Quid du Test dans un développement logiciel ?

Isabelle BLASQUEZ
@iblasquez

2020

Isabelle BLASQUEZ

@iblasquez

Enseignement : Génie Logiciel

Recherche : Développement logiciel agile

ICSTUG #IUTAgile

CodeWeek.

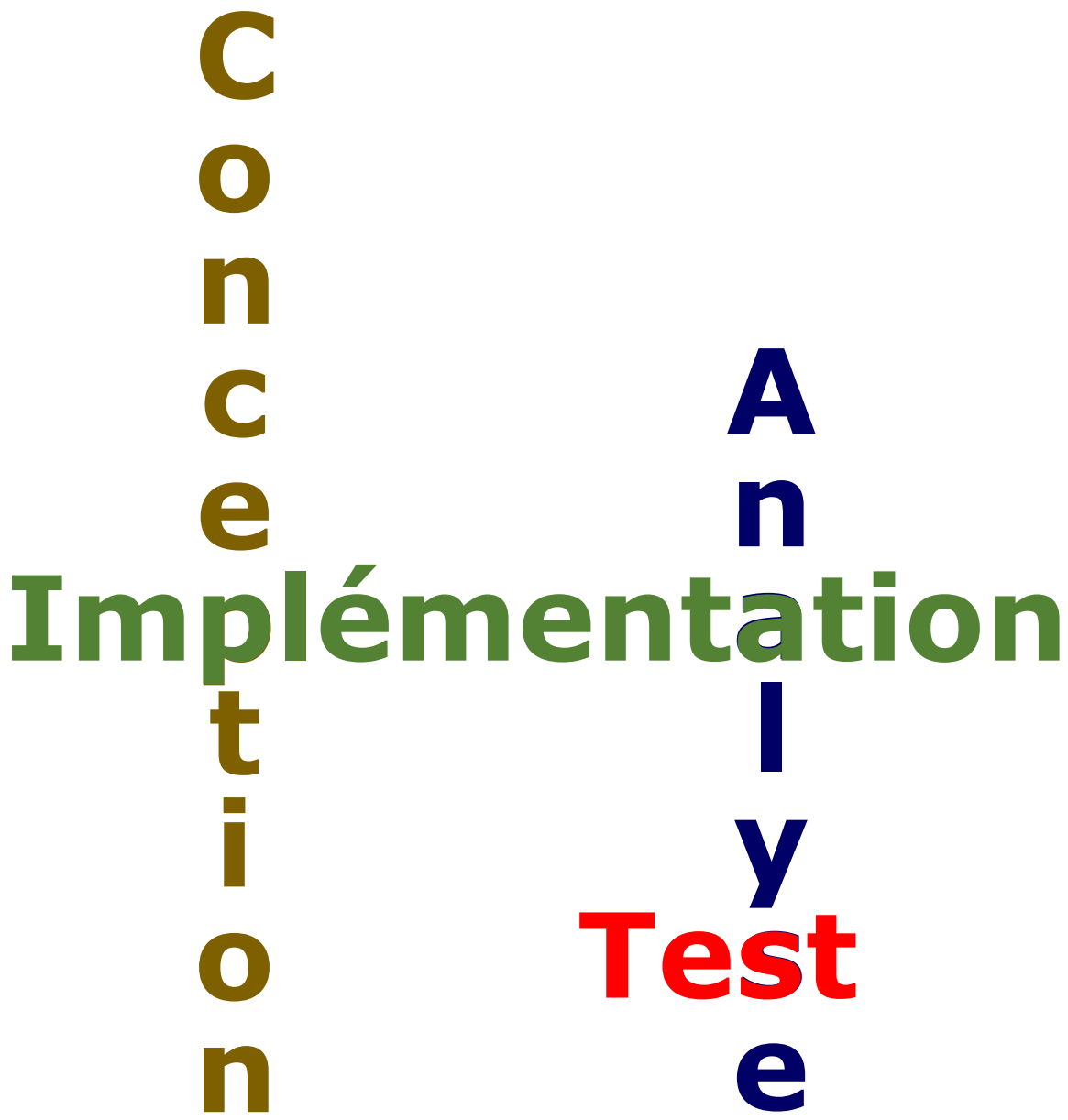
#Software Craftsmanship

MUSEOMIX LIMOUSIN

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

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)**.

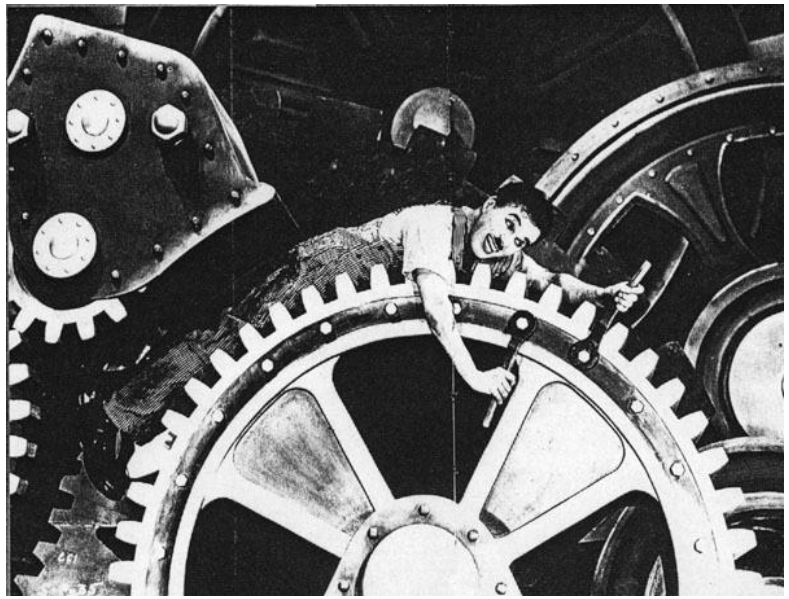
Les tâches de cette phase se concentrent autour du ***code*** *où les composants sont implémentés et testés individuellement* ***dans un langage de programmation*** *donné afin de mettre en œuvre la conception*

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

M2103

Bases de la programmation orientée objet

M2104

Bases de conception orientée objet

Implémentation (code)
Test (after)

documentation

test (first)

implémentation

Images : https://twitter.com/JS_Cheerleader/status/738527294969380868
https://openclipart.org/detail/130795/trollface
http://www.frenchweb.fr/craftsman-lartisan-du-code/221948

Isabelle BLASQUEZ

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

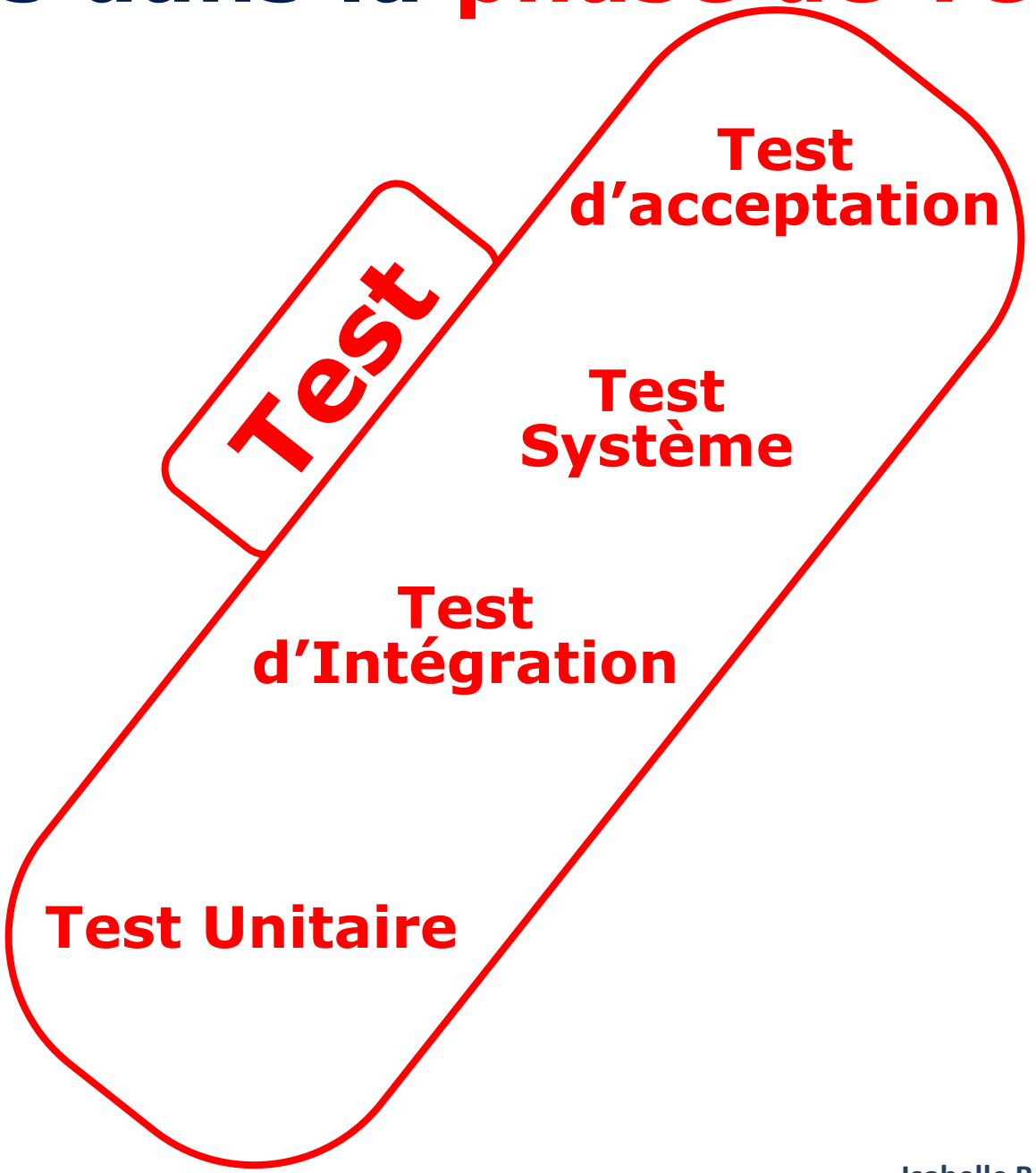
Image : http://www.99hdwallpaper.com/waterfall/waterfall-wallpapers-for-desktop.html

