6 Testy jednostkowe

Pojęcia: testy jednostkowe, asercja, namiastki (mocks, stubs, dummies, fakes), JUnit, test fixture, gtest, googlemock, EasyMock.

6.1 Lektura

Przeczytaj artykuły o testach jednostkowych (dostępne na stronie prowadzącego):

M. Fowler – Mocks Aren't Stubs, Kolawa – Unit Testing Best Practices, J. O Coplien – Why Most Unit Testing is Waste oraz przeglądnij dokumentację najnowszej wersji JUnit (www.junit.org).

6.2 JUnit I

Klasę zawierającą przypadki testowe definiujemy jako klasę pochodną od junit.framework.TestCase:

- **konstruktor** powinien przyjmować jako parametr *String* z nazwa metody i przekazywać go klasie bazowej. Parametr ten decyduje, który spośród przypadków testowych dany obiekt będzie wykonywać;
- **przypadki testowe** definiujemy w publicznych metodach, których nazwy rozpoczynają się od słowa test np. *testMySqrt*. Mechanizm refleksji rozpoznaje takie metody automatycznie, dzięki czemu nie ma potrzeby ręcznego tworzenia zestawów testów;
- w metodach tych sprawdzamy czy działanie testowanej klasy jest zgodne z oczekiwaniami definiując asercje:
  - *assertTrue*([*String* wiadomosc], *bool* warunek);
  - *assertFalse*([*String* wiadomosc], *bool* warunek);
  - *assertEquals*([*String* wiadomosc], oczekiwanaWartosc, faktycznaWartosc, [tolerancja]);
  - *assert*(*Not*)*Null*([*String* wiadomosc], *Object* obiekt);
  - *assert*(*Not*)*Same*([*String* wiadomosc], *Object* oczekiwany, *Object* faktyczny); *fail*([*String* wiadomosc]) – ręczne poinformowanie, że dany przypadek testowy nie jest spełniony.
- Metody *setUp* i *tearDown* są wykonywane odpowiednio przed i po każdym teście jednostkowym i służą przygotowaniu środowiska dla testów np. nawiązaniu połączenia z bazą danych, konfiguracją systemu przed rozpoczęciem testów itp.
- Dzięki metodzie *suite* zwracającej *TestSuite* możemy zdecydować które testy mają być uruchamiane (domyślnie - wszystkie metodyz nazwą rozpoczynającą się od „test").

Przykład

```
import junit.framework.TestCase;

public class MathTest extends TestCase {
        Double a, b, wynik;
        public void setUp(){
                a = new Double(2.0);
                b = new Double(3.0);
                wynik = new Double(5.0);
        }
        public void testAdd(){
                assertEquals(wynik, a+b);
        }
        public void testMultiply(){
                assertFalse(wynik.equals(a*b));
        }
        public void tearDown(){
                a = b = wynik = null;
        }
}
```

JUnit 4

W wersji 4 JUnit nie jest konieczne definiowanie metod o określonych nazwach, w zamian wykorzystuje się adnotacje:

- @Test – dana metoda jest testem jednostkowym; adnotacja ma dwa opcjonalne parametry:
  - *expected* – deklaruje ze metoda musi rzucić wyjątek danej klasy np. @Test(expected=MyException.class)
  - *timeout* – określa w milisekundach limit czasu dla metody,
- @Before – metoda będzie wykonana przed każdym testem jednostkowym,
- @After – metoda będzie wykonana po każdym teście jednostkowym.

Asercje zdefiniowane są jako metody statyczne klasy org.junit.Assert. Zaleca się ich statyczny import, ponieważ stosowanie *assertTrue* zamiast *Assert.assertTrue* zwiększa czytelność kodu.

