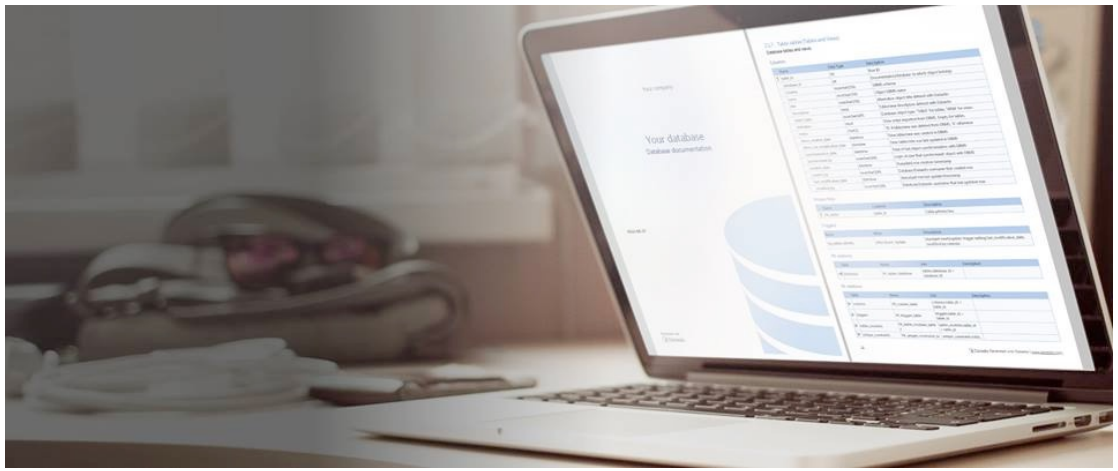
IUT Lyon 1

l'excellence technologique

Refactoring, Tests et TDD

V. Deslandres, IUT Univ LYON 1

S1.10 – Qualité de Developpement-1 – ASPE



Refactoring : définition

- **Martin Fowler :** *Refactoring - Improving the Design of Existing Code*
  - ***Refactoring*** *(nom): modification apportée à la structure interne d'un logiciel pour le rendre plus facile à comprendre et moins coûteux à modifier, sans changer son comportement observable.*
  - ***Refactorer*** *(verbe): restructurer, optimiser, améliorer le code d'un logiciel sans changer son comportement.*

TDD : Test Driven Development

- « Développement dirigé par les tests »
- On écrit les tests **avant** le code de la fonctionnalité
- Objectifs : traduire une spécification avec toutes ses conditions extrêmes (un test par condition), càd. définir les **tests d'acceptation :**
  - Ex. affecter une prime salariale en fin d'année
    - Type de département (pas les mêmes budgets de prime)
    - Salariés éligibles
    - Type de salarié (pas les mêmes primes)
  - Ex.: *quand le client annule sa réservation d'hôtel, il faut vérifier s'il doit payer une pénalité (cas où non Premium ou plus de 48h avant la date). Le montant de la pénalité est aussi fonction de l'écart entre la date du séjour et la date d'annulation.*

Ex. Test d'acceptation pour ROVER

- Test unitaire de déplacement (pivot) d'un robot :

```
@Test
public void roverFaceNord_SiPivoterDroite_AlorsFaceEst(){
        rover.pivoterDroite();
        assertEquals(Orientation.EST, rover.orientation);
}
```

