TP bonus sur les tests: NFA035 – Bibliothèques et patterns

V. Aponte, P. Courtieu, S. Rosmorduc

March 5, 2014

Tests JUnit.

Un exemple de test avec JUnit4

Tester les fonctions

Que signifie *tester* une fonction en programmation? Il s'agit d'invoquer la fonction avec des arguments valides, puis de comparer le résultat obtenu avec le résultat que la fonction est censée calculer dans ce cas.

Par exemple, pour tester une fonction `factorielle(int x)` qui doit calculer la factorielle d'un entier naturel, nous l'invoquons sur une entrée valide, par exemple 0, puis nous comparons le résultat obtenu avec 1, qui est notre résultat attendu. S'ils sont égaux, ce premier test est réussi. Évidemment, un seul test est rarement suffisant. Il faut donc tester la fonction sur plusieurs *cas de test*. Nous devons ainsi commencer par définir les cas de test intéressants, par exemple à l'aide d'une table, et ensuite les mettre en oeuvre de manière automatisée, par exemple avec la librairie JUnit. Dans cette table on mettra une colonne par nom de paramètre de la fonction à tester, et une colonne pour le résultat attendu. *Chaque ligne correspondra à un cas de test*.

x	résultat attendu
0	1
1	1
4	24

Figure 1: Quelques cas de tests pour la fonction `factorielle(int x)`

Établir de cas de tests est très utile pour valider du code écrit, mais c'est aussi un outil précieux pour réfléchir à ce que *doit faire* notre fonction avant même d'avoir écrit une ligne de code. Écrire les cas des tests avant d'écrire la fonction permet de garder à l'esprit *tous les cas du problème*, plutôt que *tous les cas prévus dans notre code*. Écrire les cas de test *avant* de coder les fonctions est utilisé comme principe de développement dans le *Extreme Programming*. Nous vous conseillons d'employer cette technique.

Mise en oeuvre des cas de test en JUnit4

Nous allons transcrire les cas de tests de la figure 1 en JUnit4, et cela avant même d'écrire le code de la fonction `factorielle(int x)`. Nous commençons par écrire une classe où se trouve cette fonction, mais avec un corps minimal pour permettre à la fonction de compiler:

```
/** Calcule la factorielle d'un entier naturel
 * @param x   entier naturel (qu'on suppose supérieur ou égale a zéro)
 * @return    la factorielle de x
*/
```

```
class Factorielle {
   public static int factorielle(int x){
      return -1;
  }
}
```

Sous Eclipse, après avoir demandé à utiliser les librairies JUnit4 (clic droit sur le répertoire du projet + *Build Path* + *Add Libraries* + JUnit. Attention: choisir JUnit4 et non 3!), vous demanderez à générer un cas de test pour cette classe (clic droit sur la classe à tester + *New ... JUnit Test*). Vous obtiendrez une nouvelle classe de nom `FactorielleTest`, où chaque cas de test (lign de notre table est à compléter par vous, sous la forme d'une méthode de *nom différent*. Cette méthode suit le schéma suivant:

```
        @Test
        public void test1() {
                int x =0;
                int expected = 1;
                int res = Factorielle.factorielle(x);
                assertEquals(expected,res);
        }
```

L'annotation `@Test` indique à JUnit qu'il s'agit d'un cas de test à jouer. La première ligne nous donnons la valeur pour le premier argument de la fonction, et dans la deuxième, le résultat attendu en retour d'appel. La 3ème, calcule le résultat de l'appel sur la valeur x, et la dernière invoque une méthode de JUnit pour comparer le résultat obtenu avec celui attendu et pour comptabiliser l'échec ou la réussite des tests dans ses statistiques (qu'il affiche pendant les tests). Voici comment on peut transcrire la table de la figure 1 par des tests Junit4:

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

public class FactorielleTest {

        @Test
        public void test1() {
                int x=0;
                int expected = 1;
                int res = Factorielle.factorielle(x);
                assertEquals(expected,res);
        }

        @Test
        public void test2() {
                int x=1;
                int expected = 1;
                int res = Factorielle.factorielle(x);
                assertEquals(expected,res);
        }

        @Test
        public void test3() {
                int x=4;
                int expected = 24;
                int res = Factorielle.factorielle(x);
                assertEquals(expected,res);
        }
}
```

Une fois tous les cas de test écrits, vous devrez les employer pour tester votre fonction avec clic droit + *Run as ... JUnit test*. Comme votre fonction ne fait pour l'instant pas grande chose, tous vos tests ou presque devraient

échouer. JUnit vous donnera le pourcentage de tests réussis et, dans le détail, les noms des méthodes de test ayant réussi et échoué.

Tester JUnit4

Maintenant, il ne vous reste qu'à écrire le code de la fonction, la vraie, et de rejouer dessus les cas des tests. S'il y a des échecs, cela signifie que votre fonction ne se comporte pas comme prévu par les cas de test. A vous de trouver l'erreur, de le corriger et de recommencer. Bien entendu, il faut rejouer tous les tests après chaque modification, même ceux qui avaient réussi avant: il s'agit de s'assurer que vos modifications n'ont pas "cassé" ce qui fonctionnait bien avant. Cela s'appelle faire des *tests de non régression*.

Tester les exceptions

Pour écrire un test pour lequel le comportement attendu est une exception vous pouvez utiliser le paramètre `expected` de la balise `@Test`.

```
@Test(expected=RuntimeException.class)
public void test4_dernierIndiceDe() {
        /* l'appel de méthode qui doit lever l'exception RuntimeException */
}
```

Exercice 1: test de fonctions statiques avec JUnit4

Dans cet exercice vous devez compléter un jeu de test existant pour une méthode statique déjà définie:

`int dernierIndice(int n,int t[])`

et une autre à définir:

`int compteMemeCases(int u[],int t[]).`

Ces fonctions sont dans la classe `Atester.java` et les commentaires spécifient leur comportement.