

IA – 2014 / 2015

Programmation Orientée Objet

Cours 7

Gérald Oster <oster@loria.fr>

Plan du cours

- Introduction
- Programmation orientée objet :
 - Classes, objets, encapsulation, composition
 - 1. Utilisation
 - 2. Définition
- Héritage et polymorphisme :
 - Interface, classe abstraite, liaison dynamique
- Exceptions
- Généricité
- Tests

II^{ème} Partie :

Introduction aux tests

Pourquoi vérifier et valider ?

- Sonde Mariner I, 1962 : problème de spécification
 - http://fr.wikipedia.org/wiki/Mariner_I
- Ariane V vol 501, 1996 : problème de conversion de nombres
 - Coût : 370 millions de dollars
 - http://fr.wikipedia.org/wiki/Vol_501_d'Ariane_5
- Therac-25, 1985-87 : problème de synchronisation, de dépassement capacité
 - Coût : plusieurs vies
 - <http://fr.wikipedia.org/wiki/Therac-25>
- Panne d'électricité aux États-Unis et au Canada, 2003
 - Impact sur 55 millions d'habitants. Coût : 6 milliards de dollars
 - http://fr.wikipedia.org/wiki/Panne_de_courant_nord-am%C3%A9ricaine_de_2003

Pourquoi vérifier et valider ?

- Du point de vue utilisateur :
 - Limiter coûts économiques, humains et environnementaux
- Du point de vue développeur/fournisseur :
 - Limiter coût de la correction des bugs
 - En phase d'implémentation coût = 1
 - En phase d'intégration (bug de conception) x 10
 - En phase de recette (bug de spécification) x 100
 - En phase d'exploitation > x 1000
- Pour réduire les coûts :
 - Vérification et validation :
 - Environ 30% du développement d'un logiciel standard
 - Plus de 50% du développement d'un logiciel critique
 - Phase de test souvent plus longue que les phases de spécification, de conception et de réalisation réunies

Pourquoi vérifier et valider ?

- Pour assurer la qualité du logiciel :
 - Capacité fonctionnelle : réponse aux besoins des utilisateurs
 - Facilité d'utilisation : prise en main et robustesse
 - Fiabilité : tolérance aux pannes
 - Performance : temps de réponse, débit, fluidité, etc.
 - Maintenabilité : facilité à corriger ou à faire évoluer le logiciel
 - Portabilité : aptitude à fonctionner dans un environnement différent de celui prévu initialement

Vérification vs. Validation

- Vérification : « *Are we building the product right?* »
 - But : Assurer que le système fonctionne correctement
 - Résultat : Preuve formelle de propriétés sur un modèle du système (*model-checking*, preuve, etc.)
 - Techniques apparentées : génération de code, synthèse de programme (à partir d'une spécification)
- Validation : « *Are we building the right product?* »
 - But : Assurer que le système fonctionne selon les attentes de l'utilisateur
 - Résultat : Assurance d'un certain niveau de confiance dans le système
 - Techniques apparentées : relecture de code (*code review*), inspection de programmes

Vérification vs. Validation

- **Vérification** : Le système (ou le modèle) vérifie-t-il la propriété P ?
Objectif : prouver la correction du système
- **Validation** : Le système est-il conforme au cahier des charges (à tous les niveaux) ?
Objectif : détecter les non-conformités

Vérification vs. Validation

- Test :
 - Non exhaustif mais relativement facile à mettre en œuvre
- Preuve :
 - Exhaustif mais très technique
- Model-checking :
 - Exhaustif et moins technique que la preuve
- Mais,
 - Preuve et model-checking sont limités par la taille du système et vérifient des propriétés sur un modèle du système (distance entre le modèle et la réalité ?)
 - Test repose sur l'exécution du système réel, quelles que soient sa taille et sa complexité

Test : Définition

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

ISO/IEC/IEEE. 2010. Systems and Software Engineering - System and Software Engineering Vocabulary. Geneva, Switzerland: International Organization for Standardization (ISO)/ International Electrotechnical Commission (IEC)/ Institute of Electrical and Electronics Engineers (IEEE). ISO/IEC/IEEE 24765:2010.

⇒ Comparaison entre système et spécification

⇒ Validation dynamique (exécution du système)

Test : Définition

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

⇒ Objectifs : détection de bugs.

