



Testowanie oprogramowania - TDD

Łukasz Bednarski



Testy jednostkowe

5 minutowe przypomnienie...

Testy jednostkowe - Definicja



„Test jednostkowy to fragment kodu (zazwyczaj metody), który wywołuje inny fragment kodu i sprawdza poprawność pewnych założeń. Jeśli założenia okażą się błędne, test jednostkowy nie powiedzie się. „Jednostka” jest metodą lub funkcją.”

Testy jednostkowe. świat niezawodnych aplikacji

Roy Osherove



Właściwości testów jednostkowych

Szybkość: Testy jednostkowe powinny uruchamiać się szybko, tak aby można było je uruchamiać bardzo często. Szybko oznacza dużo poniżej 1 s.

Izolacja: Testy powinny być od siebie odizolowane i niezależne od siebie. Test nie powinien uruchamiać innego testu.

Powtarzalność: Testy powinny być powtarzalne na każdym środowisku. Testy nie mogą mieć stanów początkowych, ani zasobów do wyczyszczenia. Oznacza to także brak zależności w stosunku do zasobów zewnętrznych (baza danych, system plików, itd.)

Zgodność: Testy powinny dawać ten sam rezultat za każdym uruchomieniem.

Atomiczność: Atomiczne testy oznaczają, że test jednoznacznie jest zielony lub czerwony. Nie ma przypadku, gdy test jest zielony, ale testowana logika nie działa prawidłowo z jakichś powodów. Nie ma testów częściowo poprawnych.

Testy Integracyjne - Definicja



„Testowanie integracyjne jest testowaniem jednostki pracy bez pełnej kontroli nad jej wszystkimi elementami oraz z wykorzystaniem jednej lub kilku jej rzeczywistych zależności, takich jak czas, sieć, baza danych, wątki, generatory liczb losowych itp.”

Testy jednostkowe. świat niezawodnych aplikacji

Roy Osherove

Testy jednostkowe i testy integracyjne



Zagadnienie	Test jednostkowy	Test integracyjny
Zależności	Testowany jednostkowy element (klasa, metoda) w izolacji.	Testowana więcej niż jedna wewnętrzna lub zewnętrzna zależność.
Punkt awarii (<i>failure point</i>)	Tylko jeden potencjalny punkt awarii	Wiele potencjalnych punktów awarii.
Szybkość działania	Bardzo szybko, dużo poniżej 1 sekundy.	Może trwać długo, ze względu na czasochłonne operacje np. dostęp do bazy danych, I/O, operacje na sesji.
Konfiguracja	Test musi działać na każdej maszynie bez dodatkowej konfiguracji.	Test może być zależny od konfiguracji, np. machine.config (login/hasło) do bazy danych.



Testy jednostkowe - Przykład



Źródło: <http://www.rpmgo.com/automotive-articles/car-parts.html>



Test integracyjny - Przykład



Źródło: <http://indianautosblog.com/2008/12/iihs-suzuki-sx4-crash-test>



JUnit





„JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks which is collectively known as xUnit that originated with SUnit.”

<https://en.wikipedia.org/wiki/JUnit>



@Test - występuje przed każdą metodą testową i oznacza pojedynczy test jednostkowy.

@BeforeClass - występuje przed metodą, która jest wykonywana przed uruchomieniem testów. Można w niej utworzyć obiekt klasy testowej, zainicjować jej atrybuty itp.

@AfterClass - oznacza metodę, która ma być wywołana po zakończeniu wszystkich testów. Można w niej np. zamknąć połączenie z bazą danych, zapisać plik itp.

@Before - metoda oznaczona tą adnotacją, będzie wywoływana przed wykonaniem każdej metody testowej.

@After - oznacza metodę, która ma być wywoływana po każdej metodzie testowej.

@Ignore - oznacza test który będzie ignorowany



JUnit – asercje

Asercje są to metody statyczne pozwalające na porównanie wyników działania określonej funkcji z oczekiwanaą wartością.

Najpopularniejsze asercje to:

Asercja	Opis
assertEquals(expected, actual)	Porównanie dwóch wartości - otrzymanej po działaniu i oczekiwanej
assertNotNull(object)	Sprawdzenie czy obiekt został zainicjalizowany
assertNull(object)	Sprawdzenie czy obiekt nie został zainicjalizowany
assertTrue(value)	Sprawdzenie czy pole lub metoda zwraca wartość true w przypadku typu boolean
assertFalse (value)	Sprawdzenie czy pole lub metoda zwraca wartość false w przypadku typu boolean
assertArrayEquals(expected, resultArray)	Porównanie dwóch tablic czy posiadają tą samą zawartość



Testy jednostkowe - przykład

1. Nowa klasa testowa
2. Testowany obiekt
3. Metoda poprzedzona adnotacją @Before będzie wywołana przed każdym testem
4. Metoda testująca dodawanie
5. Assercja sprawdzająca czy otrzymana wartość po operacji dodawania jest wartością oczekiwana

```
public class CalculatorTest {  
    private Calculator calculator;  
  
    @Before  
    public void setup(){  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void testAdding() {  
        int result = calculator.add(3, 5);  
        assertEquals(8, result);  
    }  
}
```



Testy jednostkowe - wzkazówka

Popularnym „stylem” pisania testów jest podejście Given - When -Then. Dzięki temu testy są usystematyzowane oraz bardziej czytelne. Bardzo wiele osób dodaje także komentarze oddzielające opisywane sekcje.

