leurs interactions* **(IEEE, 1990)**.

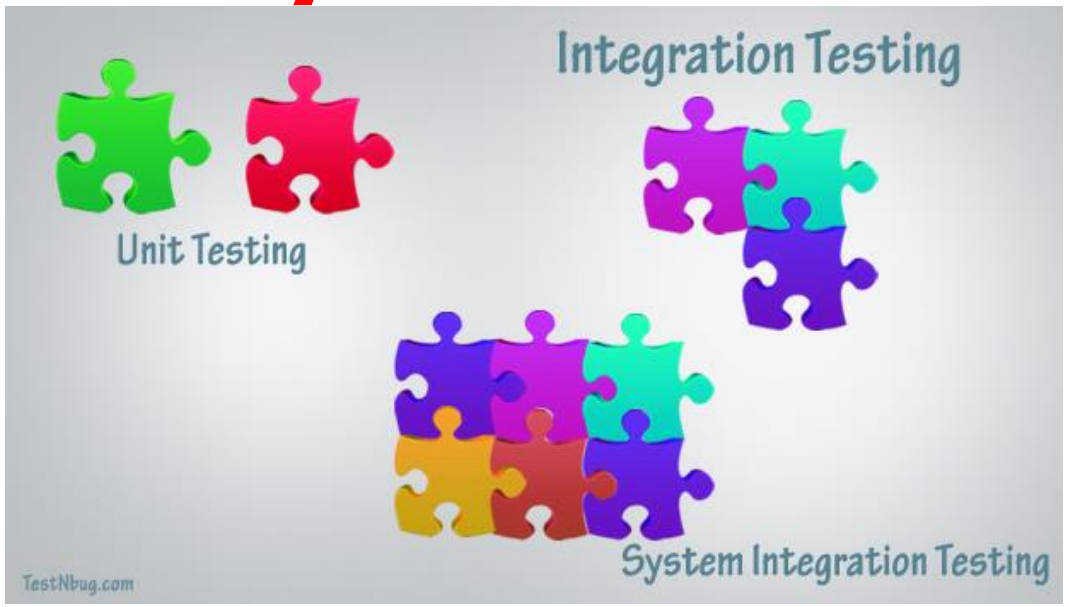
Extrait : http://keynote.hoa-project.net/ForumPHP12/

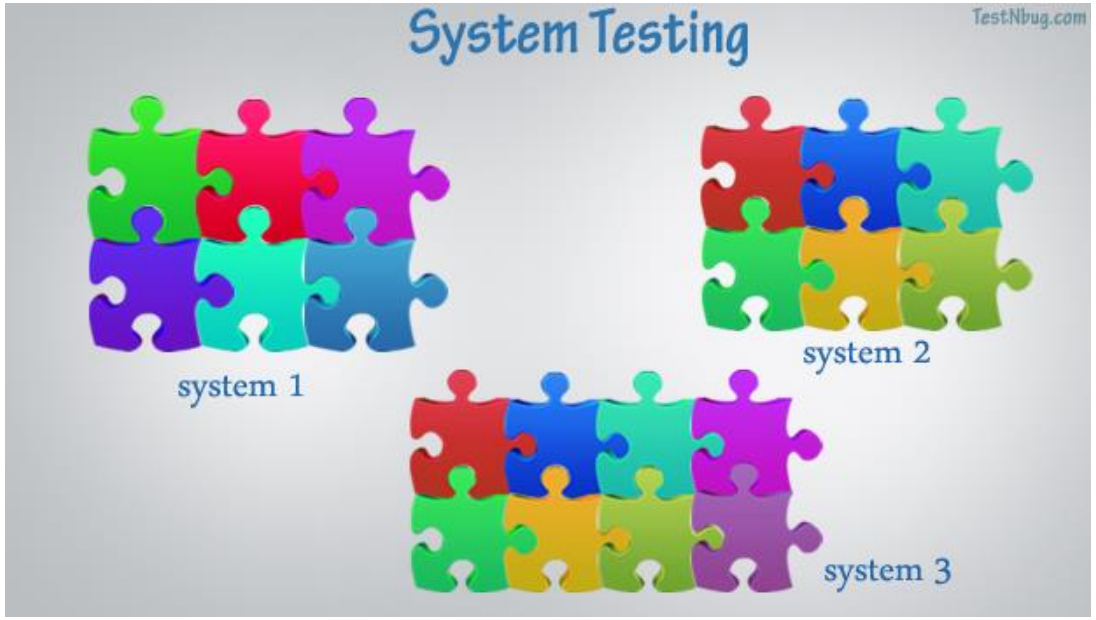
Les tests « isolés » ne suffisent pas …



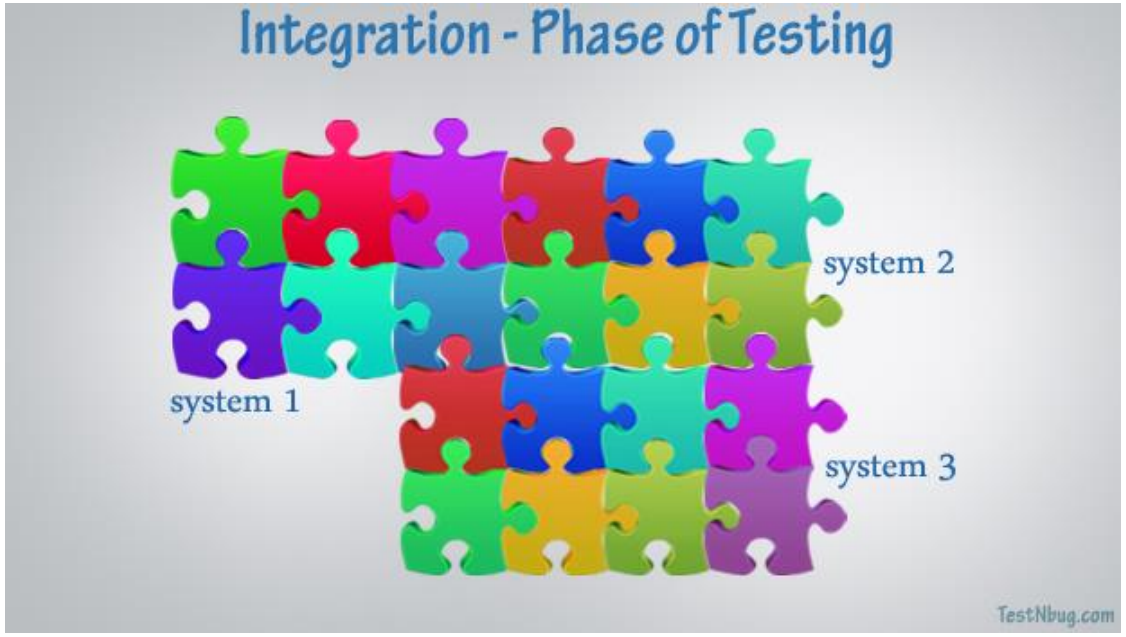
Source : https://twitter.com/ThePracticalDev/status/733908215021199360
https://twitter.com/catriona/status/733911005986578434

Test Système





Un test système *est un test mené sur un système entièrement intégré afin de vérifier si celui-ci respecte les exigences spécifiées* **(IEEE, 1990)**.



→ à effectuer au plus proche de l'environnement de production

→ comportement global
(adéquation des fonctionnalités par rapport aux spécifications fournies)

Extraits : http://www.testnbug.com/2015/01/levels-of-testing/

Test d'acceptation

(Validation des exigences par **rapport au produit livré**)