« Testing can only prove the presence of bugs, not their absence », E. W. Dijkstra

« Tester peut révéler la présence d'erreurs mais jamais leur absence. »

⇒ Vérification partielle : le test ne peut pas montrer la conformité du système (nécessité d'une infinité de tests)

Remarque : Débogage = Processus de recherche des erreurs produisant un échec observé

Terminologie

- Anomalie (fonctionnement) [*error*]: différence entre comportement attendu et comportement observé
 - Ex: boîte de message indiquant : « Bonjour, M. <null value> »
- Défaut (interne) [*fault*]: élément ou absence d'élément dans le logiciel entraînant une anomalie
 - Ex: un numéro de compte qui n'est pas initialisé
- Défaillance [*failure*] : résultat incorrect dû à un défaut
- Erreur (programmation, conception) [*mistake*]: comportement du programmeur ou du concepteur conduisant à un défaut

Exemple

- **Spécification.** Le programme prend en entrée trois entiers, interprétés comme étant les longueurs des côtés d'un triangle. Le programme retourne la propriété du triangle correspondant (scalène, isocèle ou équilatéral).
- **Exercice.** Ecrire un ensemble de tests pour valider la conformité du programme.

Exemple

Cas valides

Triangle scalène valide

Triangle isocèle valide + permutations

Triangle équilatéral valide

Triangle plat ($a+b=c$) + permutations

Cas invalides

Pas un triangle ($a+b < c$) + permutations

Une valeur à 0

Toutes les valeurs à 0

Une valeur négative

Une valeur non entière

Mauvais nombre d'arguments

Exemple

Cas valides	Données	Résultat attendu
Triangle scalène valide	(10,5,7)	scalène
Triangle isocèle valide + permutations	(3,5,5)	isocèle
Triangle équilatéral valide	(3,3,3)	équilatéral
Triangle plat ($a+b=c$) + permutations	(2,2,4)	scalène
Cas invalides		
Pas un triangle ($a+b < c$) + permutations	(2,1,5)	triangle invalide
Une valeur à 0	3,0,4	triangle invalide
Toutes les valeurs à 0	0,0,0	triangle invalide
Une valeur négative	2,-1,6	triangle/entrée invalide
Une valeur non entière	'x',4,2	entrée invalide
Mauvais nombre d'arguments	(3,5)	entrée invalide

Exemple

- 16 cas correspondant aux défauts constatés dans des implémentations de cette spécifications
- Moyenne des résultats obtenus par un ensemble de développeurs expérimentés : 55%
- Les programmeurs ne sont pas forcément les meilleurs testeurs :
 - Programmer est une activité constructive (essayer de faire fonctionner les choses)
 - Tester est une activité destructive (essayer de faire échouer les choses)
- « *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.* » B. Kernighan

Vocabulaire du test

- Objectif de test : comportement du système à tester
 - Données de test : données à fournir en entrée au système de manière à déclencher un objectif de test
 - Résultats d'un test : conséquences ou sorties de l'exécution d'un test (affichage à l'écran, modification des variables, envoi de messages, etc.)
- ⇒ Cas de test : données d'entrée et résultats attendus associés à un objectif de test

Standard Glossary of Terms used in Software Testing, ISTQB (International Software Testing Qualifications Board)

Faux-positifs et faux-négatifs

- Validité des tests : les tests n'échouent que sur des programmes incorrects.
⇒ Faux-échec (*false-fail*) : fait échouer un programme correct
- Complétude des tests : les tests ne réussissent que sur des programmes corrects
⇒ Faux-succès (*false-pass*) : fait passer un programme incorrect
- Validité indispensable : complétude impossible en pratique (il faut toujours s'assurer que les tests sont valides !!!)

Processus de test

- Choisir les comportements à tester (objectifs de test)
- Choisir les données de test permettant de déclencher ces comportements + décrire le résultat attendu
- Exécuter les cas de test sur le système + collecter les résultats
- Comparer les résultats obtenus aux résultats attendus pour établir un verdict

Exécution d'un test

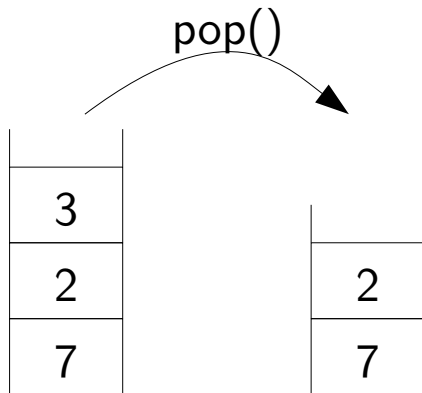
Scénario de test :

- Préambule : suite d'actions (instructions) amenant le programme dans l'état nécessaire pour exécuter le cas de test
- Corps : exécution des fonctions du cas de test
- Identification (facultatif) : opération d'observations rendant la comparaison possible
- Postambule : suite d'actions permettant de revenir à l'état initial

