Université De La Rochelle

Licence 3 - Informatique

Génie logiciel

TP9 Test Unitaire

Mohammed Benaou

Responsable :
Pr. Armelle Prigent

06 Avril 2017

Table des matières

Table des figures

1 Introduction

Le principal de ce TP numero 9 est de s'assurer que le code répond toujours aux besoins même après d'éventuelles modifications.Nous allons maintenant voir comment les mettre en pratique grâce à JUnit, le framework de test unitaire de Java.

2 Exercice 1 : prise en main de Junit et premiers tests unitaires

Après l'ajout de la librairie JUnit à mon projet, on a crée la classe ForOperator.

2.1 classe ForOperator

voila maintenant le code correspondant à cette classe :

```
package FirstPackage;

public class ForOperator {
public int add (int a, int b){
return a+b;
}

public int sub (int a , int b) {
return a-b;
}

public int mult( int a  , int b ){
return a*b;
}

public int div (int a , int b ){
return a/b;
}
}
```

2.1.1 Ecriture des séquences de test

```
package SecondPackage;
```

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

import org.junit.Test;

import FirstPackage.ForOperator;

public class ForOperatorTest {

@Test
public void testAdd() {
 int result = 13 + 5;
 ForOperator simpleCalculator = new ForOperator();
 assertEquals(result, simpleCalculator.add(13, 5));
}

@Test
public void testSub() {
 int result = 13 - 5;
 ForOperator simpleCalculator = new ForOperator();
 assertEquals(result, simpleCalculator.sub(13, 5));
}

@Test
public void testMult() {
int result = 13 * 5;
 ForOperator simpleCalculator = new ForOperator();
 assertEquals(result, simpleCalculator.mult(13, 5));
}

