Testowanie: Junit + Mockito

Przemysław Grzesiowski
15 września 2017

> *Programming is like sex: one mistake and you're providing support for a lifetime.*
> *(Michael Sinz)*

> *A computer lets you make more mistakes faster than any invention in human history–with the possible exceptions of handguns and tequila.*
> *(Mitch Radcliffe)*

Agenda

- wstęp
- Część I - jak ugryźć jUnit
  - dodawanie wielu liczb – przykład testów
  - struktura typowego projektu
  - dobre praktyki
  - więcej testów dodawania – ćwiczenia
  - adnotacje @After, @Before, @AfterClass, @BeforeClass
  - jak uruchamiać testy (IJ vs maven)
  - weryfikacja wyników (asercje w jUnit, hamcrest, assertJ)
  - ćwiczymy używanie kropki (assertJ)

Agenda

- Część II - Jak wypić mockito
  - inicjacja mocka
  - stubbing
  - weryfikacja
  - ćwiczenia
- po co to wszystko?
- podsumowanie

Tworzenie oprogramowania bez błędów jest trudne, a wręcz niemożliwe, ale ilość tych błędów można znacząco ograniczyć przez poprawne testowanie.

Pierwsza krew - testujemy dodawanie

1. Stwórz nowy projekt (archetyp maven-archetype-quickstart)

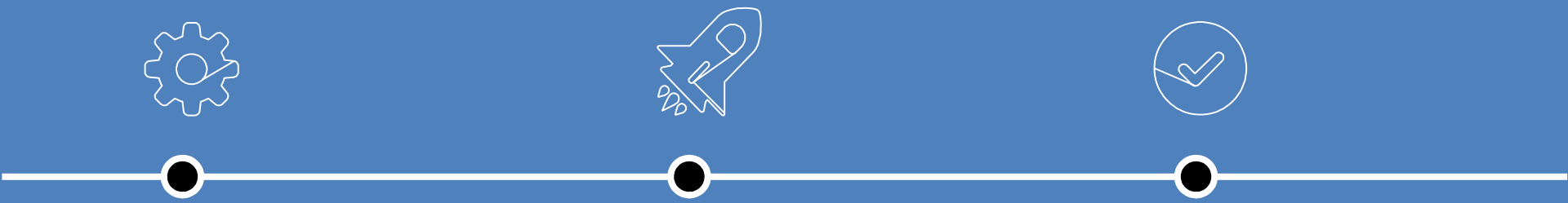
2. Upewnij się, że wygenerowany pom ma zależność jUnit

Pierwsza krew - testujemy dodawanie

3. Utwórz w kodzie produkcyjnym metodę dodającą dowolna liczbę argumentów, np.:

```
public Integer dodaj(Integer ... args) {
   Integer sum = 0;
   for (Integer i : args) {
       sum += i;
   }
   return sum;
}
```

Testy jednostkowe
Struktura (zasada 3 * A)



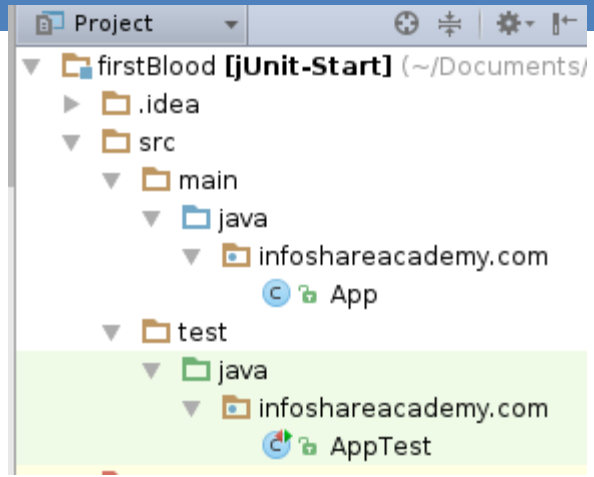

```
App app = new App();

                                Integer wynik = app.dodaj(1, 3);

                                                                assertTrue(wynik == 4);
```

Pierwsza krew - testujemy dodawanie

```
@Test                                                   // adnotacja @Test !!
public void dodawanie_dwaArgumenty() {
        // inicjacja obiektu(ów), przygotowanie warunków początkowych (test fixture)
                App app = new App();
        // wywołanie metody którą testujemy (method under test)
                Integer wynik = app.dodaj(1, 3);
        // weryfikacja wyników (asercja)
                assertTrue(wynik == 4);
}
```