Exécution d'un test

Exemple :

- `pop()` (dépiler le sommet d'une pile)
- Cas de test :



Exécution du test :

Préambule :

```
push(7)
push(2)
push(3)
```

Corps :

```
pop()
```

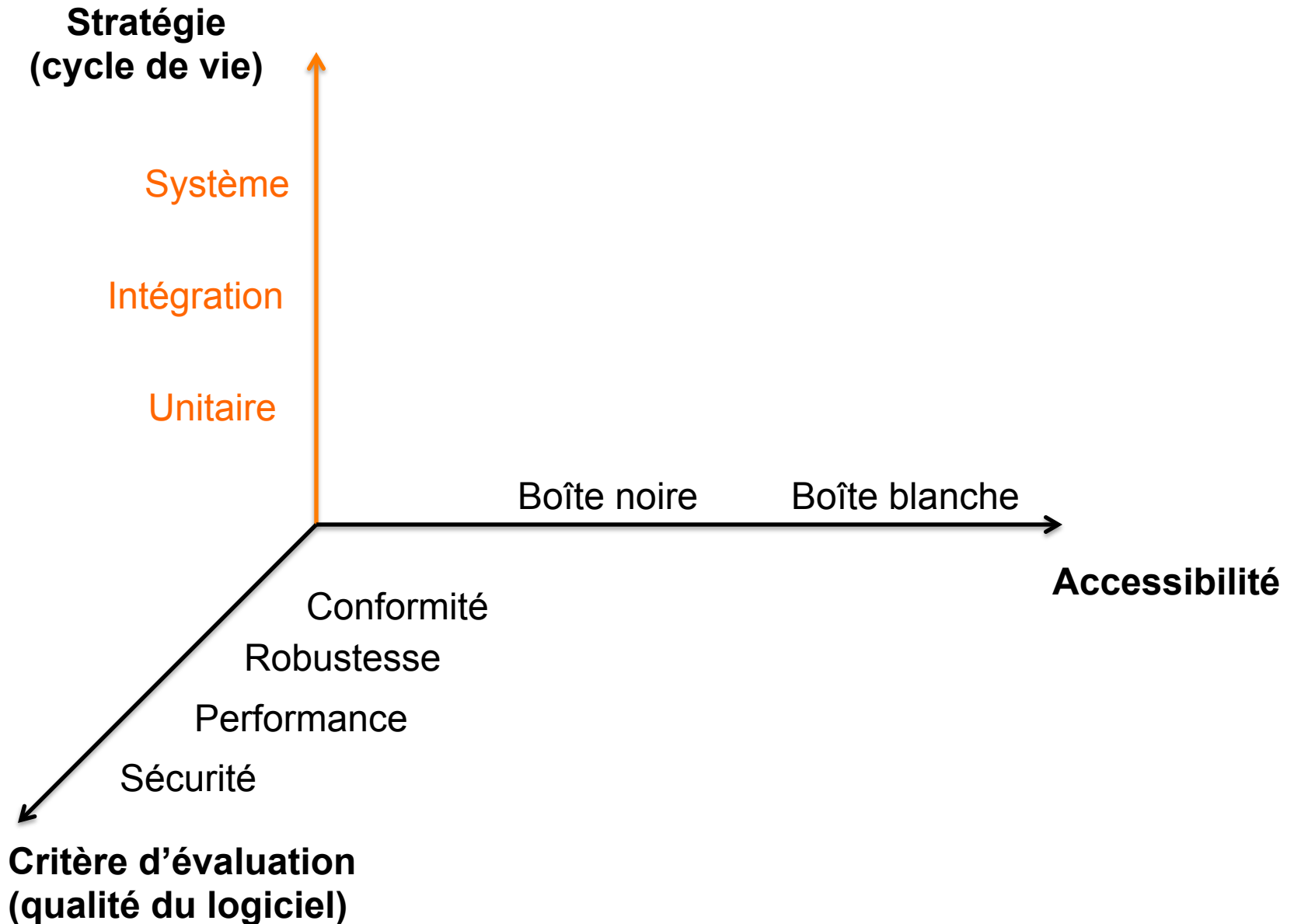
Identification :

```
top() == 2
pop()
top() == 7
pop()
top() == empty
```