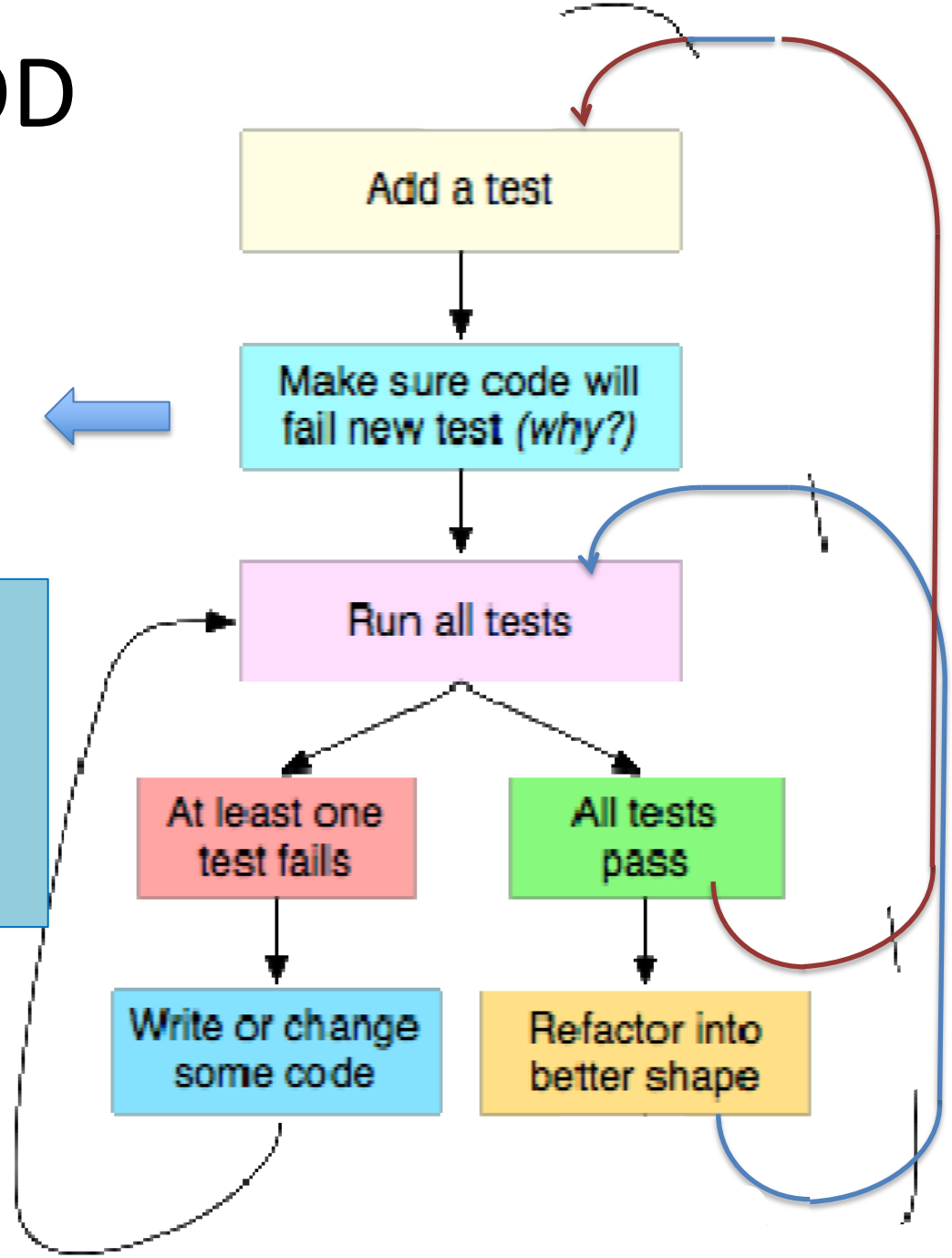
➔ test réel avec le robot

Cycle du refactoring / TDD

On écrit le test avant d'écrire le code de la fonctionnalité

*Le test est une **sécurité** : on est certain de ne pas abîmer la fonctionnalité quand on modifie son code*