- Given określa sekcję w której tworzymy początkowe założenia. Ustawiamy stan, tworzymy instancję obiektów, ładujemy konfigurację itp. na potrzebny do testów.
- When określa sekcję w której wykonujemy akcję którą chcemy testować.
- Then określa sekcję w której wykonujemy sprawdzenia czy aplikacja zachowała się zgodnie z oczekiwaniami. Najczęściej poprzez wykorzystanie asercji lub interakcji z mockami.

```
@Test  
public void testAdding() {  
    //Given  
    Calculator calculator = new Calculator();  
  
    //When  
    int result = calculator.add(3, 5);  
  
    //Then  
    Assert.assertEquals(8, result);  
}
```



Maven - dodanie zależności

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```



Testy jednostkowe - zadanie

Pobierz projekt o nazwie `UnitTest` a następnie zainportuj go w swoim IDE. Przejdź do pakietu `pl.sda.tests.calculator` i otwórz klasę `Calculator`. Klasa ta zawiera 4 metody wykonujące podstawowe działania matematyczne. Aby zweryfikować poprawność ich implementacji, utwórz klasę testową i uzupełnij brakujące testy.

Sprawdź czy zaimplementowane w nim metody zwracają odpowiednie wartości. Do każdej z metod dodaj przynajmniej 2-3 asercje.

Aby zainportować poprawnie projekt do Eclipse wybierz:

1. Kliknij w górnym menu `File` a następnie `Import`
2. Wybierz `Existing Maven Project`
3. Kliknij `Browse` i wstaż katalog z projektem.



Testy parametryzowane



Testy parametryzowane

W jaki sposób przetestować określoną funkcjonalność na podstawie wielu różnych danych wejściowych? Np:

1. Danych poprawnych
2. Danych błędnych
3. Skrajne przypadki
4. Itd...

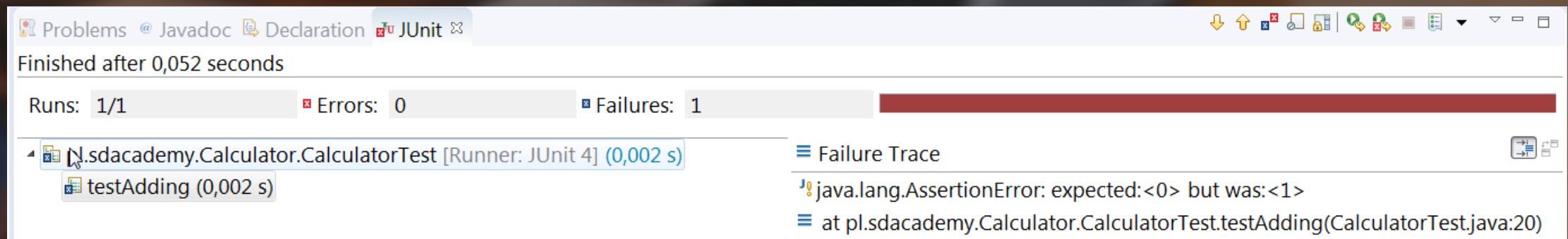
Jakie błędy i problemy może powodować takie podejście?

```
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Before  
    public void setup(){  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void testAdding() {  
        Assert.assertEquals(8, calculator.add(3, 5));  
        Assert.assertEquals(0, calculator.add(1, -1));  
        Assert.assertEquals(0, calculator.add(0, 1));  
        Assert.assertEquals(0, calculator.add(5, -5));  
        Assert.assertEquals(-5, calculator.add(-10, 5));  
        Assert.assertEquals(1, calculator.add(3, -2));  
    }  
}
```



Testy parametryzowane

1. Bardzo wiele asercji w tej samej metodzie testującej.
2. Możemy wydzielić asercje dla każdego przypadku ale wówczas powstanie nam duża nadmiarowość kodu w postaci wielu testów.
3. W razie niepowodzenia nie wiemy w którym miejscu test się nie powiodł.



Dlatego też powstały testy parametryzowane.



Testy parametryzowane

1. Parametryzowane testy jednostkowe służą do testowania tego samego kodu w różnych warunkach.
2. Dzięki parametryzowanym testom możemy sprawdzić poprawność działania określonej metody, dla wielu różnych danych wejściowych.
3. Ogólną ideą jest łatwiejsze testowanie różnych warunków przy użyciu tej samej metody testowania, co ogranicza ilość pisanej kodu i sprawia, że kod testowy jest bardziej czytelny i spójny.



Testy parametryzowane

Do pisania testów parametryzowanych można wykorzystać gotowe narzędzia. Np.

1. Parametrized

1. Wbudowane narzędzie w JUnita
2. Nie potrzebujemy dodatkowych bibliotek
3. Więcej informacji i dokumentacja:

[http://junit.org/junit4/javadoc/4.12/org/junit/runners/
Parameterized.html](http://junit.org/junit4/javadoc/4.12/org/junit/runners/Parameterized.html)

2. JUnitParams

1. Projekt polskiej firmy Pragmatics
2. Łatwa i szybka integracja z JUnitem
3. Dużo więcej możliwości i czytelniejsze testy
4. Więcej informacji i dokumentacja:

<https://github.com/Pragmatists/JUnitParams>



Parametrized - przykład

Definiowanie testu parametryzowanego

Zmienna przechowująca wartości kolejnych parametrów

Utworzenie konstruktora przez który przekazywane będą wartości do kolejnych wywołań testowych

Adnotacja @Parameters definiuje metodę z parametrami do kolejnych testów

Zdefiniowanie tablicy obiektów która zawiera listę przypadków/wartości do przetestowania

Wartość przekazana z metody getParameters() jako kolejny przypadek do testowania

```
@RunWith(Parameterized.class)
public class CalculatorTest {

    private Calculator calculator;
    private Integer value;

    public CalculatorTest(Integer value) {
        this.value = value;
    }

    @Parameters
    public static Collection getParameters() {
        return Arrays.asList(
            new Integer[][] {
                { 1000 },
                { 2000 },
                { 3000 }
            });
    }

    @Test
    public void testAdding() {
        calculator = new Calculator();

        int result = calculator.add(value, value);

        assertEquals(value + value, result);
    }
}
```



JUnitParams - przykład

Definiowanie testu parametryzowanego

Anotacja @Parameters zawierająca kolejne wartości do przetestowania

Wartość przekazana z metody adnotacji jako kolejny przypadek do testowania

```
@RunWith(JUnitParamsRunner.class)
public class CalculatorTest {

    private Calculator calculator;

    @Test
    @Parameters({ "10", "20", "30" })
    public void testAdding(int value) {
        calculator = new Calculator();

        int result = calculator.add(value, value);

        assertEquals(value + value, result);
    }
}
```

Wyniki działania testu parametryzowanego

The screenshot shows the JUnit interface with the following details:

- Test summary: Finished after 0,018 seconds.
- Test results: Runs: 3/3, Errors: 0, Failures: 0.
- Test tree:
 - pl.lbednarski.test.CalculationDate.CalculatorTest
 - testAdding (0,000 s)
 - testAdding(10) [0] (0,000 s)
 - testAdding(20) [1] (0,000 s)
 - testAdding(30) [2] (0,000 s)
 - Failure Trace: A button to view detailed failure information.