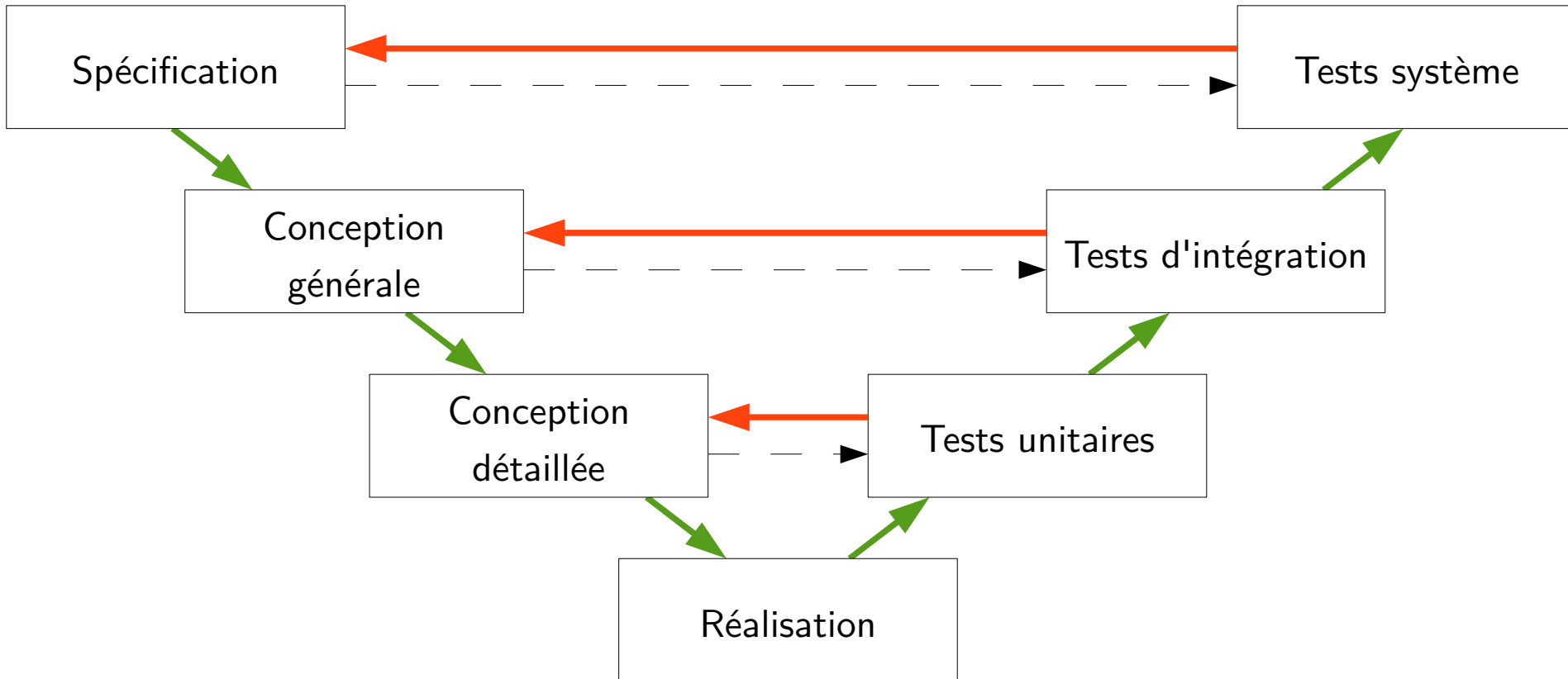
Postambule :

```
rien
```

Types de tests



Exemple : le cycle en V



Test unitaire

- Test des unités de programme de façon isolée, indépendamment les unes des autres, c'est à dire sans appel à une fonction d'un autre module, à une base de données, etc.
- Exemple d'unité : méthode, classe, module, composant