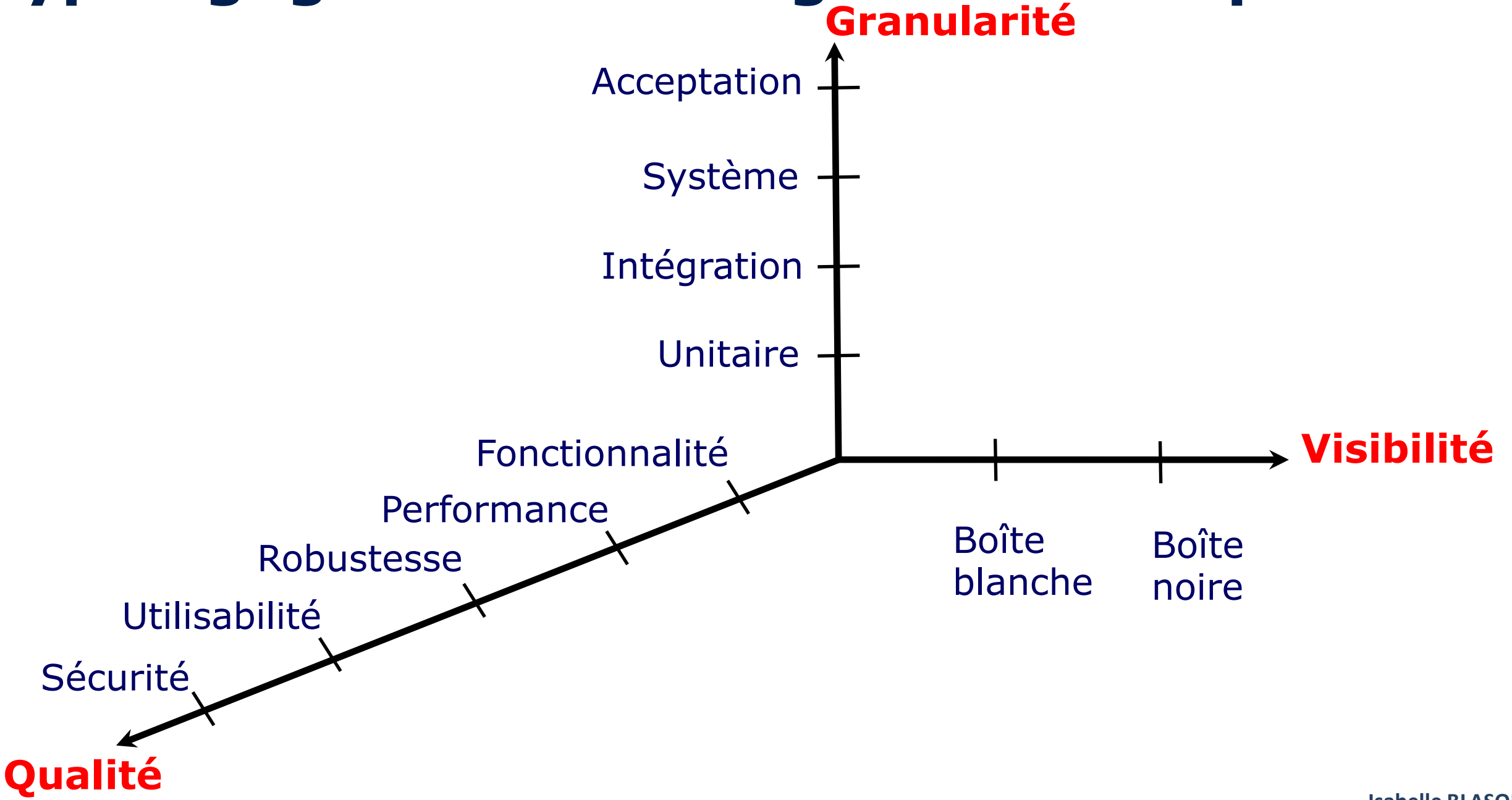
Right Product ?

Validité du système du point de vue client …

Un test d'acceptation *(ou anciennement test de recette) permet de déterminer si un système satisfait ou non à ses critères d'acceptation et permet au client d'accepter ou non le système* **(IEEE, 1990)**.

Sources : http://lesjoiesducode.fr/post/98960240314/quand-je-respecte-les-specs-du-client-a-la-lettre
http://www.commitstrip.com/fr/2014/10/15/when-i-thoroughly-follow-the-requirements/

Typologogie des tests logiciels classiques



Conception d’un cas de test

Isabelle BLASQUEZ

Quelques définitions : test, cas de test

Un test *est défini comme un ensemble* ***de cas de test****, accompagné éventuellement* ***de procédures de test*** **(IEEE, 1990)**

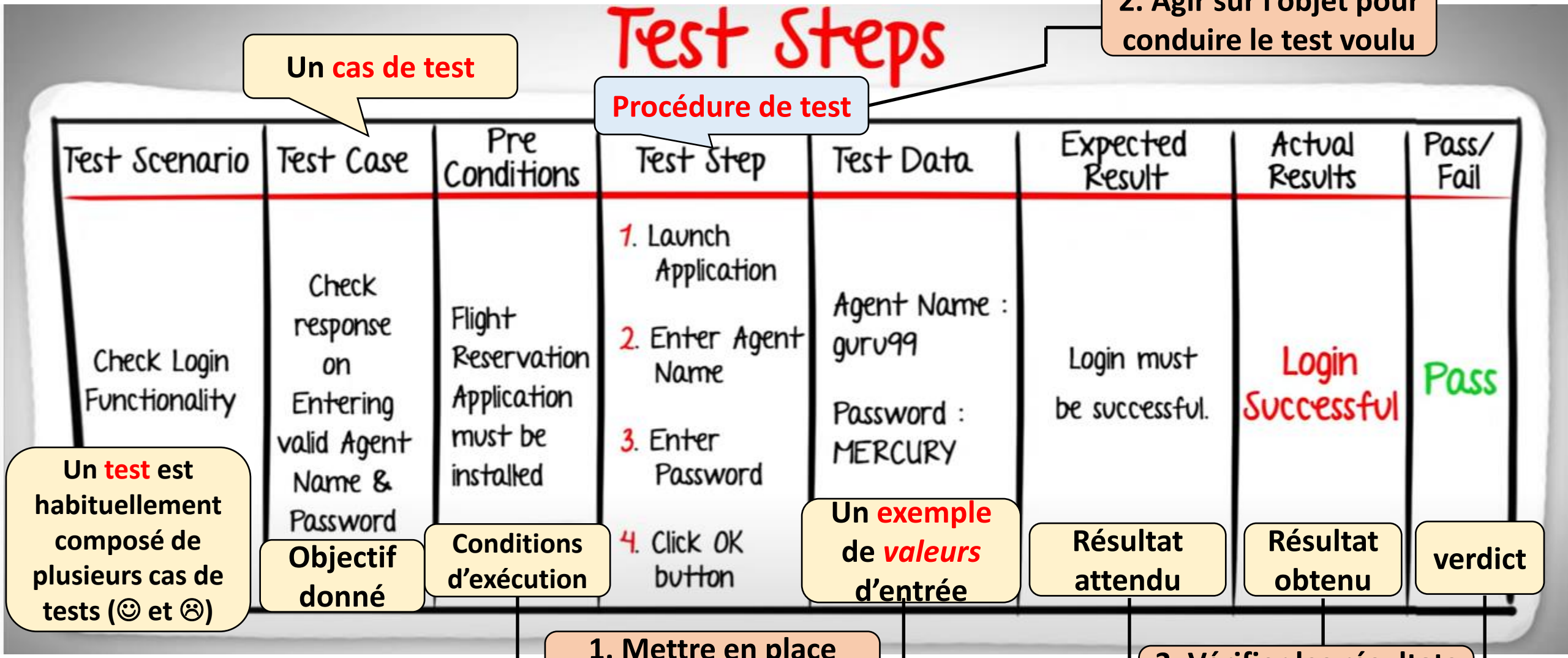
Un cas de test *se compose d'un* ***ensemble de valeurs d'entrée****,* **de conditions d'exécution***, de* ***résultats attendus*** *et qui est développé pour un* ***objectif donné*** *ou une condition de test particulière, tel qu'exécuter un chemin particulier d'un programme ou* ***vérifier le respect d'une exigence spécifique*** **(IEEE, 1990)**

Une procédure de test *contient le* ***détail des instructions*** *pour la mise en place, l'exécution et l'évaluation des résultats des tests pour un cas de test donné* **(IEEE, 1990)**

IEEE Standard Glossary of Software Engineering Terminology

Abstract: IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*, identifies terms currently in use in the field of Software Engineering. Standard definitions for those terms are established.
Keywords: Software engineering; glossary; terminology; definitions; dictionary

Conception de cas de test : illustration



Extrait de la vidéo : https://www.youtube.com/watch?v=BBmA5Qp6Ghk à regarder pour avoir toutes les infos,
Ainsi que https://www.guru99.com/test-case.html#test-case-management-tools

Conception de cas de test : pattern AAA

ARRANGE *Mettre en place les conditions du test*

ACT *Agir sur l'objet pour conduire le test voulu*

ASSERT *Vérifier les résultat attendu*

Pour concevoir des tests pertinents qui permettront de vérifier et valider correctement les exigences du client, nous devons comprendre sans ambiguïté les règles métier

Illustrer ses règles avec des exemples est indispensable pour favoriser et valider cette *bonne* compréhension

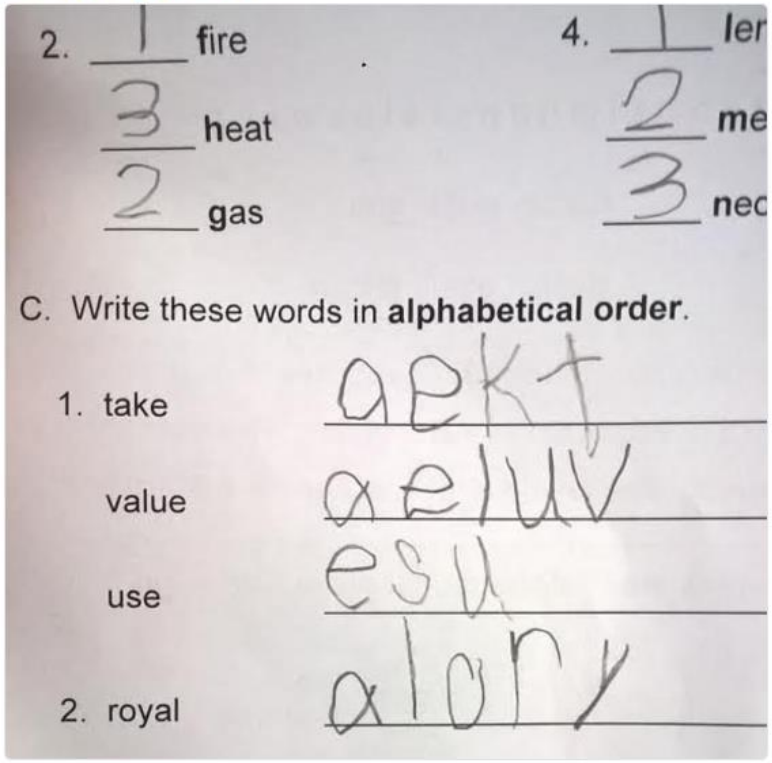
Test Data	Expected Result
Agent Name : guru99 Password : MERCURY	Login must be successful.

