

Wprowadzenie do systemów rozproszonych

Sprawozdanie z ćwiczeń 1d + 1e

Testy jednostkowe z wykorzystaniem bibliotek
junit oraz mockito

Mateusz Jabłoński

1. Wstęp

Celem ćwiczenia jest dokończenie implementacji testów jednostkowych dla ćwiczeń 1d oraz 1e. W przypadku ćwiczenia 1d należy wykorzystać bibliotekę junit, natomiast w ćwiczeniu 1e powinna zostać użyta biblioteka mockito.

2. Rozwiązanie

- Pojęcia:
 - **Testy jednostkowe** - metoda testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów (jednostek) programu – np. metod lub obiektów w programowaniu obiektowym lub procedur w programowaniu proceduralnym.
 - **assertThat** - mechanizm asercji, który jako jeden z parametrów przyjmuje matcher.
 - **Matcher** – mechanizm odpowiadający za operacje dopasowywania.
 - **Mock** – atrapa rzeczywistego obiektu, która w kontrolowany sposób naśladuje jego zachowanie.
- Modyfikacje kodu
 - exercise1d
 - **CalculatorTest**

```
3 import static org.hamcrest.CoreMatchers.is;
4 import static org.junit.Assert.assertEquals;
5 import static org.junit.Assert.assertThat;
6
7 import org.junit.Before;
8 import org.junit.Test;
```

```

66 @Test
67 public void testMax_shouldReturnTheOnlyValue() {
68     // given
69     int[] values = {-5};
70
71     // when
72     int max = calculator.max(values);
73
74     // then
75     assertThat(values[0], is(max));
76 }
77 @Test
78 public void testMax_shouldReturnTheLargestOfAllValues() {
79     // given
80     int[] values = {2, -3, 5};
81
82     // when
83     int max = calculator.max(values);
84
85     // then
86     assertThat(5, is(max));
87 }
88 @Test(expected = IllegalArgumentException.class)
89 public void testMax_shouldRaiseAnExceptionForNullArgument() {
90     // given
91     int[] values = null;
92
93     // when
94     calculator.max(values);
95
96     // then
97     // empty
98 }
99 @Test(expected = IllegalArgumentException.class)
100 public void testMax_shouldRaiseAnExceptionForEmptyArgument() {
101     // given
102     int[] values = {};
103
104     // when
105     calculator.max(values);
106
107     // then
108     // empty
109 }
110 }

```

Dodanie testów dla metody max z klasy Calculator. W celu sprawdzenia poprawności zwracanego rezultatu wykorzystano metodę assertThat z biblioteki junit oraz metodę statyczną klasy Matcher is(). Należy pamiętać o ręcznym dodaniu biblioteki **org.hamcrest.CoreMatchers.is**, co mam miejsce w linii 3, ponieważ bez tego metoda is() nie będzie działać, a biblioteka nie jest automatycznie dodawana przez IDE.

- ArrayConverterTest

```
1 package wdsr.exercise1.conversions;
2
3 import static org.hamcrest.CoreMatchers.is;
4
5
6
7
8
9 public class ArrayConverterTest {
10     private ArrayConverter arrayConverter;
11
12     @Before
13     public void setup() {
14         arrayConverter = new ArrayConverter();
15     }
16
17     @Test
18     public void testConvertToInts_shouldReturnTheOnlyValue() {
19         // given
20         String[] values = {"234"};
21
22         // when
23         int[] convertedValues = arrayConverter.convertToInts(values);
24
25         // then
26         assertThat(234, is(convertedValues[0]));
27     }
28
29     @Test
30     public void testConvertToInts_shouldReturnTableOfIntegers() {
31         // given
32         String[] values = {"234", "1", "0", "-988776655", "02321"};
33         int[] expectedTab = {234, 1, 0, -988776655, 2321};
34
35         // when
36         int[] convertedValues = arrayConverter.convertToInts(values);
37
38         // then
39         assertThat(expectedTab, is(convertedValues));
40     }
41
42     @Test(expected = NullPointerException.class)
43     public void testConvertToInts_shouldRaiseAnExceptionForNullArgument() {
44         // given
45         String[] values = null;
46
47         // when
48         arrayConverter.convertToInts(values);
49
50         // then
51         // empty
52     }
53
54     @Test(expected = NumberFormatException.class)
55     public void testConvertToInts_shouldRaiseAnExceptionForNotNumberArgument() {
56         // given
57         String[] values = {"a", "cvfd"};
58
59         // when
60         arrayConverter.convertToInts(values);
61
62         // then
63         // empty
64     }
65 }
```