Parametrized vs JUnitParams

```
@RunWith(Parameterized.class)
public class CalculatorTest {

    private Calculator calculator;
    private Integer value;

    public CalculatorTest(Integer value) {
        this.value = value;
    }

    @Parameters
    public static Collection getParameters() {
        return Arrays.asList(
            new Integer[][] {
                { 1000 },
                { 2000 },
                { 3000 }
            });
    }

    @Test
    public void testAdding() {
        calculator = new Calculator();

        int result = calculator.add(value, value);

        assertEquals(value + value, result);
    }
}
```

```
@RunWith(JUnitParamsRunner.class)
public class CalculatorTest {

    private Calculator calculator;

    @Test
    @Parameters({ "10", "20", "30" })
    public void testAdding(int value) {
        calculator = new Calculator();

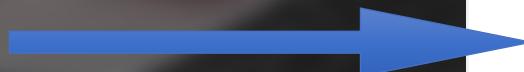
        int result = calculator.add(value, value);

        assertEquals(value + value, result);
    }
}
```



Możliwe jest także przekazywanie więcej niż jednego parametru.

Przekazanie 2 parametrów



```
@RunWith(JUnitParamsRunner.class)
public class CalculatorTest {

    private Calculator calculator;

    @Before
    public void setup() {
        calculator = new Calculator();
    }

    @Test
    @Parameters({ "3,5", "8,-5", "0,-5" })
    public void testAdding(int a, int b) throws Exception {
        assertEquals(a + b, calculator.add(a, b));
    }
}
```

Przekazywać można także wartości typu boolean itp.



Główne ulepszenia dostępne w JUnitParams w stosunku do Parametrized:

- Bardziej jawne - przypadki testowe są w parametrach metody testowej, a nie w polach klasy.
- Mniej kodu - nie potrzebujesz konstruktora do ustawiania parametrów.
- Można mieszać parametryzowane z nieparametryzowanymi metodami w jednej klasie.
- Parametry można przekazać pobierając je z pliku CSV lub klasy dostawcy parametrów.
- Klasa dostawcy parametrów może mieć tyle parametrów, ile ma być dostarczone, tak aby można było grupować różne przypadki



Maven - dodanie zależności:

```
<dependency>
    <groupId>pl.pragmatists</groupId>
    <artifactId>JUnitParams</artifactId>
    <version>1.1.0</version>
    <scope>test</scope>
</dependency>
```



Zadania do wykonania

Twoim zadaniem jest przećwiczenie możliwości poznanych testów parametryzowanych. W tym celu wróć do testu Kalkulatora i zrefaktoryzuj go w taki sposób aby każdą z metod przetestować przynajmniej dla 3 różnych przypadków.

Do swoich testów wykorzystaj JUnitParams, jednakże warto napisać także 1-2 testy z wykorzystaniem Parametrized, aby zobaczyć różnicę.

Spróbuj także przetestować metodę sprawdzającą czy liczba jest parzysta, przekazując liczbę oraz true/false jako wartość oczekiwana.

Zadanie dodatkowe:

Wejdź na stronę: <https://github.com/Pragmatists/junitparams/wiki/Quickstart> i sprawdź w jaki sposób dostarczyć parametry do metody testowej w bardziej zaawansowany sposób:

1. Z innej metody
2. Z innej klasy
3. Z pliku CVS

Spróbuj wykonać dodatkowy test pobierając dane z zewnątrz.



Test Driven Development



Test Drive Development - Definicja

„**Test-driven development (TDD)** - technika tworzenia oprogramowania, zaliczana do metodyk zwinnych. Pierwotnie była częścią programowania ekstremalnego (ang. *extreme programming*), lecz obecnie stanowi samodzielna technikę (...)

Programowanie techniką test-driven development wyróżnia się tym, że najpierw programista zaczyna od pisania testów do funkcji, która jeszcze nie została napisana. Na początku testy mogą nawet się nie kompilować, ponieważ może nie być jeszcze elementów kodu (metod, klas) które są w testach użyte.

Na początku zaczyna się od przypadku, który nie przechodzi testu - zapewnia to, że test na pewno działa i może wyłapać błędy.”

https://pl.wikipedia.org/wiki/Test-driven_development



Czym TDD jest, a czym nie jest.

Czym TDD nie jest?

1. Nie jest to technika służąca do pisania testów
2. Głównym celem Test Driven Development pomimo nazwy nie jest pisanie testów
3. Nie chodzi także o wyeliminowanie potrzeby pracy testera

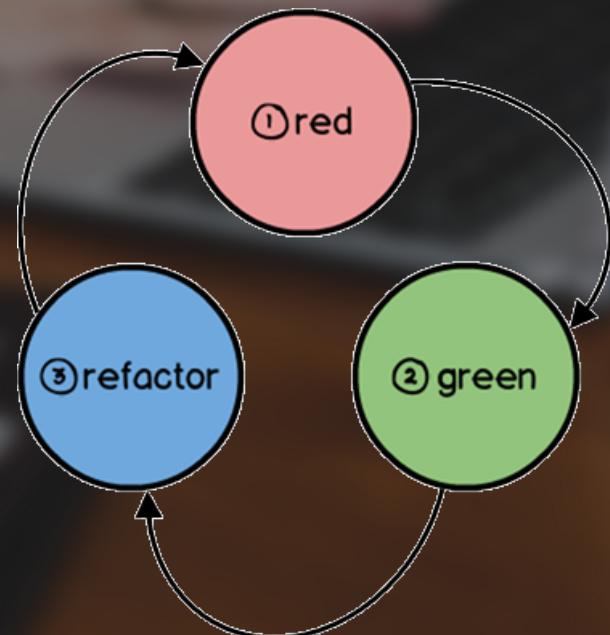
Więc czym jest TDD?