Conception de cas de test : De l'importance d'un exemple pour lever toute ambiguïté sur les règles métiers …

The YUNiversity
@The_YUNiversity

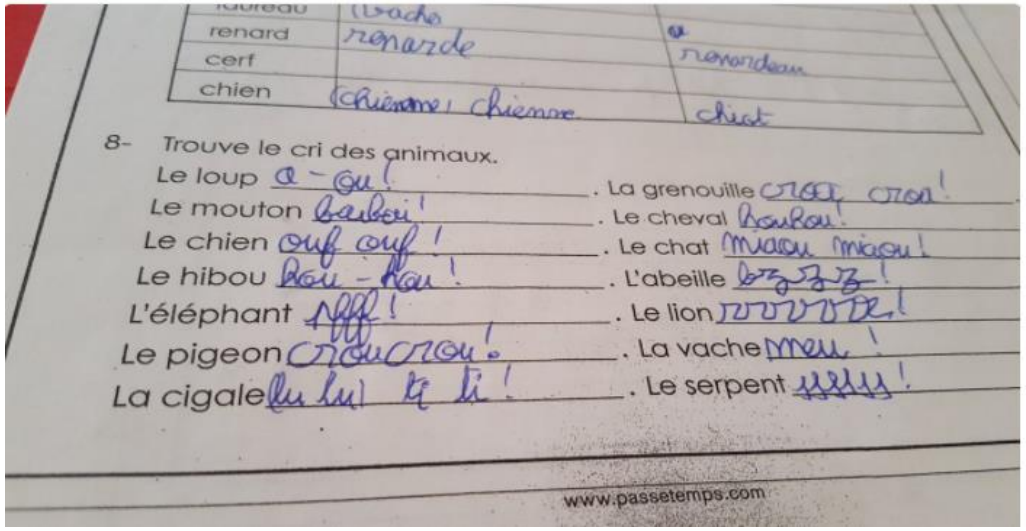
Good job, kid.

bit.ly/2r6qO4g



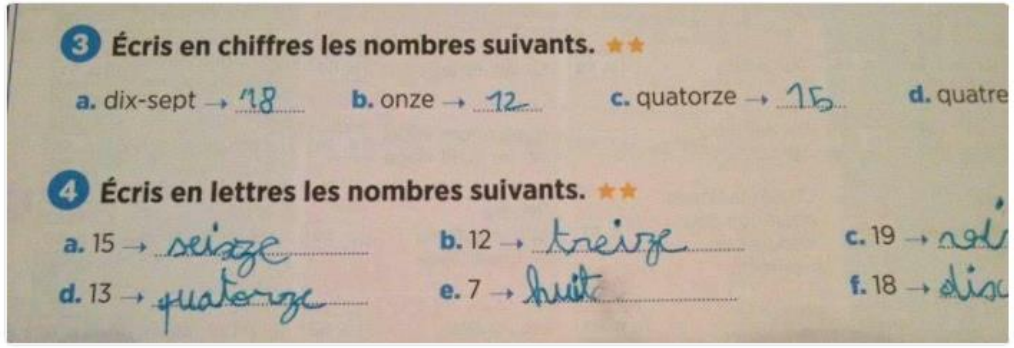
Ninoche
@Prof_Ninoche

Parfois quand t'es instit, tu imagines que ta consigne est claire et puis … #Gilbert



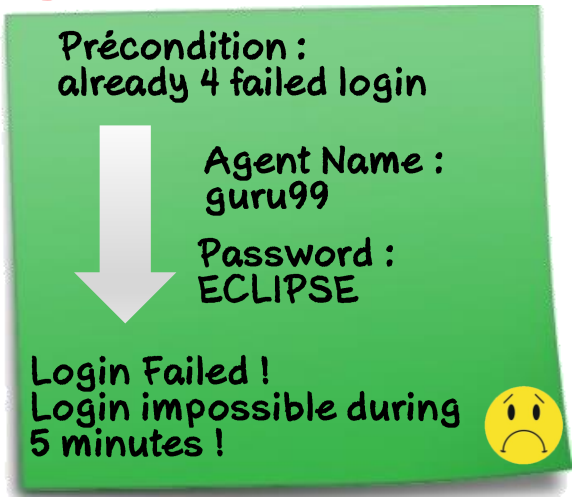
Nils Lesieur
@Nils_Back

La spec semble claire et précise mais la réponse n'est pas celle attendue… => #SpecificationsByExample !!



Extraits : https://twitter.com/Nils_Back/status/813377842356568064 https://twitter.com/The_YUNiversity/status/862759315974828034 https://twitter.com/Prof_Ninoche/status/840299435489951744 (Les bonnes réponses : http://www.animaux-nature.com/liste-cri.php)

1. Les Exemples permettent d'*illustrer et bien comprendre les règles métiers* (exigences)

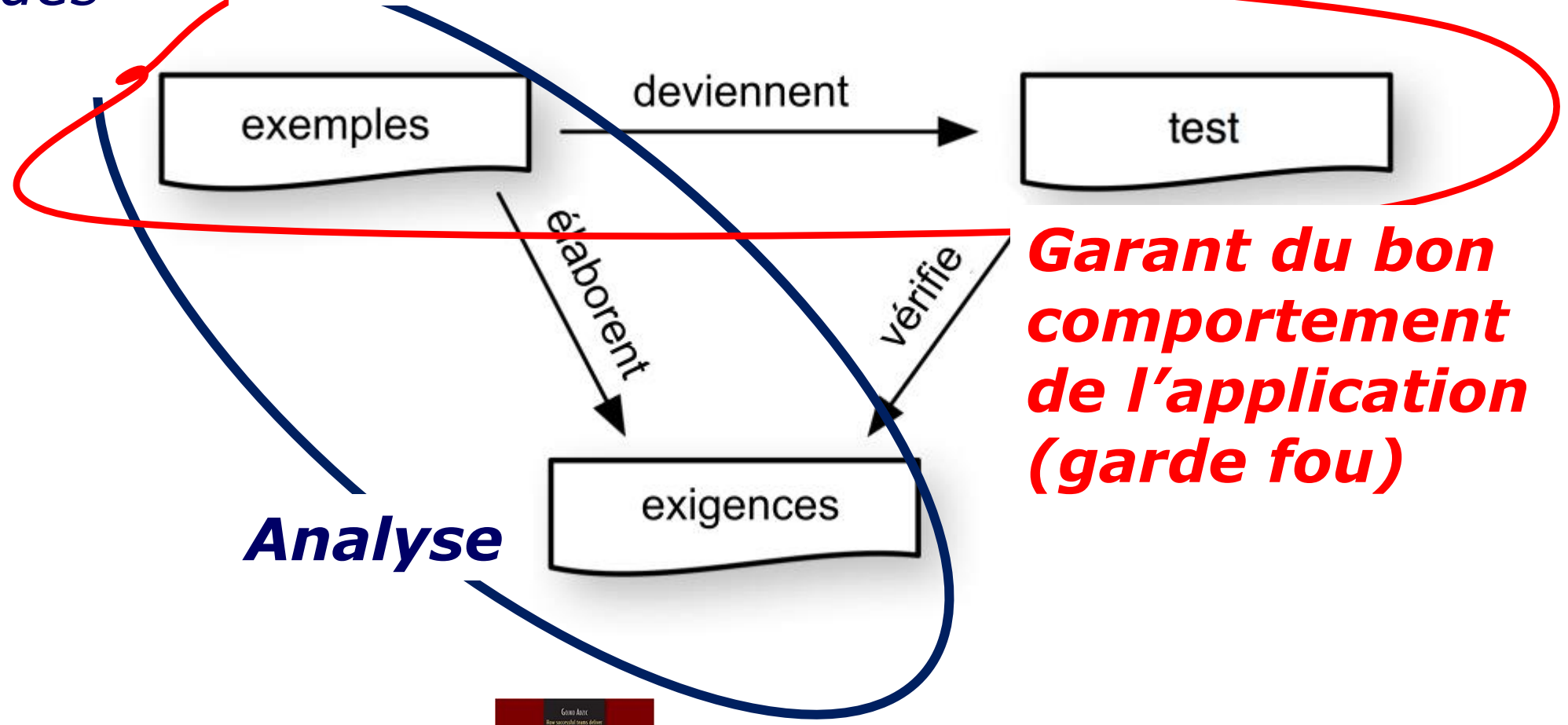


2. Les Exemples peuvent être transformés en tests si les 3 étapes *Arrange, Act et Assert* sont *jouées (déroulées) de manière manuelle ou automatique afin d'obtenir un verdict*.