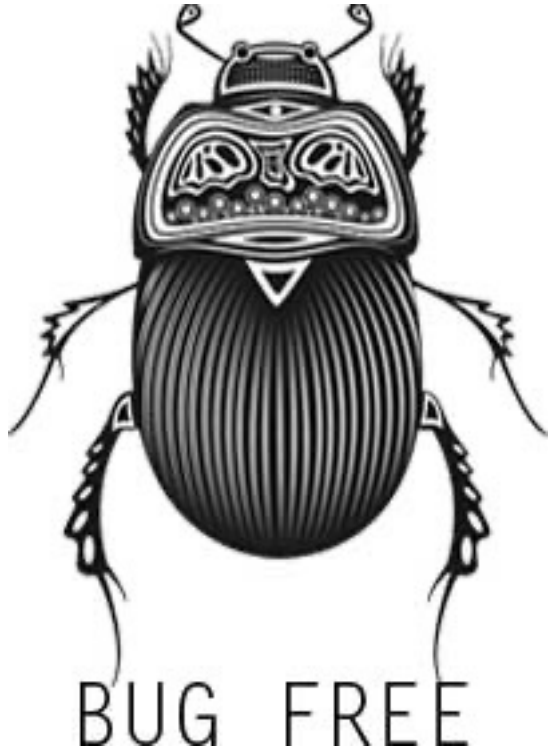
On crée ou on fait évoluer le code de la fonctionnalité



Pourquoi les tests ?

- Sans les tests, vous êtes obligé d'écrire le **code métier**, le code de **persistance** et le code de **l'IHM,** avant de pouvoir vraiment tester quoi que ce soit.

  - Les tests permettent au contraire de tester vite la logique *métier* sans avoir à développer les 2 autres parties.
  - Les tests apportent une **plus-value** au code (qualité, robustesse).

