**SOMMAIRE**

I) Introductions aux tests  
 A) Cycle en V  
II) Tests Unitaires  
 A) Assertions  
 B) Programmation par contrat  
III) Tests unitaires avec JUnit  
 A) Création d’une classe de test  
 B) Lancement des classes de test  
 C) Annotations  
 D) Répéter un test, avec différentes valeurs  
IV) Programmation défensive  
 A) Introduction  
 B) Comment choisir nos tests  
V) Test-driven development (TDD) et Behavior driven development (BDD)  
 A) Test-driven development (TDD)  
 B) Behavior driven development (BDD)

1. Introductions aux tests

Le but premier est de vérifier si un logiciel fait bien ce qu’il doit faire.

Il existe deux grandes familles de tests et une famille intermédiaire :

* boite noire (on ne connait pas le code, seulement entrés&sorties)
* boite blanche (on connait tout : code, entrés et sorties).

Et une famille intermédiaire :

* boite grise (on a des informations partielles)

On parle de test static lorsqu’on analyse le code sans le tester.

1. Cycle en V

Le cycle en V est une méthode de développement de projets en 3 phases :  
 - phase de conception : pense le projet  
 - phase de réalisation : code en faisant des tests unitaires  
 - phase de validation : on test le projet

Analyse des besoins et faisabilité   
(discute avec le client)  
 Spécifications (delais)  
 Conception Architecturale  
 (MVC, cahier des charges)  
 Conception détaillée  
 (diagrammes de classes)  
 CODAGE

Recette  
 (version beta)  
 Validation (vérifie a système, comportements)  
 Test d’intégration  
 (vérifie les briques)  
 Tests unitaires (vérifie bout)  
  
  
  
  
CODAGE a

N’étant pas flexible, il n’est pas toujours adapté à chaque situation. Il s’agît d’un des cycles de développement en informatique.

1. Tests Unitaires

On teste une partie du logiciel. On met un main dans chaque classe pour la tester. Il s’agit de vérifier qu’une toute petite partie du code fait ce qu’elle doit faire.

1. Assertions

Les assertions permettent de vérifier (condition) une qu’une information est vraie à un moment du programme. Elles sont désactivées de base.  
 - "-ea" (-enableassertions) à java pour les activer partout  
 - "-da:package" à java pour une activation dans certains packages

Une fois activées, on place dans notre code un assert qui lève une AssertionError si la condition est évaluée à false.  
 assert(condition) ou assert(condition) : MessageErreur

Remarque : Les étant rarement lues, on préférera utiliser des exeptions pour les méthodes publiques (=programmation défensive)

1. Programmation par contrat

La programmation par contrat par du principe que si les préconditions sont correctes alors les postconditions les sont aussi.

On peut utiliser une assertion pour vérifier les pré/post conditions et les invariants d’une méthode.  
 - préconditions d’une méthode privée  
 - postconditions d’une méthode  
 - invariants

Les préconditions représentent le fait que tout est correct lors de l’appel de la méthode (arguments/attributs…)  
Les postconditions représente le fait que tout est correct après l’appel de la méthode. (vérifier résultat, vérifier valeurs…)  
Un invariant est une propriété/quelque chose qui reste vrai avant et après l’appel de la méthode. (taille liste si on inverse deux éléments…)

Ex : un objet ne doit pas être NULL : Object.requireNonNull(Object)  
 un objet est compris dans un intervalle : if(…<0) throw new Exception

Remarque : Si un répète un test pour une précondition, on peut créer une classe qui se charge de vérifier la précondition.

1. Test unitaires avec JUnit

On devrait faire des tests unitaires pour vérifier que :  
 - toutes les méthodes ont été exécutées  
 - toutes les structures ont été testées pour les deux cas  
 - tous les chemins ont été exécutés  
si on convient qu’il est intéressant de faire des tests sur une méthode.

JUnit permet de facilement gérer les tests unitaires pour décider quels tests exécuter. (parce que dans le main, ça devient compliqué)

Installer JUnit : [JUnit - Environment Setup](https://www.tutorialspoint.com/junit/junit_environment_setup.htm)

On associe à chaque fichier, son fichier de test généralement nommé :  
 - contient le mot Test  
 - contient la méthode testée ou la classe  
 - un numéro  
Il n’y a pas vraiment d’ordre et les 3 éléments sont optionnels.

1. Création d’une classe de test

Une classe de test est une classe normale cependant devant chaque méthode de test, on met l’annotation @Test. (org.junit.Test)

Par principe, une méthode = 1 test.

JUnit nous offre de nouveaux types d’asserts (org.junit.Assert) :

|  |  |
| --- | --- |
| assertEquals(Object, Objet) | Regarde si deux objets sont égaux |
| assertFalse(condition) assertTrue(condition) | Vérifie condition fausse Vérifie condition vraie |
| assertNotNull(Objet) assertNull(objet) | Vérifie qu’un objet n’est pas null Vérifie q’un objet est null |
| assertSame(Object, Object) assertNotSame(…) | Vérifie que 2 instances ont même référence et réciproquement |
| assertArrayEquals(t1, t2) | Regarde si t2 est égal à t1 |
| fail() | Échoue un test sans message |

La classe TestCase (org.junit.TestCase) contient de nombreuse méthodes pour obtenir le nombre de tests exécutés, les noms des tests, lancer un test…

1. Lancement des classes de test

On lance l’exécution des classes de tests avec la méthode :  
 JUnitCore.runClasses(ClasseTest0.class,…);  
Remarque : l’avantage d’utiliser la méthode est qu’elle renvoi un objet de type Result qui contient les messages d’erreur et de succès.