Procesem tworzenia oprogramowania, który opiera się na powtarzaniu krótkich cykli rozwoju, w trakcie których najpierw tworzymy test a później dopiero konkretną implementację. Dzięki takiemu podejściu kod jest (powinien być) przede wszystkim bardziej przemyślany, mniej złożony i krótszy ponieważ implementujemy tylko tyle, ile potrzebujemy aby test był zakończony sukcesem. „Efektem ubocznym” jest duże pokrycie kodu testami.



Red – Green - Refactor

Kluczowym aspektem TDD jest zachowanie pewnego cyklu. Najpierw piszemy testy, następnie implementujemy funkcjonalność, a na końcu refaktorujemy. Cykl nazywany jest najczęściej Red-Green-Refactor lub TDD Mantra i składa się z trzech etapów:



Red – Green - Refactor

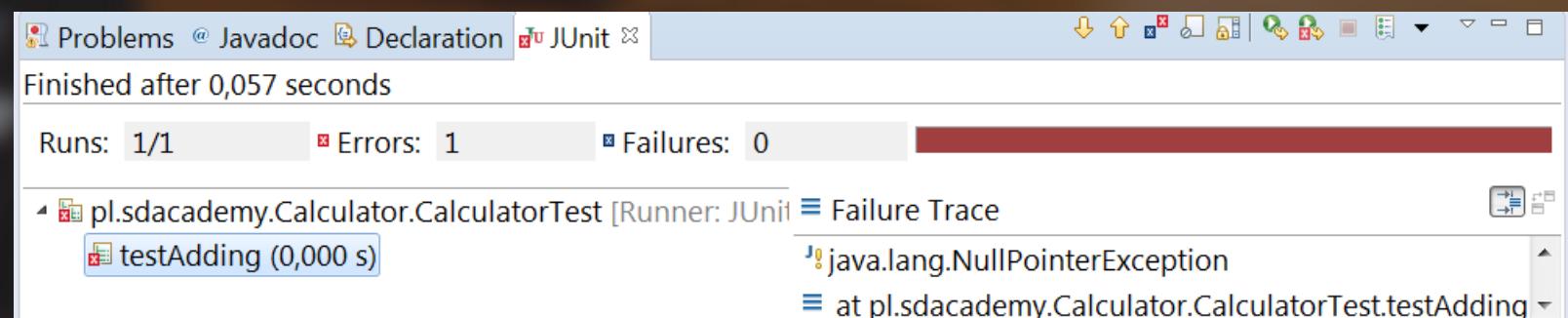


Wstęp: Jeśli jest to pierwszy test bądź nowa klasa tworzymy nową klasę testową

1. Tworzymy nowy test
2. Uruchamiamy test
3. Test zakończył się nie powodzeniem

Red

```
public class CalculatorTest {  
  
    @Test  
    public void testAdding() {  
        Calculator calculator = new Calculator();  
  
        int result = calculator.add(2, 5);  
  
        assertEquals(8, result);  
    }  
}
```



Red – Green - Refactor



1. Pozostawiamy na chwilę testy i przechodzimy do implementacji funkcjonalności
2. Piszemy kod w taki sposób aby rezultat testu był „zielony”
3. Nie implementujemy od razu wszystkich możliwych przypadków użycia funkcji, nie interesują nas na tą chwilę także wartości skrajne, wartości błędne itp.



Red

Green

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

Problems @ Javadoc Declaration JUnit

Finished after 0,036 seconds

Runs: 1/1 Errors: 0 Failures: 0

pl.sdacademy.Calculator.CalculatorTest [Runner: JUnit 4] (Failure Trace)



Red – Green – Refactor

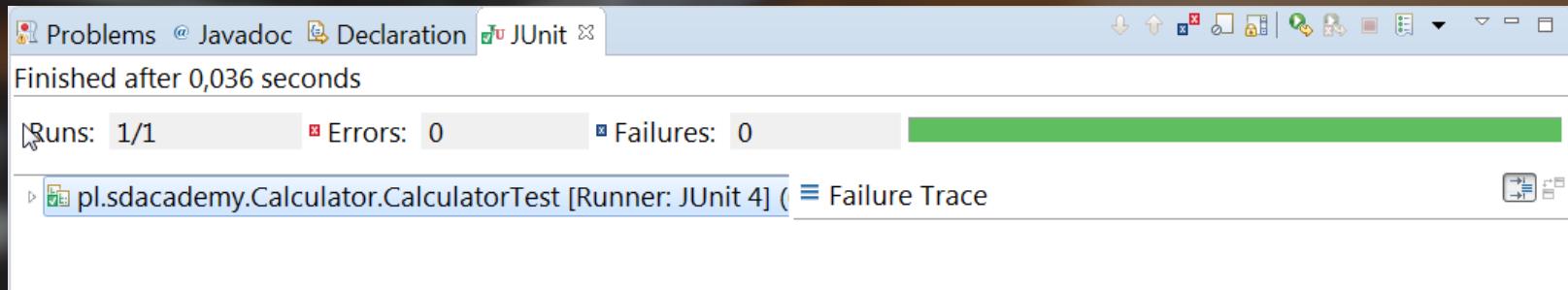
1. Refaktoryzujemy kod który napisaliśmy poprzez wprowadzenie zmian, takich jak np. usunięcie duplikacji w kodzie, nieużywanych zmiennych itp.
2. Ponownie uruchamiamy testy
3. Jeśli zakończyły się pozytywnie, wracamy do pierwszego stanu i dodajemy kolejny test...

