Tests unitaires avec jUnit

MEDEV

Basé sur les transparents de G. Moreau

Introduction

- Tester un programme
  - Pourquoi faire ?
    - Euh…
    - Test de non-régression : « je ne comprends pas, ça marchait hier »
    - Gagner du temps
  - Comment ?
    - Automatiser les tests ?
    - Automatiser et systématiser leur exécution
      - À la livraison mais surtout à chaque modification même mineure
  - Quand ?
    - eXtreme Programming et TDD

Test Driven Development

- Motivations
  - Affiner l'analyse, les tests sont des *uses cases* au sens d'UML
  - Ils permettent de modéliser les pré et les post conditions d'un algorithme
  - Eviter d'écrire du code inutile
  - Une forme de documentation technique
    - Le test est un exemple d'utilisation de la classe qu'on vient d'écrire

Motivation

- Les logiciels évoluent
  - Modification de fonctionnalités (exigences)
  - Supprimer les bugs
  - Optimisations
  - Restructuration du code (refactoring)
- Peut-on faire ces changements en toute confiance ?
  - Ne pas casser le comportement actuel
  - Ne pas introduire de nouveaux bugs
  - Hypothèse raisonnable : pas de connaissance complète de la base de code
- Ecrire des tests automatisés est une condition necessaire pour passer au niveau supérieur

Pas assez de temps pour faire des tests !

- On aurait peut-être plus de temps si on avait écrit les tests avant. . .
- On ne peut pas vraiment se permettre de ne pas écrire de tests
  - Effectuer de grosses modifications dans un code sans tests, c'est comme débuter le trapèze sans filet (M. Feathers "Working Effectively with Legacy Code")
  - Les bugs détectés tardivement coûtent un ordre de grandeur de plus à résoudre
    - Ca se ressent sur le projet

Sans les tests appropriés. . .

- A un moment donné, on ne pourra plus faire de modifications dans le projet sans doute raisonnable
- On aura l'impression qu'il n'y a plus d'espoir qu'un jour tous les bugs soient enfin réparés
  - chaque bug réparé en crée un ou deux autres
  - on en vient à livrer avec des bugs pas trop critiques
  - Le moral de l'équipe baisse et le turnover augmente
- Ca vous rappelle quelque chose ?

Écrire des tests = du temps pour les nouvelles fonctions !

- Comment ceci peut-il être vrai ?
  - Rendre le code testable amène souvent à une meilleure architecture
    - Oblige à prendre en compte les besoins du client
    - *testable* signifie souvent réutilisable et peu dépendant, i.e. faire plus en moins de temps
  - S'ils sont fait correctement, les tests ne devraient pas prendre très longtemps à écrire
  - tester chaque module isolément petits ou grands tests ?

Petits tests plutôt que grands tests ?

- Grands tests
  - ce sont les tests systèmes, les tests d'intégration, de non-régression les tests avec les utilisateurs
  - Difficile de localiser les erreurs détectées
  - Lents et arrivent à la fin du processus
  - Qui peut prouver qu'ils sont effectivement tous effectués ?
- Petits tests
  - les fameux tests unitaires
  - objectif : tester de façon isolée un module, une classe, une fonction
  - Rapides et faisables très tôt dans le processus de développement
  - A vérifier avant de fournir votre code aux autres développeurs !
- Bilan : les grands tests sont importants mais ils ne dispensent pas des petits tests

Les tests en java : jUnit

- Au départ, une série de classes
- Intégration complète à NetBeans [plugins] (et à d'autres IDE)
- Principe
  - Une classe de test hérite de junit.framework.TestCase
  - Écrire des méthodes de test d'une autre classe
    - Commençant par le nom testXXXX()
    - Ou en utilisant l'annotation @Test (jUnit 4.x)
    - Mettant en œuvre les assertions
  - Lancer le tout en fournissant la classe à un TestRunner

Terminologie

- Test unitaires : test d'une classe
- Cas de tests (Test case) : teste les réponses d'une méthode à un ensemble particulier d'entrées
- Suite de tests (Test suite) : collection de cas de test
- Testeur (Test Runner) : programme d'exécution de tests qui construit un rapport des résultats

Les assertions

- De l'anglais *assert* : affirmer
- Dans le package org.junit.Assert
- assertTrue(*expression*) : l'expression doit être vraie sinon le test échoue
- assertEquals(a,b) : a=b vrai sinon le test échoue
- assertFail(« Echec ») : fait échouer un test
- Il existe des assertions beaucoup plus puissantes
  - Voir la classe Assert et ses méthodes statiques
  - http://junit.org/junit4/javadoc/latest/index.html

jUnit en mode texte

```
public class FirstTest {

        @Test
        public void testEmptyCollection() {
                Collection collection = new ArrayList();
                assertTrue(collection.isEmpty());
        }

        @Test
        public void testTrue() {
                assertTrue(true);
        }

        @Test
        public void testFalse() {
                assertFalse(true);
        }

        public static void main(String[] args) {
                org.junit.runner.JUnitCore.main("tests.FirstTest");
        }
}
```

Remarques

- @Test : définit une méthode qui servira pour le test
- Le main() n'est pas obligatoire
- Résultat [tronqué d'exécution]

```
JUnit version 4.8.1
...E
Time: 0,009
There was 1 failure:
1) testFalse(tests.FirstTest)
java.lang.AssertionError:
…
FAILURES!!!
Tests run: 3,  Failures: 1
```