@Test
public void testDiv() {
int result = 15 / 5;
 ForOperator simpleCalculator = new ForOperator();
 assertEquals(result, simpleCalculator.div(15, 5));
}
}
```

2.1.2 Exécution des séquences de test

2.1.3 Test d'un code ne respectant pas la spécification

on a insereé une erreure dans la fonction sub pour générer des erreurs au moment de l'exécution des tests
La trace d'exécution du test produit l'erreur détectée .

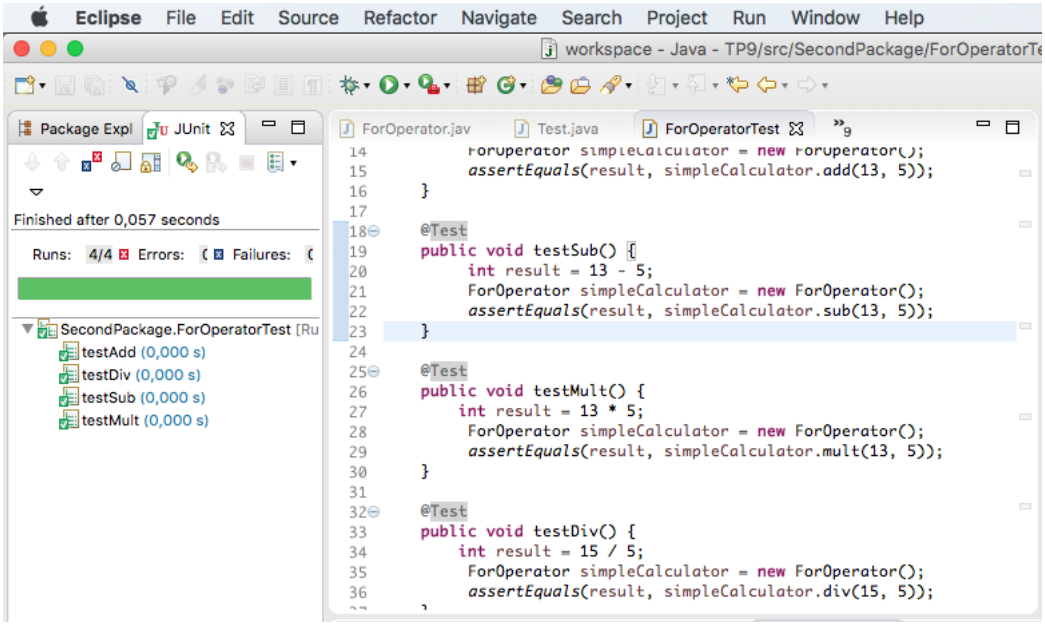

FIGURE 1 – execution sans erreur

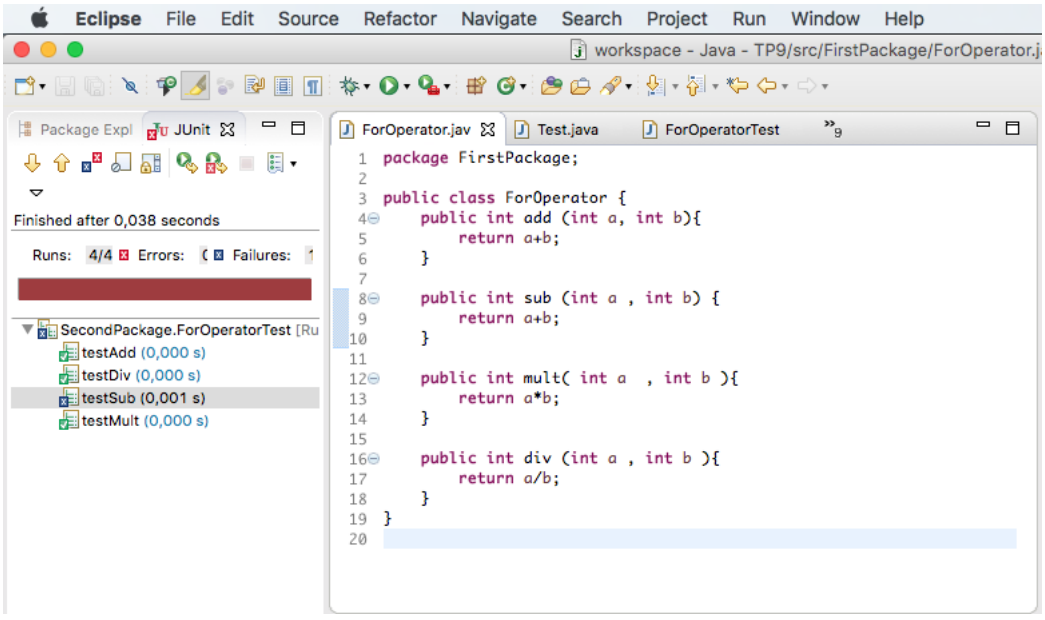

FIGURE 2 – execution avec erreur

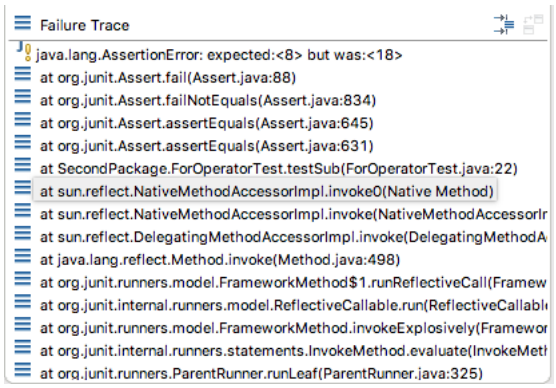

FIGURE 3 – La trace d'exécution

3 Exercice 2 : Ecriture des suites de test

en regroupant l'exécution de plusieurs cas de test dans une unique suite de test (la classe ForOperatorTest).

```
package FirstPackage;
public class ForOperator {
public int add (int a, int b){
return a+b;
}

public int sub (int a , int b) {
return a-b;
}

public int mult( int a  , int b ){
return a*b;
}

public int div (int a , int b ){
if (b==0){
return 0;
}
else

return a/b;
}
}
```

on a crée La classe TestLimite qui permet de tester le résultat obtenu par ces 4 opérateurs avec la modification pour gerer le cas de division par 0

```
package SecondPackage;

import static org.junit.Assert.*;

import org.junit.Test;

import FirstPackage.ForOperator;

public class TestLimite {

@Test
public final void TestAddNegative() {
int result = -13 + 5;
 ForOperator simpleCalculator = new ForOperator();
 assertEquals(result, simpleCalculator.add(-13, 5));
}

@Test
        public final void testSubNegative() {
  int result = -13 -(-5);
 ForOperator simpleCalculator = new ForOperator();
 assertEquals(result, simpleCalculator.sub(-13, -5));
}