Struktura projektu

- *src/main*
- *scr/test*



Testy jednostkowe

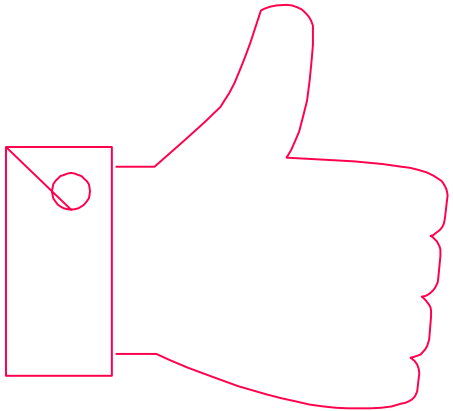
Dobre praktyki

- Testujemy nie tylko przypadki optymistyczne, ale i brzegowe i wyjątkowe
- Dbamy o czytelność i zwięzłość (clean code)
- Pojedyncza odpowiedzialność
- Nie umieszczamy w testach warunków i pętli

Testy jednostkowe

Zasady
(F.I.R.S.T Principles of Unit Testing)

- Fast
- Independent (Isolated)
- Repeatable
- Self-checking
- Timely



Testy jednostkowe

Zasady
(F.I.R.S.T Principles of Unit Testing)

- Fast
  run tests quickly (you'll be running them all the time)
- Independent (Isolated)
  no tests depend on others, so can run any subset in any order
- Repeatable
  run N times, get same result
- Self-checking
  test can automatically detect if passed (no human checking of output)
- Timely
  written about the same time as code under test (with TDD, written first!)

Kolejne testy - czemu nie?

IntelliJ:
- **Ctrl + Shift + F10 – uruchom test**
- **Ctrl + F5 – powtórz ostatnie testy**

dodaj nowe przypadki testowe:

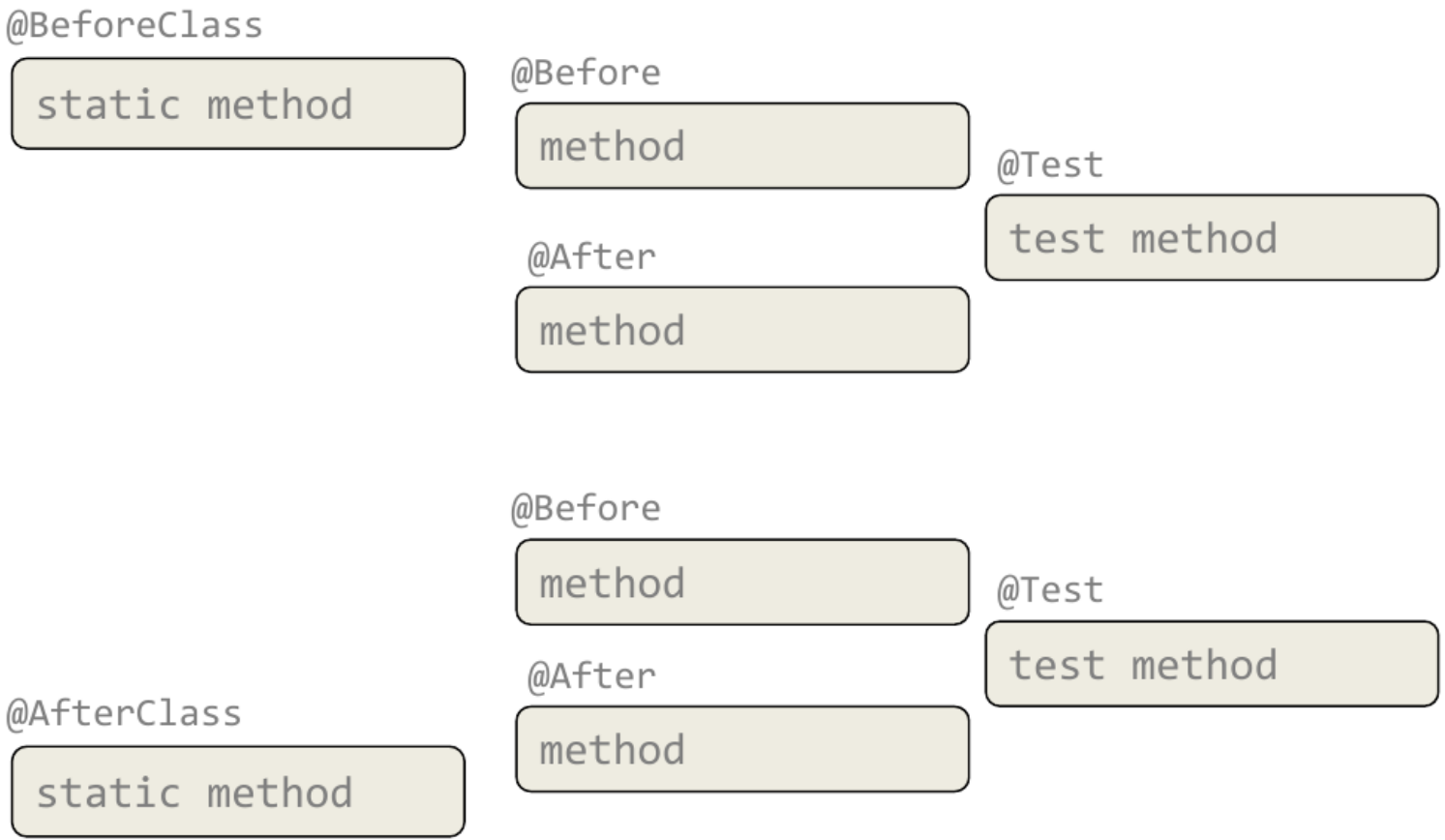
1. dodawanie 3 liczb
2. dodawanie gdzie jedna z liczb jest ujemna
3. dodawanie bardzo dużych liczb (gdzie suma wykracza poza zakres integera)
4. jeden z argumentów wejściowych to null