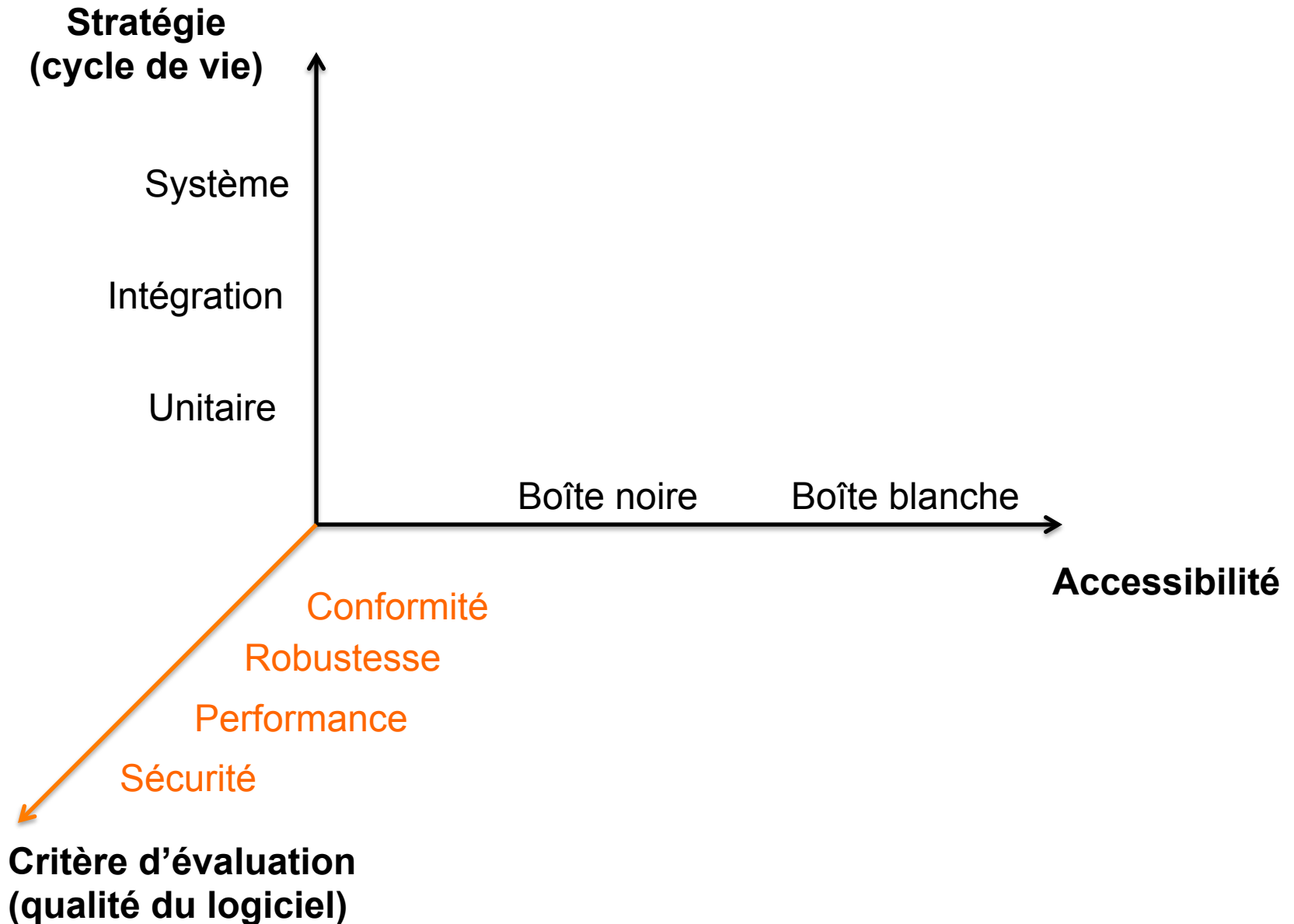
Test d'intégration

- Test la composition de plusieurs modules via leur interface :
 - Communications entre modules, appels de procédure, etc.
- Exemple d'unité : méthode, classe, module, composant

Test système

- Test la conformité du produit fini par rapport au cahier des charges, effectué en boîte noire au travers de son interface

Types de tests



Test de conformité

- But : Assurer que le système présente les fonctionnalités attendues par l'utilisateur
- Méthode : Sélection des tests à partir des spécification, de façon à contrôler que toutes les fonctionnalités spécifiées sont implémentées selon leurs spécifications

Test de robustesse

- But : Assurer que le système supporte les utilisations imprévues
- Méthode : Sélection des tests en dehors des comportements spécifiés (entrées hors domaines, utilisation incorrecte de l'interface, environnement dégradé)