Red

Green

Refactor

```
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Before  
    public void setup(){  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void testAdding() {  
        int result = calculator.add(3, 5);  
  
        assertEquals(8, result);  
    }  
}
```





1. Wróć do programu kalkulator
2. Rozbuduj kalkulator o kolejne funkcjonalności:
 1. Sprawdź czy liczba jest parzysta
 2. Podniesienie liczby całkowitej do kwadratu
 1. Podniesienie liczby całkowitej X do potęgi Y
 3. Obliczanie VAT (np. 19%) z kwoty X
 4. Obliczanie kwoty brutto na podstawie wartości netto X oraz Y procent VAT.

PAMIĘTAJ aby stosować technikę TDD.

TDD – Zadanie 2 – dodatkowe



Firma SDA prowadzi księgarnię internetową. Zostałeś(aś) poproszony o dodanie nowej funkcjonalności, a czas na wykonania zadania otrzymałeś do zakończenia bloku tematycznego.

Twoim zadaniem jest napisanie programu, który będzie reprezentował koszyk sklepowy do którego klient może umieścić zamawiane produkty.

Utwórz klasę reprezentującą książkę o nazwie Book. Dla ułatwienia przyjmijmy że książka na początku posiada tylko 3 parametry:

1. Tytuł książki
2. Autor książki
3. Cena

Dodaj także klasę reprezentującą koszyk o nazwie Basket. Klasa ta musi posiadać listę pozwalającą przechowywać obiekty klasy Book oraz wykonywać dodatkowe operacje.



Funkcjonalności:

1. Tworzenie obiektów klasy Book podając tytuł, autora i cenę książki
2. Możliwość zmiany ceny książki
3. Pobranie listy wszystkich książek dodanych do koszyka
4. Dodawanie książek do koszyka
5. Wyczyszczenie koszyka (usunięcie wszystkich elementów koszyka).

Dodatkowo można również zaimplementować:

1. Suma wszystkich produktów znajdujących się w koszyk
2. Usuwanie konkretnej książki z koszyka na podstawie nazwy
3. Sumowanie książek w koszyku. Jeśli do koszyka została dodana kolejna taka sama książka, to w koszyku powinna ona widnieć jako np. ilość 2x.



Testowanie wyjątków



Wyjątki przypomnienie

Wyjątki w Javie to specjalne obiekty, które poza standardowymi operacjami na obiektach możemy także rzucić za pomocą słowa kluczowego throw, co powoduje natychmiastowe przerwanie działania wątku (w najprostszym przypadku - aplikacji) oraz przejście do pierwszego napotkanego miejsca, które ten wyjątek jest w stanie obsłużyć. Nieobsłużony wyjątek uśmierca bieżący wątek.



Najpopularniejsze wyjątki

- **NullPointerException** - rzucany kiedy próbujesz wywołać metodę na zmiennej, której wartość to null
- **IllegalArgumentException** - rzucany, kiedy przekazywany argument jest z jakiegoś powodu nieprawidłowy (walidacja wewnątrz metod)
- **IOException** (wyjątki po nim dziedziczące) - rzucany w przypadku problemów z systemem wejścia/wyjścia, np. kiedy wystąpi problem przy pracy z plikami lub z transmisją danych za pośrednictwem internetu
- **NumberFormatException** - rzucany, kiedy próbujemy zamienić na liczbę np. obiekt typu String, który zawiera nie tylko cyfry
- **IndexOutOfBoundsException** - rzucany, kiedy próbujemy się odwołać do nieistniejącego elementu tablicy lub listy



Przykład wyjątku w Javie

Poniżej przedstawiono przykład rzucaniem wyjątku klasy `ArithmeticException`, w momencie wywołania metody `divide` z argumentem o wartości 0.

```
public class Calculator {  
  
    public Double divide(Double a, Double b) {  
        if (a == 0 || b == 0) {  
            throw new ArithmeticException("Incorrect value");  
        }  
  
        return a / b;  
    }  
}
```

Wyjątek klasy `ArithmeticException` wraz z komunikatem.



Testowania wyjątków

Po co testować wyjątki ?

1. Chcemy mieć pewność i świadomość że system poprawnie reaguje np. na błędne wartości (np. dzielenie przez 0).
2. Chcemy mieć pewność że podczas określonych problemów system „rzuca” wyjątki odpowiednich klas
3. Chcemy mieć pewność że wiadomość/informacja zawarta w rzuconym wyjątku jest poprawna

Przykładowe sposoby testowania wyjątków:

1. Try - catch
2. Słowo kluczowe - expected
3. Adnotacja @Rule



Expected – Przykład

Przykład testowania z wykorzystaniem adnotacji expected. Minusem takiego podejścia jest fakt że nie mamy możliwości zweryfikowania przesłanego komunikatu.

Deklaracja testu który oczekuje że podczas jego działania rzucony zostanie wyjątek.

Wywołanie metody rzuca wyjątkiem więc test zakończy się pozytywnie.

Dodatkowa asercja sprawdzająca wynik metody.

```
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Before  
    public void setup() {  
        calculator = new Calculator();  
    }  
  
    @Test(expected = ArithmeticException.class)  
    public void testDivision() {  
        Double result = calculator.divide(new Double(1), new Double(0));  
  
        assertNull(result);  
    }  
}
```

WAŻNE! W parametrze expected należy podawać konkretną klasę wyjątku. Nie należy więc umieszczać tam wyjątków typu Exception.class ponieważ taki test staje się całkowicie bezużyteczny, pomimo iż pozornie może działać poprawnie.



Try – catch

Konstrukcja try - catch służy do obsługi wyjątków w Javie w trakcie działania programu. Dzięki temu możliwe jest wyłapywanie wyjątków i obsługa nieoczekiwanej błędu. Możemy też try-catch wykorzystać do przeprowadzenia testów jednostkowych.