A co gdy test kończy się porażką?

1. Zmodyfikuj kod produkcyjny tak, żeby produkował niepoprawny wynik (np. odejmowanie zamiast dodawania)
2. Uruchom wszystkie testy dodawania które napisałeś
3. Zwróć uwagę jaką informacje dostajesz gdy test nie przeszedł
4. Dodaj własną wiadomość (message) do istniejącej asercji
5. Zamiast “assertTrue” użyj asercji “assertEquals”, powtórz pkt. 3.

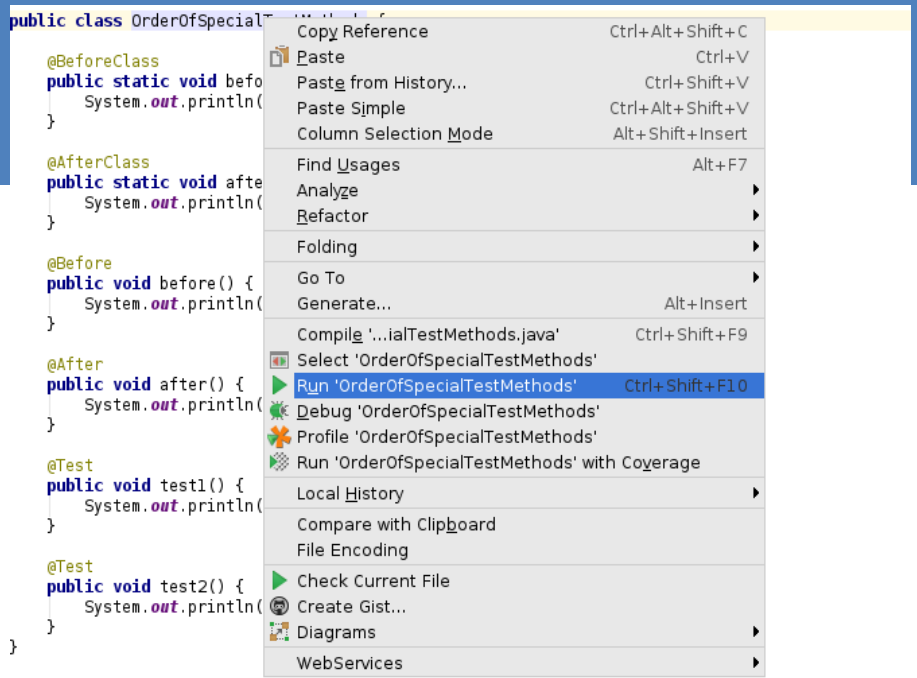
Junit - @After @Before @AfterClass @BeforeClass

1. Pobierz kod z github: https://github.com/infoshareacademy/jjdd2-materialy-junit-mockito/
2. Otwórz projekt firstBlood, klasę *OrderOfSpecialTestMethods.java*
3. Wykonaj ćwiczenia oznaczone jako “todo A” zgodnie z opisem w klasie.



