METODY PROGRAMOWANIA

Testy jednostkowe

8 grudnia 2017

Krzysztof Pawłowski
kpawlowski@pjwstk.edu.pl

PO CO NAM TESTY?

- weryfikacja poprawności
- sprawdzanie regresji
- specyfikacja
- dokumentacja
- wymuszanie dobrego stylu programowania

RODZAJE TESTOW

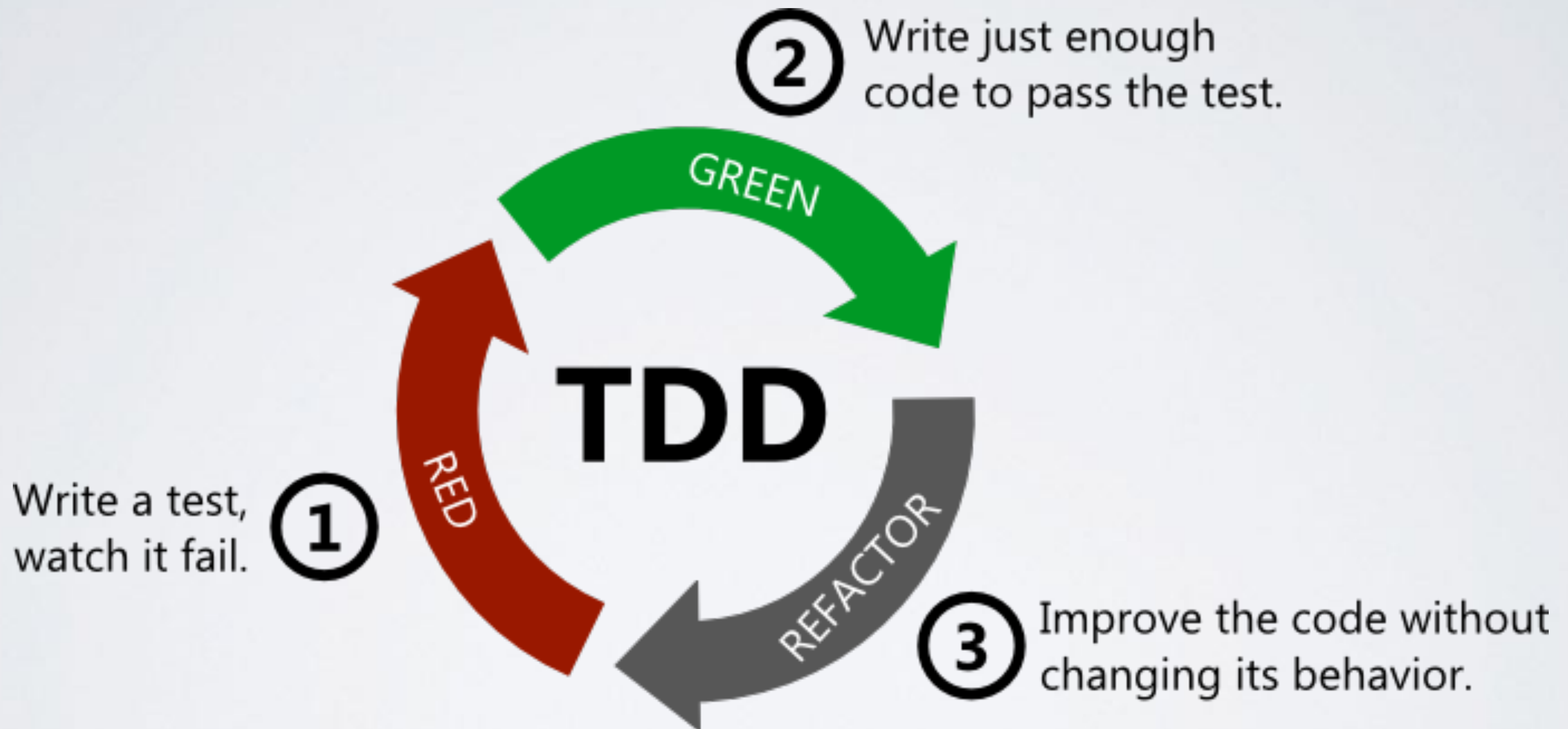
TESTY JEDNOSTKOWE

- testy na poziomie klas/metod
- używają "zaślepek" (ang. *mocks*)
- sprawdzają funkcjonalną poprawność i kompletność
- mają wysoki poziom szczegółowości