3. Le verdict du Test permet de *vérifier & valider les exigences* du client (ses besoins)

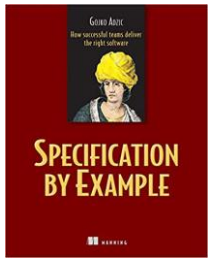
Analyse
(Elaboration des exemples)

Documentation
(Re)lecture des exemples



Garant du bon comportement de l'application (garde fou)

Analyse



Adaptation du schéma proposé par Godjko Adzick dans Specification by Example

Comment savoir si un test est de qualité ? (*Clean* Test)

Un test de qualité : *Clean Test*

FAST

INDEPENDANT

REPEATABLE

SELF-VALIDATING

TIMELY

Un test de qualité respecte le Pattern AAA (vu précédemment)

ARRANGE *Mettre en place les conditions du test*

ACT *Agir sur l'objet pour conduire le test voulu*

ASSERT *Vérifier les résultat attendu*

Independant
Repétable
Self-Validating
(F**IRS**T)

Focus sur l'Automatisation des tests unitaires

au plus proche du code avec
le *framework* **`XUnit`** disponible
dans quasiment tous les langages de programmation)

Isabelle BLASQUEZ

JUnit 5 (pour Java)

Dernière version du *framework* de tests unitaires pour le langage Java développé par Kent Beck a Erich Gamma

Site de référence : https://junit.org/junit5

JUnit 5 = *JUnit Platform* + *JUnit Jupiter* + *JUnit Vintage*

Le cœur du système, cadre général permettant d'exécuter les tests

Bibliothèque nécessaire à l'exécution des tests JUnit 5

Pour exécuter des tests JUnit 3 et JUnit 4 depuis JUnit 5

*Attention ! **JUnit 5** requires **Java 8 (or higher)** at runtime.*

Ecriture d'une méthode de test pour JUnit

✓ Méthode annotée avec `@Test` (à ne pas oublier => erreur fréquente du débutant)

✓ Méthode **publique**

✓ **Méthode d'instance** : pas de ~~`static`~~!!!

✓ Méthode qui ne **renvoie rien** : `void`

✓ Méthode qui ne **prend aucun paramètre d'entrée** : `()`

✓ Méthode avec un **nom explicite** qui montre **l'intention du test**
c-a-d qui décrit le comportement à tester

Bonne pratique de nommage : commencer par **`test`** ou **`doit`** (`test` ou `should`)

✓ Implémentation de la méthode respecte le **pattern AAA**

Implémenter un *clean* test avec JUnit

Fast (FIRST)

ARRANGE

ACT

ASSERT

Annotation indispensable pour que cette méthode soit vue comme un test et exécutée par JUnit !

```
@Test
public void shouldReturnHelloWorldWhenNothing()
    {
     MaClasse monObjet = new MaClasse();


     String resultat = monObjet.maMethodeATester();


     Assertions.assertEquals("Hello World!", resultat);
    }
```

`assertEquals` est une méthode *statique* de la classe **`Assertions`**

résultat attendu (`expected result`)

résultat obtenu une fois le test joué (**`actual result`**)

Méthodes de base de la classe `Assertions`

Javadoc

Method Summary

All Methods | Static Methods | Concrete Methods

Modifier and Type	Method
static void	**assertAll**(**String** heading, **Collection**<**Executable**> executables)
static void	**assertAll**(**String** heading, **Stream**<**Executable**> executables)
static void	**assertAll**(**String** heading, **Executable**... executables)
static void	**assertArrayEquals**(boolean[] expected, boolean[] actual)
static void	**assertArrayEquals**(int[] expected, int[] actual)
static void	**assertArrayEquals**(**Object**[] expected, **Object**[] actual)
static void	**assertTrue**(boolean condition)
static void	**assertFalse**(boolean condition)

Package org.junit.jupiter.api

Class Assertions

java.lang.Object

org.junit.jupiter.api.Assertions

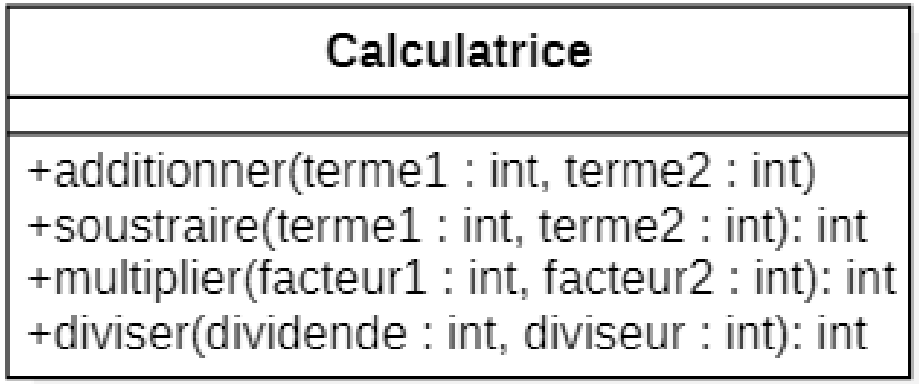
```
import org.junit.jupiter.api.Assertions;
```

… tout petit extrait de la *javadoc* de la classe **`Assertions`** …
… à consulter en fonction de ses besoins …
https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html

Exemple d'écriture de cas de tests avec JUnit :

Calculatrice (pour des entiers)

Une classe métier pour manipuler des entiers en cours d'implémentation.




```java
public class Calculatrice {

 public int additionner(int terme1, int terme2) {
  return terme1 + terme2;
 }

 public int soustraire(int terme1, int terme2) {
  return terme1 - terme2;
 }

 public int multiplier(int facteur1, int facteur2)
{
  //TODO
 }

 public int diviser(int diviseur, int dividende) {
 //TODO
 }

}
```

Bonne pratique autour de la classe de test

✓ **Test Unitaire** : **une classe de test** pour chaque classe métier à tester

✓ **Convention de nommage de la classe :** *NomClasseATester***Test**

✓ Une classe de test est **un ensemble de méthode de tests**

✓ Une méthode de test est un **cas de test**

Une classe de test par classe métier

```java
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class CalculatriceTest {

@Test
void doitAdditionerDeuxEntiersPositifs() {

    int terme1 = 4;
    int terme2 = 6;
    Calculatrice calculatrice = new Calculatrice();

   int somme = calculatrice.additionner(terme1, terme2);


   assertEquals(10, somme);
}

}
```

Respect de la convention de nommage de la classe Test : XXXTest

Méthode de @Test dont le nom montre explicitement l'intention du test c-a-d le comportement à tester

