INF2015 – Développement de logiciels dans un environnement Agile

Tests unitaires

Jacques Berger

Objectifs

Introduire la pratique des tests unitaires

Introduire JUnit 4

Présenter quelques techniques pour faciliter la mise en test

Présenter une méthode pilotée par les tests

Prérequis

Java

Refactoring

Tests

Test unitaire

Un petit test qui ne vérifie qu'une petite partie du logiciel

Test fonctionnel

Vérifie une fonctionnalité du système, peut être fait à partir du GUI

Tests

Test de régression

Vérifie que la fonctionnalité n'a pas été brisée par un changement

Test de stress

Habituellement pour les applications web ou distribuées, vérifie le comportement de l'application lors de forts achalandages

Tests

Test d'intégration

Vérifie l'intégration de différents composants du système

Test d'acceptation

Le client approuve que la fonctionnalité développée correspond à ce qu'il voulait

Tests

Toutes les formes de tests sont les bienvenues dans un projet de développement de logiciel

Tests unitaires

Du code pour tester le code

La forme de test la plus légère, avec la portée la plus limitée

Ne s'applique qu'à une seule fonctionnalité d'une classe

Plusieurs tests unitaires peuvent être nécessaires sur une méthode

Tests unitaires

Sert de test fonctionnel à très petite échelle

Sert de test de régression

Tests unitaires

Sur une classe, on peut tester :

- La fonctionnalité
- La non-fonctionnalité
- La gestion des erreurs
- Les exceptions
- Les cas limites
- Les cas hors bornes

Tests unitaires

Les propriétés d'un test unitaire :

- Exécution très rapide
- Code simple
- Indépendant des autres tests
- Doit pouvoir s'exécuter en tout temps

Tests unitaires

Un test unitaire ne doit pas :

- Manipuler un fichier
- Traiter avec une base de données
- Effectuer une communication sur un réseau
- Nécessiter un environnement de test

Vocabulaire

Classe du domaine : Une classe contenant de la logique qu'il faut tester

Classe de test : Une classe qui contient les tests pour une classe du domaine

Suite de tests : Une suite contenant plusieurs classes de tests

Avantages

Permet de détecter les erreurs plus tôt

Transfert de connaissances

Facilite la maintenance et le refactoring

Très payant à long terme

Inconvénients

Couplage fort entre la classe du domaine et la classe de tests

Plus de code à maintenir (code de test)

Les tests peuvent être bogués

Activité souvent difficile

Framework

Le framework de tests unitaires par excellence avec Java est JUnit

Assertions

On vérifie les résultats dans un test unitaire à l'aide d'assertions

Une assertion est une condition qui doit toujours être vraie

Bonnes pratiques

Entretenir le code de test comme si c'était du code de production
- Éliminer la duplication
- Faire du refactoring
- Appliquer des patrons de test

Essayer de n'avoir qu'une assertion par test

Bonnes pratiques

Avoir une bonne couverture de tests

Exécuter nos tests après chaque modification du code

Bogues

Un bogue est un test oublié!

Lorsqu'on détecte un nouveau bogue, on tente de l'isoler dans un test unitaire

Ce test nous assure qu'on ne réinjectera pas le bogue une deuxième fois dans le code

JUnit 4

JUnit est une plateforme xUnit pour Java

On retrouve des implémentations de xUnit dans plusieurs langages

Emplacement

En général, on regroupe les tests pour une classe du domaine dans une classe de tests JUnit

Cette classe de tests peut être placée à peu près n'importe où

Emplacement

Pratique courante : on crée un répertoire test à la racine (au même niveau que src), ensuite on crée un package de tests pour chaque package du domaine qu'on veut tester

On place la classe de tests dans le même package que la classe du domaine, mais pas nécessairement dans le même répertoire

Test unitaire

Exemple

```
public class SqlTransformerTest {

    @Test
    public void testTransformIdentifierNormal() {
        assertEquals(
            "FIRST_NAME",
            SqlTransformer.transformIdentifier("first_name"));
    }
}
```

Junit 3

Avant, il fallait :

- Hériter de TestCase
- Le nom de la méthode devait commencer par Test

JUnit 4 utilise les annotations de Java plutôt qu'une convention de nommage

Annotations

Les annotations Java ajoutent de la flexibilité à JUnit.

Les méthodes de test doivent avoir l'annotation @Test

Annotations

On peut ignorer un test avec @Ignore

On peut ajouter un timeout qui fera échouer le test si la méthode est trop longue à s'exécuter, le timeout est en millisecondes @Test(timeout=10)

Annotations

On peut tester l'envoie d'une exception avec @Test(expected=IOException.class)

@Before indique que la méthode sera exécutée avant chaque test, habituellement nommée setUp

@After indique que la méthode sera exécutée après chaque test, habituellement nommée tearDown

Annotations

@BeforeClass exécutera la méthode une seule fois avant l'ensemble des tests de la classe

@AfterClass exécutera la méthode une seule fois après l'ensemble des tests de la classe

Assertions

fail : fait échouer le test

assertTrue : fait échouer le test si le paramètre vaut false

assertFalse : fait échouer le test si le paramètre vaut true

Assertions

assertEquals : vérifie l'égalité de 2 valeurs

assertNull : vérifie la nullité

assertNotNull : vérifie la non nullité

assertSame : les deux instances sont le même objet

assertNotSame : les instances sont différentes

Pratique

On exécute les tests aussi souvent que possible

Toutes les classes ne sont pas faciles à tester, certaines sont presque impossibles à tester

Un petit refactoring pourrait être nécessaire avant de pouvoir créer un test sur une méthode

Pratique

Techniques utiles

- Fake objects
- Mock objects
- Injection de dépendance

Fake Object

Un faux objet qui se fait passer pour un autre objet dont nous dépendons

Les méthodes du fake retourneront habituellement des valeurs hard-coded

Ex. : Simuler une base de données

Mock Object

Une forme plus évoluée d'un fake object dans lequel le faux objet ajoute de la fonctionnalité de test

Ex. : Simuler une base de données et vérifier la génération des requêtes SQL

Injection de dépendance

Lorsqu'une classe encapsule l'utilisation d'une autre classe, il peut être très utile de sortir la création de l'objet de la classe et de le passer à l'instance par un setter ou un constructeur

Design for Testability

Faire la conception de façon à favoriser l'écriture des tests

- Favoriser l'injection de dépendance
- Ajouter des constructeurs par défaut
- Permettre des références null

TDD

Test-Driven Development

Créé par Kent Beck

Méthode Agile qui prône la création de tests unitaires et le refactoring

TDD

Idée de base : On veut une couverture de test la plus haute possible et l'on rédige toujours nos tests avant le code

TDD

Red – green – refactor

1: Écrire un test pour une nouvelle fonctionnalité

2: Écrire une implémentation qui fait échouer le test

3: Écrire une implémentation qui fait marcher le test

4: Faire du refactoring jusqu'à l'obtention d'un design satisfaisant

TDD

Avantages

- Première réflexion sur l'utilisation de la classe
- Grande banque de tests
- Meilleure maintenabilité
- Favorise le refactoring
- Design for Testability

TDD

Inconvénients

- Nouvelles fonctionnalités seulement
- Discipline personnelle

Liens

jUnit

http://www.junit.org/