Implementacja testów dla klasy `ArrayConverter`. Metoda w linii 18 sprawdza czy w przypadku, jeżeli tablica przekazana jako parametr do metody `convertToInts` będzie zawierała tylko jedną wartość, to zostanie zwrócona tablica intów tylko z tą jedną wartością. Metoda w linii 30 sprawdza, czy jeżeli tablica stringów będzie zawierała więcej wartości, które będą spełniały takie kryteria jak: liczba ujemna czy liczba zaczynająca się od 0 to zostanie zwrócona tablica intów zawierająca odpowiednie wartości. Ostatnie dwa testy, których definicje znajdują się w liniach 42-64 sprawdzają, czy jeżeli metoda `convertToInts` jako argument otrzyma null lub tablice zawierającą inne znaki niż liczby, to zostaną zwrócone odpowiednie wyjątki.

- exercise1e
 - **CalculatorUtilDivisionTest**

```
16 public class CalculatorUtilDivisionTest {
17     private Calculator calculator;
18     private CalculatorUtil calcUtil;
19
20     @Before
21     public void init() {
22         calculator = Mockito.mock(Calculator.class);
23         calcUtil = new CalculatorUtil(calculator);
24     }
25
26     @Test
27     public void test16dividedBy4() {
28         // given
29         doReturn(4.0).when(calculator).divide(anyInt(), anyInt());
30
31         // when
32         String result = calcUtil.getDivisionText(16, 4);
33
34         // then
35         assertEquals("16 / 4 = 4.0", result);
36         verify(calculator).divide(anyInt(), anyInt()); // check if our calculator mock was really
37                                                         // invoked.
38     }
39
40     @Test(expected = IllegalArgumentException.class)
41     public void testDivisionByZero() {
42         // given
43         doThrow(new IllegalArgumentException()).when(calculator).divide(anyInt(), eq(0));
44
45         // when
46         calcUtil.getDivisionText(3, 0);
47
48         // then
49         // empty - exception expected
50     }
51 }
```

W każdej z tych 4 klas: `CalculatorUtilAdditionTest`, `CalculatorUtilDivisionTest`, `CalculatorUtilModuloTest`, `CalculatorUtilSubtractionTest` znajduje się blok kodu przedstawiony w liniach 17-24. W pierwszych dwóch liniach, następuje deklaracja zmiennych referencyjnych do obiektów typu `Calculator` i `CalculatorUtil`. Następnie, w celu przeprowadzenia testów jednostkowych klasy `CalculatorUtil`, konieczne jest utworzenie jej instancji, jednak do tego potrzebny jest obiekt typu `Calculator`, którego nie możemy utworzyć,

ponieważ Calculator jest interfejsem. Możemy sobie z tym poradzić, poprzez wykorzystanie mocka obiektu typu Calculator, który jest tworzony, a następnie przypisany do zmiennej referencyjnej w linii 22. Kiedy już posiadamy atrapę obiektu typu Calculator jesteśmy w stanie utworzyć obiekt typu CalculatorUtil w linii 23.

W klasie CalculatorUtilDivisionTest zaimplementowane są 2 testy metody getDivisionText klasy CalculatorUtil. Pierwszy z nich (linie 26-38) sprawdza czy podczas wywołania tejże funkcji zwracana jest poprawna wartość. W tym celu niezbędne jest skorzystanie z mocka klasy Calculator, co ma miejsce w linii 29, gdzie w metodzie doReturn określamy jaką wartość ma zostać zwrócona „**doReturn(4.0)**” w wyniku wywołania z dla obiektu Calculator „**when(calculator)**”, metody divide z 2 argumentami typu int „**divide(anyInt(), anyInt())**”. W linii 32 następuje właściwe wywołanie testowanej metody i przypisanie zwróconej wartości do zmiennej result, która jest następnie porównywana z oczekiwaną wartością w linii 35. Na koniec w linii 36 następuje weryfikacja, czy nasza atrapa została rzeczywiście wywołana. W drugim teście w liniach 40-50 na początku określamy, że oczekiwanym rezultatem testu jest zwrócenie wyjątku IllegalArgumentException. W linii 43 w przeciwieństwie do poprzedniego testu, zamiast metody doReturn używamy doThrow, ponieważ zamiast wartości, która ma zostać zwrócona, określamy typ wyjątku, który zostanie zwrócony, podczas wywołania metody divide z drugim argumentem równym „**eq(0)**”. Na koniec zostało już tylko wywołanie testowanej metody „**calcUtil.getDivisionText(3, 0)**”.

