LES TESTS UNITAIRES

OBJET :	Aide au développement
RÉFÉRENCE :	Tests unitaires, junit, cobertura, tdd, test driven developpement
ÉQUIPE :	Méthode
AUTEUR(S) :	POITEVINEAU Romain
DESTINATAIRE(S) :	Developpeurs - testeurs
VERSION / DATE :	1.0, le 12/02/2012
STATUT :	DRAFT
VALIDÉ PAR :	Developpeurs - testeurs

Remarque

Cette documentation est faite pour apporter une introduction aux tests unitaires, ces principes de Fonctionnements et pourquoi nous allons les utilisées dans le projet.

En cas de problème rencontré ne figurant pas dans la dans la documentation merci de transmettre Le problème à l'équipe méthode. Merci également d'en faire parvenir la solution si vous l'avez trouvé

Sommaire

I - Introduction aux tests unitaires

1 - Qu'est ce que c'est ?

Le test unitaire est un procédé automatisé et donc renouvelable à l'infini permettant de s'assurer du bon fonctionnement d'une partie déterminée d'un programme (généralement une fonction ou une suite de fonctions). Le test unitaire est un code, indépendant du programme testé qui permet de s'assurer que ce dernier répond aux spécifications fonctionnelles et qu'il fonctionne correctement en toutes circonstances. Un test unitaire est particulièrement utile lors de l'évolution d'une partie de code, afin de s'assurer qu'il n'y a aucun impact ou effets de bord non maîtrisés.

2 - Gain de temps ?

La programmation de test unitaire n'est pas quelque chose d'anodin. Le temps passé a penser et coder ces tests a un impact réel sur les temps de développement, qu'il ne faut surtout pas négliger lors des estimations. Ce temps n'est cependant pas investit à perte puisqu'il permet d'augmenter grandement la qualité du code produit et de diminuer considérablement la phase de maintenance corrective. Grâce à eux la phase de maintenance évolutive est également facilitée puisque les risques de régressions sont bien moindres. Afin de ne pas trop pénaliser le développement il convient aux développeurs de bien choisir les portions de code à tester et les scénarii de test à passer. Il est inutile de s'éterniser sur les fonctions simples ou sans impacts tandis qu'il faudra se montrer extrêmement pointilleux et exhaustive lors du test des points critiques de l'application.

3 - Quand coder les tests unitaires ?

Pour être réellement efficace les tests unitaires doivent être développés en même temps que la fonction testée (au pire juste après et au mieux avant le code testé) de manière à servir de garant du bon fonctionnement de cette dernière. Ils servent en quelque sorte de contrat de travail.

Lors de la livraison de code source de la part d'un developpeurs, demandez également à recevoir les tests unitaire associé. **En fait un code source livrée sans les tests DOIT être considéré comme incomplet**. N'hésitez surtout pas à repasser ces tests de manière à vous assurer de la qualité du travail que vous avez réceptionné.

4 – Couverture de code par les tests unitaires

Il existe des outils permettant aux développeurs de juger de la couverture de code unitaire. L'outil préconisé est Cobertura. Il ne se focalise pas sur le nombre de fonction testé, mais le nombre de ligne de chaque méthode. Il permet par exemple de savoir que certaines conditions d'une fonction ne sont pas couvertes par le code. Il ne faut surtout pas prendre le résultat brut de Cobertura (ou de tout autre outil de test de couverture de code), à savoir le pourcentage de l'application couverte par les tests unitaire pour argent comptant, mais de l'utiliser comme indicateur pour vérifier que tout les zone de code identifié comme critique sont convenablement testé.

II – Outils JAVA

Les outils de tests unitaires préconisés peuvent s'utiliser à travers l'outil de développement **Eclipse** ou via une ligne de commande Ant ou Maven. Ils peuvent donc être à la fois utilisés par les développeurs tout au long de leurs développements pour valider la qualité du travail réalisé.

1 – JUnit

L'utilisation de framework de tests unitaires est essentielle à la constitution d'un code robuste. Le framework préconisé est JUnit. Il permet une création rapide de test unitaire, afin de faire gagner le maximum de temps au développeur et un résultat clair lors du passage des tests pour indiquer de manière simple quels tests ont réussis et quels tests ont échoués et pourquoi.