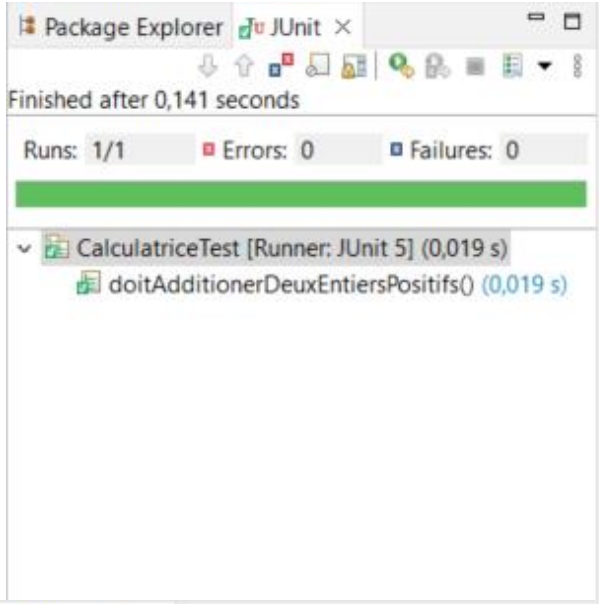
Arrange

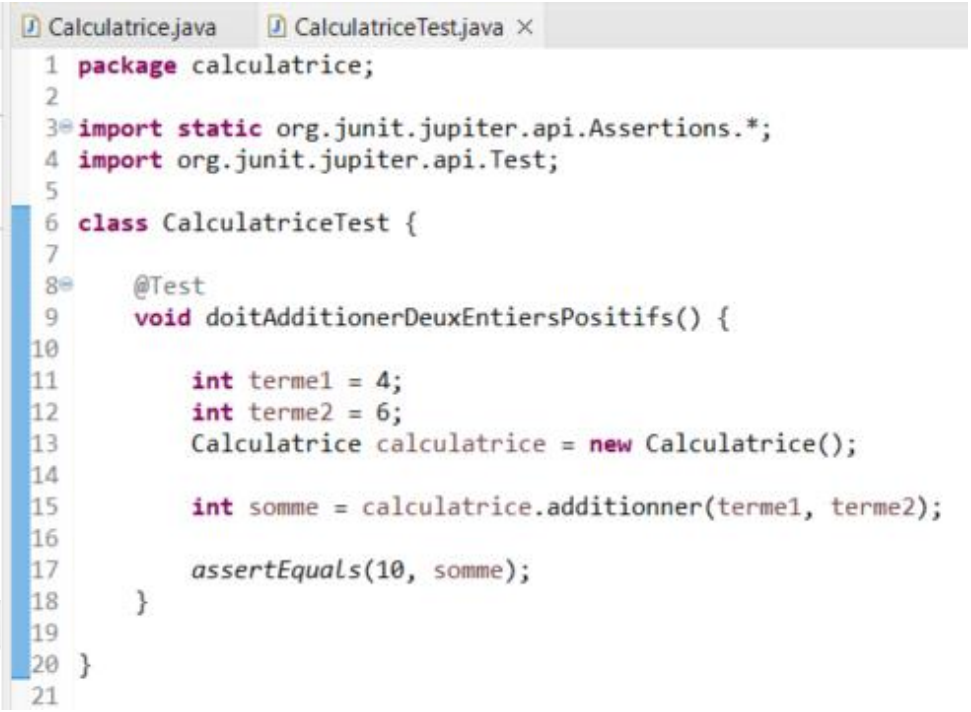
Act

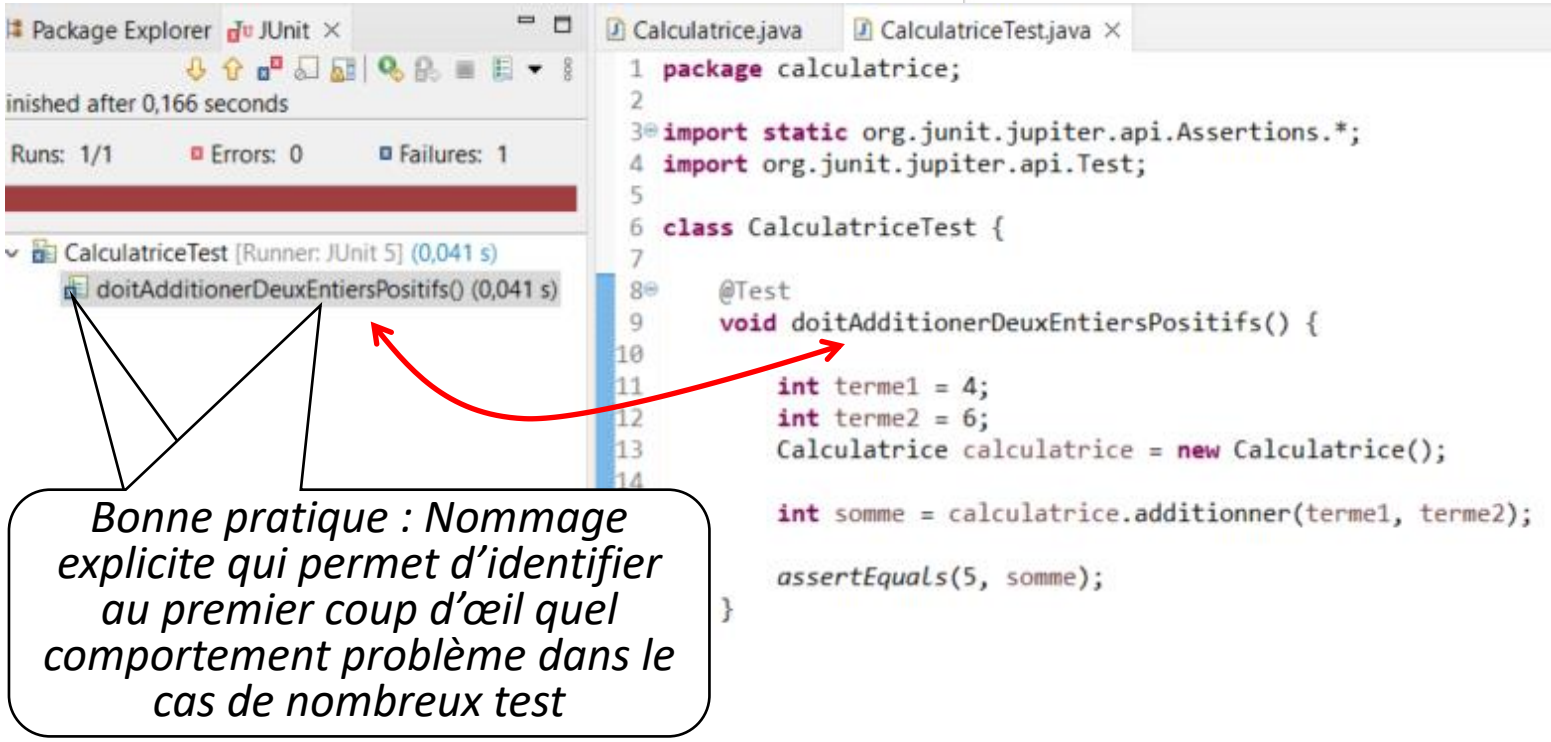
Assert

JUnit joue le test et donne son verdict ...

Le test passe !







Bonne pratique : Nommage explicite qui permet d'identifier au premier coup d'œil quel comportement problème dans le cas de nombreux test

Le test échoue ...

Mais combien de tests, faut-il écrire ?

… **assez** pour que les tests servent de « ***garde-fou*** »
c-a-d **garantissent que le comportement** du système reste inchangé dès que le code vient à être remanié

⇒ c'est ce que l'on appelle éviter la régression du code
(ou assurer la **non-regréssion**)

… mais pas trop pour que l'exécution de la suite de test reste rapide…
(**F**IRST)

La couverture de code par les tests peut vous aider à alléger votre suite de test pour qu'elle reste rapide et efficace !

A propos de la couverture de code par les tests

Isabelle BLASQUEZ

La couverture de code par les tests

La couverture de code est une mesure
qui permet d'identifier la **proportion du code testé**.

En analysant quelle partie du code est **atteinte** (***couverte***) durant l'exécution d'un test,
les outils de couverture permettent de réaliser cette **mesure**,
d'identifier les parties de code non couvertes et de les **visualiser**.

Extraits : https://fr.wikipedia.org/wiki/Couverture_de_code
et Définition de la couverture de code illustrée avec Junit : http://www.junit.fr/2012/04/03/definition-de-la-couverture-de-code-illustree-avec-junit/

Plusieurs niveaux de couverture de code

→ couverture des fonctions (***Function Coverage***) :

*Chaque **fonction** dans le programme a-t-elle été appelée ?*

→ couverture des instructions (***Statement Coverage***) :

*Chaque **ligne du code (instruction)** a-t-elle été exécutée et vérifiée ?*

→ couverture des points de tests (***Condition Coverage***) :

*Chaque **point d'évaluation** (tel que le test d'une variable) a-t-il été exécuté et vérifié ?*

→ couverture des chemins d'exécution (***Path Coverage***) :

*Chaque **parcours possible** (par exemple les 2 cas vrai et faux d'un test) a-t-il été exécuté et vérifié*

Extraits : https://fr.wikipedia.org/wiki/Couverture_de_code
et Définition de la couverture de code illustrée avec Junit : http://www.junit.fr/2012/04/03/definition-de-la-couverture-de-code-illustree-avec-junit/

Exemple de couverture de code (réalisée par l'IDE)

Code de Production
(dans un source folder **src**)

Calculatrice.java

```java
package calculatrice;

public class Calculatrice {

    public int additionner(int terme1, int terme2) {
        return terme1 + terme2;
    }

    public int soustraire(int terme1, int terme2) {
        return terme1 - terme2;
    }

    //TODO : multiplier et diviser

}
```

Code de Test
(dans un source folder **test**)

CalculatriceTest.java

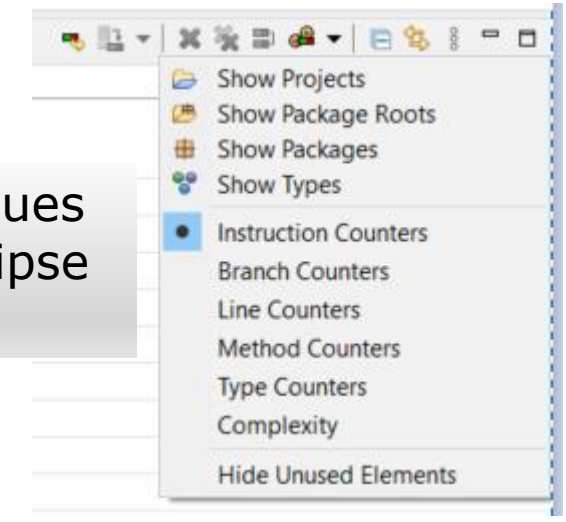
```java
package calculatrice;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class CalculatriceTest {

    @Test
    void testAdditionerDeuxEntiersPositifs() {

        Calculatrice calculatrice = new Calculatrice();

        assertEquals(10, calculatrice.additionner(6, 4));
    }

}
```

Visualisation directe dans le code de production

Et mise à disposition de **métriques** ..