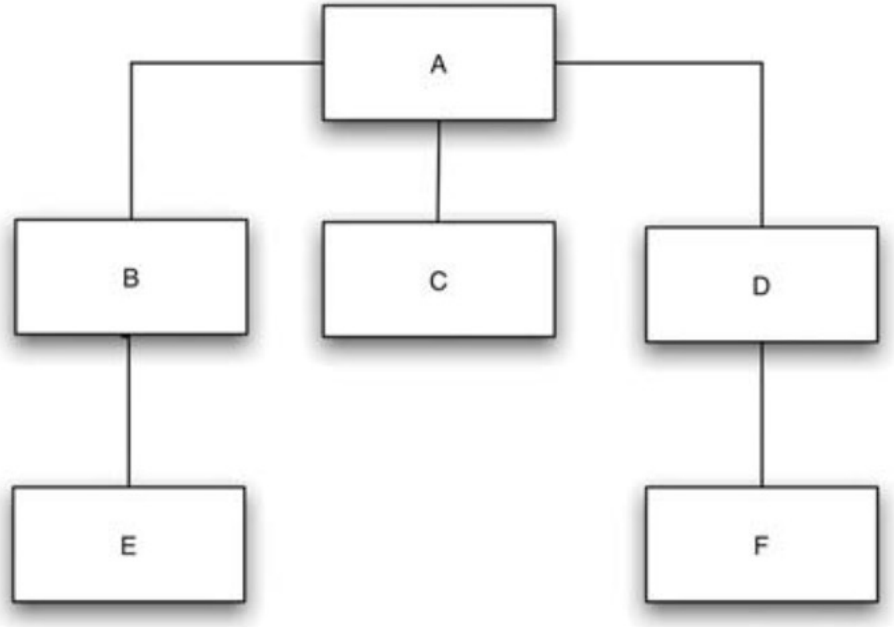


Types de tests

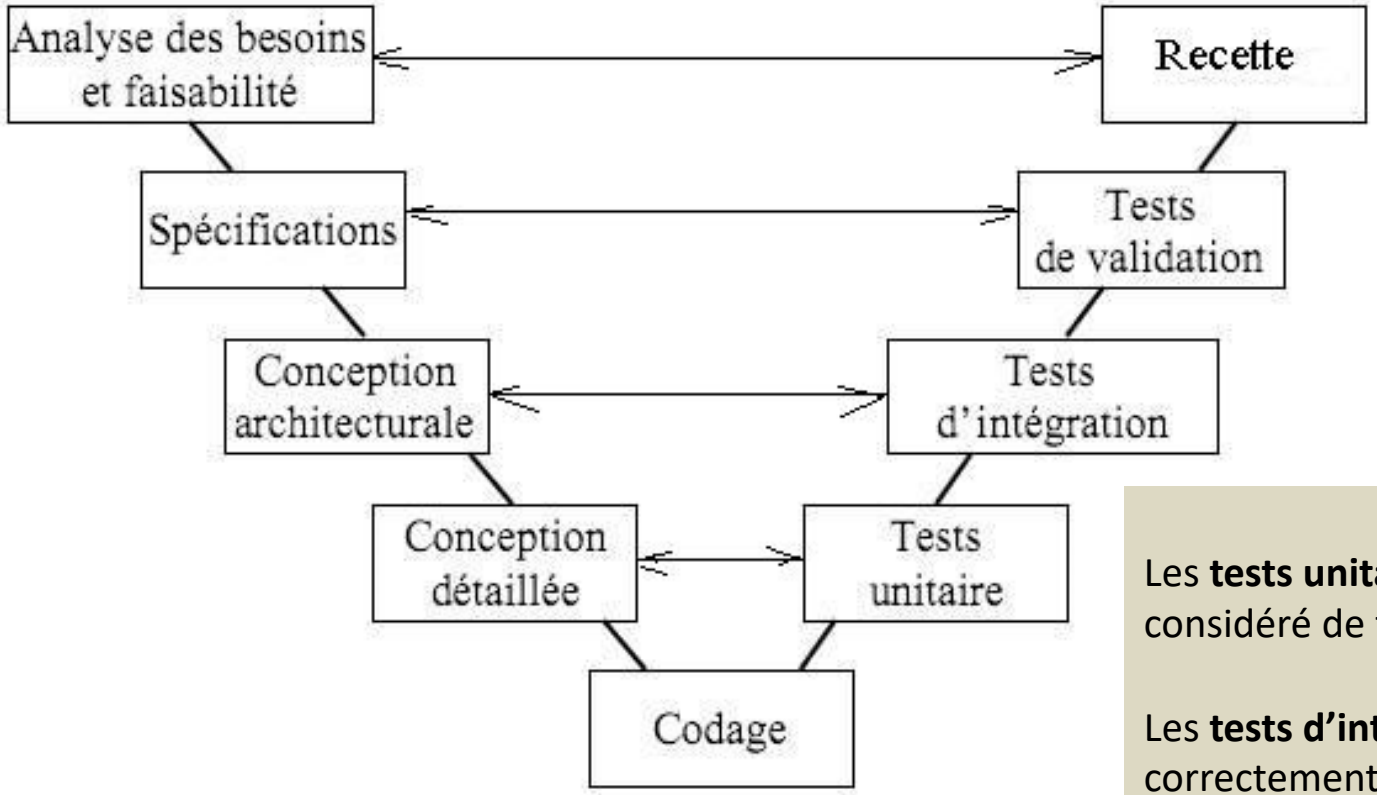
- Tests unitaires : test de méthode, de module
- Tests d'intégration

Ex.: un module A appelle le modules B, C et D.
Le module B appelle le module E, etc.

En phase de test unitaire : on utilise des « **stub modules** » qui simulent l'exécution normale des composants dont on dépend.



- Aussi : tests d'acceptation, etc.
  - Cf schéma ci-après



Les **tests unitaires** concernent un module, une classe ou une méthode, considéré de façon isolée.

Les **tests d'intégration** vérifient que les modules couplés s'articulent correctement.

Les **tests de validation ou tests fonctionnels** couvrent les exigences globales (processus utilisateur, ex.: retirer de l'argent).

+ test **d'installation**

+ **tests système** correspondent à la traduction des exigences de l'application dans son environnement d'exécution.