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

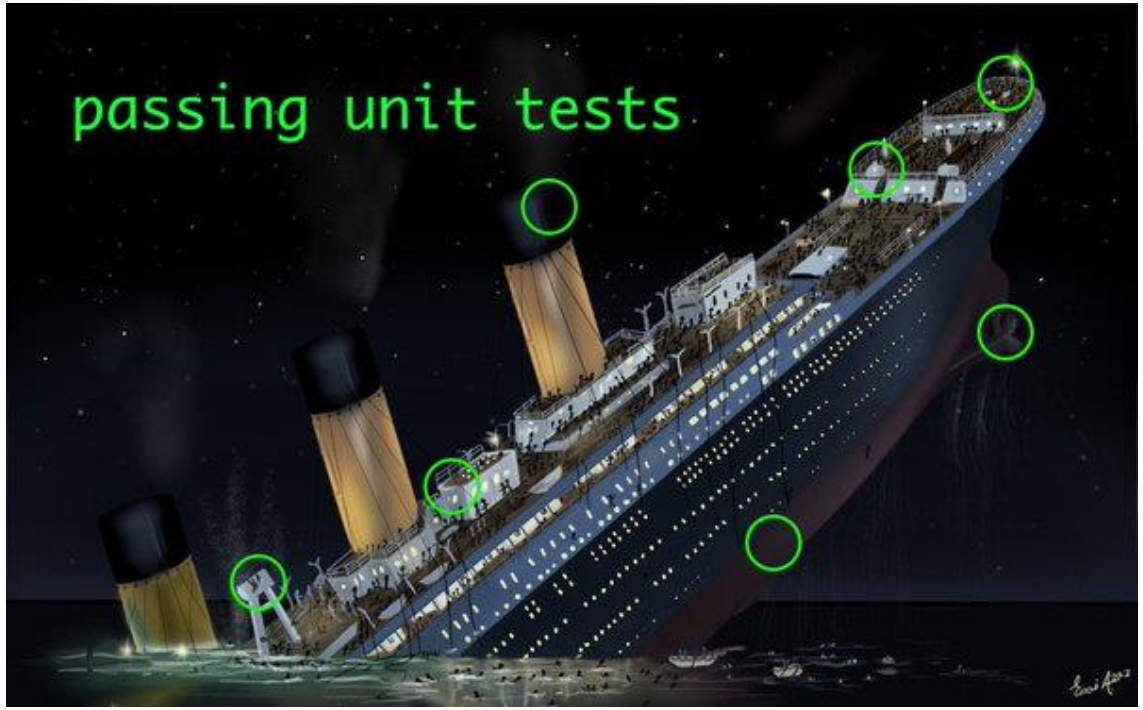
(Vérification des connexions d'interface des composants)

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)**.

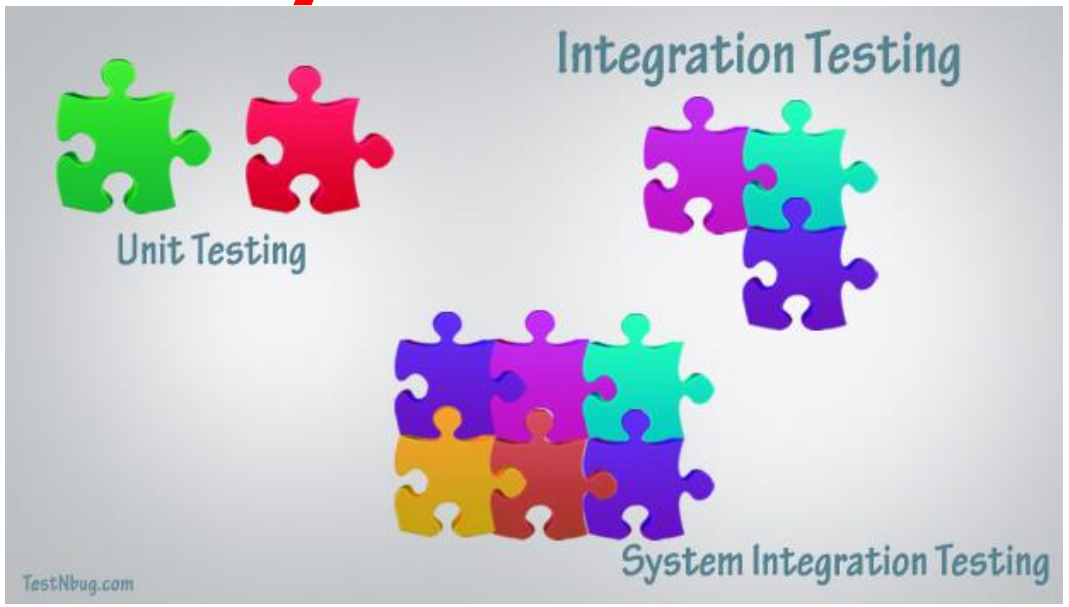
Extrait : http://keynote.hoa-project.net/ForumPHP12/

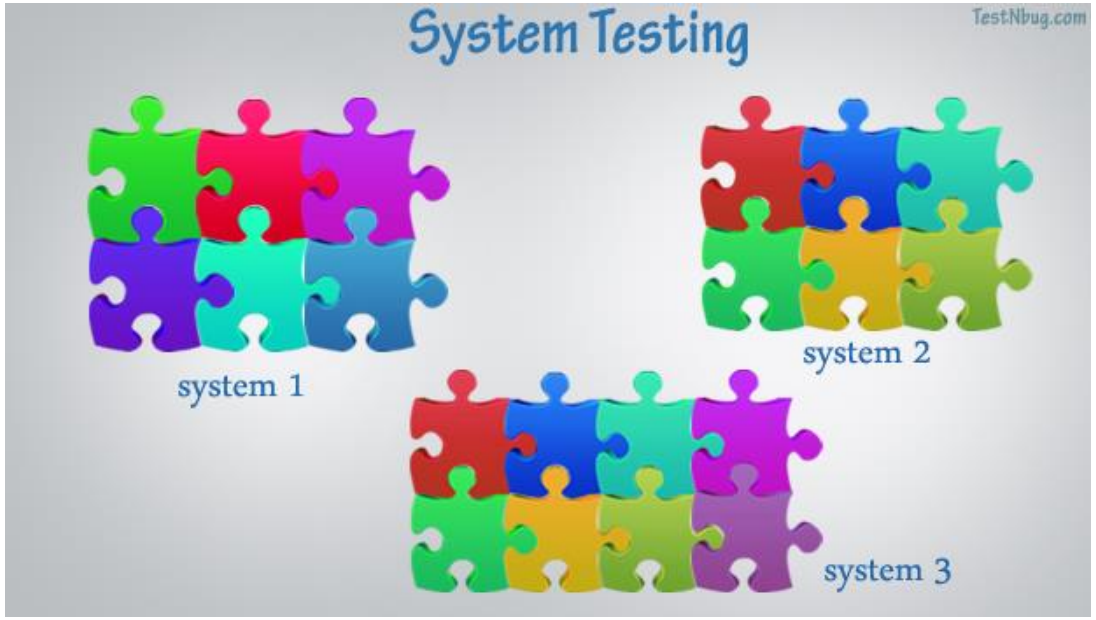
Les tests « isolés » ne suffisent pas …



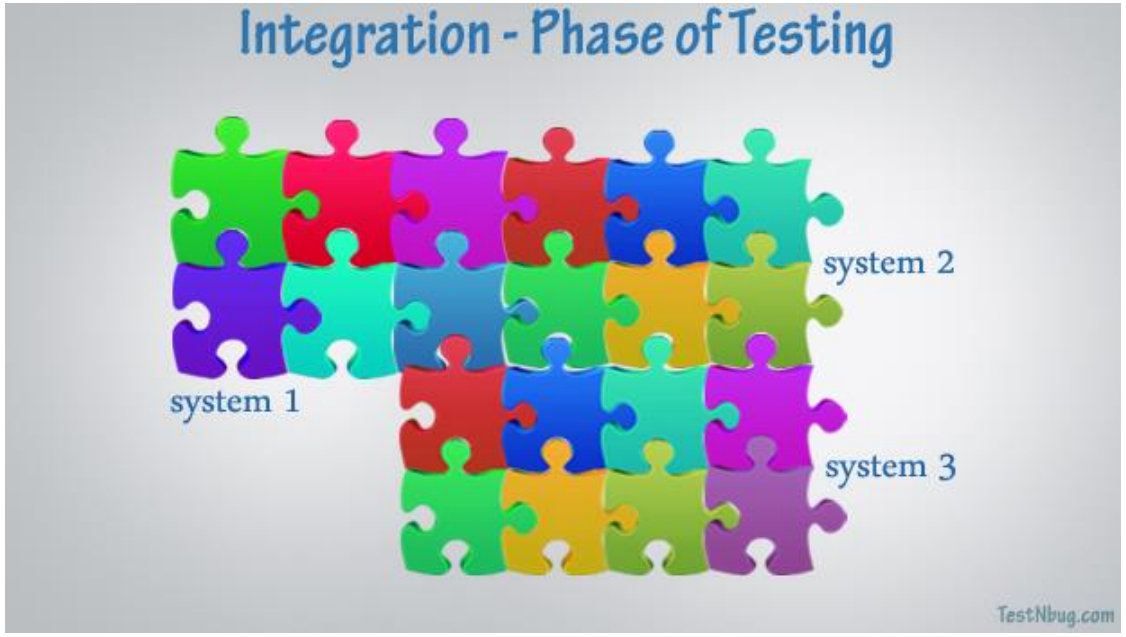
Source : https://twitter.com/ThePracticalDev/status/733908215021199360
https://twitter.com/catriona/status/733911005986578434

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**)

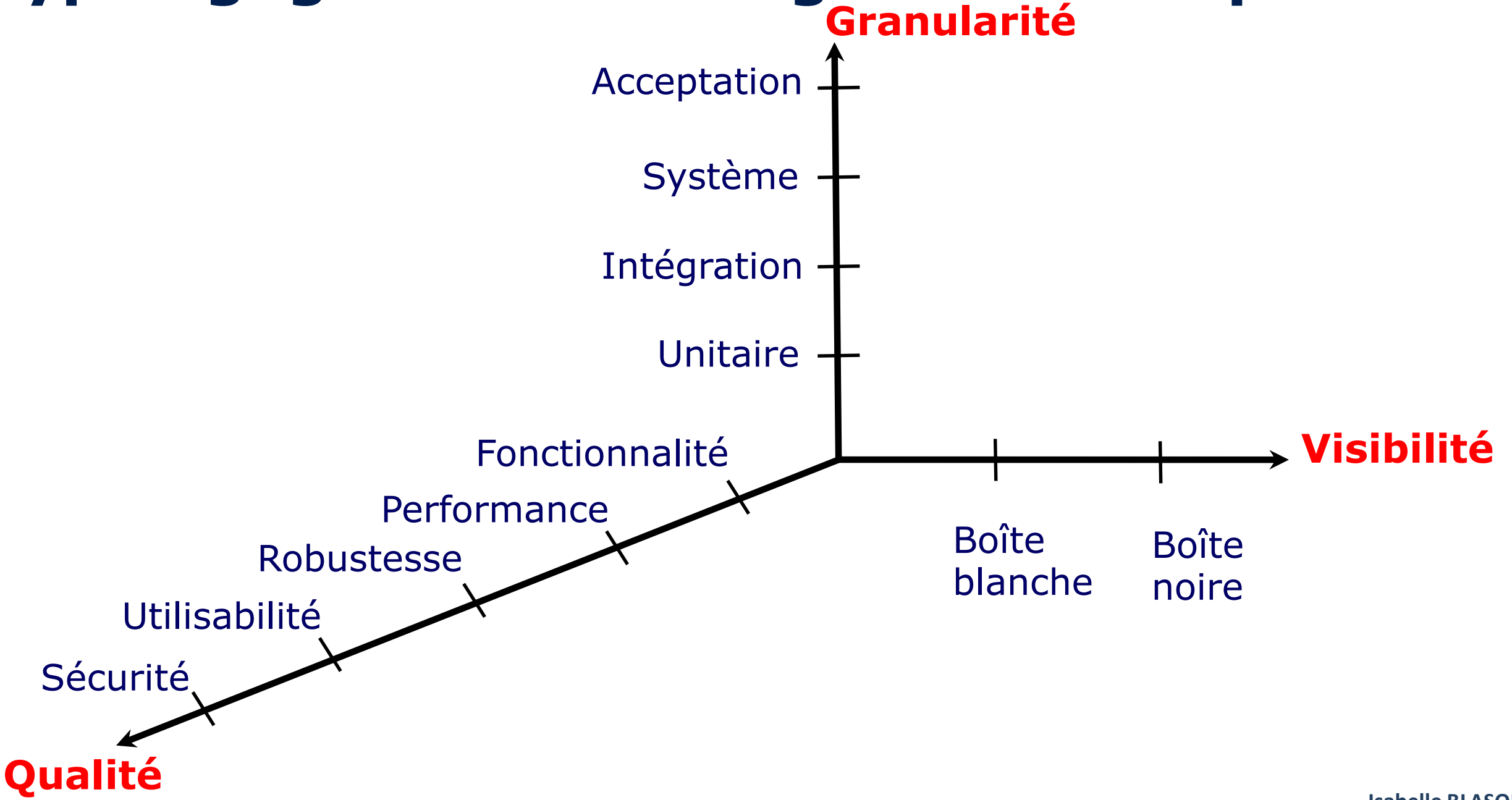
Right Product ?