Przykład

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

public class MathTest2 {
        Double d2, d3, d5;
        @Before
        public void przygotuj() {
                d2 = new Double(2.0); d3 = new Double(3.0); d5 = new Double(5.0);
        }
        @Test
        public void dodawanie(){
                assertEquals(d5, d2+d3);
        }
        @Test(expected=java.lang.ArithmeticException.class)
        public void dzielenie(){
                assertEquals(Double.POSITIVE_INFINITY, (d2/new Double(0.0)));
                assertTrue(new Double(Double.NaN).equals(0/0));
        }
        @After
        public void sprzataj() {
                d2 = d3 = d5 = null;
        }
}
```

Zadaniem jest przetestować klasę **StringPair**. Obiekty **StringPair** będą używane jako klucze w obiektach typu map, dlatego muszą implementować metody: *equals* oraz *hashCode*. Te dwie metody muszą być spójne tzn. jeżeli *equals* zwróci *true* dla dwóch obiektów to *hashCode* musi zwrócić tą samą wartość dla obu obiektów. **StringPair** ma dwie właściwości: *right* i *left*. Ponieważ dla **StringPair** ważna jest wydajność, klasa ta przechowuje *hashCode* by nie musiał być obliczany za każdym razem. Wartość *hashCode* powinna być ustawiana na –1 za każdym razem gdy któraś z wartości *left* lub *right* uległa zmianie. Jest to znak, że wartość *hashCode* powinna być przeliczona przy następnej próbie pobrania tej wartości.

W pliku StringPairTest.java znajduje się implementacja testów jednostkowych dla klasy **StringPair** wykorzystująca framework JUnit 3. Zawiera ona kilka często popełnianych błędów. Zlokalizuj je, a następnie pogrupuj podobne błędy oraz opisz w czym tkwi problem.

6.3 JUnit II

Popraw implementację klasy **StringPairTest** tak aby testowała wszystkie istotne aspekty API wyspecyfikowane w poprzednim zadaniu.
W pliku StringPair.java znajduje się przykładowa implementacja klasy **StringPair**.

6.4 JUnit III

W poniższym teście zlokalizuj, opisz i usuń błędy.

```
import junit.framework.TestCase;
public class MathTest extends TestCase {
	public void testAdd(){
		double expected = 3;
		double result =0;
		for(int i=0; i<30; ++i){
			result += 0.1;
		}
		assertTrue(expected == result);
	}
}
```

6.5 JUnit IV

A) Dla interfejsu **IAnagramChecker** w oparciu o szablon JUnit napisz zestaw testów jednostkowych testujących klasę **AnagramChecker** implementującą ten interfejs.

```
/**
* Interfejs obiektu który sprawdza czy dane słowa są anagramami.
* Anagram jest słowem lub frazą, która powstała
* przez zmianę kolejności liter w oryginalnym słowie lub frazie.
* Zobacz kilka przykładów na http://www.wordsmith.org/anagram/hof.html
*/
public interface IAnagramChecker {
/** Sprawdza czy jedno slowo jest anagramem drugiego.
* Wszystkie niealfanumeryczne znaki są ignorowane.
* Wielkość liter nie ma znaczenia.
* @param word1 dowolny niepusty string różny od null.
* @param word2 dowolny niepusty string różny od null.
* @return true wtedy i tylko wtedy gdy word1 jest anagramem word2.
*/
  boolean isAnagram(String word1, String word2);
}
```

Czy można ten proces jeszcze bardziej zautomatyzować?

B) Zaimplementuj klasę **AnagramChecker**, przetestuj ją swoim zestawem przypadków testowych i usuń znalezione błędy. Następnie odpowiedz na pytania:

- Czy wcześniejsze napisanie testów pomogło Ci w implementacji klasy?
- Czy podczas implementacji przyszły Ci na myśl pomysły nowych testów? Jeśli tak, to jakie?
- Czy w implementacji były błędy które nie zostały wyłapane przez testy jednostkowe?
- Czy dodałeś odpowiednie przypadki testowe ujawniające znalezione błędy przed usunięciem błędu czy też najpierw usunąłeś błąd?