Ex. de tests système

Examples of Presentation, Business, and Data Tier Testing

Presentation Tier	*Business Tier*	*Data Tier*
• Ensure fonts are the same across browsers.	• Check for proper calculation of sales tax and shipping charges.	• Ensure database operations meet performance goals.
• Check to make sure all links point to valid files or Websites.	• Ensure documented performance rates are met for response times and throughput rates.	• Verify data are stored correctly and accurately.
• Check graphics to ensure they are the correct resolution and size.	• Verify that transactions complete properly.	• Verify that you can recover using current backups.
• Spell-check each page.	• Ensure failed transactions roll back correctly.	• Test failover or redundancy operations.
• Allow a copy editor to check grammar and style.	• Ensure data are collected correctly.	
• Check cursor positioning when page loads to ensure it is in the correct text box.		
• Check to ensure default button is selected when the page loads.		

Source : The Art of Software Testing, G. J. Myers

Quand détecte-t-on les bugs ?

Hypothetical Estimate of When the Errors Might Be Found

	Coding and Logic-Design Errors	*Design Errors*
Module test	65%	0%
Function test	30%	60%
System test	3%	35%
Total	98%	95%

Source : The Art of Software Testing, G. J. Myers, pré-cité

Les tests automatisés…

- "*Je ne comprends pas pourquoi, hier ça marchait !* «
  - Leitmotiv des développeurs (et des étudiants en démo ;-))
  - Le Chef de projet, le client, le prof… ne veut plus entendre ça.
- Au fait, la maintenance de votre code va être effectuée par qui ? Vous ?
  - Le développeur effectue des tests manuels tout en codant, il a la fonctionnalité en tête, c'est facile ; mais le test ne concerne qu'un contexte donné (souvent avec un scenario **nominal**)
  - Mais dans 6 mois ? Quand il aura ajouter d'autres fonctionnalités qui peuvent avoir un impact sur ce premier code ?
    - Réécrire ce code, l'enrichir

Les tests automatisés (2)

- Les tests automatisés servent à **sécuriser** les personnes chargées de faire évoluer votre code
- Quand on commence à avoir 20 fonctionnalités, il devient impossible de déduire instinctivement TOUS les impacts.

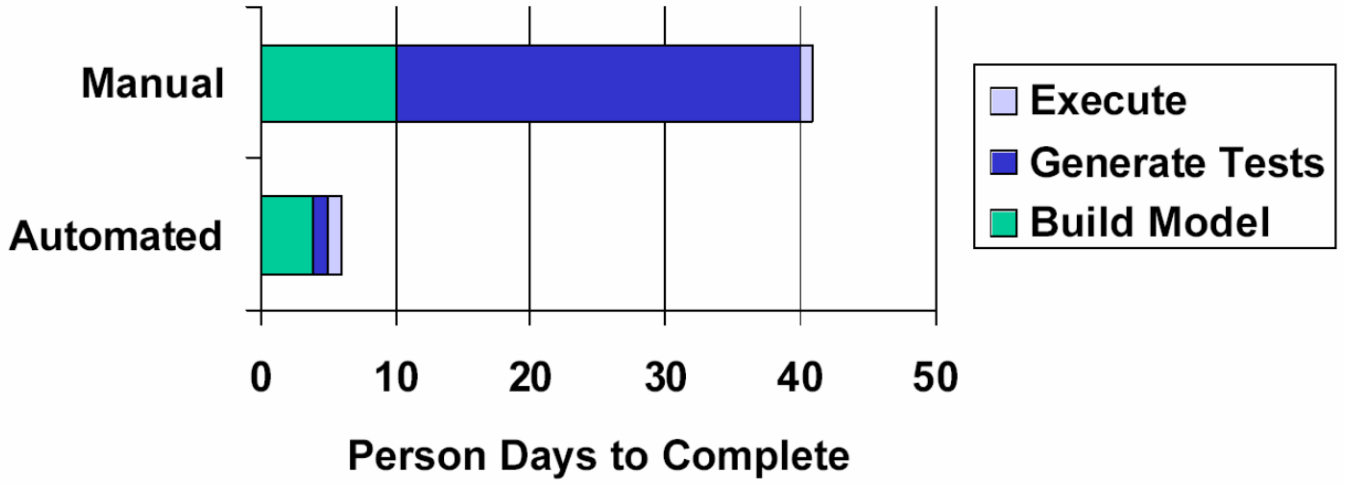
- Prendre quelques minutes à écrire des tests pour chaque partie de code, c'est gagner des **heures de débogage**