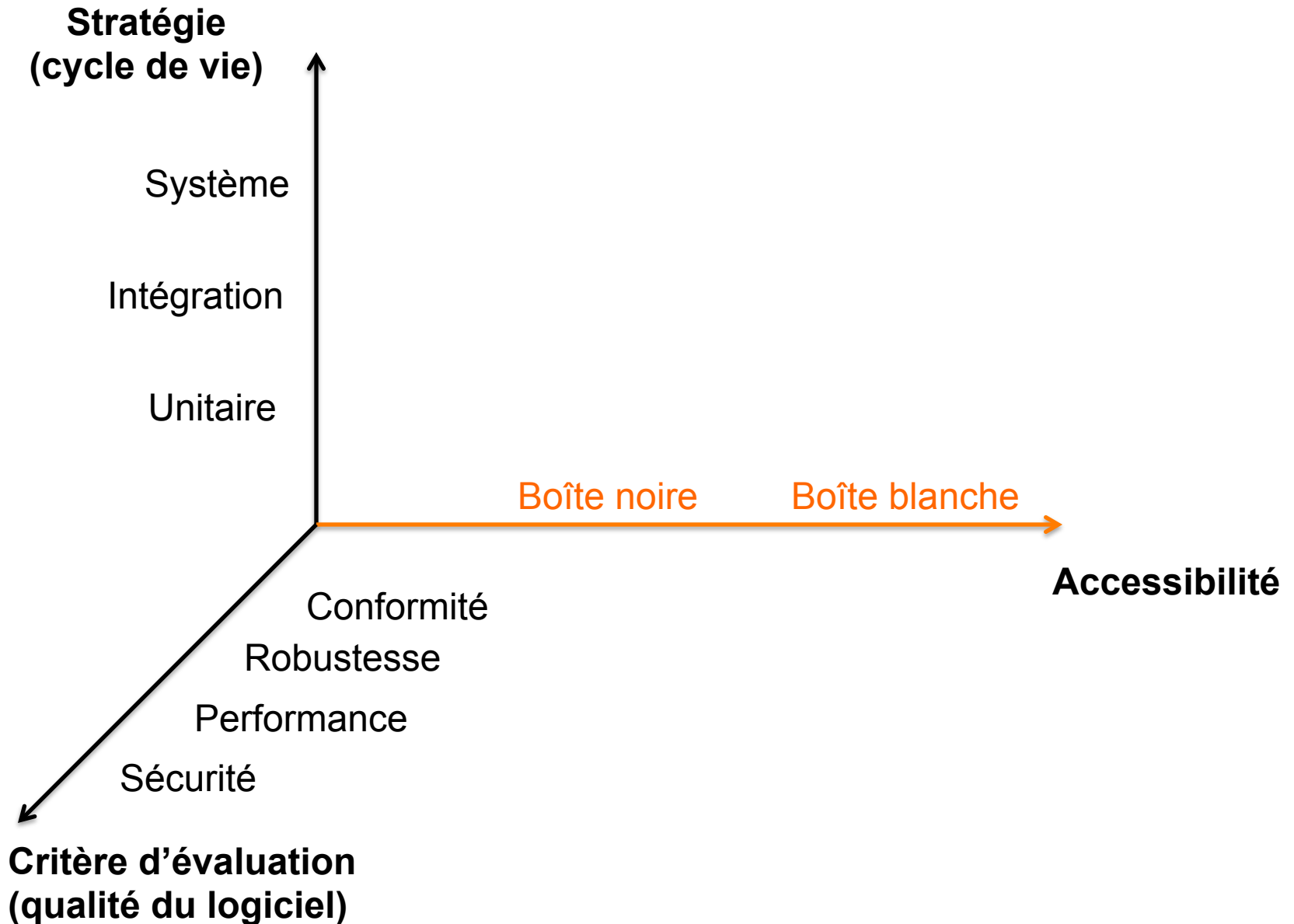
Test de sécurité

- But : Assurer que le système ne possède pas de vulnérabilités permettant une attaque de l'extérieur
- Méthode : Simulation d'attaques pour découvrir les faiblesses du système qui permettraient de porter atteinte à son intégrité

Test de performance

- But : Assurer que le système garde des temps de réponse satisfaisants à différents niveaux de charge
- Méthode : Simulation à différents niveaux de charge d'utilisateurs pour mesurer les temps de réponse du système, l'utilisation des ressources, etc.

Types de tests



Test « boîte noire »

- Composant à tester = boîte noire
 - Black-box testing: tester les fonctionnalités sans prendre en considération la structure interne de l'implémentation
- ⇒ Comportement déterminé uniquement par les entrées / les sorties
- Dériver les tests d'une spécification externe (description du système)
 - Couvrir autant que possible le comportement spécifier
 - Ne peut révéler les erreurs dû aux détails d'implémentation

Test « boîte noire »

- Spécification : entrée une valeur entière et l'afficher

Test « boîte noire »

- Spécification : entrée une valeur entière et l'afficher

```
public void printNum(int num) {  
    if (param < 1024)  
        System.out.println(param);  
    else  
        System.out.println(param/1024 + "K");  
}
```

- Comment imaginer à partir de la spécification ce comportement erroné lorsque le paramètre est supérieur à 1024 ?