Annotations utiles

- @Test : méthodes de test
  - @Test(expected=XXXException) : teste si une méthode lève bien une exception
- @Before public void method()
  - Méthode exécutée avant chaque test
- @After public void method()
  - Méthode exécutée après chaque test
- @BeforeClass public void method()
  - Méthode exécutée avant le début de tous les tests
- @AfterClass public void method()
  - Méthode exécutée après tous les tests
- @Ignore : test pas exécuté

Une classe à tester

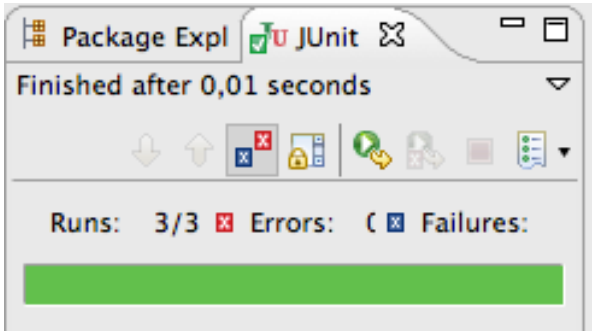
```java
/** retourne la valeur absolue de a
 *
 * @param a input
 */
public static int methode(int a) {
    if (a < 0) {
        return -a;
    }
    else {
        return a;
    }
```

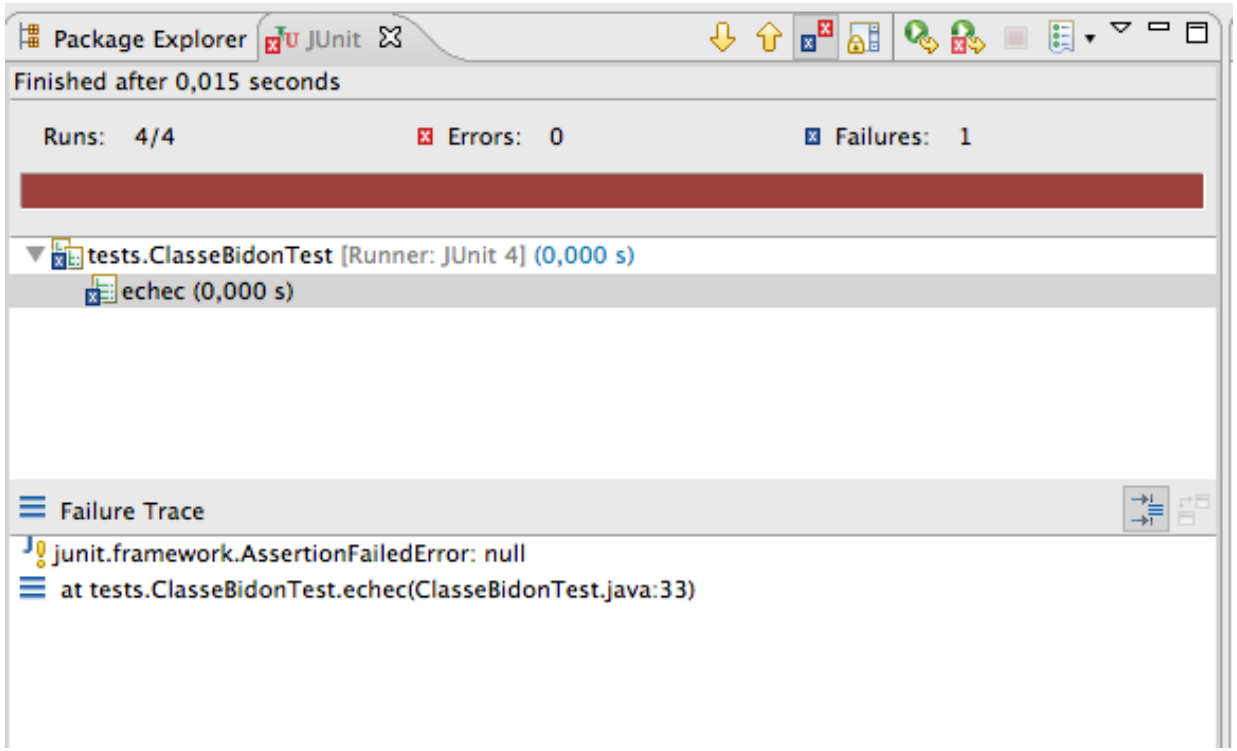
Code de test (un peu exagéré)

```
@Test
public void Absnulle() {
    assertEquals(0,tests.ClasseBidon.methode(0));
}

@Test
public void AbsPos() {
    assertEquals(1,tests.ClasseBidon.methode(1));
    assertEquals(10,tests.ClasseBidon.methode(10));
    assertEquals(120,tests.ClasseBidon.methode(120));

}

@Test
public void AbsNeg() {
    assertEquals(1,tests.ClasseBidon.methode(-1));
    assertEquals(10,tests.ClasseBidon.methode(-10));
    assertEquals(120,tests.ClasseBidon.methode(-120));
```

Lancer les tests (via Run)

- Résultats (quand tout va bien)
  - Le reste de la fenêtre pour les erreurs et les détails





Remarques

- Bonnes pratiques
  - Une classe, un test
    - Convention de nommage TestXXXX pour une classe XXXX
- Mauvaises pratiques
  - Écrire des tests triviaux (cf. use cases)
  - Effets de bord dans les tests (et en particulier dans les couches de persistance)
  - Présumer de l'ordre d'exécution des tests