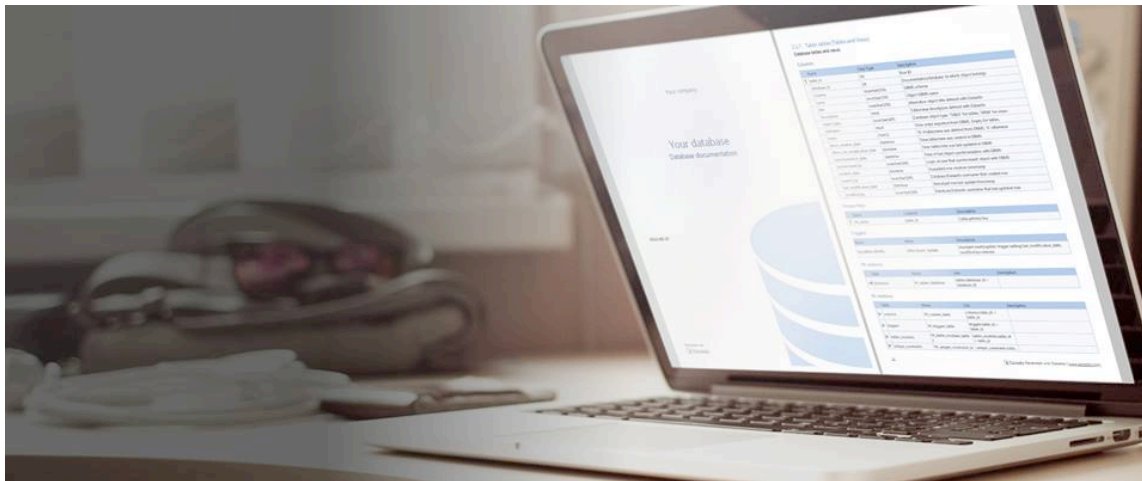


Génération de Documentation et Tests Unitaires

V. Deslandres, IUT de LYON

Module CVDA – s2



3 types de commentaires

- Le commentaire pour la [javadoc](#) `/** */`
- Le commentaire multi-lignes `/* */`
 - Souvent utilisé pour ignorer une partie de code
- Le commentaire d'une seule ligne `//`
 - Souvent informatif

sample

Class Vectors

java.lang.Object
sample.Vectors

```
public final class Vectors
extends java.lang.Object
```

Sample utility class for vector algebra.

Method Summary

All Methods

Static Methods

Concrete Methods

Modifier and Type

Method and Description

static boolean

equal(int[] a, int[] b)

Checks whether the given vectors are equal.

static int

scalarMultiplication(int[] a, int[] b)

Scalar multiplication of given vectors : la somme des produits des entiers de chq tableau

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Il n'y a pas que JavaDoc !

- Générateurs de documentations

- JavaDoc
- Doxygen
- PhpDoc
- ...



- Ce qu'on peut générer :

- Html
- Txt
- Man
- Pdf

Un exemple

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

Description
fonctionnelle

Les tags @

Tests automatisés

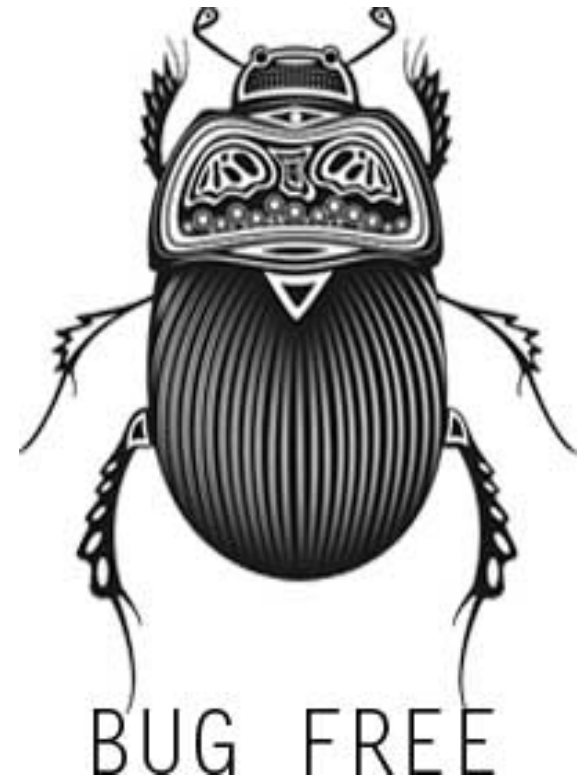


JUnit, un pas vers un meilleur code

LES TESTS UNITAIRES

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 le métier sans avoir à développer les 2 autres parties.



Les tests automatiques...

- *"Je ne comprends pas pourquoi, hier ça marchait !" «*
 - lieu commun 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 ?**
 - Les tests servent aussi à **sécuriser** les personnes chargées de faire évoluer votre code
- Prendre quelques minutes à écrire quelques tests, c'est gagner des **heures de débogage.**

Test dit de « non régression »

- C'est la certitude que procurent les tests en garantissant 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
 - On parle de « refactoring » de code :
réécriture / modification du code
- Une fois qu'on est habitué, on ne peut plus s'en passer !