Validité du système du point de vue client …

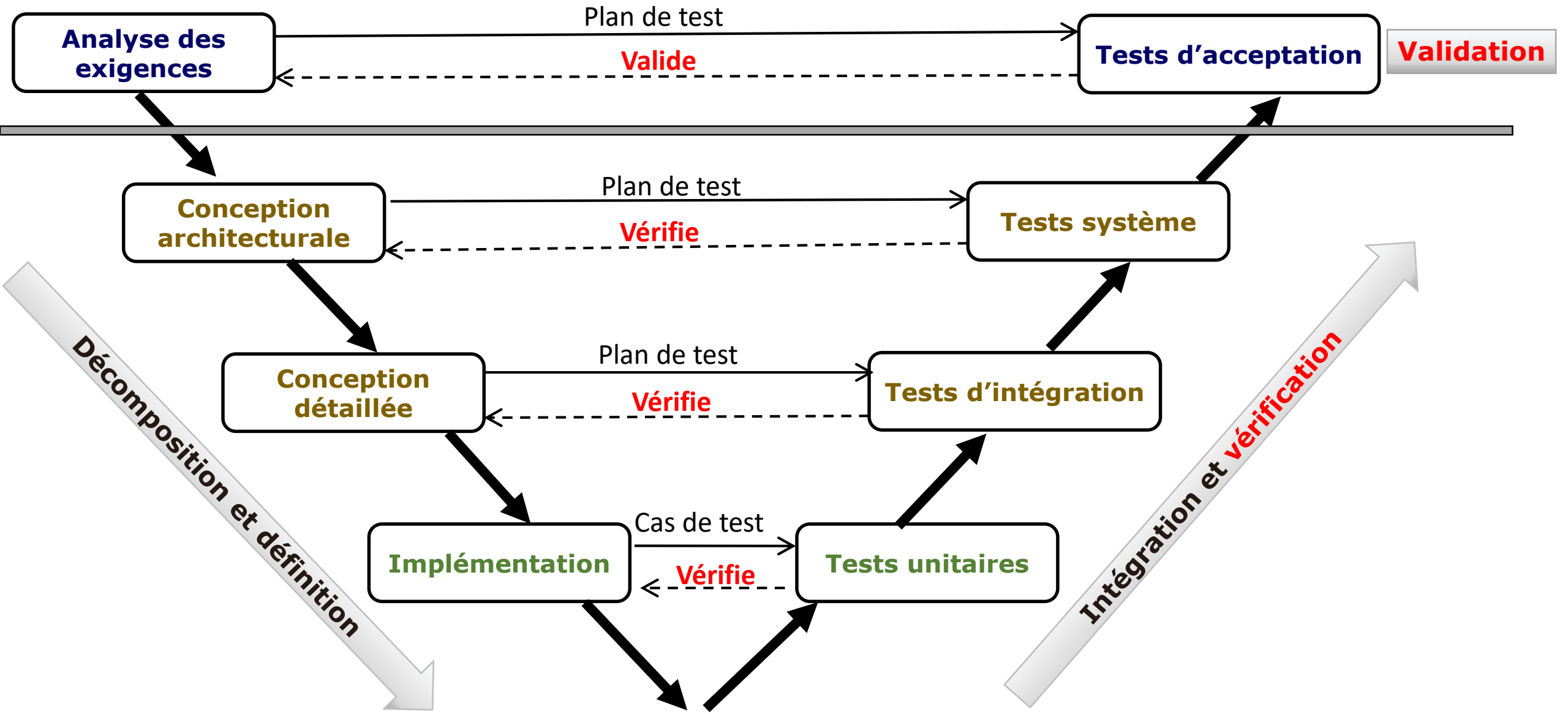
Un test d'acceptation *(ou anciennement test de recette) permet de déterminer si un système satisfait ou non à ses critères d'acceptation et permet au client d'accepter ou non le système* **(IEEE, 1990)**.

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



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



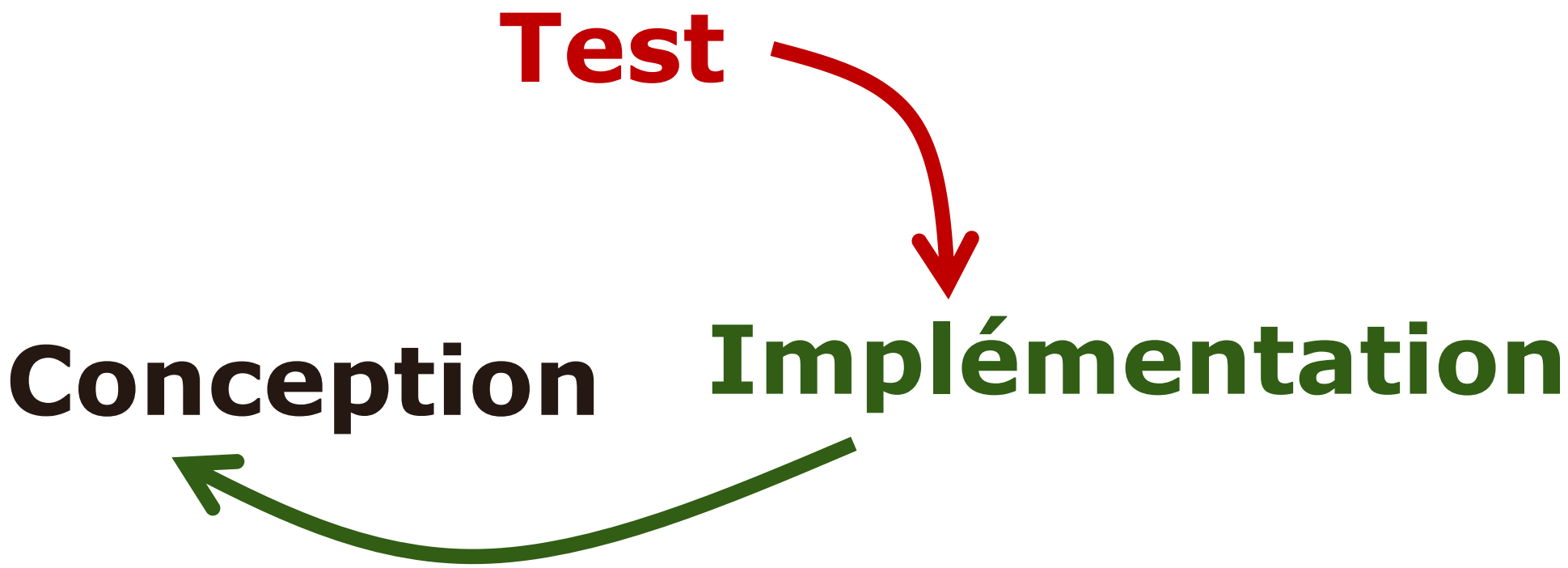
En savoir plus sur : http://fr.wikipedia.org/wiki/Cycle_en_V

Quand tester ?

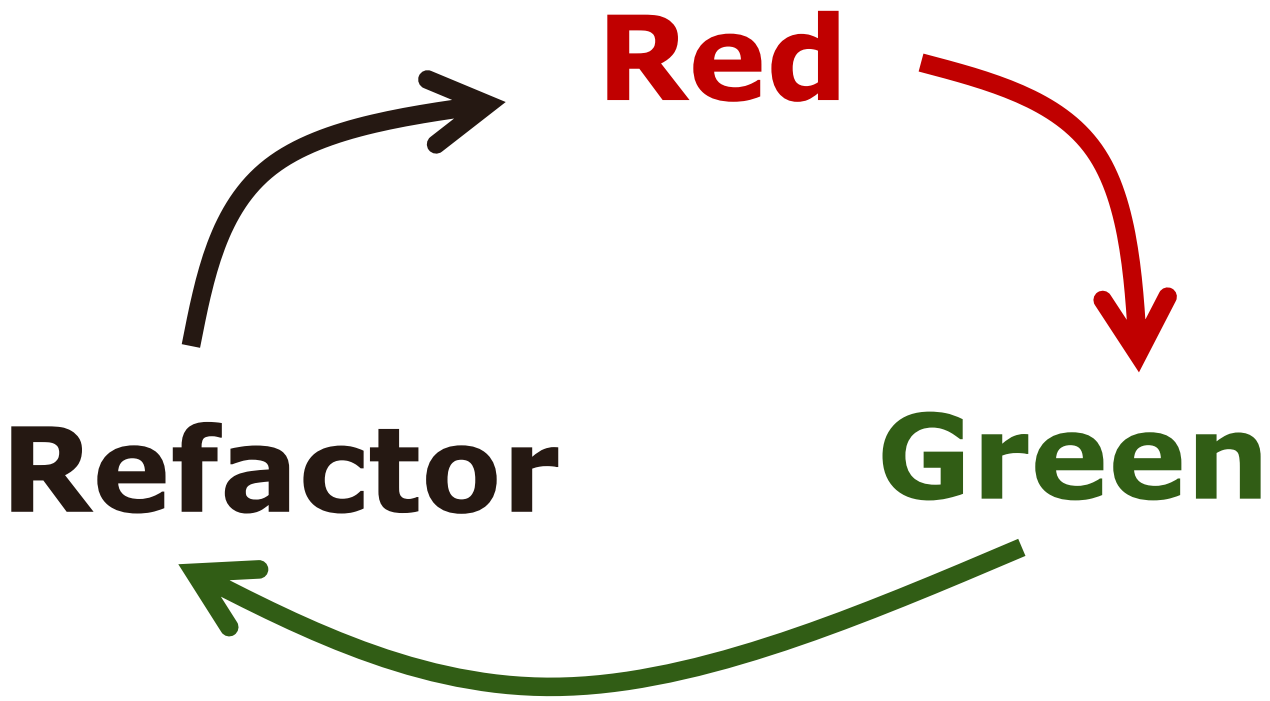
Développement classique
(type Cycle en V)
⇒ Test APRES (a posteriori)…