Start testów w IJ

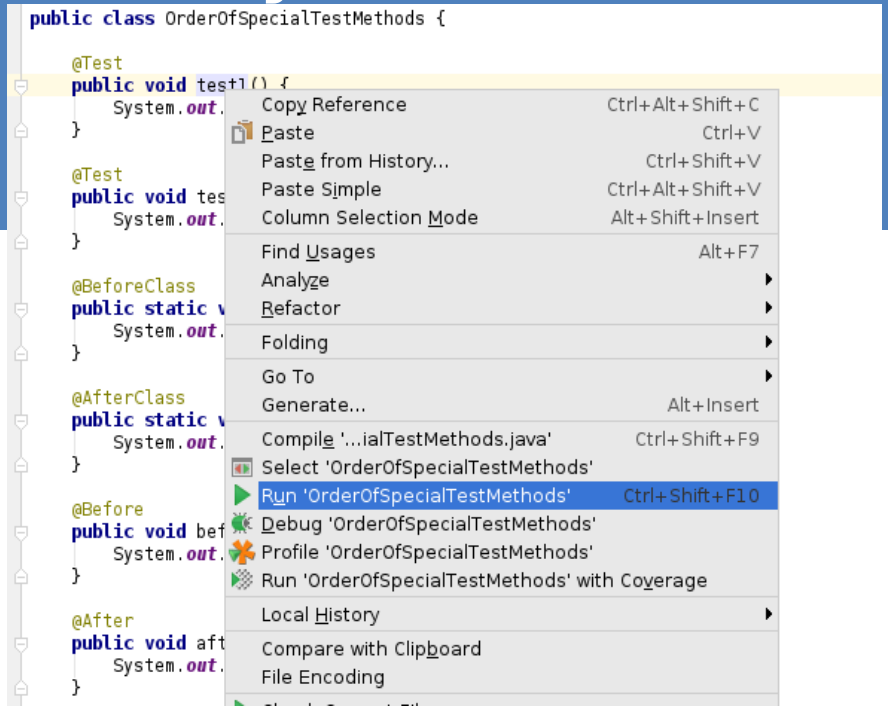
Prawy click na nazwie klasy → Run 'your Class'



Start testów w IJ

Prawy click na nazwie metody

→ Run 'your test method'



Start testów – maven
(surefire plugin)

> mvn test -Dtest=OrderOfSpecialTestMethods

> mvn test -Dtest=OrderOfSpecialTestMethods#test2

Junit - @After @Before @AfterClass @BeforeClass

4. Wykonaj ćwiczenia oznaczone jako “todo B” zgodnie z opisem w klasie.

Asercje z pakietu org.junit.Assert

http://junit.sourceforge.net/javadoc/org/junit/Assert.html

Przyjrzyj się sygnaturom następujących metod:

assertFalse, assertTrue
assertEquals(Object expected, Object actual)
assertNotNull, assertNull
assertArrayEquals
fail
assertThat(...) ← punkt “zaczepny” dla hamcresta

Hamcrest – bardziej wyszukane asercje

public static <T> void **assertThat**(T actual, Matcher<? super T> matcher)

“Matcher” to interfejs, otwórz link:
http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/Matcher.html