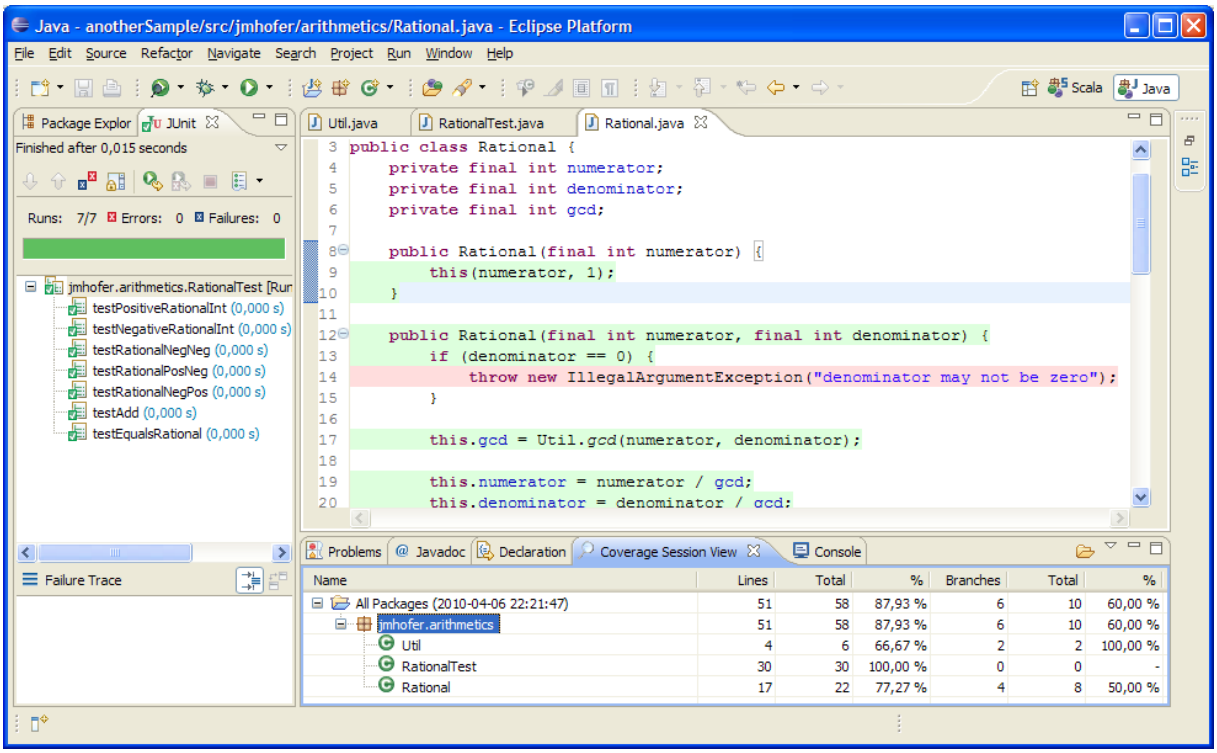
Le framework JUnit: https://github.com/KentBeck/junit/downloads

2 – Cobertura

Cobertura permet de calculer le pourcentage de code couvert par des tests unitaire. Il permet d'avoir une vue et des rapports aussi bien macroscopique (valeur pour l'application et les paquets) que microscopique (ligne de code réellement parcourut par les tests unitaires). Les comptes rendu de ces analyses peuvent être publiés sous forme html pour une consultation aisée. C'est donc l'outil idéal pour une surveillance générale et fréquente des parties globales du programme et une analyse poussée sur les zones critiques de l'application.

Plugin Cobertura Eclipse : **http://ecobertura.johoop.de/update/** (nécessite Eclipse 3.5+)

Plugin Cobertura pour eclipse :



III – Ecriture de tests

1 – Ecrire un test

Avec JUnit, on va créer une nouvelle classe pour chaque classe testée. On crée autant de méthodes que de tests indépendants : imaginez que les tests peuvent être passés dans n'importe-quel ordre (différent de celui dans lequel elles apparaissent dans le code source).