- **Couverture fonctionnelle** = % du code qui est testé

- Les tests **automatiques** permettent aussi de tester une application de manière exhaustive
  - CR de tests fournis au Client, preuve de qualité
- Les tests **manuels** sont :
  - Longs et fastidieux (jamais exhaustifs)
  - Pas assez fréquents
    - Un petit bug laissé mijoté devient un gros bug, long à déceler
  - Peu fiables (pour le client)

Un test automatisé est entre **5 et 10x plus rapide à faire** qu'un test manuel.

Les tests automatiques augmentent généralement la couverture fonctionnelle.



Source : The Art of Software Testing, G. J. Myers, pré-cité

Tests dit de « non régression »

- C’est la certitude que le code ne sera pas **abîmé** quand on le modifie
- Les tests donnent **confiance** : on avance de façon plus sereine dans l’écriture et la modification du code
  - Utilisé pour le « refactoring » de code : réécriture / modification du code
- Une fois qu’on est habitué, on ne peut plus s’en passer !

A quoi d’autres servent les tests ?

- (en plus de tester)

- A **documenter**
  - Le test est un exemple d'utilisation et de manipulation de ses classes : il permet donc de **documenter** celles-ci.

Comment faire les tests unitaires ?

Avec Java :

- JUnit est un **framework** open source pour le développement et l'exécution de tests unitaires automatisables
- Repose sur des **assertions** qui testent le résultat du code par rapport à ce qui est attendu

Il existe des équivalents avec *PHPUnit*, *pytest*, etc.

Souvent ces frameworks sont intégrés dans les IDE.

Assertions de JUnit

Méthode	Rôle
assertEquals()	Vérifier l'égalité de deux valeurs de type primitif ou objet (en utilisant la méthode equals()). Il existe de nombreuses surcharges de cette méthode pour chaque type primitif, pour un objet de type Object et pour un objet de type String
assertFalse()	Vérifier que la valeur fournie en paramètre est fausse
assertNull()	Vérifier que l'objet fourni en paramètre soit null
assertNotNull()	Vérifier que l'objet fourni en paramètre ne soit pas null

Assertions de Junit (2)

assertSame()	Vérifier que les deux objets fournis en paramètre font référence à la même entité Exemples identiques : assertSame("Les deux objets sont identiques", obj1, obj2); assertTrue("Les deux objets sont identiques ", obj1 == obj2);
assertNotSame()	Vérifier que les deux objets fournis en paramètre ne font pas référence à la même entité
assertTrue()	Vérifier que la valeur fournie en paramètre est vraie

Bonnes Pratiques des tests unitaires

- Un test ne concerne **qu'une seule** fonctionnalité
- Les *asserts* sont **séparés** des actions (sur différentes lignes)
- Eviter les tests **liés à l'implémentation**
- Ne pas mettre **trop de vérifications** dans un seul test
  - pas de "scénario" de test complexe dans un test unitaire
- S'assurer de la **reproductibilité** des tests

  (« test fixture » : le contexte est bien défini et reproductible)

Bonnes Pratiques

Bonnes Pratiques des tests unitaires (2)

- Ne pas faire de tests **inconsistants**, càd. :
  - Des tests qui utilisent des valeurs aléatoires
  - Des tests qui utilisent la date/heure courante
  - Des tests qui supposent un ordre d'exécution des tests
  - Des tests non unitaires (ex : dépendance à une base de données)

Bonnes Pratiques

Exploiter les annotations avec JUnit

Les annotations sont des indications pour le compilateur.