Wywołanie metody divide w bloku try.
„Złapanie” rzuconego wyjątku.
Możliwość sprawdzenia poprawności
wiadomości dodanej do wyjątku.

```
@Test
public void testDivision() {
    Double result = null;

    try {
        result = calculator.divide(new Double(1), new Double(0));
    } catch (ArithmetricException exception) {
        assertNull(result);
        assertEquals(exception.getMessage(), "Incorect value");
    }
}
```

Finished after 0,02 seconds

Runs: 1/1 Errors: 0 Failures: 1

pl.lbednarski.test.CalculationDate.CalculatorTest [Runner: JUnit 4]

testDivision (0,000 s)

Failure Trace

org.junit.ComparisonFailure: expected:<Incor[r]ect value> but was:<Incor[.]ect value>

at pl.lbednarski.test.CalculationDate.CalculatorTest.testDivision(CalculatorTest.java:1)



ExpectedException oraz @Rule

Adnotacja Rule pojawiła się w JUnit dopiero w wersji 4.7
Służy do bardziej rozbudowanego testowania wyjątków.

Utworzenie obiektu klasy ExpectedException z
adnotacją @Rule do wyłapywania wyjątków.

Zdefiniowanie jakiej klasy wyjątku oczekujemy
oraz jaką wiadomość powinien zawierać.

Wywołanie metody divide która rzuca wyjątkiem.

Dodatkowa asercja sprawdzająca czy wynik na
pewno nie został zwrócony.

```
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Rule  
    public ExpectedException thrown = ExpectedException.none();  
  
    @Before  
    public void setup() {  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void testDivision() {  
        thrown.expect(ArithmeticException.class);  
        thrown.expectMessage("Incorrect value");  
  
        Double result = calculator.divide(new Double(1), new Double(0));  
        assertNull(result);  
    }  
}
```



Testowanie wyjątków - zadanie

Przejdź do programu UnitTests. W pakiecie pl.sda.tests.users znajduje się imitacja prostej bazy danych systemu do zarządzania użytkownikami. Zaimplementowano kilka metod które pozwalają dodawać, usuwać oraz wyszukiwać użytkowników. Przygotowano także prostą metodę pozwalającą przeprowadzić logowanie użytkowników.

Metody saveUser oraz removeUser zawierają fragment kodu pozwalający zweryfikować czy przekazany obiekt typu User nie jest null'em. Odpowiada za to metoda: Objects.requireNonNull(user); Jeśli przekazano null, rzucony zostanie wyjątek typu NullPointerException.



Testowanie wyjątków - zadanie

Dodatkowo utworzony został wyjątek `UserExistException`, i jest on rzucany w momencie kiedy ktoś spróbuje dodać do bazy użytkownika o loginie który już w bazie istnieje.

Przeprowadź serię testów pozwalającą zweryfikować czy faktycznie system zachowuje się w prawidłowy sposób i wyjątki są rzucane.

Pamiętaj aby przeprowadzić testy dla 3 różnych sposobów testowania wyjątków:

- `expected`
- `try - catch`
- `@Rule`

Jeśli jest to możliwe, przetestuj także komunikaty wysłane razem z wyjątkami



Testowanie wyjątków - zadanie

1. Wróć do testów kalkulatora. Dodaj test sprawdzający czy po podaniu wartości 0 jednej z liczb do dzielenia, zwrócony zostanie wyjątek. Sprawdź także poprawność komunikatu.
 - Jeśli chcesz, możesz zrefaktoryzować metodę divide w taki sposób aby wiadomość zawarta w wyjątku zawierała informację która liczba była błędna. Dodaj także test.
2. Przejdź do pakietu pl.sda.test.users i dodaj nowy test. Sprawdź czy wyjątek zostanie rzucony kiedy do metody saveUser oraz removeUser zamiast użytkownika przekazany zostanie null.
3. Spróbuj zarejestrować użytkownika który już istnieje w bazie danych i przetestuj czy rzucony zostanie wyjątek UserExistException.
 - Dodatkowo przetestuj także czy zawarty w nim komunikat jest poprawny
3. Dodaj do bazy danych np. 5 użytkowników, następnie pobierz listę wszystkich zarejestrowanych a z listy tej spróbuj wyciągnąć użytkownika o indeksie np. 10. Sprawdź czy rzucony zostanie wyjątek, i jeśli tak to jakiej klasy.



Biblioteki matcherów



Czym są i po co używać

Biblioteki matcherów powstały aby poprawić przede wszystkim jakość i czytelność pisanych przez nas testów. Ich największymi zaletami są przede wszystkim:

1. Poprawienie czytelności i przejrzystości testów.
2. Komunikaty o błędach podczas niepowodzenia się testu są także zdecydowanie bardziej zrozumiałe.
3. Testy mogą być znacznie bardziej dokładne.
4. Łatwiejsza analiza napisanych testów.

Biblioteki matcherów w Javie



1. Fest Assert
 - Deprecated
2. AssertJ
 - <http://joel-costigliola.github.io/assertj>
3. Hamcrest
 - <http://hamcrest.org/JavaHamcrest/>



AssertJ

„The AssertJ project provides fluent assertion statements for Java. These assert statements are typically used with Java JUnit tests. AssertJ is a library for simplifying the writing of assert statements in tests. It also improves the readability of asserts statements. It has a fluent interface for assertions, which makes it easy for your code completion to help you write them. The base method for AssertJ assertions is the assertThat method followed by the assertion.“

<http://www.vogella.com/tutorials/AssertJ/article.html>



AssertJ – przykład użycia

Przykład testu bez AssertJ:

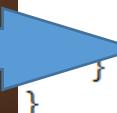
```
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Before  
    public void setup(){  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void testAdding() {  
        int result = calculator.add(3, 5);  
  
        assertEquals(8, result);  
    }  
}
```



Pytanie:
Assert.assertEquals(var, var2);