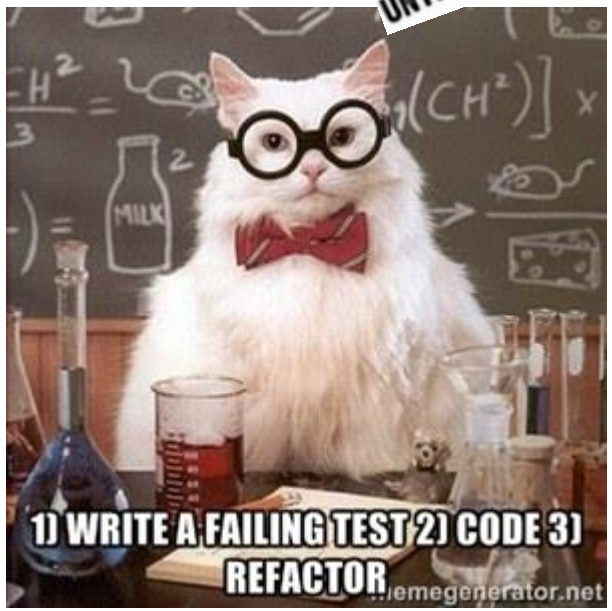
… Et si on testait AVANT ?
(a priori)

Test FIRST



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

Un test de qualité : *Clean Test*

FAST

INDEPENDANT

REPEATABLE

SELF-VALIDATING

TIMELY

Un test lisible : pattern AAA

ARRANGE

ACT

ASSERT

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

    String resultat = monObjet.maMethodeATester();

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

Exemple Simple de TDD

Code de tests

```
import static org.junit.Assert.*;
import org.junit.Test;

public class MaClasseShould {

@Test
public void returnHelloWorldWhenNothing() {
    MaClasse monObjet = new MaClasse();
    String resultat = monObjet.maMethodeATester();
    assertEquals("Hello World!", resultat);
   }
}
```

Ne pas écrire du code de production sans (code de) test !

Un nouveau test = un nouveau comportement

Code de Production

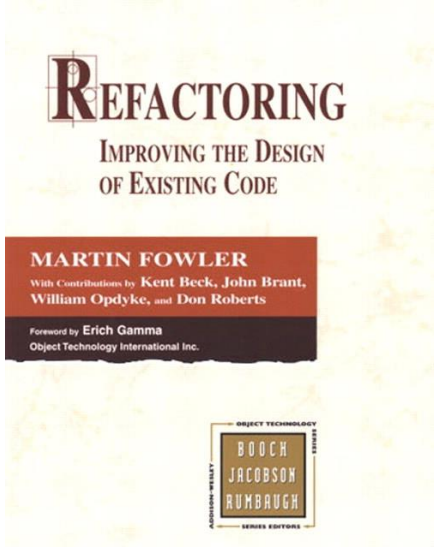
```
public class MaClasse {

public String maMethodeATester() {
    return "Hello World!";
   }

}
```

Ecrire au plus vite un code qui fait passer les tests

Refactoring pour produire un code de qualité



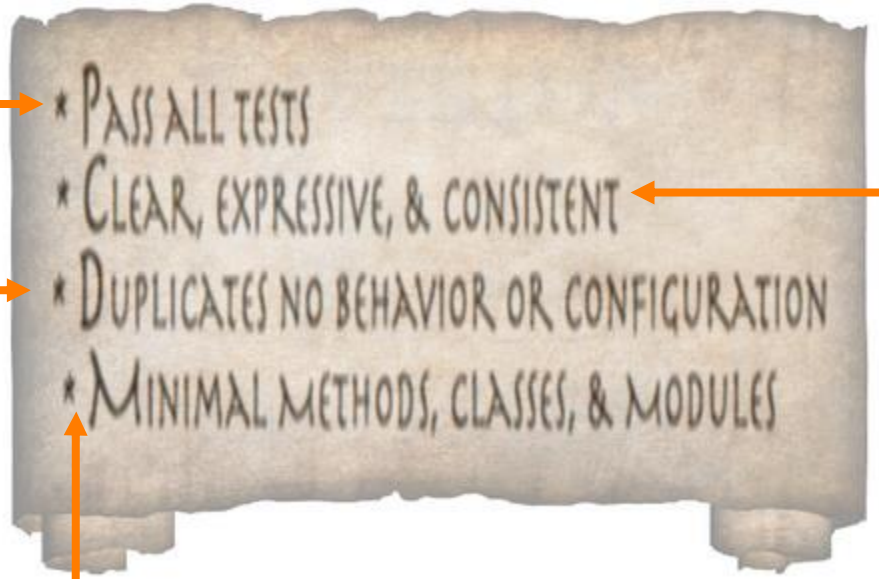
Un refactoring (remaniement) consiste à changer la structure interne d'un logiciel sans en changer son comportement observable (M. Fowler)

Le refactoring contribue à travers tous les cycles du TDD à l'obtention d'une ***conception simple et évolutive*** *appelée* ***conception émergente***

lisibilité
duplication
complexité

Les règles de simplicité (eXtreme Programming)

1. Tous les tests passent



2. Révéler l'intention du code (lisibilité) & consistance

3. DRY : pas de duplication (Don't Repeat Yourself)

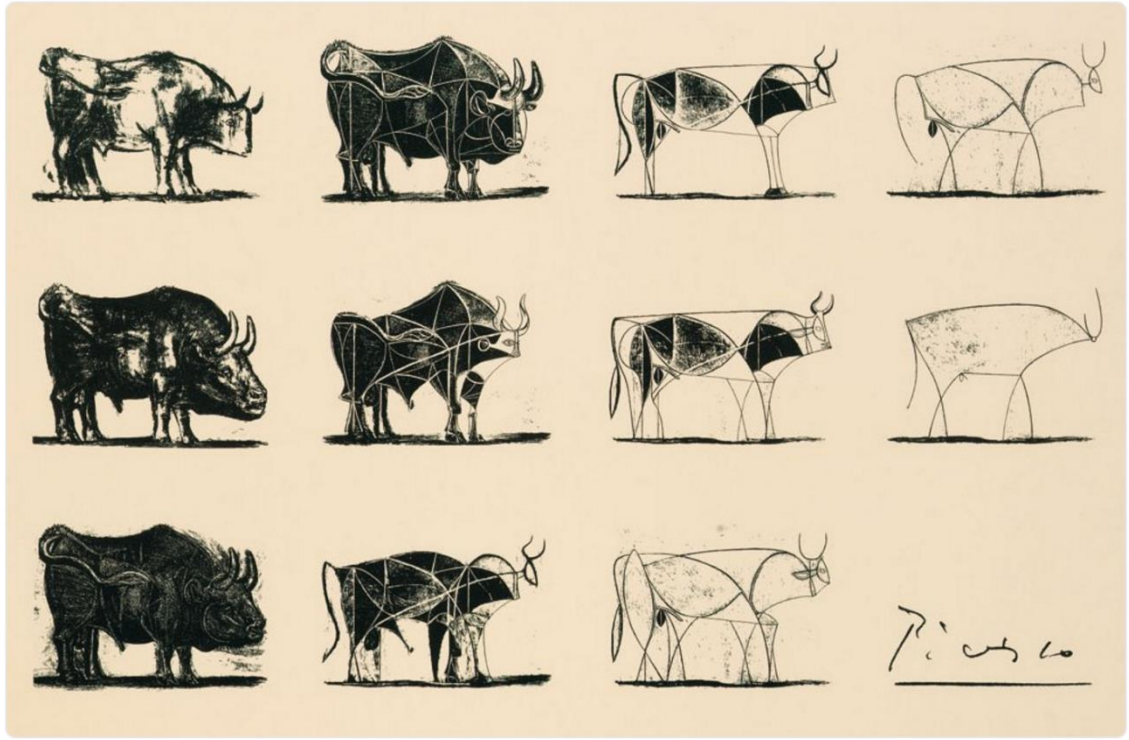
4. Éléments les plus petits possibles (complexité)

Extrait : http://c2.com/cgi/wiki?XpSimplicityRules
A lire également : http://martinfowler.com/bliki/BeckDesignRules.html
et https://leanpub.com/4rulesofsimpledesign

La simplicité n'est pas *simple (facile)* à mettre en oeuvre

Simplicity is the result of working incredibly hard at planning and refinement.

À l'origine en anglais



Extrait : https://twitter.com/hardmaru/status/842431377605713921

Kata Fizz Buzz : Le « Hello world ! » du TDD !

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz?".

