Les Tests unitaires

Les tests sont appelés et classifiés de très nombreuses manières différentes.
On pourra entendre parler de :

- Tests unitaires
- Tests fonctionnels
- Smoke tests (test rudimentaire pour s'assurer du bon fonctionnement général du logiciel)
- Tests d'intégration
- Tests de non-régression
- Tests en boîte blanche
- Tests en boîte noire
- Tests de validation
- ... et beaucoup d'autres

Ces termes recoupent parfois les mêmes idées, et certains supposent des points de vue particuliers.

Ici on ne commencera à ne parler que de tests unitaires.

Les tests unitaires entrent dans la couverture de code

https://fr.wikipedia.org/wiki/Couverture_de_code

Exemple : SonarQube

Comme le nom le suggère, un test unitaire s'applique à une "unité" de code, quelque chose de réduit comme une méthode.

Pour à peu près toutes les méthodes que l'on écrit, on sait ce qui doit en sortir quand on lui applique une entrée spécifique.

Le test unitaire sert à s'assurer que c'est bien ce qui arrive.

Si la méthode que l'on cherche à tester a besoin de services extérieurs, comme une base de données, ce ne peut pas être un test unitaire. On entre alors dans l'univers des tests d'intégration que l'on n'aborde pas ici.

Les tests unitaires font partie des tests en boîte blanche : en écrivant le test, on sait à quoi ressemble le code qui est testé. On sait précisément ce qu'il doit faire et on va essayer tous les cas particuliers possibles.

On prendra pour cible les classes 'métiers' qui sont de parfaits candidats aux tests unitaires :

Certains setters acceptent ou rejettent des valeurs particulières, on sait donc assez clairement ce qui autorisé.

Les classes métiers sont le coeur du programme, il est important qu'elles soient testées

Les autres classes sont beaucoup plus difficiles à tester : on ne peut pas vraiment écrire de code pour dire à quoi doit ressembler nos fenêtres Swing.

On pourrait en revanche tester que le click sur un bouton déclenche bien telle ou telle action, mais il nous faudrait des outils supplémentaires qui ne sont pas à l'ordre du jour : les Mocks.

JUnit

JUnit est et a toujours été la solution de tests unitaires la plus populaire sous Java.

TestNG a un temps été un concurrent sérieux mais est en grande perte de vitesse.

JUnit est un membre de la famille des "xUnit", des frameworks de tests unitaires écrit pour différents langages, initié par 'SUnit' (pour le langage SmallTalk) qui ont tous une structure similaire.

Par exemple :

- PHPUnit
- PyUnit en Python
- NUnit en C#
- HUnit en Haskell
- JSUnit en JavaScript (abandonné)
- hUnit pour Bash
- etc.

On utilisera ici **JUnit *5***, qui est une réécriture majeure sortie en 2017, mais on gardera à l'esprit qu'un très grand nombre de documentations sur internet concernent encore **JUnit 4**.

Ecrire une classe de test

- Créer un package au même niveau que le package main
- Sélectionner la classe
- Appuyer sur Alt Enter et sélectionner Create test
- La structure des classes et packages de tests fait miroir aux classes testées. Exemple, la classe de Test pour la classe Foo doit s'appeler FooTest
- La classe contient des méthodes annotées avec @Test, pour les tests simples.

- Les méthodes de test ne doivent pas être privées, mais ne doivent pas nécessairement être publiques
- Dans chaque méthode on écrit un code à tester. Si jamais une exception est lancée sans être attrapée, le test échoue.
- Si un test se termine sans encombre, il est considéré comme passé.
- Les méthodes d'assertion lancent des exceptions quand l'assertion n'est pas vérifiée

  Définition d'une assertion : Proposition que l'on avance et que l'on soutient comme vraie

- Par défaut, l'écriture de chaque test est en trois parties :
  1. *Arrange*, préparer les données
  2. *Act*, effectuer l'opération
  3. *Assert*, vérifier l'état des résultats.
- On n'hésitera pas à briser les conventions de nommage si cela sert à lisibilité

Les tests comme aide au développement

En plus de vérifier que le résultat de notre code est celui attendu, les tests unitaires ont pour effet de bord de vous inciter à aller vers les bonnes pratiques.

Si la méthode semble trop compliquée à tester, c'est peut-être :

- Qu'elle est trop longue
- Trop compliquée
- Fait trop de choses
- Dépend de trop de paramètres
- etc.

Peut-être aussi qu'une fonction serait une bonne candidate aux tests, mais on l'en a inutilement privée ?

La TDD (Test-Driven Development) est une méthode qui consiste à écrire les tests *avant* le code réel et oblige ainsi à ne pas oublier le test et à réfléchir davantage à ce que doit concrètement faire la fonction. Quand le réflexe est installé, il structure la démarche.

Les Assertions les plus utiles

assertTrue(); // La paramètre à l'entrée doit être vrai
assertFalse(); // Je vous laisse deviner

assertEquals(); // Les deux paramètres sont de valeur égale
assertSame(); // Les deux paramètres sont les même objets

assertNull();
assertNotNull();

// Le premier paramètre est la classe de l'exception attendue, le deuxième
// est une lamdba qui doit lancer ladite exception
assertThrows();
// Un seul paramètre : un lambda qui ne lance rien
assertDoesNotThrows();

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