Hamcrest – większa czytelność asercji

Junit Assert
assertEquals(expected, actual);

```
assertEquals(4, wynik);
```

hamcrest
assertThat(actual, is(expected));

```
assertThat(wynik, is(4));
```

Hamcrest – precyzyjny opis błędu

Junit Assert

```
assertTrue(expected.contains(actual));
```

```
java.lang.AssertionError at ...
```

hamcrest

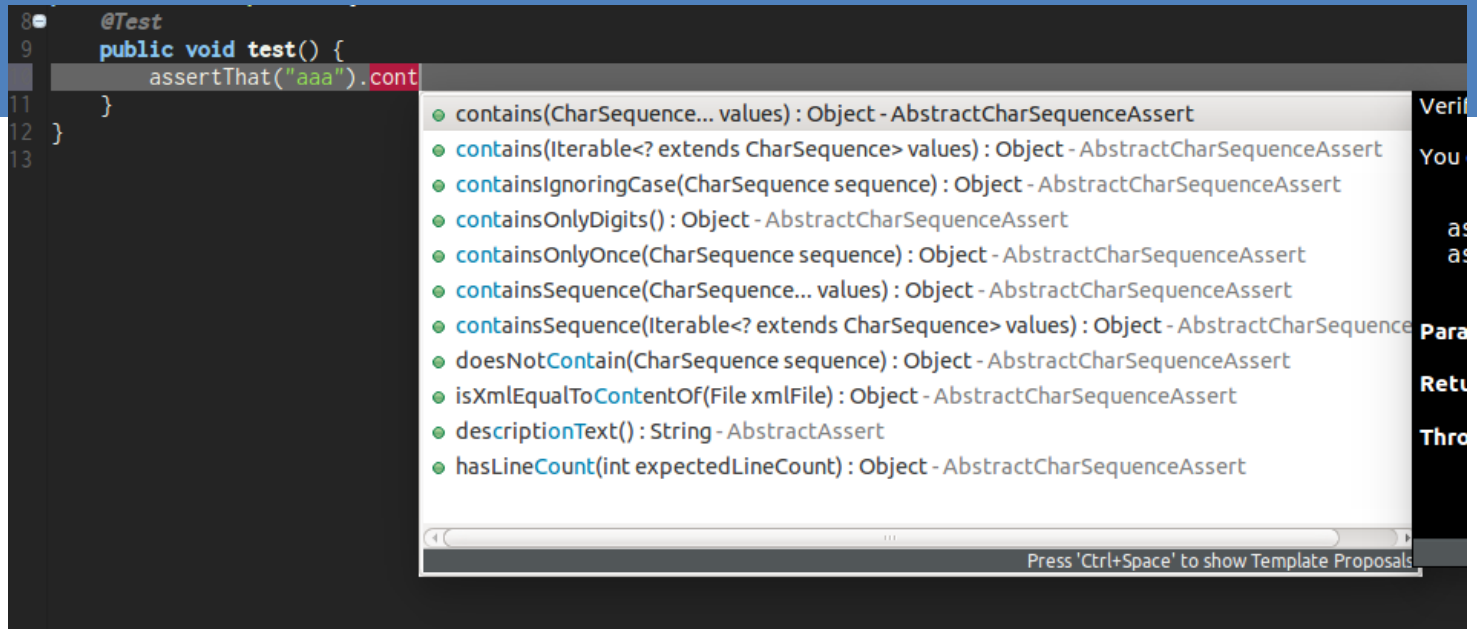
```
assertThat(actual, containsString(expected));
```

```
java.lang.AssertionError:
Expected: a string containing "info"
got: "share"
```

AssertJ – cały świat asercji kryje się za kropką

assertThat(obiektDoZbadania).
// code completion -> dobiera asercje obserwując typ obiektu

AssertJ – fluent assertions dla leniwych programistów



AssertJ – bogate API dla różnych typów danych - ćwiczymy

1. Otwórz projekt assertJExcercises
2. Zapoznaj się z kodem produkcyjnym (klasa NamesStats), co Twoim zdaniem robi ten kod?
3. Wykonaj ćwiczenia oznaczone jako “todo C” zgodnie z opisem w klasie NamesStatsTest.

MOCK

EASYMOCK

Wyobraź sobie, że Twój ulubiony portal internetowy wp.pl / onet.pl * wyświetla wykrzyknik w prawym górnym rogu jeśli jest duże ryzyko gwałtownych zjawisk pogodowych w kraju.

* - niepotrzebne skreślić

Wyobraź sobie, że Twój ulubiony portal internetowy wp.pl / onet.pl * wyświetla wykrzyknik w prawym górnym rogu jeśli jest duże ryzyko gwałtownych zjawisk pogodowych w kraju.
Dane pobierane są z zewnętrznego serwisu pogodowego.