Sample output:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
... etc up to 100
```

Enoncé complet sur :
http://codingdojo.org/cgi-bin/index.pl?KataFizzBuzz

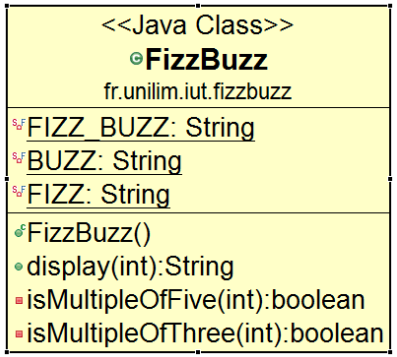
Une vidéo sur le Kata Fizz Buzz sur la chaine Crafties
https://www.youtube.com/watch?v=RWYvBNX9wcU

Image : http://agilekatas.co.uk/static/img/katas/kata_fizzbuzz.png

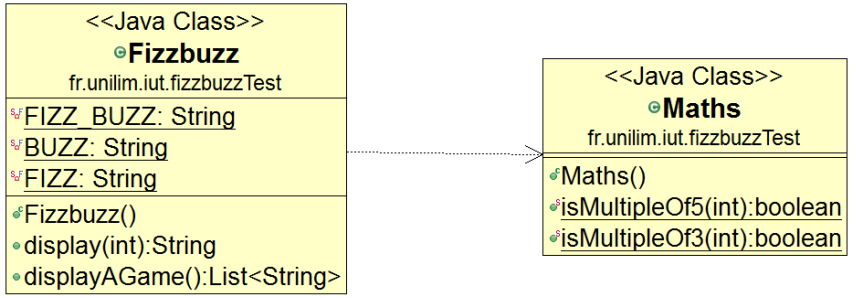
Isabelle BLASQUEZ

Exemples de conception pouvant émerger suite au développement du `FizzBuzz` en TDD (1/2)

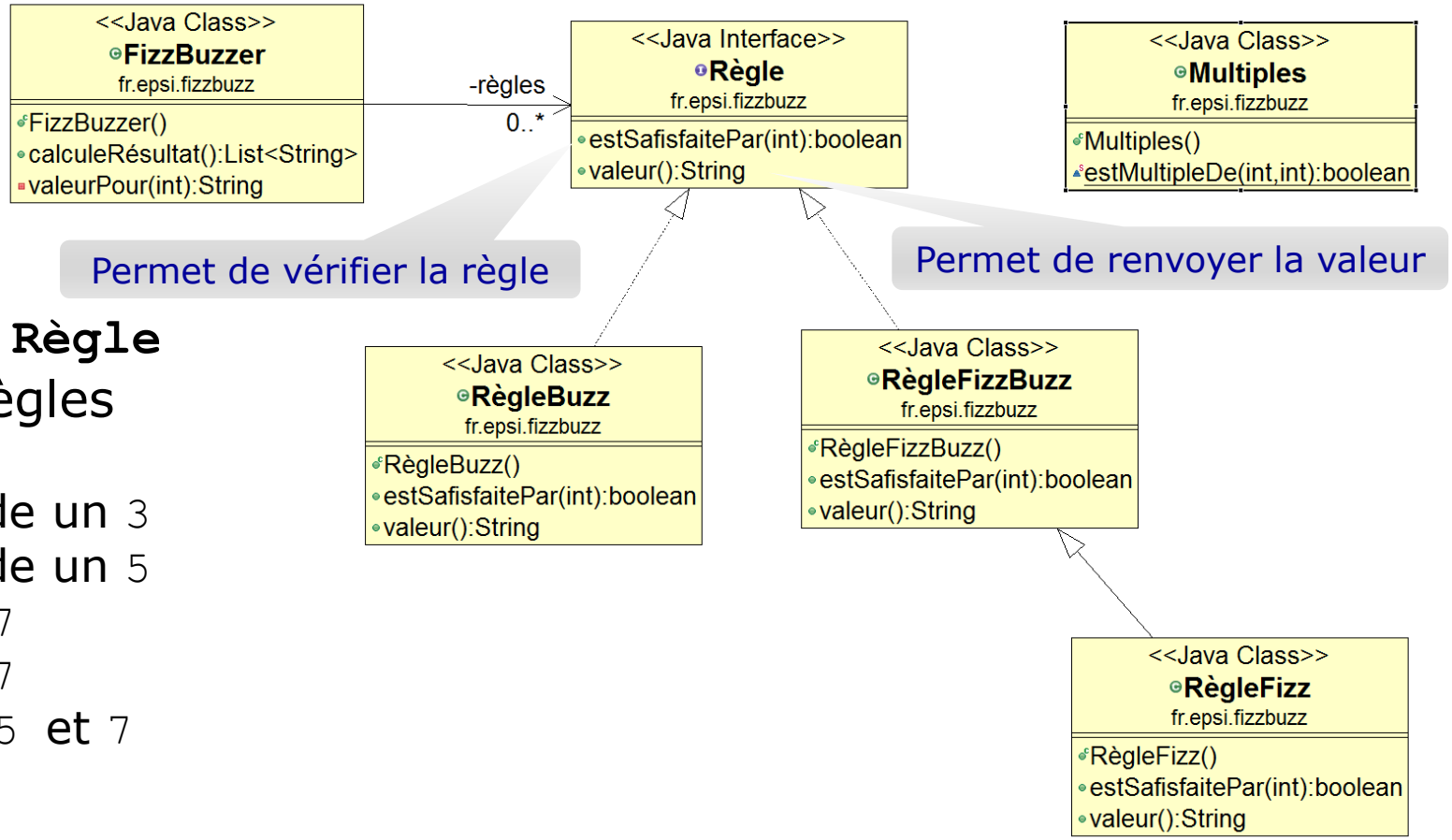
→ Le plus simple



→ Avec une classe **`Maths`**
Pour isoler la responsabilité de la reconnaissance des multiples



Exemples de conception pouvant émerger suite au développement du `FizzBuzz` en TDD (2/2)



→ En faisant apparaître une classe **`Règle`** pour faciliter l'ajout de nouvelles règles comme par exemple :

- dire `Fizz` pour un nombre qui possède un `3`
- dire `Buzz` pour un nombre qui possède un `5`
- dire `FizzBang` si multiple de `3` et de `7`
- dire `BuzzBang` si multiple de `5` et de `7`
- dire `FizzBuzzBang` si multiple de `3, 5` et `7`
- …

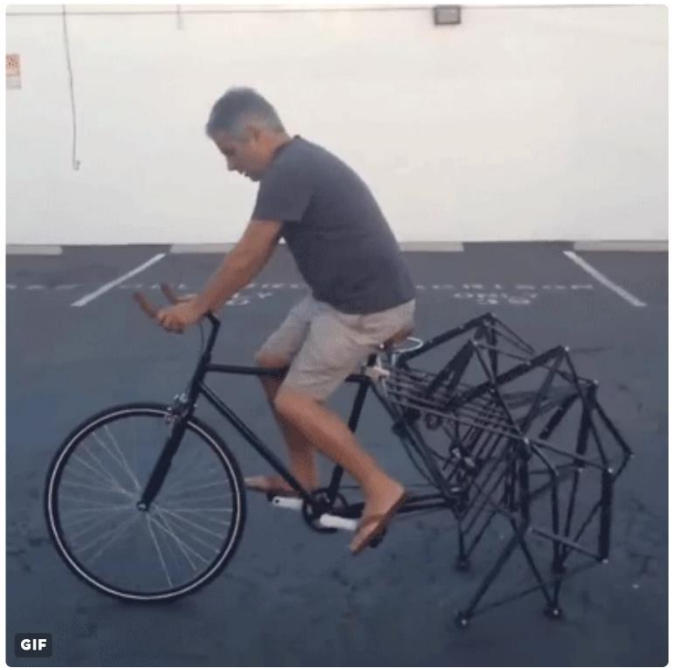
Diagramme de classes obtenu à partir du code disponible sur : https://github.com/BodySplash/FizzBuzzJava

Une implémentation sur le même principe est aussi disponible ici : https://github.com/JeffLeFoll/KataJava-BaseGradle

Restez KISS, évitez l'Over-Engineering !

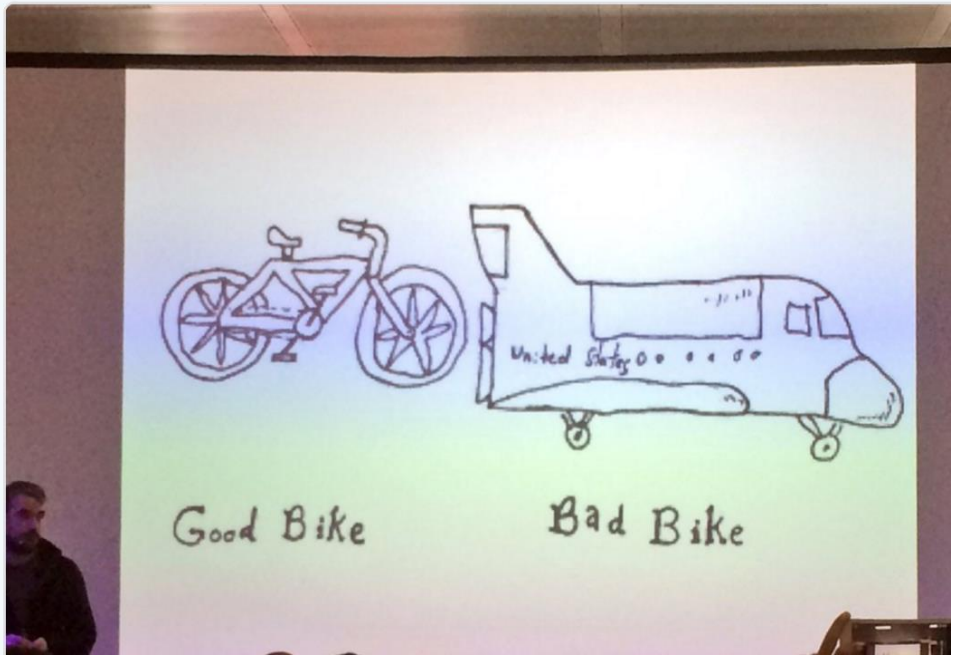
When I prematurely overengineer my code.
#programming

À l'origine en anglais



Extrait https://twitter.com/victorporof/status/833021513347559424

This! @gregyoung on Over-Engineering #dddx



Extrait : https://twitter.com/cyriux/status/741186682507137024

Voir le Kata Fizz Buzz Enterprise :

https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition

FizzBuzz Enterprise Edition is a no-nonsense implementation of FizzBuzz made by a serious businessman for serious business purposes.

TDD : une technique de conception (émergente)

The act of writing a unit test
is more an act of design than of verification.
It is also more an act of documentation than of verification
(Robert. C. Martin)

T*est* **D***riven*

D*eveloppement*
D*esign*

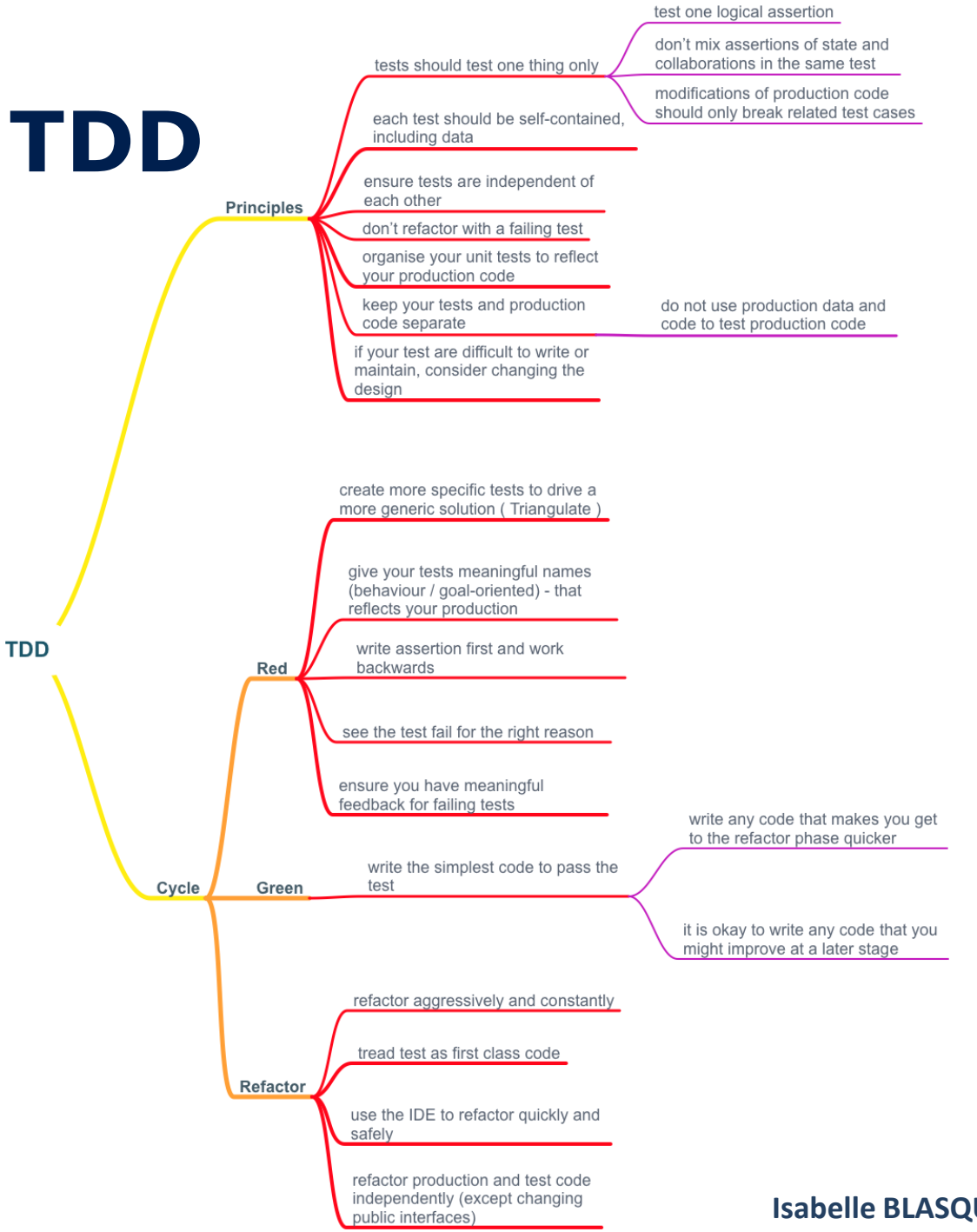
Good Habits Manifesto : Les bonnes pratiques du TDD sous forme de Mindmap

Source : https://github.com/neomatrix369/refactoring-developer-habits/blob/master/02-outcome-of-collation/tdd-manifesto/tdd-good-habits-manifesto.md

Voir aussi : http://fr.slideshare.net/neomatrix369/refactoring-developer-habits-62785350
https://github.com/neomatrix369/refactoring-developer-habits

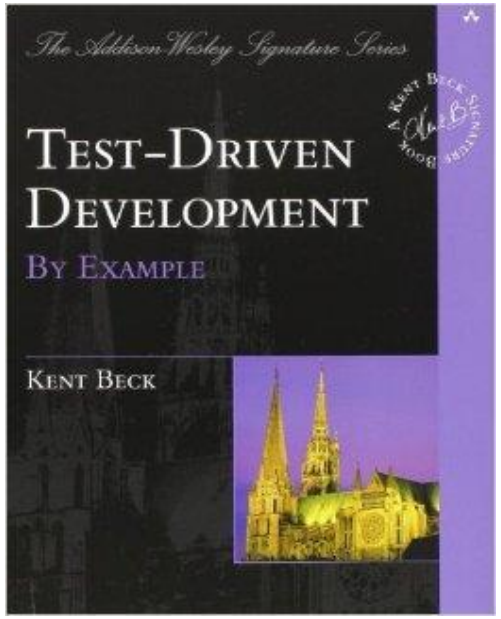
mani
neomatrix369

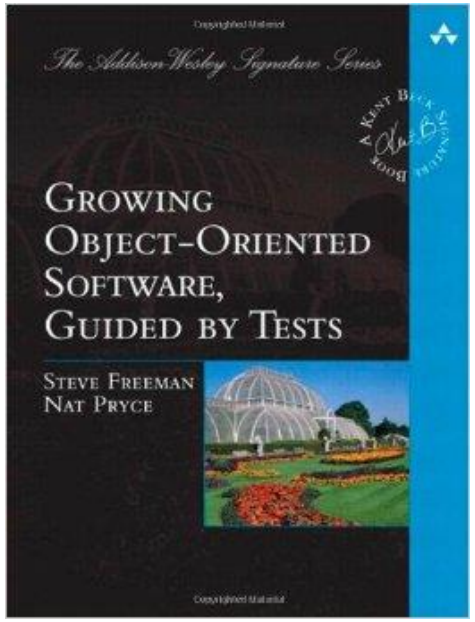
Passionate Java/JVM developer, advocate to F/OSS projects. A strong supporter of software craftsmanship principles. Developer communities, blogger & speaker.

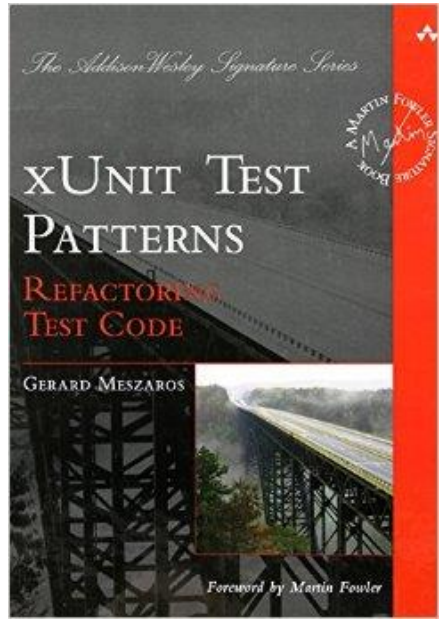


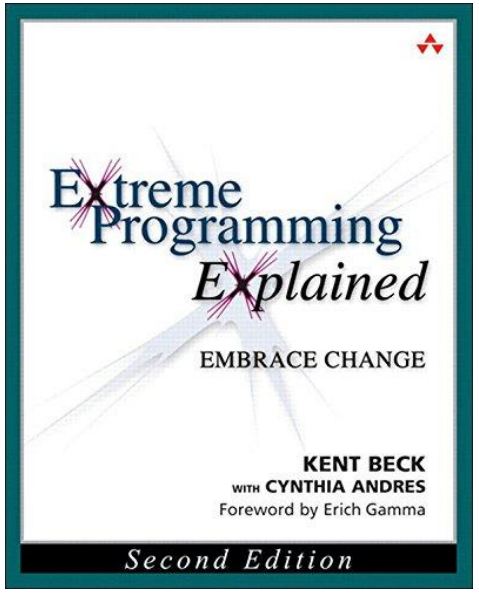
Isabelle BLASQUEZ

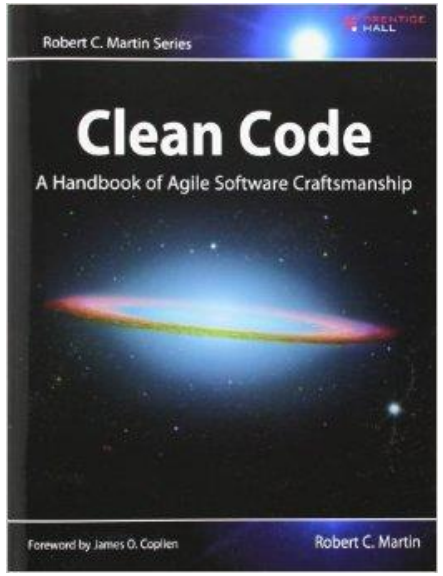
TDD : une discipline de programmation …

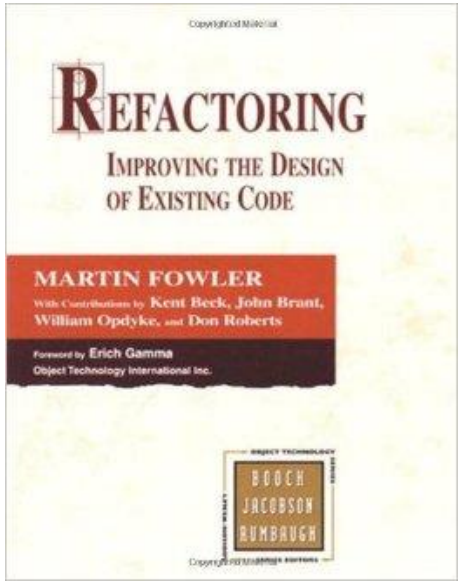


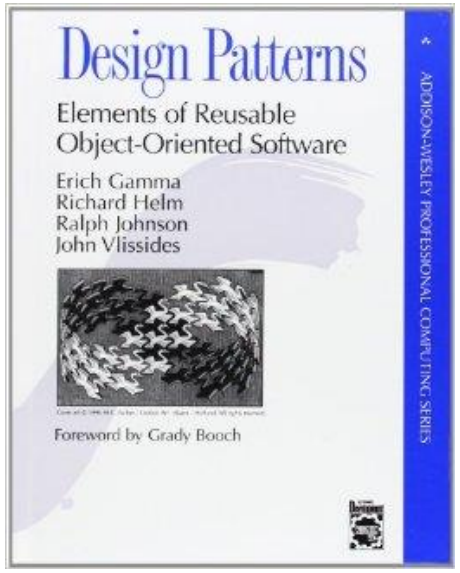








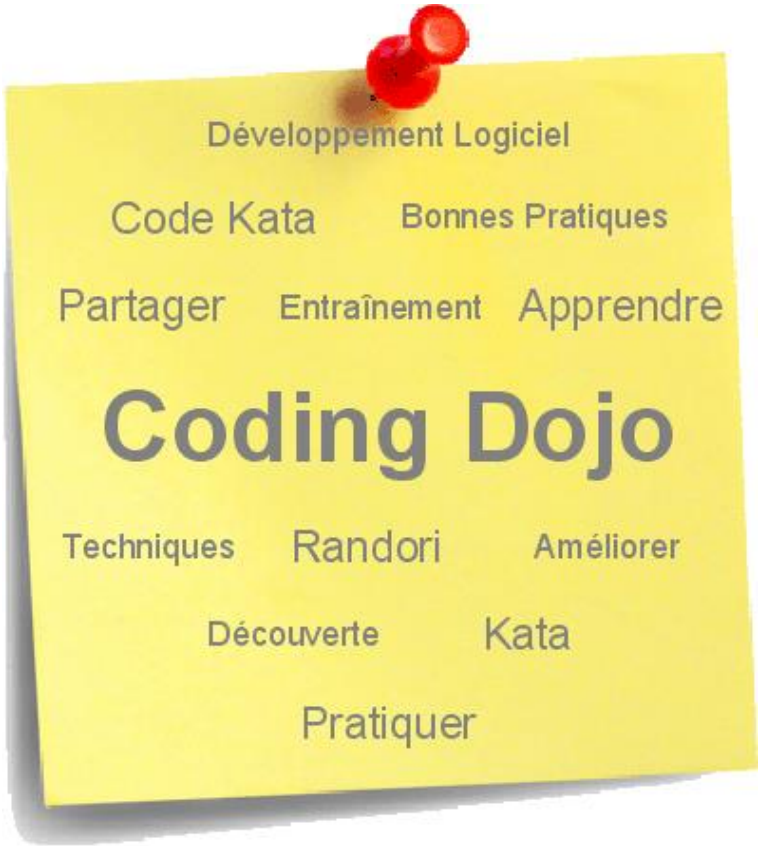


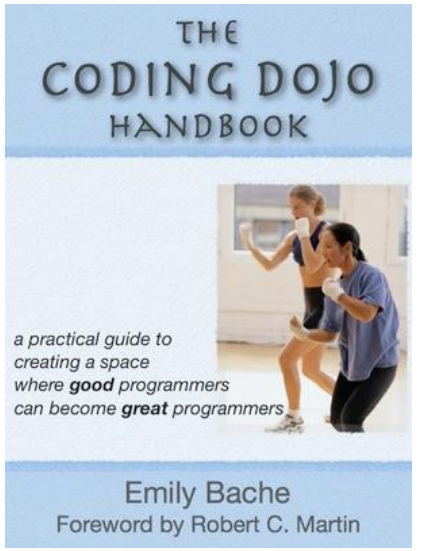


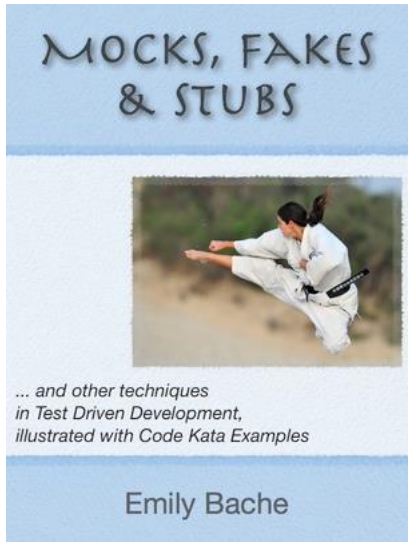
… qui demande un entrainement

De nombreux katas sur :

http://codingdojo.org/kata/
https://github.com/emilybache
https://medium.com/alebaffa-blog/code-kata-catalog-43ad0c6227d