@Test
public void testMultNagative() {
int result = -13 * 5;
 ForOperator simpleCalculator = new ForOperator();
 assertEquals(result, simpleCalculator.mult(-13, 5));
}

@Test
public void  testDivZero() {
 ForOperator simpleCalculator = new ForOperator();
 assertEquals(0, simpleCalculator.div(13, 0));

}
}
```

3.0.1 la recherche de bugs

Après l'importation du fichier .jar voilà le code de la classe TestList

```
package Package3;

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

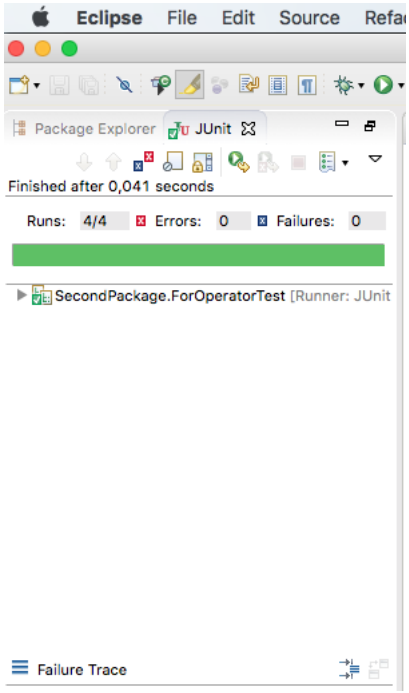

FIGURE 4 – la classe ForoperatorTest

```
import java.util.ArrayList;
import org.junit.Test;
import FirstPackage.ForOperator;
import liste.ListeErreur;
import liste.*;
public class TestListe extends ListeErreur {

ListeErreur L = new ListeErreur();


@Test
public void testEstVide() {

 assertTrue(L.estVide());
}

@Test
public void testNonVide(){
L.ajouterAuDebut(1);
assertTrue("le nombre n'existe pas", L.contient(-1)==false);
```

```
}

@Test
public void testGetPremier() {
L.ajouterAuDebut(0);
ElementListe liste=L.getPremier();
int val =liste.getValeur();

assertTrue("c'est la premierre val ",val==0);
}

@Test
public void testAjouterAuDebut() {
    L.ajouterAuDebut(2);
ElementListe elment=L.getPremier();
int val=elment.getValeur();
assertTrue("La valeur est bien ajouter ", val==2); }


@Test
public void testAjouterALaFin() {
L.ajouterALaFin(10);
ElementListe lm=L.getDernierElement();
int x=lm.getValeur();
assertTrue("cette valeur est a la fin de la liste ",x==10);
}
public void testNonLafin(){
L.ajouterALaFin(3);
ElementListe Elm=L.getDernierElement();
int y=Elm.getValeur();
assertTrue("cette valeur n'est pas à la fin de la liste  ",y==9);
}

@Test
public void testGetDernierElement(){
L.ajouterALaFin(3);
L.ajouterAuDebut(7);
ElementListe ele=L.getDernierElement();
int valeur=ele.getValeur();
assertTrue("La valeur est au dernier element de la liste ",valeur==3);

}

@Test
public void testGetLongueur() {
 L.ajouterALaFin(1);
```

```
 int longeur=L.getLongueur();
 assertTrue("La longeur", longeur>-1);
}

@Test
public void testNeContientPas() {
 L.ajouterAuDebut(10);
 assertTrue("la nombre n'existe pas", liste.contient(-1)==false);

}

@Test
public void testGetDernierFalse() {
L.ajouterAuDebut(4);
L.ajouterALaFin(3);
ElementListe element=L.getDernierElement();
int valeur=element.getValeur();
boolean b=false;
if(valeur!=3)
b=true;
assertEquals(true, b);

}

@Test
public void testContient() {
ListeErreur liste=new ListeErreur();
 liste.ajouterAuDebut(5);
 assertTrue("la nombre éxiste", liste.contient(10)==true);
}

public void testNonLaFin() {
ListeErreur liste=new ListeErreur();
liste.ajouterALaFin(10);
ElementListe lm=liste.getDernierElement();
int x=lm.getValeur();
assertTrue("cette valeur n'est pas la derniere dans la liste",x==4);
}

@Test
public void testRetirerPremiereOccurrence() {
L.ajouterAuDebut(5);
L.ajouterALaFin(5);
ElementListe element=L.getPremier();
int val1=element.getValeur();
element=L.getDernierElement();
```

```
int val2=element.getValeur();
boolean tr=false;
if(val1==val2){
tr=true;
}
//assertEquals(val1,val2);
L.retirerPremiereOccurrence(5);
assertTrue("la premier occurrence est bien retiré",tr==true);

}

@Test
public void testConcatener() {
L.ajouterAuDebut(5);
L.ajouterALaFin(6);
ListeErreur liste2=new ListeErreur();
liste2.ajouterAuDebut(50);
liste2.ajouterALaFin(60);
int sizeApres=L.getLongueur();
L.concatener(liste2);
int sizeAvant=L.getLongueur();
assertTrue("La liste est bien concaterner ",sizeAvant>sizeApres);

}

}
```