Test « boîte blanche »

- White-box testing: considérer l'implémentation de la structure interne lors de la conception des tests
- Dériver les tests :
 - En utilisant les connaissances sur les structures internes
Ex: structure interne = un tableau de taille 256
Tester les valeurs aux limites proches des bornes (0, 255, 256, 257)
 - En couvrant les branches :
Ex : conditionnelle à deux branches (if [cond] then ... else ...)
Tester pour cond == vrai et cond == faux

Test « boîte blanche »

```
public int fun(int num) {  
    int result;  
    result = param / 2;  
    return result;  
}
```

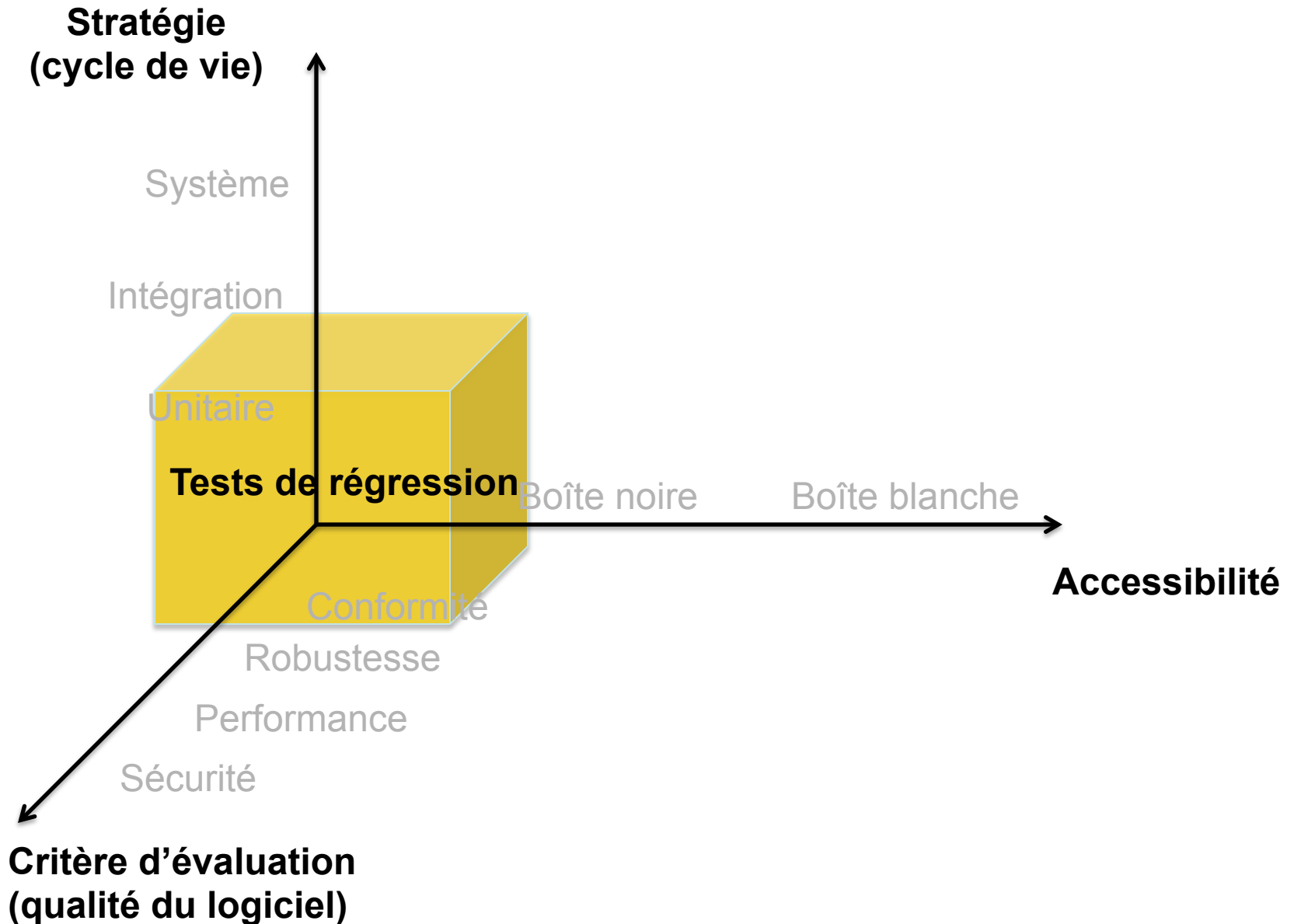
Test « boîte blanche »

- Spécification : entrée une valeur entière et retourne la moitié de cette valeur si elle est paire sinon la valeur non modifiée

```
public int fun(int num) {  
    int result;  
    result = param / 2;  
    return result;  
}
```

- Comment imaginer que la spécification du problème était celle-ci ?