TDD

- Test Driven Development - metodologia wytwarzania oprogramowania oparta na testach
- Testy są pisane na początku
- Krótkie iteracje
- Nacisk na wymagania i przypadki testowe
- Łatwe “ulepszanie” kodu (*refactoring*)

TDD



ZASADY FIRST

- Robert C. Martin (Uncle Bob) w Clean Code pisze, że testy powinny spełniać następujące warunki:
  - Fast (szybkie)
  - Independent (niezależne)
  - Repeatable (powtarzalne)
  - Self-validating (samowalidujące)
  - Timely (napisane w odpowiednim czasie)

TESTY JEDNOSTKOWE - DOBRE PRAKTYKI

- Jeden test testuje jedną rzecz
- Konkretne komunikaty w razie niepowodzenia testu
- Logika w kodzie produkcyjnym nie powinna być tylko na potrzeby testu
- Powinny być proste w zrozumieniu
- Brak instrukcji warunkowych, pętli, obsługi wyjątków
- Jakość kodu testów powinna być równie wysoka, jeśli nie wyższa niż kodu produkcyjnego

Dilbert.com DilbertCartoonist@gmail.com

3-24-11 © 2011 Scott Adams, Inc./Dist. by UFS, Inc.

JUNIT

- framework do pisania przypadków testowych dla języka Java
- “open source”
- testy definiowane przy użyciu adnotacji (ang. *annotation*)

JUNIT - ADNOTACJE

- `@Test`
- `@BeforeClass`
- `@Before`
- `@AfterClass`
- `@After`

Adnotacja `@BeforeClass`

```
@BeforeClass
public static void setUpClass() {
  // kod wykonany przed pierwszą
  // metodą testową
}
```

Adnotacja `@Before`

```
@Before
public void setUp() {
  // kod wykonany przed każdą
  // metodą testową
}
```

Adnotacja `@AfterClass`

```
@AfterClass
public static void tearDownClass() {
  // kod wykonany po ostatniej
  // metodzie testowej
}
```

Adnotacja @After

```
@After
public void tearDown() {
  // kod wykonany po każdej
  // metodzie testowej
}
```

Adnotacja @Test

```
@Test
public void test1() {
  // kod pierwszego testu
}

@Test
public void test2() {
  // kod drugiego testu
}
```

Adnotacja `@Ignore`

```
@Ignore
@Test
public void test() {
  // ten test się nie wykona
}
```

TIMEOUT

- Jeśli test wykonuje się za długo możemy wymusić jego niepowodzenie.

```
@Test(timeout=1000)
public void testWithTimeOut() {
  // jeśli wykonywanie tego testu nie
  // zakończy się po 1 s. test
  // zakończy się niepowodzeniem
}
```

GLOBALNY TIMEOUT

- Możemy ustawić timeout dla wszystkich metod testowych w klasie

```
public class GlobalTimeOut {

  @Rule
  public Timeout globalTimeout = new Timeout(1000);

  @Test
  public void testWithTimeOut() {
    // jeśli wykonywanie tego testu nie
    // zakończy się po 1 s. test
    // zakończy się niepowodzeniem
  }
}
```

WYJĄTKI

- Możemy testować czy kod rzucił wyjątkiem

```
@Test(expected=RuntimeException.class)
public void testException() {
  // test powiedzie się tylko wtedy,
  // gdy kod rzuci wyjątkiem
  // RuntimeException
}
```

JUNIT - ASERCJE

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

assertTrue(“błąd - wyrażenie nie jest prawdziwe”, true)

assertFalse(“błąd - wyrażenie nie jest fałszywe”, false)
```

JUNIT - ASERCJE

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

assertEquals(“błąd - napisy nie są taki same”,
“napis”, “napis”);

assertSame(“błąd - liczby nie są równe”, 1, 1)

assertArrayEquals(“błąd - tablice nie są takie
same”, tab1, tab2)
```

STRUCTURE OF JUNIT TEST

```
@Test
public void shouldReturnAverage() {
  // Given
  List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);

  // When
  double res = avgCalculator.calc(list);

  // Then
  assertEquals("Wrong average", 3.0, res);
}
```

JAK NAZYWAĆ TESTY?

