Tests unitaires

Par Cyrille MBIA

Au programme

Généralités

- Pourquoi écrire des tests ?
- Différence entre test unitaire et test d'intégration
- Démarche du TDD

Le Framework JUnit

- Définition
- Structure d'un test case
- Les différentes assertions

Généralités

Pourquoi écrire des tests ?

Écrire des scénarios de tests automatisés (tant des tests unitaires que des tests d'intégration) constitue une étape incontournable du cycle de développement logiciel dans un cadre professionnel. Bien que cette tâche puisse parfois sembler rébarbative, elle est indispensable pour produire des développements de bonne qualité, notamment parce que cela permet entre autres de :

- s'assurer que l'on maîtrise les aspects fonctionnels de l'application (les besoins métier à satisfaire) ;
- détecter au plus tôt les anomalies éventuelles avant la livraison d'un projet aux utilisateurs ;

Généralités

Pourquoi écrire des tests ?

- détecter d'éventuelles régressions, suite à l'implémentation de nouvelles fonctionnalités, en automatisant l'exécution de l'ensemble de tous les scénarios de tests implémentés sous forme de tests unitaires ou tests d'intégration, de façon à ce qu'ils soient exécutés régulièrement (à minima une fois par jour par exemple).

Plus l'ensemble de vos codes sera couvert par les tests, plus vous serez à même de détecter les régressions, de maîtriser l'ensemble des fonctionnalités de votre application et ainsi de la rendre plus maintenable.

Généralités

Différences entre tests unitaires et tests d’intégration

De manière générale, le test unitaire a pour but de tester une partie précise d'un logiciel (d'où le nom "unitaire") voire une fonction précise, en essayant de couvrir au maximum la combinatoire des cas fonctionnels possibles sur cette portion de code.

Cependant, cela ne suffit pas à différencier un test unitaire d'un test d'intégration qui, lui aussi, peut se contenter de tester une portion précise du code.

Généralités

Différences entre tests unitaires et tests d’intégration

La différence fondamentale est que le test unitaire doit uniquement tester le code et ne doit donc pas avoir d'interactions avec des dépendances externes du module testé telles que des bases de données, des Web services, des appels à des procédures distantes, la lecture ou écriture de fichiers ou encore d'autres fonctions. Les dépendances devront être simulées à l'aide de mécanismes tels que les bouchons ou encore les **mocks**.

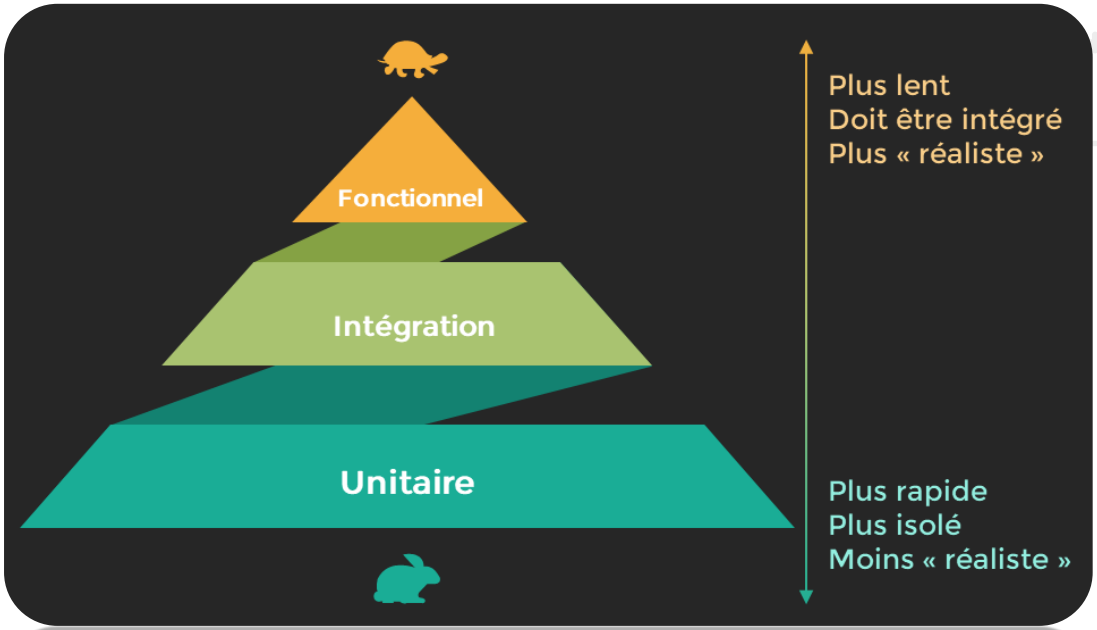
Généralités

Différences entre tests unitaires et tests d’intégration

Concernant les tests d'intégration, ceux-ci permettent, de manière générale, de vérifier la bonne intégration de la portion de code testée dans l'ensemble du logiciel et de ses dépendances. Il peut s'agir tout aussi bien de vérifier la bonne insertion de données dans un SGBDR sur une petite portion de code que de scénarios complets du logiciel, correspondant à des enchaînements d'appels de fonctions, de Web services... (c'est ce que l'on appelle aussi des tests « bout en bout »).