Która z powyższych wartości jest rezultatem a która oczekiwany wynikiem?

```
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Before  
    public void setup(){  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void testAdding() {  
        int result = calculator.add(3, 5);  
  
        assertThat(result).isEqualTo(8);  
    }  
}
```





AssertJ - przykłady

Sprawdzenie czy wartość pola name obiektu frodo równa się „Frodo”

Sprawdzenie obiekt frodo jest tym samym obiektem co obiekt sauron.

```
// basic assertions
assertThat(frodo.getName()).isEqualTo("Frodo");
assertThat(frodo).isNotEqualTo(sauron);

// chaining string specific assertions
assertThat(frodo.getName()).startsWith("Fro")
    .endsWith("do")
    .isEqualToIgnoringCase("frodo");
```

Test bardziej złożony pozwalający sprawdzić wiele różnych warunków.
Sprawdzone zostało czy wartość pola name obiektu frodo rozpoczyna się od „Fro”, kończy na „do”. Dodatkowo sprawdzone zostało czy nazywa się „frodo” jednakże przy porównywaniu ignorowane są wielkości liter.



AssertJ - przykłady

```
assertThat(fellowshipOfTheRing).hasSize(9)
    .contains(frodo, sam)
    .doesNotContain(sauron);

// as() is used to describe the test and will be shown before the error message
assertThat(frodo.getAge()).as("check %s's age", frodo.getName()).isEqualTo(33);

// Java 8 exception assertion, standard style ...
assertThatThrownBy(() -> { throw new Exception("boom!"); }).hasMessage("boom!");
```

Sprawdzenie czy lista fellowshipOfTheRing zawiera 9 elementów a w śród nich obiekty frodo oraz sam jak również nie zawiera obiektu sauron.

Sprawdzenie wieku frodo. Nowością jest metoda **as** dzięki czemu dodany jest komunikat który będzie wyświetlony w momencie niepowodzenia się testu.

Ciekawostka!
Testowanie wyjątków oraz komunikatów w Javie 8 z wykorzystaniem AssertJ.



AssertJ - rozpoczęcie

Maven - dodanie zależności:

```
<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.8.0</version>
    <scope>test</scope>
</dependency>
```



AssertJ - zadanie

Wróć do programu Calculator, a następnie zrefaktoryzuj istniejące testy w taki sposób, aby zamienić klasyczne asercje z JUnita, na bardziej rozbudowane z AssertJ. Wykorzystaj metody isEqualTo oraz isNotEqualTo. Spróbuj także wykorzystać „.as” aby dodać komunikat o błędzie w przypadku niepowodzenia testu.

Zadanie dodatkowe.

W pakiecie pl.sda.tests.users znajdują się klasy User oraz UserService. Wykonaj kilka testów z wykorzystaniem AssertJ sprawdzając poprawność ich implementacji.

1. Utwórz obiekt klasy User podając w konstruktorze login oraz hasło, a następnie sprawdź czy wartość pola login jest poprawna.
2. Dodaj kilku użytkowników do UserService, a następnie sprawdź czy ilość elementów na liście jest poprawna.
3. Utwórz nowy obiekt klasy User a następnie zapisz go w bazie za pomocą UserService. W kolejnym kroku wyszukaj dodanego użytkownika za pomocą metody findUserByLogin a następnie sprawdź czy zwrócony obiekt jest tym samym który został dodany wcześniej.



Mockowanie



Mockowanie - Wstęp

W trakcie rozwoju oprogramowania, naturalnym jest że kod staje się bardziej złożony. Każdy projekt wykorzystuje także zewnętrzne biblioteki, bazy danych itd. W jaki sposób testować więc taki kod? Co jeśli obiekt klasy X:

1. Zwraca niedeterministyczny wynik?
 - Np. Aktualną godzinę, temperaturę, ciśnienie itp.
2. Posiada stany, które są trudne do wywołania lub zreplikowania?
 - Np. Cykliczne wywołanie funkcji Y każdego dnia o konkretnej godzinie.
3. Podczas normalnego działania korzysta z ogromnych zasobów danych?
 - Np. informacji o klientach pobranych z bazy nadych.
4. Nie jest kompletna, nie istnieje lub też jej zachowanie może się zmienić?
 - Np. Kolega z zespołu pracuje nad poprawieniem implementacji klasy.
5. Do działania potrzebuje wielu innych obiektów?
 - Np. Obiekt klasy MainBoard do działania potrzebuje obiektów klas Processor, HardDrive, Memory, PowerSuply.



Mockowanie - Definicje

„Atrapą obiektu (ang. mock object) - symulowany obiekt, który w kontrolowany sposób naśladuje zachowanie rzeczywistego obiektu. Programista tworzy zazwyczaj atrapy obiektów w celu przetestowania zachowania jakiegoś innego obiektu, podobnie jak projektanci samochodów wykorzystują manekiny do symulacji dynamiki zachowania ludzkiego ciała podczas zderzenia pojazdów.”

Źródło: https://pl.wikipedia.org/wiki/Atrapa_obiektu



Mockowanie - zalety

1. Atrypy obiektów posiadają identyczny interfejs jak obiekty które naśladują
2. Możemy symulować wiele różnych scenariuszy zachowań
3. Atrapa może być wielokrotnie uruchamiana dzięki czemu możemy sprawdzić czy metody zawsze zwracają ten sam wynik
4. Nie ma potrzeby tworzenia i przygotowywania wielu obiektów
5. Testowana klasa może być uruchamiana w wielu różnych konfiguracjach co pozwala na bardziej elastyczne testy.