Grâce aux annotations, vous n'avez pas à coder le lancement des tests ou d'autres fonctionnalités propres aux tests.

Il faut simplement ajouter des annotations, et JUnit s'occupe de tout : @BeforeEach, @AfterEach, @BeforeAll, @AfterAll, etc. mais aussi paramétrer un test :

Exemple : initialiser une instance de la classe testée avant chaque test

@BeforeEach

```
public void initCalculator() {
   System.out.println("Appel avant chaque test");
   calculatorUnderTest = new Calculator();
}
```

Illustration des annotations (suite)

- Pour mesurer le temps de traitement des tests :

```
@BeforeAll
static public void initStartingTime() {
        System.out.println("Appel avant tous les tests");
        startedAt = Instant.now();
}

@AfterAll
static public void showTestDuration() {
        System.out.println("Appel après tous les tests");
        Instant endedAt = Instant.now();
        long duration = Duration.between(startedAt, endedAt).toMillis();

        System.out.println(MessageFormat.format("Durée des tests : {0} ms", duration));
}
```

Exploiter les annotations avec Junit (fin)

Pour le paramétrage, JUnit4 permet de mentionner par exemple quand une exception doit être levée :

```
@Test(expected=IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException() {
  ArrayList emptyList = new ArrayList();
  Object o = emptyList.get(0);
}
```

Junit5 va plus loin avec @ParametrizedTest(…)

https://openclassrooms.com/fr/courses/6100311-testez-votre-code-java-pour-realiser-des-applications-de-qualite/6465561-structurez-vos-tests-unitaires-avec-les-annotations-junit

Quel niveau de test écrire ?

- Écrire des tests **triviaux** revient à perdre du temps : c'est inutile
  - Ils doivent être un peu complexe !

- Etudions l'exemple de **FizzBuzz**
- **https://www.youtube.com/watch?v=RWYvBNX9wcU**

- Le test représente **en moyenne un tier du temps** de développement ;
- Le test est **destructif** alors que l'écriture de code est une activité constructive : créer des fonctionnalités ;
- Il a **mauvaise réputation** car il est long, coûteux, et qu'il met souvent en retard les projets... mais... il **permet souvent d'éviter des anomalies** qui seraient encore plus coûteuses.

Quels scénarios imaginer ?

- Scenario **nominal** et les scenarii **d'exception**

- Être créatif et imaginer des scénarios pour mettre un logiciel en défaut
  - Exemple : si je donne un Prix Unitaire nul à mon article da la Facture, que se passe-t-il ?…
  - *(ou un PU négatif, ou …)*
- Il faut imaginer des jeux de tests pour vérifier l'ensemble des fonctionnalités et des contraintes

- Il est important que les personnes qui codent et les personnes qui testent soient différentes !

Je retiens…

- Définition du *refactoring*, *TDD*, *couverture fonctionnelle, scenario nominal*
- Les différents types de tests
- L’avantage des tests automatiques
- Le cycle du *refactoring* / TDD
- Les principales assertions de JUnit
- Les BP d’écriture de tests

Références utiles

- Les 10 commandements des tests Unitaires http://blog.xebia.fr/2008/04/11/les-10-commandements-des-tests-unitaires/
- TDD Pentamino en Visual C# (Kent Beck) http://bruno-orsier.developpez.com/tutoriels/TDD/pentaminos/

- Les erreurs classiques des TDD http://bruno-orsier.developpez.com/tutoriels/java/antipatrons-tests-unitaires/
- Vidéos de Nadia Humbert (Crafties) sur le TDD https://www.youtube.com/watch?v=RWYvBNX9wcU