A quoi servent les tests (automatiques) ?

- (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.
- **A définir des tests d'acceptation**
 - Souvent, les TU sont définis quand on spécifie le besoin avec le « client »

Ex. test d'acceptation

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

```
@Test
```

```
public void
```

```
avecRoverFaceNordQuandPivoterDroiteAlorsFaceEst() {
```

```
    rover.pivoterDroite();
```

```
    assertEquals(Orientation.EST,  
rover.orientation);
```

```
}
```

➔ test réel avec le robot



- 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 (donc jamais exhaustifs)
 - Pas assez fréquents
 - Un petit bug laissé mijoté, devient un gros bug long à trouver
 - Peu fiables (pour le client)



Comment faire avec JUnit ?

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

Assertions de JUnit

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

Assertions de Junit (2)

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

BP de tests

Bonnes
Pratiques

- Les asserts sont **séparés** des actions
 - (ie 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 pour lequel les tests passent est bien défini et reproductible)
- Ne pas faire de tests **inconsistants**, ie :
 - 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)

Gestion des exceptions avec JUnit4

JUnit 4 permet de mentionner quand une exception doit être levée

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

Objectifs des tests

Lisibilité
Maintenabilité
Fiabilité

Jouer les tests
SOUVENT

Code de production + code de test



Photo: Création Nature

Tests **automatisés**



TDD, le développement dirigé par les tests

TEST DRIVEN DEVELOPMENT

Pourquoi ne pas démarrer par les tests ?

Ecrire les tests **avant le code (TDD)** a plusieurs avantages :

- Ça permet de recenser les **cas d'utilisations**
 - Par ex. : pour un programme de traitement de chaînes, fonction de concaténation de chaînes

```
public static String concatWords(String... words) {  
    StringBuilder buf = new StringBuilder();  
    for (String word : words) {  
        buf.append(word);  
    }  
    return buf.toString();  
}
```

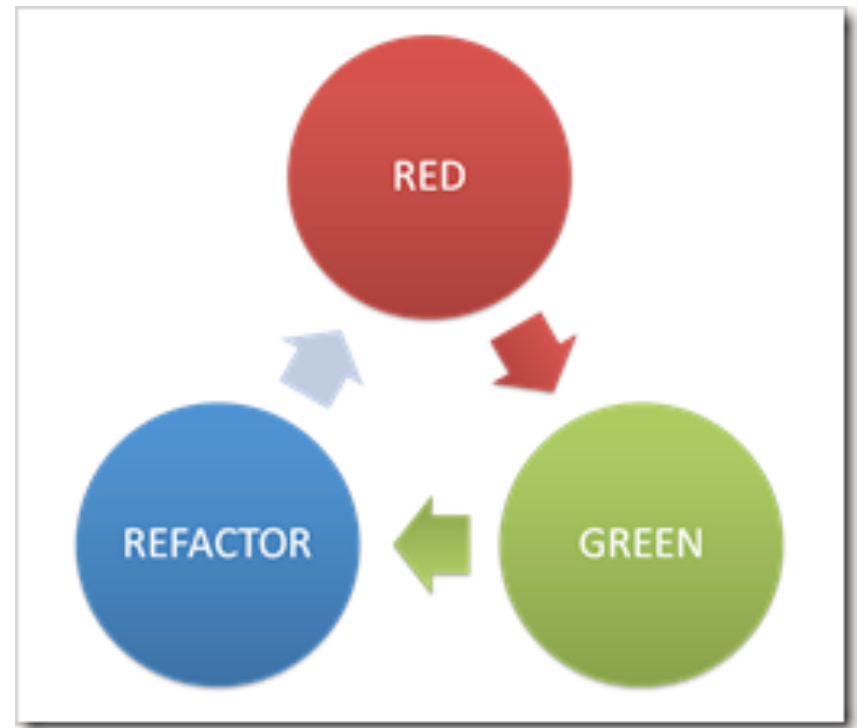
- Définir et tester toutes les règles de **pré et post conditions** de la méthode ;
- Définir les vérifications de paramètres nécessaires ;
- Développement incrémental
 - Une fois les tests (bien écrits) refactorisés au vert, on passe au point suivant.