Ou encore à la compilation directement, avec la commande :  
 java org.junit.runner.JUnitCore ClasseTest0

1. Annotations
2. Listes annotations

On a vu que l’on ajoute l’annotation @Test devant une classe de test. Il existe d’autres annotations pour dire qu’elle méthodes lancer avant/après le test, classes à ignorer…

|  |  |
| --- | --- |
| @BeforeClass | Méthode static lancé avant méthodes d’instance |
| @Before | Méthode lancé avant CHAQUE Test |
| @Test | Méthode de test |
| @Ignore | Ignorer un test, une classe de tests |
| @After | Méthode lancé après CHAQUE test |
| @AfterClass | Méthode static lancé après méthodes d’instance |

Il existe également les annotations de classes (avant la déclaration de la classe) qui indique si au lancement on doit lancer d’autres tests.  
@RunWith(Suite.class) ;  
@Suite.SuiteClasses( { ClasseTest0.class, … } );  
Lancer la classe qui contient ces lignes lance également les classes ClasseTest0…

1. L’annotation @Test

L’annotation @Test ne se contente pas de signaler une méthode de test, elle permet de :  
 - définir un temps max d’exécution (ms) : @Test( timeout = value)  
 - vérifier une exception levée : @Test( expected = Exception.class)

1. Répéter un test, avec différentes valeurs

On peut lancer encore et encore un test en changeant les paramètres.  
La classe concernée :  
 - doit être annotée @RunWith(Parameterized.class) ;  
 - doit avoir un constructeur qui correspond aux paramètre  
 - doit avoir une méthode STATIC annotée avec @Parameterized.Parameters qui renvoie une collection (Arrays.asList()) qui contient chaque objet à créer.

1. Programmation défensive
2. Introduction

On restreint les paramètres/attributs avant l’appel de la méthode et on envoi des exceptions adaptés. On teste ensuite les invariants. Enfin, on vérifie après l’exécution que les propriétés sont satisfaites.

On lève des exceptions dans les méthodes publiques (on doit bien la choisir !) mais dans le reste des cas, on fait des asserts pour les invariants et les post-conditions.

1. Comment choisir nos tests

La méthode RightBicep :   
 Right - est-ce que les résultats sont ok, ce que doit faire la méthode (faire des tests qui doivent réussir et d’autres échouer).  
 B - teste robustesse de la méthode (entré incohérente) : CORRECT  
 I - Vérifier inverse s’il en existe un  
 C - test autre méthode, vérifier en parallèle avec une source sûre  
 E - vérifier conditions d’erreurs, simule erreurs (Mock objects)  
 P - performances (étudie l’efficacité grosses valeurs, Junitperf…)

CORRECT teste la robustesse d’une méthode :  
 C - Conforme (arguments ok)  
 O - Ordre (valeurs ordonnées)  
 R - Intervalle (matrice carré ligne=colonne)  
 R - Reference (code référence quelque chose d’externe ?)  
 E - Vide (null , ça marche)  
 C - cardinalité (fait varier la taille, bon nombre de valeurs reçu)  
 T - temps : appelé dans le bon ordre (pas de méthodes avant…)

1. Test-driven development et Behavior driven development
2. Test-driven development

Le développement piloté par les tests (TDD) consiste en la création de tests avant même que l’application soit créé.

Le principe consiste à écrire des bouts de codes, et de ne les relier ensembles qu’à la fin.

Le procédé est de la forme :  
 - Crée des tests  
 - Lance les tests, vérifie qu’ils échouent car pas encore codés  
 - Écrit du code pour le réussir en réussir un  
 - Lance les tests et on vérifie qu’il passe  
 - et on recommence

Ensuite, on factorise le code et on revérifie les tests.

Pour penser nos test, on pense à quel fonctionnalités notre programme doit avoir. On pense ensuite des tests pour chacune d’entre elles.

Le développement piloté par les tests et dans la prolongation de behavior driven developpement.

1. Behavior driven development

Le behavior driven development (BDD) consiste à imaginer les comportements liés à la fonctionnalité avant de la coder.

On regardera par exemple :

* vérifier que l’on veut créer la fonctionnalité souhaitée, changement est nécessaire
* se poser les questions sur quoi implique l’ajout d’une fonctionnalité (prévoir possibles futures fonctionnalités), et demander à l’équipe de projet

La syntax du BDD est :  
 Etant donné (Given) une action (utilisateur est/fait)  
 Quand (When) cet action entraine …  
 Alors (Then) je fais …  
 Et (And) cela entraine …

On écrit les cas négatif et positifs (tous les cas).