Types de tests : transverses



Test de non régression

- But : Assurer que les corrections et les évolutions du code n'ont pas introduit de nouveaux défauts
 - Cyclicité = un bug fixé réapparaît souvent dans une version suivante
- Méthodes : A chaque ajout ou modification de fonctionnalités, rejouer les tests pour cette fonctionnalités, puis pour celles qui en dépendent, puis les tests de niveaux supérieurs

⇒ Conserver vos cas de test (*test case*)

⇒ Utiliser les tests que vous avez conservés pour les versions suivantes de votre programme

⇒ Une suite de tests (*test suite*) est un ensemble de tests que l'on peut répéter

⇒ Automatiser le processus de test au maximum (intégration continue)

Couverture de test

- Mesure combien de parties du programme sont testées
- Assurez vous que chaque partie de vos programmes sont testées au moins une fois par un test
i.e. Chaque branche d'une conditionnelle devrait être testée

Right BICEPS

- Difficile de penser à tous les cas de test
- Guide pour « débutant » :
 - Right : déterminer ce que sont les valeurs correctes
 - B (Boundaries) : tester les résultats aux limites des intervalles sont corrects
 - I (Inverse) : tester les inversions possibles
 - insert/remove, push/pop, compression/décompression
 - C (Cross-check) : tester de manière croisée par un autre algorithme
 - E (Error condition) : tester en générant les cas d'erreur
 - P (Performance) : tester les performances par rapport aux attentes

En pratique...

- Utiliser un canevas de test (JUnit, TestNG, etc.)
- Exemple : une simple méthode de test JUnit

```
import static org.junit.Assert.assertEquals
public class MyBasicCalculatorTest {
    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero {
        MyCalculator calc = new MyCalculator();

        assertEquals("10 x 0 must be 0", 0, calc.multiply(10, 0));
        assertEquals("0 x 10 must be 0", 0, calc.multiply(0, 10));
        assertEquals("0 x 0 must be 0", 0, calc.multiply(0, 0));
    }
}
```

Principales annotations

Annotation	Description
<code>@Test</code> <code>public void method()</code>	Une méthode de test
<code>@Test(expected=Exception.class)</code>	Test échoue si la méthode ne lève pas une exception
<code>@Test(timeout=100)</code>	Test échoue si le temps d'exécution de la méthode dépasse 100ms
<code>@Before</code> <code>public void setUp()</code>	Méthode exécutée avant chaque test
<code>@After</code> <code>public void tearDown()</code>	Méthode exécutée après chaque test
<code>@BeforeClass</code> <code>public void static method()</code>	Méthode exécutée avant tous les tests
<code>@AfterClass</code> <code>Public void static method()</code>	Méthode exécutée après tous les tests

Principales assertions

Assertion	Description
<code>fail(String)</code>	La méthode échoue si cette instruction est exécutée
<code>assertTrue([msg], condition) / assertFalse([msg], condition)</code>	Vérifie que la condition est vraie / fausse
<code>assertEquals([msg], expected, actual)</code>	Tests que deux valeurs sont égales (pour un tableau ce sont les références qui sont comparées)
<code>assertEquals([msg], expected, actual, tolerance)</code>	Tests que les deux valeurs réelles sont égales (tolerance est le nombre décimale souhaitée)
<code>assertNull([msg], ref) / assertNotNull([msg], ref)</code>	Tests si ref est null / non null
<code>assertSame([msg], expected, actual) assertNotSame([msg], expected, actual)</code>	Tests si les deux références (ne) sont (pas) identiques

Allez encore plus loin...

- Développement dirigé par les tests (*Test-driven development* – *TDD*)
- Méthodologie pour écrire du code
- Principe : écrivez d'abord les tests (avant le code)
 - Seul le code utile est écrit
 - Améliore les interfaces des composants

Références

- Cours
 - Introduction aux test de logiciels, Delphine Longuet, LRI
 - TOP, Martin Quinson, TELECOM Nancy
- Mooc :
 - Software Testing, Udacity
<https://www.udacity.com/course/cs258>
 - Software Development Process, Udacity
<https://www.udacity.com/course/ud805>
- Tutoriel :
 - Unit Testing with JUnit – Tutorial
<http://www.vogella.com/tutorials/JUnit/article.html>
- Standard
 - *ISO/IEC/IEEE. 2010. Systems and Software Engineering - System and Software Engineering Vocabulary. Geneva, Switzerland: International Organization for Standardization (ISO)/ International Electrotechnical Commission (IEC)/ Institute of Electrical and Electronics Engineers (IEEE). ISO/IEC/IEEE 24765:2010.*
 - *Standard Glossary of Terms used in Software Testing, ISTQB (International Software Testing Qualifications Board)*