* - niepotrzebne skreślić

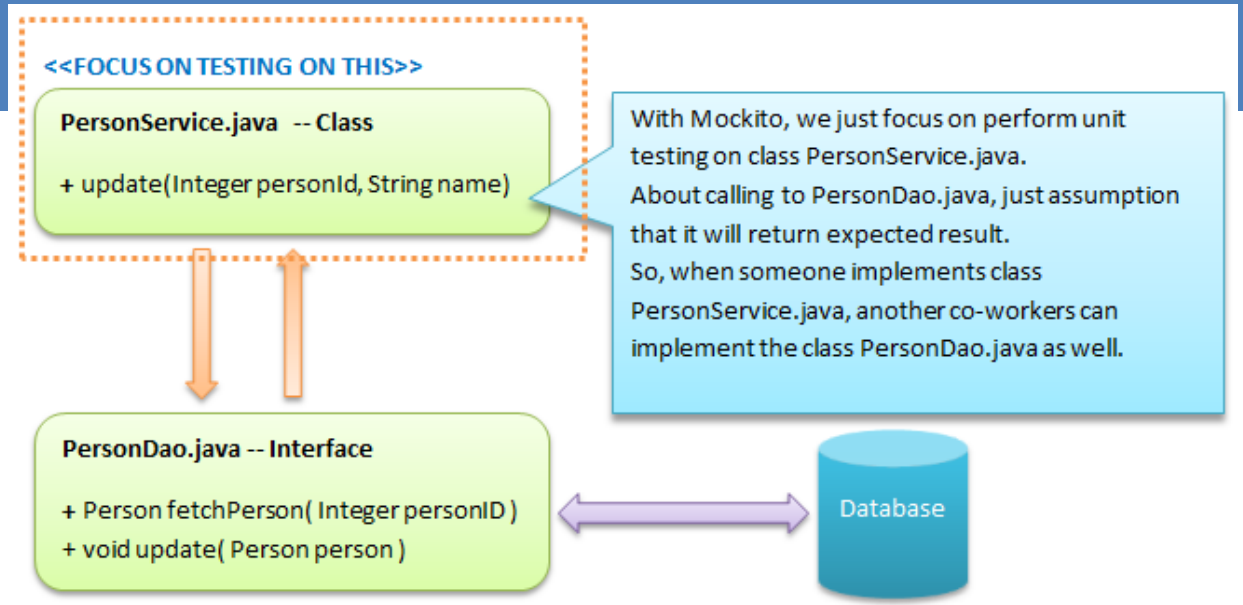
Wyobraź sobie, że Twój ulubiony portal internetowy wp.pl / onet.pl * wyświetla wykrzyknik w prawym górnym rogu jeśli jest duże ryzyko gwałtownych zjawisk pogodowych w kraju.
Dane pobierane są z zewnętrznego serwisu pogodowego.
Jak to przetestujesz?

* - niepotrzebne skreślić

Twój sklep internetowy potrzebuje paypal, ale gdy transakcja się nie powiedzie 2 razy z rzędu (lub gdy paypal ma przerwę konserwacyjną) chcesz dasz użytkownikowi opcję płatności przelewem.
Jak to przetestujesz?

W “urodziny miesiąca” sklep internetowy “W Siódmym Niebie” udziela rabatu 10 % na wszyskie artykuły z działu “bielizna dla niej” Jak to przetestujesz?

Odpowiedź brzmi: MOCK IT !



Jak to się pije?

- inicjacja obiektu
- stubbing (określenie zachowania)
- weryfikacja

Inicjacja

Musimy poprosić mockito, żeby “zaopiekowało się naszym obiektem”. W ten sposób powstaje fake(mock) - obiekt udający oryginał, którego zachowanie możemy sami kształtować/szpiegować itp.

```
    List mockedList = mock(List.class);

//lub:  ( @RunWith(MockitoJUnitRunner.class) )
    @Mock
    List mockedList;
```

STUBBING

"When the x method is called then return y", np.