Calculatrice.java	63,6 %
Calculatrice	63,6 %
soustraire(int, int)	0,0 %
additionner(int, int)	100,0 %

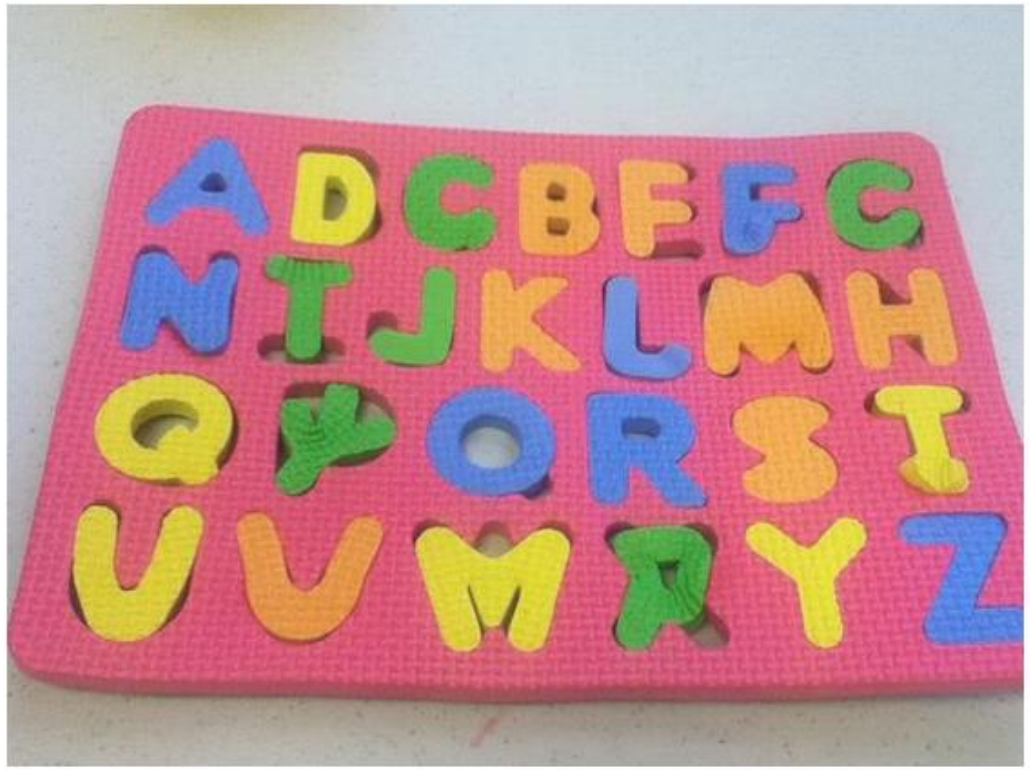
Différentes métriques possibles sous Eclipse avec EclEmma



Mais, gardez aussi bien à l'esprit que la couverture ne garantit pas la justesse du code

100% code coverage
"@bloerwald: SUCCESS: 26/26 (100%)
Tests passed "

Répondre Retweeté Ajouté aux favoris Plus



Extrait : https://twitter.com/francesc/status/449206943987793920

Extrait : https://martinfowler.com/bliki/TestCoverage.html

Et toujours pensez à aller chercher les cas aux limites du système à tester

Pour notre **`Calculatrice`**, pour l'opération diviser, quid d'un **diviseur égal à zéro** ?

⇒ Pour un programme robuste de qualité, il faudra penser à écrire le test et à implémenter le bon comportement ;-)

Extrait : https://twitter.com/yukselisim/status/1105430379132084224

Quelques mots sur les annotations de JUnit

Annotations (@) de JUnit5 ...

Package org.junit.jupiter.api

`package org.junit.jupiter.api`

Annotation Interface	Description
AfterAll	`@AfterAll` is used to signal that the annotated method should be executed *after* **all** tests in the current test class.
AfterEach	`@AfterEach` is used to signal that the annotated method should be executed *after* **each** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, and `@TestTemplate` method in the current test class.
BeforeAll	`@BeforeAll` is used to signal that the annotated method should be executed *before* **all** tests in the current test class.
BeforeEach	`@BeforeEach` is used to signal that the annotated method should be executed *before* **each** `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, and `@TestTemplate` method in the current test class.
Disabled	`@Disabled` is used to signal that the annotated test class or test method is currently *disabled* and should not be executed.
DisplayName	`@DisplayName` is used to declare a custom display name for the annotated test class or test method.
DisplayNameGeneration	`@DisplayNameGeneration` is used to declare a custom display name generator for the annotated test class.
IndicativeSentencesGeneration	`@IndicativeSentencesGeneration` is used to register the `DisplayNameGenerator.IndicativeSentences` display name generator and configure it.
Nested	`@Nested` is used to signal that the annotated class is a nested, non-static test class (i.e., an *inner class*) that can share setup and state with an instance of its enclosing class.
Order	`@Order` is an annotation that is used to configure the order in which the annotated element (i.e., field, method, or class) should be evaluated or executed relative to other elements of the same category.
RepeatedTest	`@RepeatedTest` is used to signal that the annotated method is a *test template* method that should be repeated a specified number of times with a configurable display name.
Tag	`@Tag` is a repeatable annotation that is used to declare a *tag* for the annotated test class or test method.
Tags	`@Tags` is a container for one or more `@Tag` declarations.
Test	`@Test` is used to signal that the annotated method is a *test* method.
TestClassOrder	`@TestClassOrder` is a type-level annotation that is used to configure a `ClassOrderer` for the `@Nested` test classes of the annotated test class.
TestFactory	`@TestFactory` is used to signal that the annotated method is a *test factory* method.
TestInstance	`@TestInstance` is a type-level annotation that is used to configure the lifecycle of test instances for the annotated test class or test interface.
TestMethodOrder	`@TestMethodOrder` is a type-level annotation that is used to configure a `MethodOrderer` for the *test methods* of the annotated test class or test interface.
TestTemplate	`@TestTemplate` is used to signal that the annotated method is a *test template* method.
Timeout	`@Timeout` is used to define a timeout for a method or all testable methods within one class and its `@Nested` classes.

@Test déjà connu ...

Extrait de https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/package-summary.html

Annotations relatives au cycle de vie du test : Eviter la duplication (DRY) ...

```java
public class MonTest {

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

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

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

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

  @Test
  void simpleTest() {
    System.out.println("---");
    System.out.println("inside a test");
    System.out.println("---");
    Assertions.assertTrue(true);
  }

  @Test
  void secondTest() {
  System.out.println("---");
    System.out.println("inside another test");
    System.out.println("---");
    Assertions.assertTrue(true);
  }
}
```

1 seule fois AVANT l'exécution du premier test de la classe

AVANT chaque méthode

APRES chaque méthode

1 seule fois APRES l'exécution de tous les tests de la classe

Affichage console à l'exécution des tests

```
beforeAll
beforeEach
---
inside a test
---
afterEach
beforeEach
---
inside another test
---
afterEach
afterAll
```

Annotations pour écrire des *tests paramétrés* … et rester DRY ;-)

2.15. Parameterized Tests

Parameterized tests make it possible to run a test multiple times with different arguments. They are declared just like regular `@Test` methods but use the `@ParameterizedTest` annotation instead. In addition, you must declare at least one *source* that will provide the arguments for each invocation and then *consume* the arguments in the test method.

The following example demonstrates a parameterized test that uses the `@ValueSource` annotation to specify a `String` array as the source of arguments.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

Ecriture d'un seul test au lieu de trois !!!

When executing the above parameterized test method, each invocation will be reported separately. For instance, the `ConsoleLauncher` will print output similar to the following.

```
palindromes(String) ✓
├─ [1] candidate=racecar ✓
├─ [2] candidate=radar ✓
└─ [3] candidate=able was I ere I saw elba ✓
```