- MethodName_StateUnderTest_ExpectedBehavior
  - isAdult_AgeLessThan18_False
- MethodName_ExpectedBehavior_StateUnderTest
  - isAdult_False_AgeLessThan18
- test[Feature being tested]
  - testIsNotAnAdultIfAgeLessThan18
- Feature to be tested
  - IsNotAnAdultIfAgeLessThan18

JAK NAZYWAĆ TESTY?

- Should_ExpectedBehavior_When_StateUnderTest
  - Should_ThrowException_When_AgeLessThan18
- When_StateUnderTest_Expect_ExpectedBehavior
  - When_AgeLessThan18_Expect_isAdultAsFalse
- Given_Preconditions_When_StateUnderTest_Then_ExpectedBehavior
  - Given_UserIsAuthenticated_When_InvalidAccountNumberIsUsedToWithdrawMoney_Then_TransactionsWillFail

Źródło: https://dzone.com/articles/7-popular-unit-test-naming

MOCKITO

- Klasa powinna być testowana bez zależności
- Zależności do innych klas zastępujemy zaślepkami (ang. *mock*)
- Mockito - framework to tworzenia zaślepek

DLACZEGO ZAŚLEPKI?

- Chcemy testować tylko naszą klasę
- Zależność może jeszcze nie być zaimplementowana

JAK UTWORZYĆ ZAŚLEPKĘ?

- Adnotacja pola klasy

```
@Mock
private List<Integer> mockedList;
```

- Deklaracja zmiennej

```
List<Integer> mockedList =
mock(List.class);
```

INTERAKCJE Z ZAŚLEPKĄ

```
mockedList.get(0);
mockedList.clear();

verify(mockedList).get(0);
verify(mockedList).clear();
```

KONFIGUROWANIE ZAŚLEPKI

- Domyślnie metody wołane na zaślepce zwracają null

- Możemy przypisać wartość używając poniższej konstrukcji:

```
when(mockList.get(0)).thenReturn(1);
```

KONFIGUROWANIE ZAŚLEPKI

- Możemy przypisać wartości zwracane przy kolejnych wywołaniach tej samej metody

```
when(mockList.get(0)).thenReturn(1)
  .thenReturn(2);
```

KONFIGUROWANIE ZAŚLEPKI

- Możemy wywołać rzucenie wyjątku przy wołaniu metody:

```
when(mockList.get(0)).thenThrow(new
RuntimeException());
```

DOPASOWYWANIE ARGUMENTÓW

```
when(mockList.get(anyInt()))
   .thenReturn(1);
```

WERYFIKACJA WYWOŁANIA ZAŚLEPKI

```
verify(mockList).get(anyInt());

verify(mockList).get(0);
```

WERYFIKACJA LICZBY WYWOŁAŃ

```
verify(mockList, times(2))
  .get(anyInt());

verify(mockList, atLeast(2))
  .get(anyInt());

verify(mockList, atMost(2))
  .get(anyInt());
```

- domyślna wartość to *times(1)*

WERYFIKACJA KOLEJNOŚCI WYWOŁAŃ

- kolejność weryfikacji wywołań ma znaczenie

```
verify(mockList).add(“first el”);
verify(mockList).add(“second el”);
```

PRZECHWYTYWANIE ARGUMENTÓW

```
ArgumentCaptor<Person> arg =
ArgumentCaptor.forClass(Person.class);

verify(mock).doSomething(arg.capture());

Person person = arg.getValue();
assertEquals(“Joe”, person.getName());
```

ASSERTJ

- Biblioteka służąca do definiowania asercji
- Asercje przypominają język naturalny

ASSERTJ - PRZYKŁADY

- Zwykłe asercje z JUnit:

```
assertNotNull(list);
assertEquals(3, list.size());
```

- mogą być zastąpione przez:

```
assertThat(list).isNotNull().hasSize(3);
```

ASSERTJ - WIĘCEJ PRZYKŁADÓW

- `assertThat(birthday).isToday();`
- `assertThat(aDate).isEqualTo(“2013-08-08”);`
- `assertThat(aMap).containsKeys(key1, key2, key3);`
- Więcej przykładów: http://joel-costigliola.github.io/assertj/assertj-core-features-highlight.html