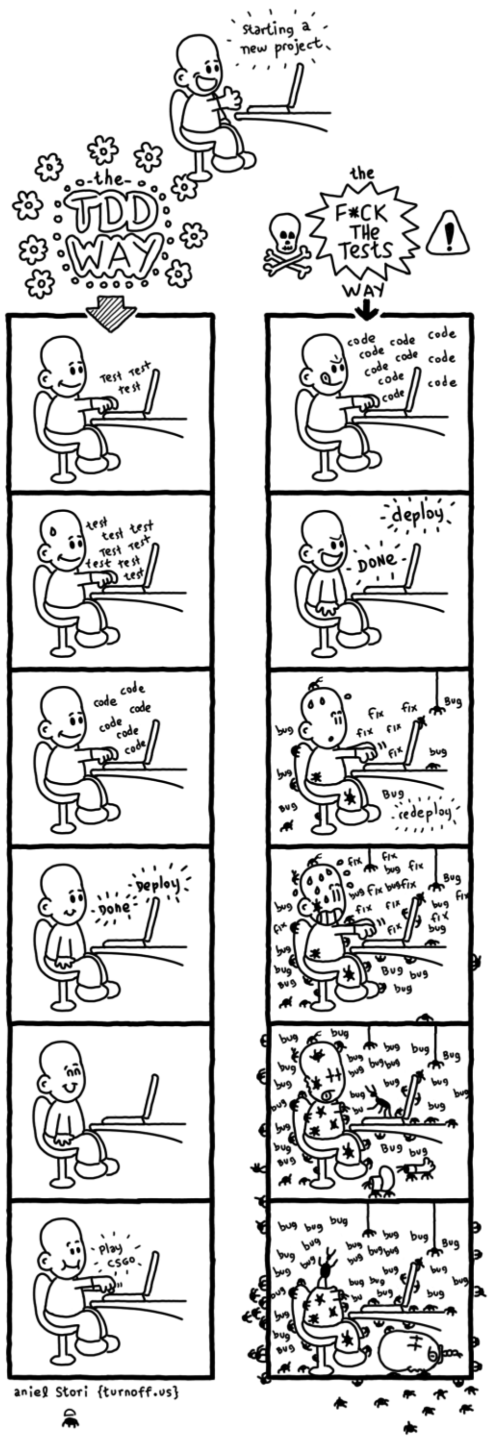




https://leanpub.com/codingdojohandbook
https://leanpub.com/mocks-fakes-stubs

TDD versus FTT
(Forget The Tests)

Extrait : http://turnoff.us/geek/tdd-vs-ftt/



Isabelle BLASQUEZ

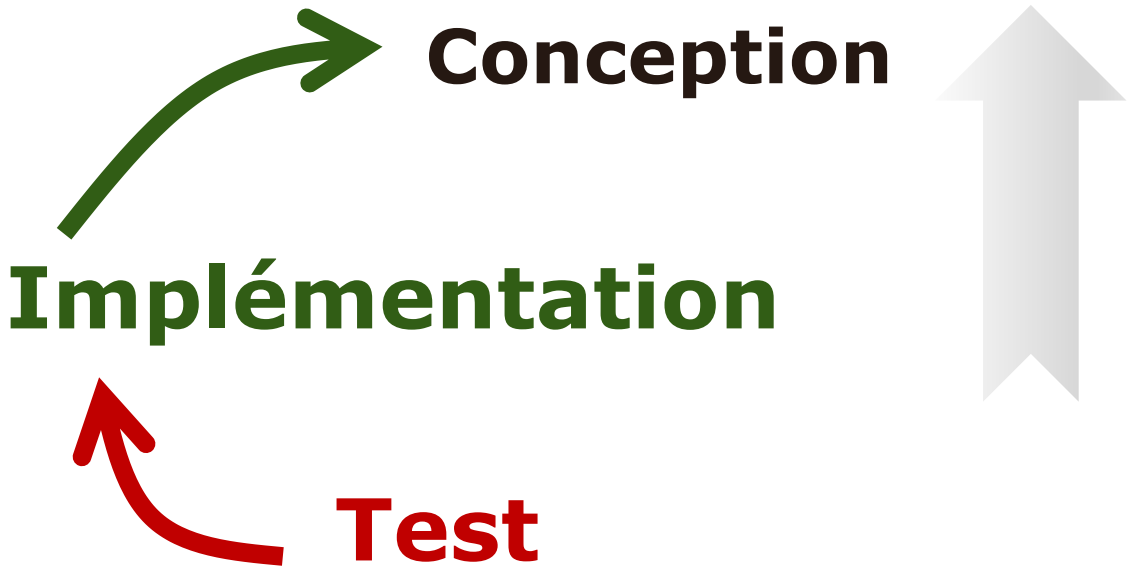
Comparatif des deux approches de développement logiciel

Isabelle BLASQUEZ

Deux approches pour le développement logiciel

Bottom-up

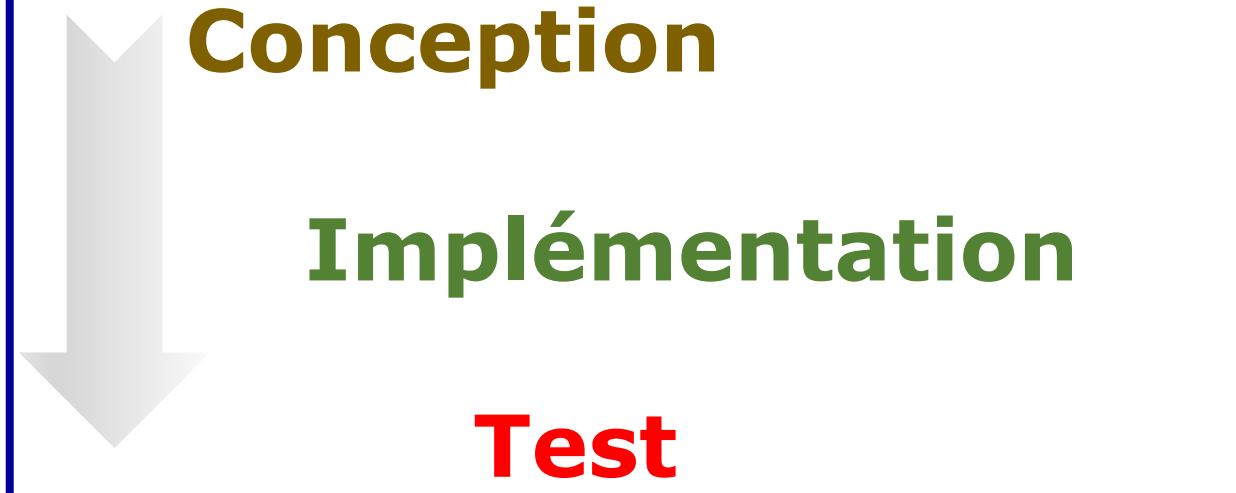
Approche ascendante :
Conception émergente
grâce au refactoring du code et aux tests



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

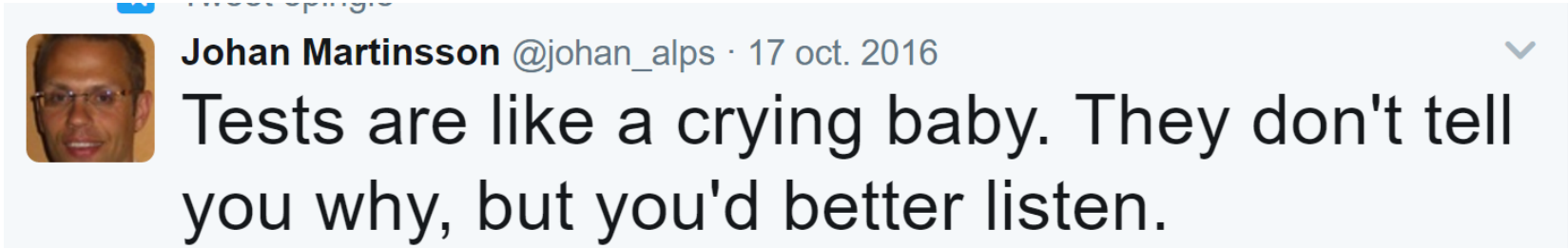
Top-down

Approche descendante :
Des modèles au code …



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

Isabelle BLASQUEZ



https://twitter.com/johan_alps/status/788126453133107202

A propos de la couverture de code par les tests

Isabelle BLASQUEZ

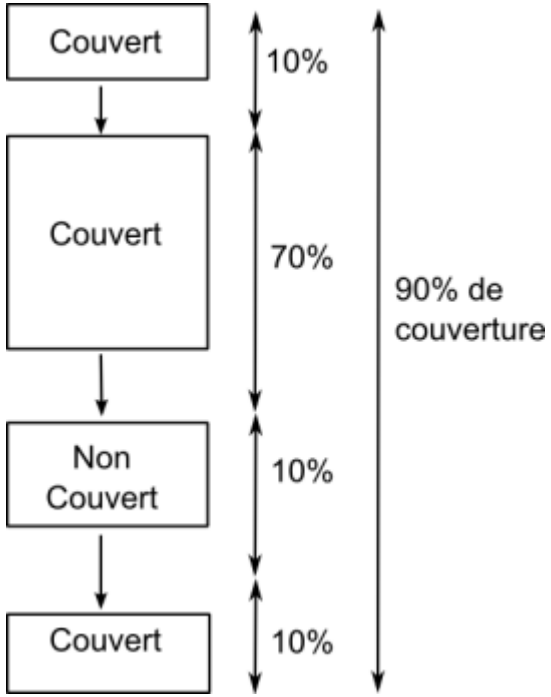
La couverture de code par les tests

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**.

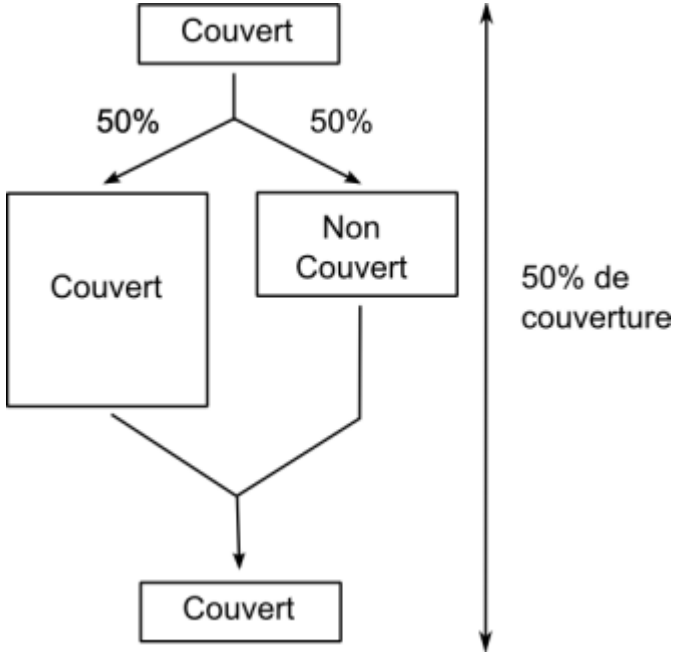
Extraits : https://fr.wikipedia.org/wiki/Couverture_de_code
et Définition de la couverture de code illustrée avec Junit : http://www.junit.fr/2012/04/03/definition-de-la-couverture-de-code-illustree-avec-junit/

La couverture linéaire est
la couverture en lignes de code
(*couverture des instructions ou* ***statement coverage***)
(la plus répandue et la plus simple à visualiser : chaque ligne compte pour un élément)



La couverture linéaire ne suffit pas pour donner une vision satisfaisante de la qualité intrinsèque du code.

La couverture conditionnelle calcule la proportion des chemins logiques couverts
(***condition coverage et path coverage***)
(plus compliquée à appréhender car nécessite une vision arborescente du code)



La couverture conditionnelle pondèrent les proportions testées en partant du principe que chaque chemin a le même poids indépendamment de sa longueur

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 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/

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

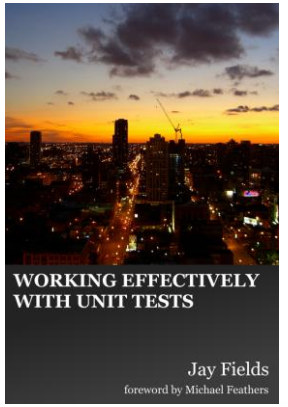
Extrait : https://martinfowler.com/bliki/TestCoverage.html

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

Il faut aller chercher les tests aux limites…

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

Et pour finir, pourquoi créer 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 …

Extrait de : https://twitter.com/CommitStrip_fr/status/829624172628295680

Annexe

Deux références pour en savoir plus sur les annotations de `Junit`

JUnit4 Annotations : Test Examples and Tutorial

JUnit4 Annotations are single big change from JUnit 3 to JUnit 4 which is introduced in Java 5. With annotations creating and running a JUnit test becomes more easy and more readable, but you can only take full advantage of JUnit4 if you know the **correct meaning of annotations** used on this version and how to use them while writing tests. In this **Junit tutorial** we will not only understand meaning of those annotations but also we will see examples of JUnit4 annotations. By the way this is my first post in unit testing but if you are new here than you may like post 10 tips to write better code comments and 10 Object oriented design principles for Programmer as well.

JUnit 4 Annotations : Overview

Following is a list of ***frequently used Annotation***s , which is available when you include junit4.jar in your Classpath:

```
@Before
@BeforeClass
@After
@AfterClass
@Test
@Ignore
@Test(timeout=500)
@Test(expected=IllegalArgumentException.class)
```

http://javarevisited.blogspot.fr/2012/06/junit4-annotations-test-examples-and.html

Program: List of JUnit annotations.

Annotations are introduced in JUnit4. Here are the list of annotations and its descriptions

Reference: org.junit java docs

@Test: The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case. To run the method, JUnit first constructs a fresh instance of the class then invokes the annotated method. Any exceptions thrown by the test will be reported by JUnit as a failure. If no exceptions are thrown, the test is assumed to have succeeded.

```
1  public class MyTestClass {
2      @Test
3      public void myTestMethod() {
4          /**
5           * Use Assert methods to call your methods to be tested.
6           * A simple test to check whether the given list is empty or not.
7           */
8         org.junit.Assert.assertTrue( new ArrayList().isEmpty() );
9      }
10  }
```

@Test (expected = Exception.class): Sometimes we need to test the exception to be thrown by the test. @Test annotation provides a parameter called 'expected', declares that a test method should throw an exception. If it doesn't throw an exception or if it throws a different exception than the one declared, the test fails.

http://java2novice.com/junit-examples/junit-annotations/

Isabelle BLASQUEZ

TDD - A Poem

Test Driven, Test Driven...

Test First, Feature Driven...
Our **right** to test is **not employer given**
We choose to test; our right; our decision; it's called **professionalism**

Write a test, before you progress
Watch it fail, we will prevail
Write the code, make it pass **fast**
Don't worry, we'll clean it last

Run it baby run it, does it go green?
Ugh, they're **taking to long to run**, I'm gonna scream!
It's taking too long? too long!!!!????
Your code is **not isolated**; your tests are written wrong!

Still red you say? ok, don't run away
It's just a bug that we haven't seen
You're almost there, don't turn to dispair
Fix it, run all tests, now are they green?

Oh Green, Green! Yes! Yes!
I'm done! I'm done! wait happy one...you're moving too fast
Fast you say? Yes, take a chill, you gotta refactor still
Refactor you say? Yes blue, both prod & test code right away

But I need to move fast, got a deadline that'll pass
But you're not doing TDD, if you skip the blue you see...
When you skip blue, you make a **mess**
When you make a mess the **less you progress**

Refactor now, to keep your code **lean**
Refactor now, to keep your code **clean**
Refactor now, and use your talents to **design**
Refactor now, because your next test will mind

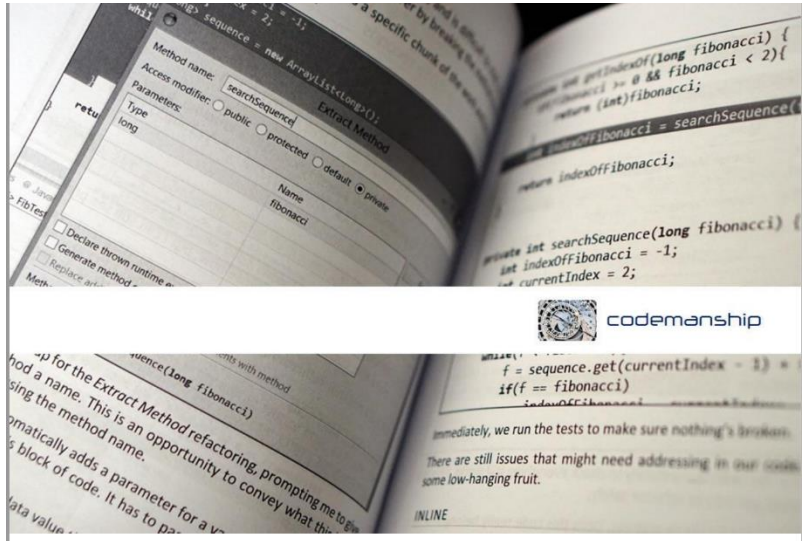
They cycle is done, what to write next?
Next test smallest step, to force you to progress
Progress you say? Yes, lets to write a little more code
Our best guess, the next **behavior** to test

Triangulate, to force your code to **bend**
Red, then green, then refactor again
Last test is finished, now we are done!
Wow, I gotta tell you this sure is fun!

Run all tests, if green check them in
Tomorrow a new feature, a new test we'll begin

About the Author: Dave Schinkel runs WeDoTDD.com a site dedicated to finding an listing teams who practice Test Driven development. He is a TDD practitioner who is also a mentor on CodeMentor.io. Book a session now with Dave and learn some unit testing!

Extraits de : https://www.codementor.io/daveschinkel13/tdd-a-poem-6bqvleahf



101 TDD Tips

TDD Tip #1: Refactoring to parameterised tests is a great way to reduce duplication while generalising the tests so they read more like a specification