Testy jednostkowe dla metod getModuloText, getSubstractionText oraz getAdditionText są analogiczne jak w przypadku getDivisionText.

- Testy
 - exercise1d

Class wdsr.exercise1.logic.CalculatorTest

all > [wdsr.exercise1.logic](#) > CalculatorTest

8	0	0	0.007s
tests	failures	ignored	duration

100%
successful

Tests

Test	Duration	Result
testMax_shouldRaiseAnExceptionForEmptyArgument	0s	passed
testMax_shouldRaiseAnExceptionForNullArgument	0s	passed
testMax_shouldReturnTheLargestOfAllValues	0.001s	passed
testMax_shouldReturnTheOnlyValue	0.002s	passed
testMin_shouldRaiseAnExceptionForEmptyArgument	0.001s	passed
testMin_shouldRaiseAnExceptionForNullArgument	0.002s	passed
testMin_shouldReturnTheOnlyValue	0.001s	passed
testMin_shouldReturnTheSmallestOfAllValues	0s	passed

Generated by [Gradle 2.11](#) at 2017-07-06 16:11:36

Class wdsr.exercise1.conversions.ArrayConverterTest

all > [wdsr.exercise1.conversions](#) > ArrayConverterTest

4	0	0	0.008s
tests	failures	ignored	duration

100%
successful

Tests

Test	Duration	Result
testConvertToInts_shouldRaiseAnExceptionForNotNumberArgument	0s	passed
testConvertToInts_shouldRaiseAnExceptionForNullArgument	0.001s	passed
testConvertToInts_shouldReturnTableOfIntegers	0.007s	passed
testConvertToInts_shouldReturnTheOnlyValue	0s	passed

Generated by [Gradle 2.11](#) at 2017-07-06 17:20:56

- exercise1e

Test Summary

6	0	0	0.107s	100% successful
tests	failures	ignored	duration	

Packages **Classes**

Class	Tests	Failures	Ignored	Duration	Success rate
wdsr.exercise1.CalculatorUtilAdditionTest	1	0	0	0.102s	100%
wdsr.exercise1.CalculatorUtilDivisionTest	2	0	0	0.003s	100%
wdsr.exercise1.CalculatorUtilModuloTest	2	0	0	0.001s	100%
wdsr.exercise1.CalculatorUtilSubtractionTest	1	0	0	0.001s	100%

Generated by Gradle 2.11 at 2017-07-06 13:32:10

3. Wnioski

Testy jednostkowe umożliwiają odpowiednio szybkie wykrycie pojawiających się w kodzie błędów oraz szybka ich lokalizację i naprawienie. Najprostszym sposobem implementacji testów jednostkowych jest wykorzystanie asercji (zalecane użycie `assertThat`), jednak w niektórych przypadkach, np. gdy rzeczywisty obiekt jest trudno dostępny, powolny lub ma złożony i trudny do odtworzenia stan, lepszym rozwiązaniem okazuje się użycie mocków, czyli atrap rzeczywistych obiektów. Niestety, wykorzystanie ich obarczone jest ryzykiem, że nie będą wiernie oddawać zachowania rzeczywistego obiektu.

4. Literatura

1. <http://www.testowanie.net/poziomy-testow/testy-modulowe-unit-tests/>
2. <https://github.com/junit-team/junit4/wiki/Matchers-and-assertthat>
3. <http://site.mockito.org/>
4. [http://static.javadoc.io/org.mockito/mockito-core/2.8.47/org/mockito/Mockito.html#mock\(java.lang.Class\)](http://static.javadoc.io/org.mockito/mockito-core/2.8.47/org/mockito/Mockito.html#mock(java.lang.Class))