Généralités

Différences entre tests unitaires et tests d'intégration



Généralités

La démarche du TDD

TDD est un acronyme pour « **Test Driven Development** » ou encore le développement dirigé par les tests en français. Il s'agit d'une démarche qui recommande au développeur de rédiger l'ensemble des tests unitaires avant de développer un logiciel ou les portions dont il a la charge.

Généralités

La démarche du TDD

Cette démarche permet de s'assurer que le développeur ne sera pas influencé par son développement lors de l'écriture des cas de tests afin que ceux-ci soient davantage exhaustifs et conformes aux spécifications. Ainsi, cette démarche permet aussi de s'assurer

que le développeur maîtrise les spécifications dont il a la charge avant de commencer à coder.

Généralités

La démarche du TDD

L'approche TDD préconise le cycle court de développement en cinq étapes :

1. écrire les cas de tests d'une fonction ;

2. vérifier que les tests échouent (car le code n'existe pas encore) ;

3. développer le code fonctionnel suffisant pour valider tous les cas de tests ;

4. vérifier que les tests passent ;

5. refactoriser et optimiser en s'assurant que les tests continuent de passer

Le Framework JUnit

Qu’est-ce que Junit

JUnit est un framework Java prévu pour la réalisation de tests unitaires et d'intégration.

JUnit est le framework le plus connu de la mouvance des frameworks xUnit implémentés dans de nombreuses technologies.

Le Framework JUnit

Qu’est-ce que Junit

JUnit permet de réaliser :

- ❖ des TestCase qui sont des classes contenant des méthodes de tests ;
- ❖ des TestSuite qui permettent de lancer des suites de classes de type TestCase.

Voir la slide suivante pour avoir la structure globale d’une classe de test écrite en JUnit >= 4.

Le Framework JUnit

Structure d'un TestCase

```
import org.junit.Before;
import org.junit.After;
import org.junit.BeforeClass;
import org.junit.AfterClass;
import org.junit.Test;

public class MaClasseDeTest {

    /** Pre et post conditions */

    @BeforeClass
    public static void
setUpBeforeClass() throws Exception
{
        // Le contenu de cette
méthode ne sera exécuté qu'une fois
avant toutes les autres méthodes
avec annotations
        // (y compris celles ayant
une annotation @Before)
    }

    @AfterClass
    public static void tearDownClass()
throws Exception {
        // Le contenu de cette méthode
ne sera exécuté qu'une fois après
toutes les autres méthodes avec
annotations
        // (y compris celles ayant une
annotation @After)
    }

    @Before
    public void setUp() throws
Exception {
        // Le contenu de cette méthode
sera exécuté avant chaque test (méthode
avec l'annotation @Test)
    }

    @After
    public void tearDown() throws
Exception {
        // Le contenu de cette méthode
sera exécuté après chaque test (méthode
avec l'annotation @Test)
    }

    /** Cas de tests */
    @Test
    public void testCas1() {
        // Code contenant l'exécution du
premier scénario avec les assertions
associées
    }

    @Test
    public void testCas2() {
        // ...
    }
}
```

Le Framework JUnit

Les différents types d’assertions

En informatique, une assertion est une expression qui doit être évaluée *vraie* ou faire échouer le programme ou le test en cas d’ échec (en levant une exception ou en mettant fin au programme par exemple).

Dans le cas de JUnit et des tests d'une manière générale, on peut assimiler une assertion à une vérification et, en cas d’échec, une exception spécifique est levée (java.lang.AssertionErrorpour JUnit).

Le Framework JUnit

Les différents types d’assertions

Voici une liste non exhaustive des fonctions d'assertions en JUnit :

- **assertEquals**: vérifier l'égalité de deux expressions ;
- **assertNotNull**: vérifier la non-nullité d'un objet ;
- **assertNull**: vérifier la nullité d'un objet ;

Le Framework JUnit

Les différents types d’assertions

- **assertTrue**: vérifier qu'une expression booléenne est vraie ;
- **assertFalse**: vérifier qu'une expression booléenne est fausse ;
- **fail** : échouer le test si cette assertion est exécutée.

Ces méthodes sont surchargées pour de nombreux objets. Par exemple, la méthode **assertEquals** va être surchargée pour tous les types primitifs et pour les classes implémentant la méthode **equals** (String par exemple). Il est également possible de fournir un message d'erreur en cas d'échec d'une assertion (en premier paramètre).

Le Framework JUnit

Exemples d'assertions simples

Imaginons que l'on veuille tester la classe suivante :

```
public class Addition {
    Integer nb1, nb2;

    public Addition(Integer nb1, Integer nb2){
        this.nb1 = nb1;
        this.nb2 = nb2;
    }
    public Integer somme(){
        return nb1 + nb2;
    }
    public Integer getNb1() {
        return nb1;
    }
    public Integer getNb2() {
        return nb2;
    }
}
```

Merci

https://itdreamtech.com

cyrille@itdreamtech.com

cyrillembia@iaicameroun.com