https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests

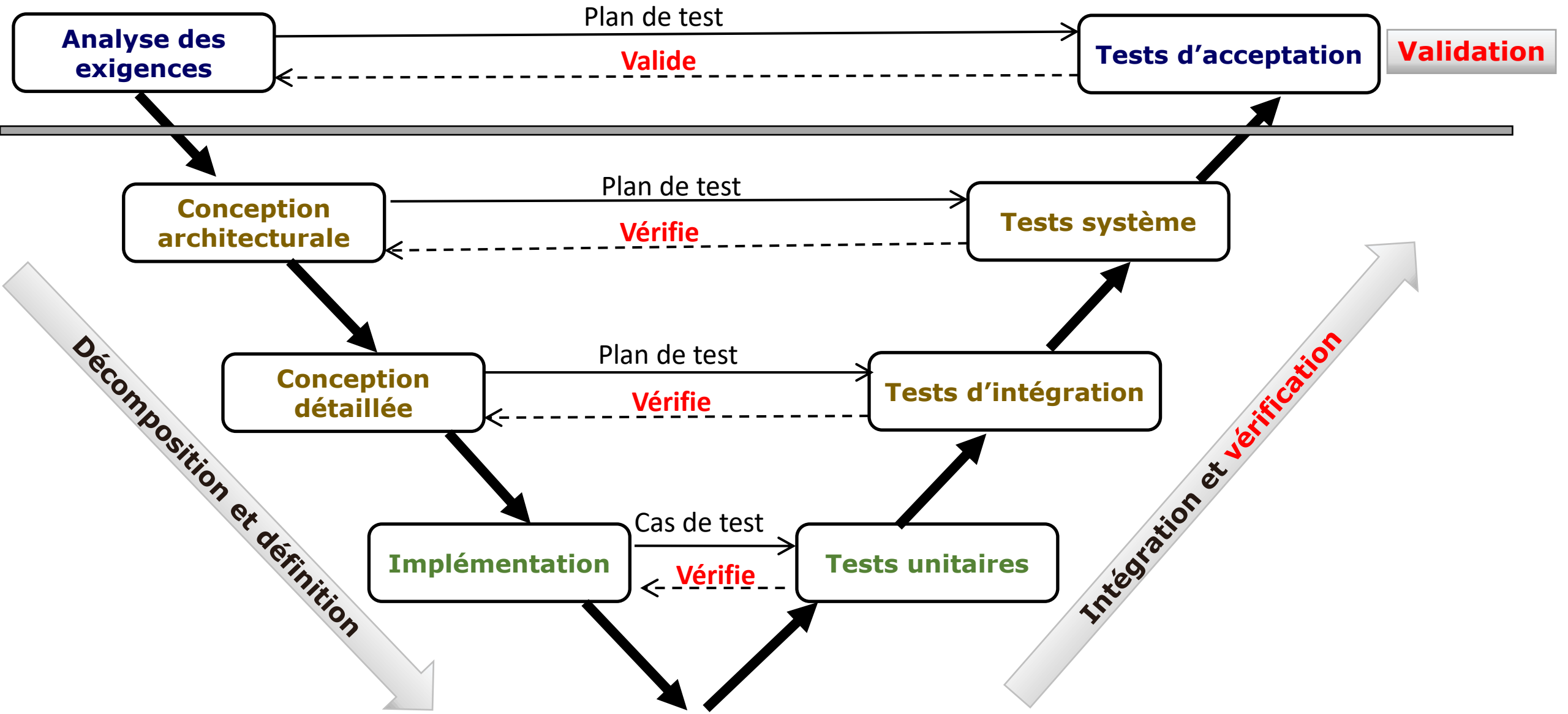
Timely
(FIRST)

Comparatif des deux approches de développement logiciel

(*par la place du test dans le cycle de développement*)

Isabelle BLASQUEZ

Le test dans un cycle en V (dév. classique)



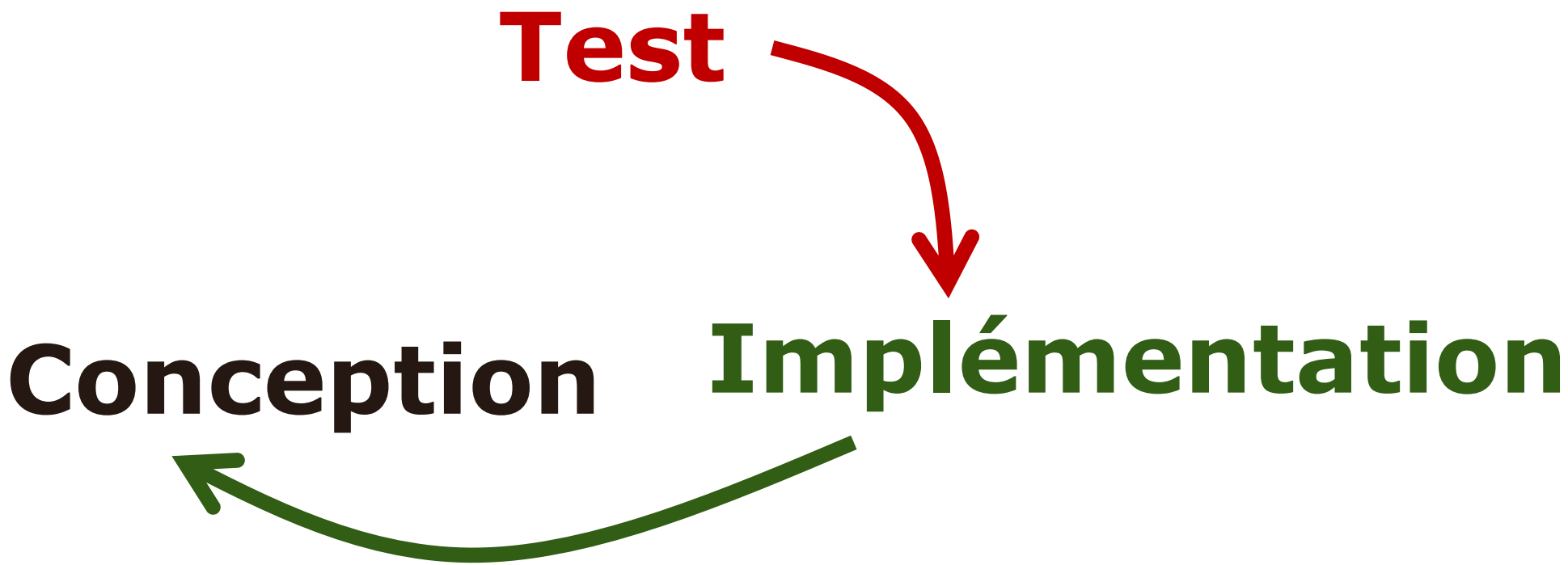
En savoir plus sur : http://fr.wikipedia.org/wiki/Cycle_en_V

Quand tester ?

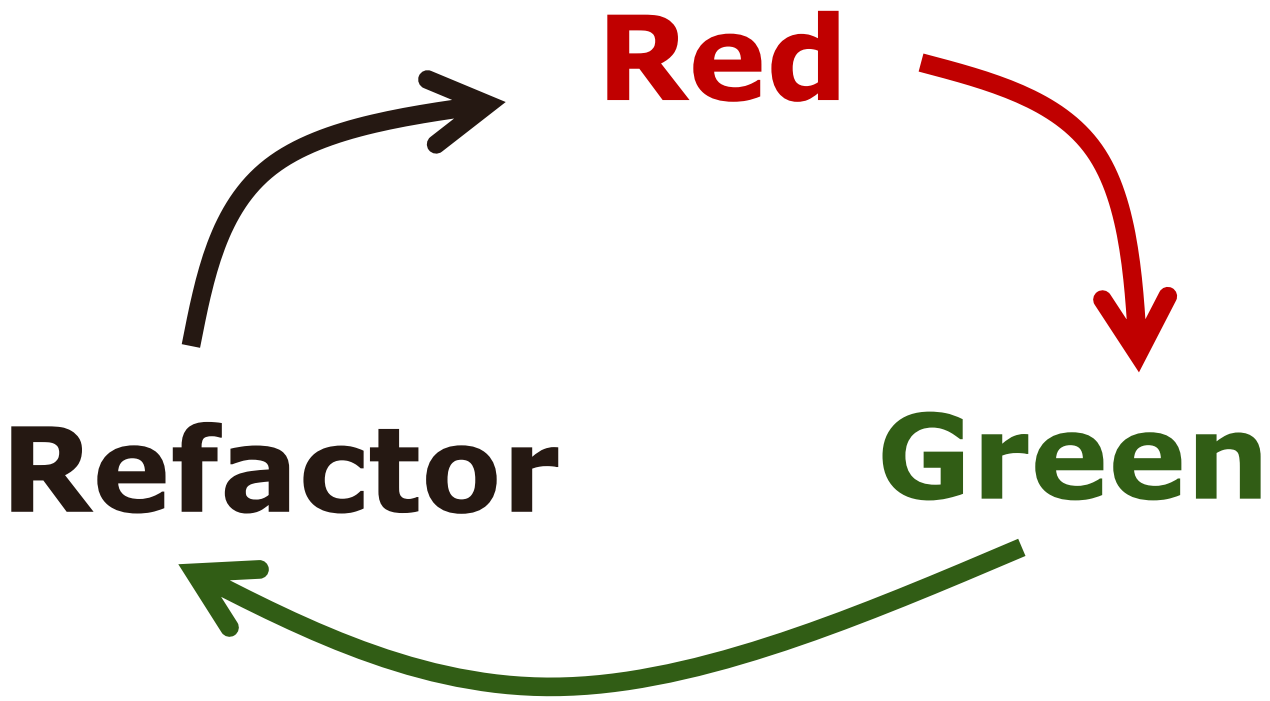
Développement classique
(type Cycle en V)
⇒ Test APRES (a posteriori)…

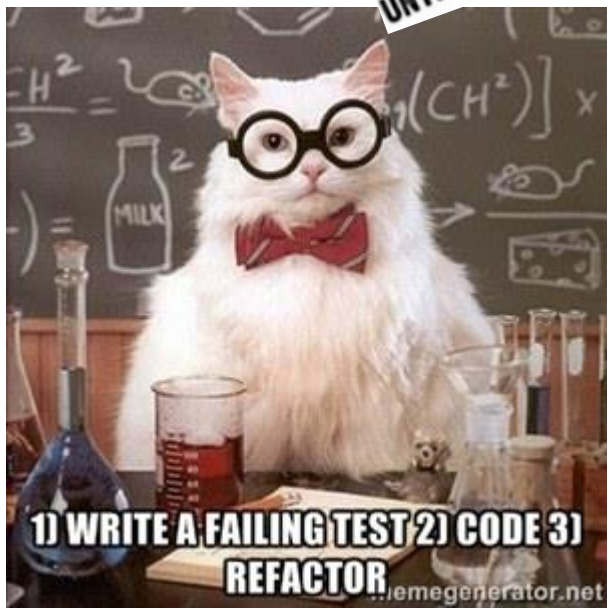
… Et si on testait AVANT ?
(a priori)

Test **FIRST** (dit aussi Test **a priori**)



Test **D**riven **D**eveloppement





Itérations « baby-step » rapides
(de quelques secondes à quelques minutes)

Image : http://blog.octo.com/coder-a-pas-de-chaton-a-lecole-du-tech-lead/

TDD en pratique

Pas de code de production sans test !