Le rapport de bug en désignant pour chaque erreur la méthode concernée et le bug observé

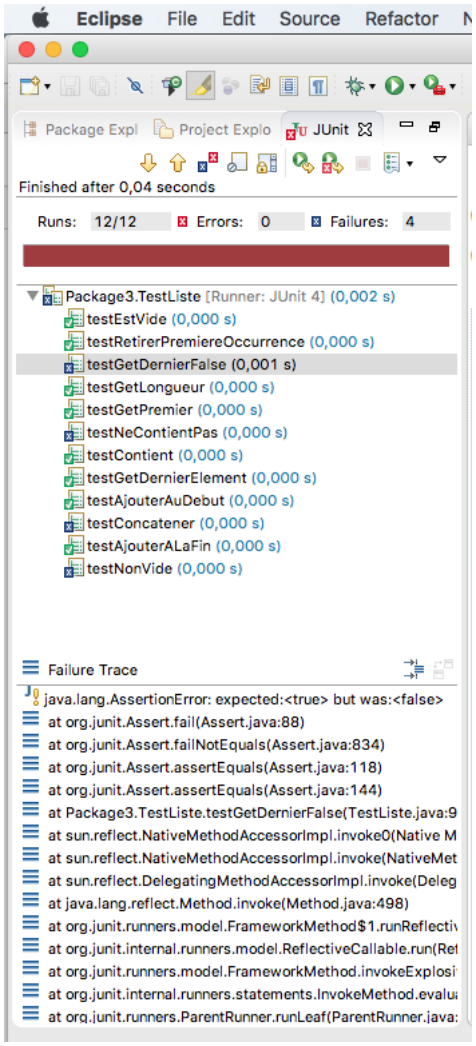

FIGURE 5 – La trace d'exécution Classe TestListe

	A	B	C	D	E	F
1	#Test	Fonction testée	Initialisation	Appel	Résultat attendu	Passage du test
2	1	ajouterAuDebut	L={1,2,3}	L.ajouterAuDebut(4)	L={4,1,2,3}	PASS
3	2	testEstVide	L={}	L.testNonVide()	true	Pass
4	3	testNonVide	L={}	L.testNonvide()	false	Fail
5	4	testGetPremier	L={1,2}	L.testGetPremier()	1	Pass
6	5	testAjouterAuDebut	L={1,2}	L.ajouterAuDebut(0);	L={0,1,2}	Pass
7	6	testAjouterALaFin	L={0,1}	L.ajouterALaFin(1);	true	Pass
8	7	testNonLafin	L={1,2,3}	L.ajouterALaFin(2);	false	Fail
9	8	testGetDernierElement	L={1,2,3}	L.getDernierElement()	3	Pass
10	9	testGetDernierFalse	L={1,2,3}	L.getDernierElement()	false	Fail
11	10	testGetLongueur	L={1,2,3}	L.getLongueur()	3	Pass
12	11	testNeContientPas	L={1,2,3}	L.contient(-1)	false	Fail
13	12	testContient	L={1,2,3}	L.contient(é)	true	Pass
14	13	testRetirerPremiereOccurrence	L={}	L.retirerPremiereOccurrer	true	Pass
15	14	testConcatener	L={1,2,3}-liste2={4,5,6}	L.concatener(liste2);	L={,1,2,3,4,5,6}	Pass

FIGURE 6 – bugReport