Mockito



1. Jedna z najpopularniejszych bibliotek do tworzenia atrap obiektów w Javie
2. Aktualna wersja stabilna 2.8.47
3. Ogromne wsparcie i bardzo wiele pomocy naukowych
4. Strona projektu: <http://site.mockito.org/>
5. Dokumentacja:
<https://static.javadoc.io/org.mockito/mockito-core/2.8.47/org/mockito/Mockito.html>



Mockito – Przykłady użycia

Czy metoda została wywołana?

```
import static org.mockito.Mockito.*;  
  
// mock creation  
List mockedList = mock(List.class);  
  
// using mock object - it does not throw any "unexpected interaction" exception  
mockedList.add("one");  
mockedList.clear();  
  
// selective, explicit, highly readable verification  
verify(mockedList).add("one");  
verify(mockedList).clear();
```

Tworzenie mocka - atrapy

Wywołanie metod

Sprawdzenie czy metody zostały wcześniej wywołana

Wynik testu JEŚLI metoda add nie została wywołana z parametrem „one”

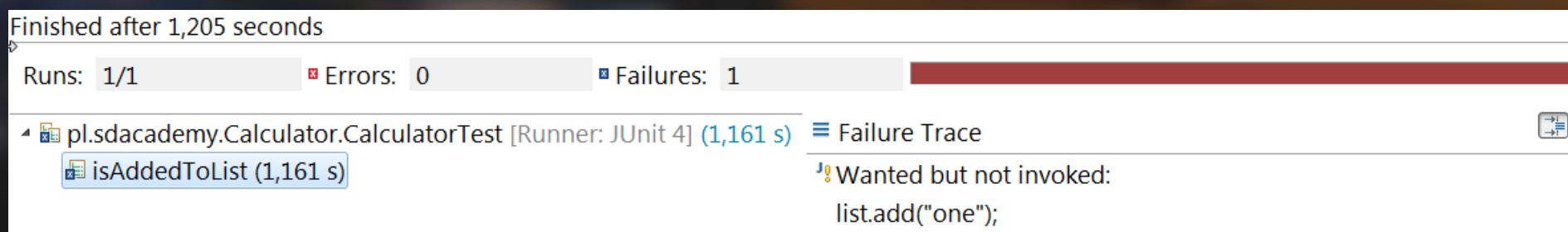
Finished after 1,205 seconds

Runs: 1/1 Errors: 0 Failures: 1

pl.sdacademy.Calculator.CalculatorTest [Runner: JUnit 4] (1,161 s) Failure Trace

isAddedToList (1,161 s)

Wanted but not invoked:
list.add("one");





Mockito – przykłady użycia

Weryfikowanie ilości wywołań metody

```
//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");
```

Verify pozwala między innymi na bardzo konkretne zweryfikowanie ilości wywołań określonej metody wraz z przekazywanym argumentem.

Czy metoda add z parametrem „once” wywołana była tylko raz?

Czy metoda add z parametrem „twice” wywołana była tylko raz a z parametrem „tree times” 3 razy?

Czy metoda add z parametrem „never happened” nie była wywołana ani razu?

Bardziej dokładne testowanie wywołań metod, np. sprawdzenie czy metoda została wywołana jako ostatnia



Mockito – Przykłady użycia

Zwracanie konkretnej wartości po wywołaniu metody

```
// you can mock concrete classes, not only interfaces  
LinkedList mockedList = mock(LinkedList.class);  
  
// stubbing appears before the actual execution  
when(mockedList.get(0)).thenReturn("first");  
  
// the following prints "first"  
System.out.println(mockedList.get(0));  
  
// the following prints "null" because get(999) was not stubbed  
System.out.println(mockedList.get(999));
```

```
//stubbing using built-in anyInt() argument matcher  
when(mockedList.get(anyInt())).thenReturn("element");
```



Bardzo przydatna funkcjonalność podczas symulowania konkretnego scenariusza, np. kiedy chcemy aby nasza metoda zwracająca wartości losowe, w teście zwracała konkretne wartości, np. mniejsze niż 0.



Mockito - adnotacje

Główne adnotacje wykorzystywane podczas korzystania z JUnit i Mockito

```
public class Person {  
  
    private Car car;  
  
    public Person(Car car) {  
        this.car = car;  
    }  
}
```

```
@RunWith(MockitoJUnitRunner.class)  
public class SampleTest {  
  
    @Mock  
    Car car;  
  
    Person person;  
  
    @Before  
    public void setUp() {  
        person = new Person(car);  
    }  
}
```

```
@RunWith(MockitoJUnitRunner.class)  
public class SampleTest {  
  
    @Mock  
    Car car;  
  
    @InjectMocks  
    Person person;  
}
```

Adnotacja dostarczająca tzw. Runner który pozwala korzystać z funkcjonalności Mockito

Adnotacja tworząca atrapę obiektu klasy Car

Adnotacja tworząca obiekt klasy person oraz wstrzyknięcie do niej wszystkich wymaganych obiektów.

Zamiast adnotacji @InjectMocks możemy obiekt klasy Car wstrzyknąć za pomocą konstruktora



Mockito – Rozpoczęcie pracy

Aby rozpocząć pracę należy dodać zależność do mavena:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.8.47</version>
    <scope>test</scope>
</dependency>
```



Mockito - zadanie

Przejdź ponownie do pakietu pl.sda.test.users

Utwórz nowy test oraz atrapę (Mocka) klasy UserService i przeprowadź serię testów.

1. Wykonaj prosty test który po zapisaniu użytkownika w bazie sprawdzi czy wywołana została metoda saveUser tylko jeden raz.
2. Wywołaj metodę login a następnie sprawdź czy metoda findUserByLogin została także wywołana.
3. Utwórz nowy obiekt klasy user a następnie wykorzystując prezentowaną wcześniej funkcję: when(function).thenReturn(value) zwróć utworzony obiekt. Sprawdź czy zwrócony obiekt jest tym samym który wcześniej utworzyłeś.
4. Utwórz atrapę klasy User, dodaj obiekt do bazy a następnie wyszukaj ten sam obiekt po jego loginie. Sprawdź czy zwrócony obiekt jest tym samym który wcześniej został dodany. Pamiętaj o poprawnym przygotowaniu atrapy.



Zakończenie

Dziękuję za uwagę 😊

Łukasz Bednarski