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**



Le cycle du TDD

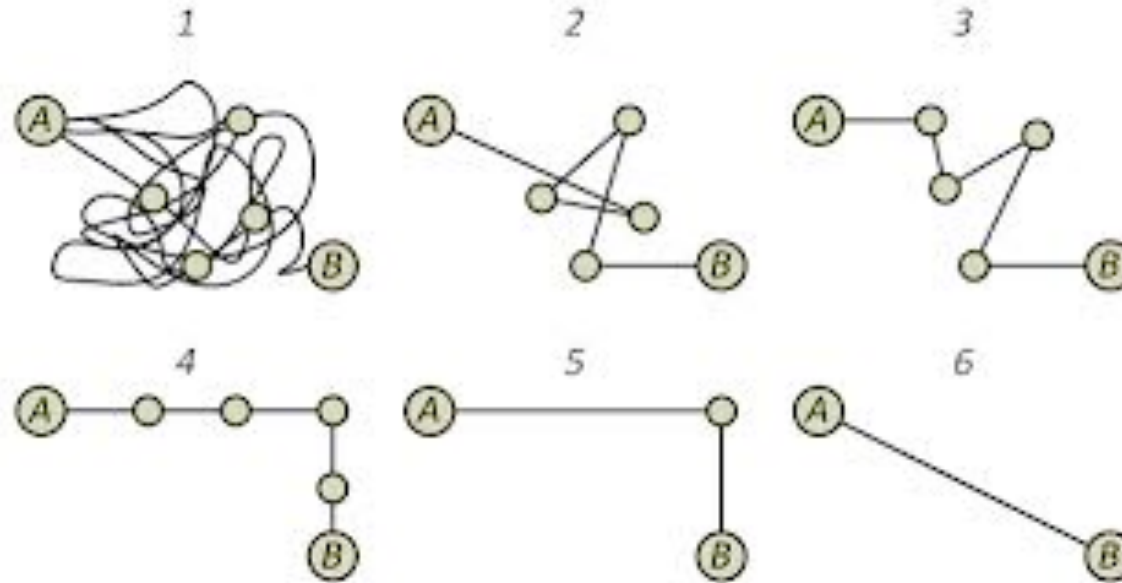


- Le test représente en moyenne 1/3 du temps de développement
- Le test a **mauvaise réputation** car il est long, coûteux, et qu'il met souvent en retard les projets... mais... il **permet souvent d'éviter des anomalies** qui seraient encore plus coûteuses.

Quels scénarios imaginer ?

- Être créatif et imaginer des scénarios pour mettre un logiciel en défaut
 - Exemple : si je donne un Prix Unitaire nul à mon article dans 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 !

Refactoring : généralités



- Les améliorations de performance concernent TRES SOUVENT les CONDITIONS :
 - Vérifier si elles sont **nécessaires**,
 - Vérifier **l'ordre d'écriture** des conditions
 - Etudier si on peut **les optimiser**

Quelques BP de **programmation**

- Nommer correctement vos variables, par ex.

```
event1.IsOverlappedBy(event2);  
october17from15to17.IsOverlappedBy(october17from16to18);
```

- Factoriser
 - créer des méthodes
 - ou par ex. pour le code de tests, utiliser @Before, @After etc.
- Si une classe est difficile à tester, c'est qu'il faut **faire du refactoring**
 - Est-elle trop volumineuse ?
 - Présente-t-elle trop de dépendances ?
 - Profitez de l'occasion pour la découper et déplacer du code dans des classes annexes.

Quelques BP de **programmation** (2)

- **Attributs** : toujours en *private*
- Pour les attributs, jamais de *new()* en dehors des constructeurs
 - Pensez à créer suffisamment de constructeurs pour rendre le code flexible

- **Pas d'affichage de sortie** dans les méthodes



<pre>public void calculTVA() { ... System.out.println("TVA=... "); }</pre>	<pre>public double calculTVA() { ... return tva; }</pre>
--	--

- Gérer les interactions dans le *main* ou via une GUI
- Une **seule responsabilité** par méthode
 - Exemple ci-après : *SRP - Single Responsibility Principle*

SRP example

Soit la classe Book suivante :

- Réfléchir à **ce qui change** d'un livre à un autre
 - Son titre, son auteur
 - Pas sa façon d'obtenir la page courante, ni d'avoir sa position dans la bibliothèque
- Penser en termes **d'utilisation** de la classe Book : quels « acteurs » en feraient quoi ?
 - Le lecteur
 - Le libraire



```
class Book {  
  
    function getTitle() {  
        return "A Great Book";  
    }  
    function getAuthor() {  
        return "John Doe";  
    }  
    function turnPage() {  
        // pointer to next page  
    }  
    function getCurrentPage() {  
        return "current page content";  
    }  
    function getLocation() {  
        // returns the position in the library  
        // ie. shelf number & room number  
    }  
}
```

Meilleure conception - SRP

```
class Book {  
  
    function getTitle() {  
        return "A Great Book";  
    }  
  
    function getAuthor() {  
        return "John Doe";  
    }  
  
    function turnPage() {  
        // pointer to next page  
    }  
  
    function getCurrentPage() {  
        return "current page content";  
    }  
}  
  
Utilisée par l'acteur  
Lecteur
```



```
class BookLocator {  
  
    function locate(Book $book) {  
        // returns the position in the library  
        // ie. shelf number & room number  
        $libraryMap->findBookBy($book-  
        >getTitle(), $book->getAuthor());  
    }  
}
```

Si la bibliothèque change, ça n'aura aucun impact sur la classe Book (et donc les instances de livres existantes).

Et réciproquement.

Références utiles

- Les 10 commandements des tests Unitaires
<http://blog.xebia.fr/2008/04/11/les-10-commandements-des-tests-unitaires/>
- SRP
<http://code.tutsplus.com/tutorials/solid-part-1-the-single-responsibility-principle--net-36074>
- 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>