Il n'y a pas de limite au nombre de tests que vous pouvez écrire. Néanmoins, on essaye généralement d'écrire au moins un test par méthode de la classe testée.

Pour désigner une méthode comme un test, il suffit de poser l'annotation @Test.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class StringTest {

    @Test
    public void testConcatenation() {
        String foo = "abc";
        String bar = "def";
        assertEquals("abcdef", foo + bar);
    }

    @Test
    public void testStartsWith() {
        String foo = "abc";
        assertTrue(foo.startsWith("ab"));
    }
}
```

2 – Les assertions

Via import static org.junit.Assert.*; , vous devez faire appel dans les tests aux méthodes statiques assertTrue, assertFalse, assertEquals, assertNull, fail, etc. en fournissant un message :

```
?
```

Votre environnement de développement devrait vous permettre de découvrir leurs signatures grâce à l'auto-complétion. À défaut, vous pouvez toutes les retrouvez dans la documentation de l'API JUnit .

3 – Lancer les tests

Pour lancer les tests, vous avez plusieurs possibilités selon vos préférences :

- La plus courante : lancer les tests depuis votre IDE
- Utiliser l'outil graphique
- Lancer les tests en ligne de commande
- Utiliser un système de construction logiciel (comme Ant ou Maven)

4 – Factoriser les éléments commum entre deux tests

On peut déjà chercher à factoriser les éléments communs à tous les tests d'une seule classe. Un test commence toujours par l'initialisation de quelques instances de formes différentes pour pouvoir tester les différents cas. C'est souvent un élément redondant des tests d'une classe, aussi, on peut factoriser tout le code d'initialisation commun à tous les tests dans une méthode spéciale, qui sera appelée avant chaque test pour préparer les données.

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class StringTest {

    static String foo;
    static String bar;

    @Before // avec cette annotation, cette méthode sera appelée avant
chaque test
    public void setup() {
        foo = "abc";
        bar = "def";
    }

    @After
    public void tearDown() {
        // dans cet exemple, il n'y a rien a faire mais on peut,
        // dans d'autres cas, avoir besoin de fermer une connexion
        // à une base de données ou de fermer des fichiers
    }

    @Test
    public void testConcatenation() {
        assertEquals("abcdef", foo + bar);
    }

    @Test
    public void testStartsWith() {
        assertTrue(foo.startsWith("ab");
    }
}
```

La méthode annotée @Before, est exécutée avant *chaque* test, ainsi, chaque test est exécuté avec des données saines : celles-ci n'ont pu être modifiées par les autres tests.

On peut également souhaiter factoriser des éléments communs à plusieurs classes de tests différentes, par exemple, écrire un test pour une interface et tester les différentes implémentations de ces interfaces. Pour cela, on peut utiliser l'héritage en écrivant les tests dans une classe de test abstraite ayant un attribut du type de

l'interface et en écrivant ensuite une classe fille par implémentation à tester, chacune de ces classes filles ayant une méthode @Before différente pour initialiser l'attribut de la classe mère de façon différente.

5 – Verifier la levée d'une exeption

Il peut être intéressant de vérifier que votre code lève bien une exception. Vous pouvez préciser l'exception attendue dans l'annotation @Test. Le test passe si une exception de ce type a été levée avant la fin de la méthode. Le test est un échec si l'exception n'a pas été levée. Vous devez donc écrire du code qui *doit* lever une exception et pas qui *peut* lever une exception.

```
@Test(expected = NullPointerException.class)
public void methodCallToNullObject() {
    Object o = null;
    o.toString();
}
```

6 – Désactivé un test temporairement

Au cours du processus de développement, vous risquez d'avoir besoin de désactiver temporairement un test. Pour cela, plutôt que de mettre le test en commentaire ou de supprimer l'annotation @Test, JUnit propose l'annotation @Ignore. En l'utilisant, vous serez informé qu'un test à été ignoré, vous pouvez en préciser la raison.

```
@Ignore("ce test n'est pas encore prêt")
@Test
public void test() {
    // du code inachevé
}
```

Historique des versions

VERSION	DATE	AUTEUR	DESCRIPTION	STATUT
1.0	12/02/12	POITEVINEAU ROMAIN		