```
@RunWith(JUnitParamsRunner.class)
public class FibonacciTests {

    @Test
    @Parameters({"0,0","1,1"})
    public void startsWithZeroAndOne(int index, int expected) {
        assertEquals(expected, getFibonacciNumber(index));
    }

    @Test
    @Parameters({"2,1", "3,2", "5,5"})
    public void thirdNumberOnIsSumOfPreviousTwo(int index, int expected){
        assertEquals(expected, getFibonacciNumber(index));
    }

    @Test(expected=IllegalArgumentException.class)
    public void indexMustBePositiveInteger() {
        getFibonacciNumber(-1);
    }

    private int getFibonacciNumber(int index) {
        return new Fibonacci().getNumber(index);
    }

}
```

codemanship

TDD Tip #2: Using variables and constants can make the meaning of test data values clearer

```
@Test(expected=MaximumExceededException.class)
public void maximumDebitAmountCannotBeExceeded() {
    BankAccount account = new BankAccount();
    account.credit(1000);
    account.debit(600.01);
}
```

```
@Test(expected=MaximumExceededException.class)
public void maximumDebitAmountCannotBeExceeded() {
    BankAccount account = new BankAccount();
    account.credit(1000);
    final double maxDebitAmount = 600.00;
    account.debit(maxDebitAmount  + 0.01);
}
```

codemanship

TDD Tip #101: TDD isn't compulsory. The choice is yours.

Choose to minimise costly misunderstandings about requirements

Choose to deliver more reliable and more maintainable code

Choose to have code that's always shippable, for faster feedback cycles and shorter lead times

Choose to be able to sustain the pace of innovation on your product for longer & outlearn the competition

codemanship

Tous les 101 conseils à lire sur :
http://www.codemanship.co.uk/files/101TddTips.pdf
Et sur : https://twitter.com/codemanship

Isabelle BLASQUEZ

Bonus : A propos du Kata Fizz Buzz & autres …

Un peu de lecture sur le Kata Fizz Buzz
http://c2.com/cgi/wiki?FizzBuzzTest

Fizz Buzz dans de nombreux langages de programmation
http://rosettacode.org/wiki/FizzBuzz

Twenty Ways to FizzBuzz
http://ditam.github.io/posts/fizzbuzz/

Kata Fizz Buzz Enterprise
FizzBuzz Enterprise Edition is a no-nonsense implementation of FizzBuzz made by a serious businessman for serious business purposes.
https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition

Dans la même idée : Enterprise Tic-Tac-Toe
In which I ridiculously over-engineer a simple game to make it "enterprise-ready"
https://fsharpforfunandprofit.com/ettt/

Dmitri Sotnikov
@yogthos

Suivre

the current state of JavaScript programming

Voir la traduction

```
Fizzbuzz npm edition:
    var zero = require("number-zero");
    var hundred = require("number-one-hundred");
    var isDivisibleBy5 = require("is-divisible-by-5");
    var isDivisibleBy3 = require("is-divisible-by-3");
    var isDivisibleBy5and3 = require("is-divisible-by-5-and-3");
    var numberToString = require("number-that-is-between-one-and-hundred-to-string");
    var fizzbuzz = require("string-fizz-buzz");
    var fizz = require("string-fizz");
    var buzz = require("string-buzz");
    var incrementByOne = require("increment-by-one");
    var forLoop = require("for-loop");
    var ifCondition = require("if-condition");
    var elseIfCondition = require("else-if-condition");
    var elseCondition = require("else-condition");
    var lessThan = require("less-than");
    var print = require("print-string");

    forLoop(zero, lessThan(hundred), incrementByOne, function(i) {
        condition(isDivisibleBy5and3, print(fizzbuzz),
            elseIfCondition(isDivisibleBy3, print(fizz),
                elseIfCondition(isDivisibleBy5, print(buzz),
                    elseCondition(print(numberToString(i))))));
    });
```

Extrait : https://twitter.com/yogthos/status/713090123894796288