Un nouveau test =
un nouveau comportement

Ecrire au plus vite un code
qui fait passer les tests

Deux approches pour le développement logiciel

Top-down

BUT 1

Approche descendante :
Des modèles au code …

Conception

Implémentation

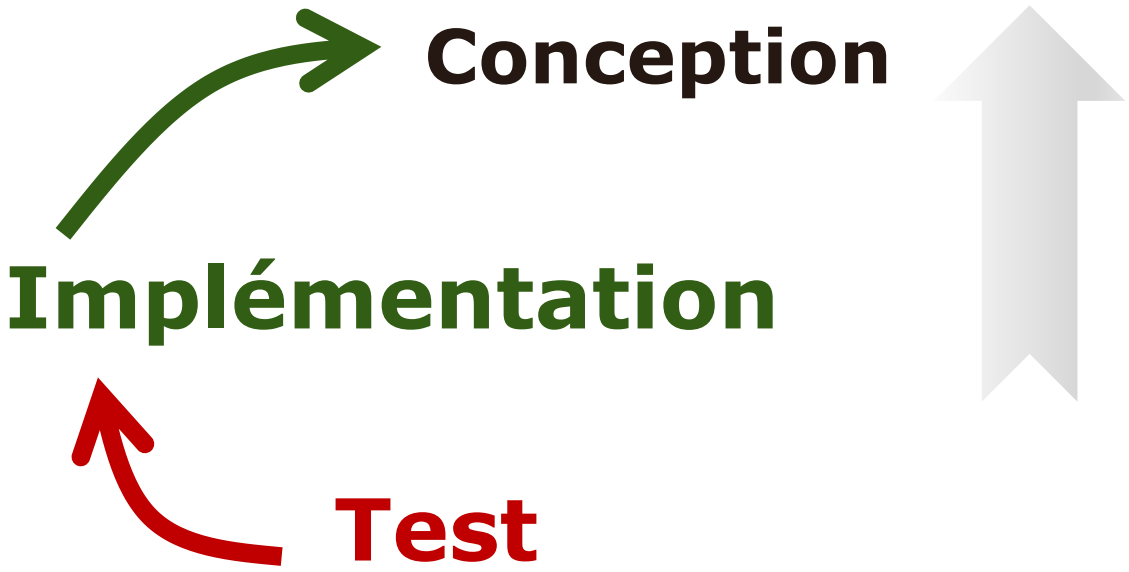
Test

comme dans le cycle en V
*approche **Model** Driven :*
modèles** au cœur du processus de **Conception

Bottom-up

BUT 3

Approche ascendante :
Conception émergente
grâce au refactoring du code et aux tests



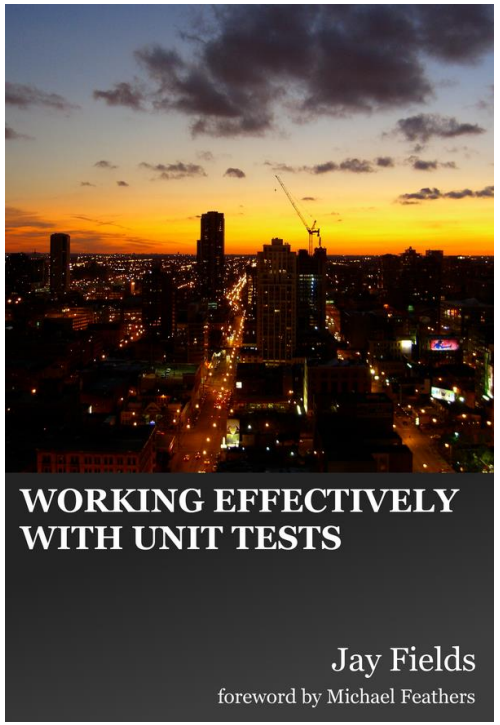
comme dans le TDD
*approche **Test** Driven :*
tests** au cœur du processus de **Conception

Conclusion

Pourquoi écrire des tests ?

There are many motivators for creating a test or several tests:

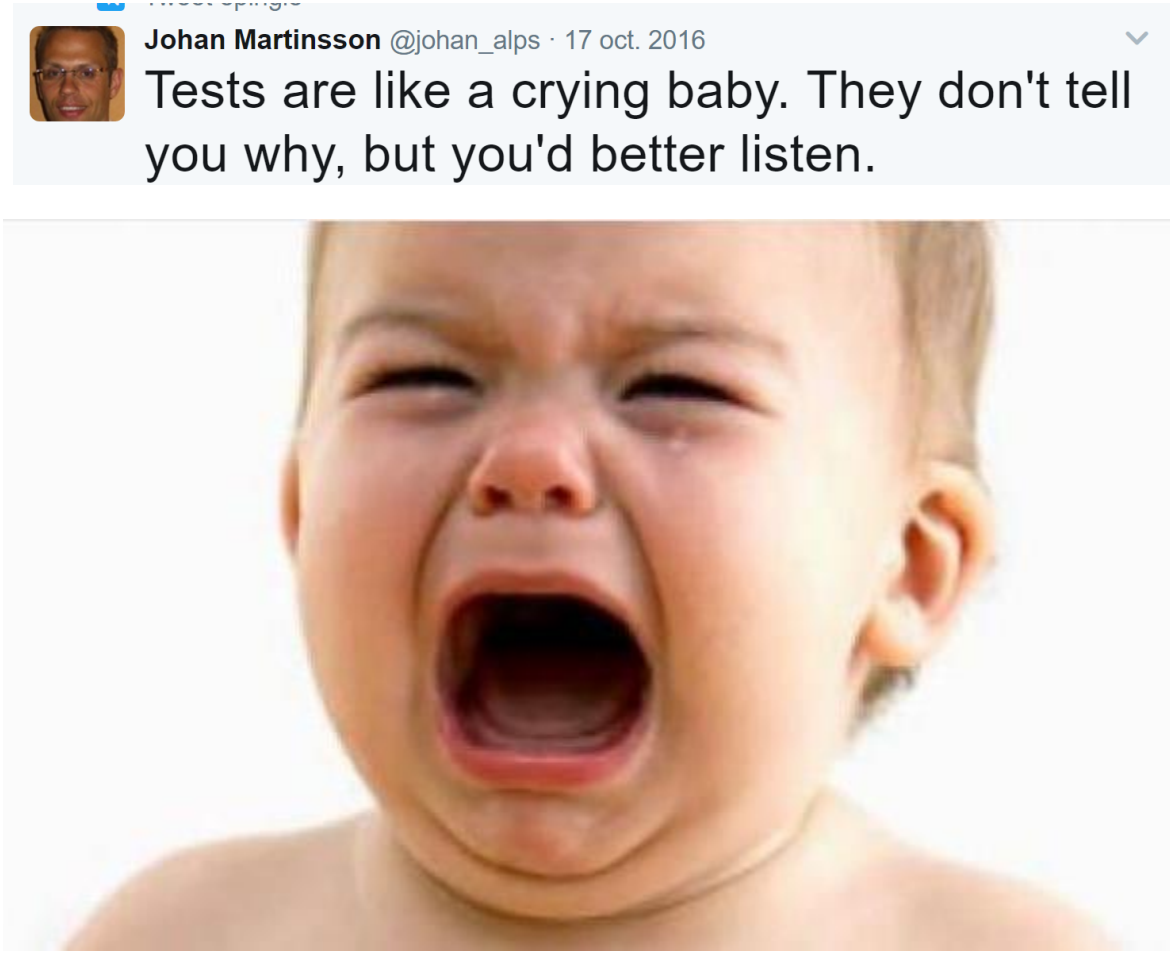
- **validate** the system
  - immediate feedback that things work as expected
  - prevent future regressions
- increase **code-coverage**
- enable **refactoring**
- **document the behavior of the system**
- your manager told you to
- **Test Driven Development**
  - improved design
  - breaking a problem up into smaller pieces
  - defining the “simplest thing that could possibly work”
- **customer acceptance**
- **ping pong pair-programming**



Extrait : https://leanpub.com/wewut

If you first understand
why you’re writing a test,
you’ll have a much better chance of
writing a test that is maintainable
and will make you more productive
in the long run.

Le mot de la fin …



Extraits de https://twitter.com/CommitStrip_fr/status/829624172628295680
https://twitter.com/johan_alps/status/788126453133107202
https://www.kangouroukids.fr/actualites/il-pleure-que-faire.html