“Jeżeli zapytam jaka jest pogoda w Sopocie, odpowiedz, że jest 21 st C”

```
when(weatherService.getTemperature(“Sopot”))
.thenReturn(21);
```

STUBBING

“Jeżeli zapytam jaka jest pogoda w dowolnym mieście, odpowiedz, że jest 30 st C”

```
when(weatherService.getTemperature(anyString())
.thenReturn(30);
```

VERIFY

Weryfikacja polega na sprawdzeniu co działo się z naszym “podstawionym” obiektem.

```
verify(emailMock)
.sendEmail(eq(“user1@wp.pl”), anyString());

verify(emailMock, times(2))
.sendEmail(anyString(), anyString());
```

4. Verifying exact number of invocations / at least x / never

```
//using mock
mockedList.add("once");

mockedList.add("twice");
mockedList.add("twice");

mockedList.add("three times");
mockedList.add("three times");
mockedList.add("three times");

//following two verifications work exactly the same - times(1) is used by default
verify(mockedList).add("once");
verify(mockedList, times(1)).add("once");

//exact number of invocations verification
verify(mockedList, times(2)).add("twice");
verify(mockedList, times(3)).add("three times");

//verification using never(). never() is an alias to times(0)
verify(mockedList, never()).add("never happened");

//verification using atLeast()/atMost()
verify(mockedList, atLeastOnce()).add("three times");
verify(mockedList, atLeast(2)).add("three times");
verify(mockedList, atMost(5)).add("three times");
```

ćwiczymy

Jak wypić mockito

1. Otwórz projekt mockitoTest

2. Zapoznaj się z kodem produkcyjnym. Wykonaj ćwiczenia z AppTest.java.

3.Zerknij na MockAndUnspecifiedMethod

Unspecified method calls

Unspecified method calls return "empty" values: (przykłady w klasie MockAndUnspecifiedMethod.java)

- null for objects
- 0 for numbers
- false for boolean
- empty collections for collections

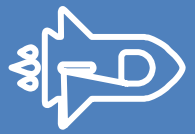
Co zyskujemy?
Zalety kodu przetestowanego

- Masz (większą) pewność, że działa
- Zadowolenie klientów
- Łatwość zmian
- Szybszy “debugging”, błyskawiczna odpowiedź o stanie kodu
- “Samopisząca” się dokumentacja
- Możesz polegać na członkach zespołu
- Oszczedzasz czas nie musząc wykonywać tak wiele testów manualnych
- Czujesz się lepiej, śpisz spokojniej

“Good code is its own best documentation.”

(Steve McConnell)

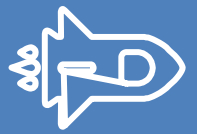
DLACZEGO TESTOWAĆ?



1. nigdy nie ufaj, zawsze testuj

```
Date date = new Date(2017,4,26);

// when
int suma= mat.dodaj(2147483647,1);
// then
assertThat(suma, is(0));
```

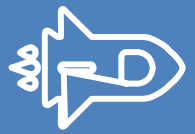
DLACZEGO TESTOWAĆ?



2. zmiany we frameworku

3. zmiany w kodzie kolegi z zespołu

DLACZEGO TESTOWAĆ?



4. Upss.. omsknął mi się palec

```
public class Mat
{

    int add(Integer a, Integer b) {
        if(b == null) return a;
        return -a + b;
    }

}
```

Jakieś wady?

Dlaczego nie robimy testów jednostkowych?

Czas developmentu
Początkowe etapy projektu wymagają dodatkowej pracy na przygotowanie testów jednostkowych

Czas utrzymania
Przygotowane zestawy testów trzeba z czasem utrzymywać by nadal przynosiły korzyści

- brak odpowiedniej wiedzy programistów
- strach kierownictwa przed wyższymi kosztami
- przekonanie, że testerzy wyłapią wszystkie błędy

> Without unit tests, you're not refactoring.
> You're just changing shit

CONS:

- Tests become part of the maintenance overhead of a project, especially wrongly written ones, fragile ones;
- Mock != real
- More time consuming (?)
- Ilość kodu testów jest zazwyczaj dużo większa niż kodu produkcyjnego, staraj się aby kod testów był również sensowny

http://rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf

INNE UWAGI

- Mockuj z głową
- nie zapominaj o testach integracyjnych
- dobrze napisane testy to świetna dokumentacja kodu
- podczas code review zacznij od czytania testów
- nigdy nie przetestujesz wszystkiego

LINKI

- https://github.com/mockito/mockito/wiki/Mockito-Popularity-and-User-Base
- http://joel-costigliola.github.io/assertj/
- http://site.mockito.org/
- http://www.vogella.com/tutorials/Mockito/article.html
- http://junit.org

Koniec