



Rafał Misiak

Lead Java Developer at Vectaury SAS



Testy jednostkowe
JUnit & AssertJ

Materiały do zajęć

<https://github.com/infoshareacademy/jjdd6-materialy-junit.git>



Wprowadzenie do
Testowania aplikacji

Jakość oprogramowania

- // skutki błędów zazwyczaj pojawiają się w czasie użytkowania
- // często odtworzenie problemu jest równie lub mniej prawdopodobne niż trafienie „6” w lotto
- // ręczne sprawdzanie każdej funkcjonalności przy każdej zmianie jest ekonomicznie niemożliwe

Co testujemy?

Testami pokrywamy istotne elementy logiki biznesowej, kod który będzie zmieniany przez innych deweloperów, kod który sprawia szczególnie duże problemy. Skupiamy się również na warunkach brzegowych.

Nie testujemy metod prywatnych, trywialnego kodu typu zwyczajny getter/setter, bibliotek 3rd party.

Realne pokrycie testami to min. 60%-70%.

Metody prywatne

Niektórzy deweloperzy rozluźniają atrybuty dostępu do metod aby być w stanie je testować. Inne przypadki sprowadzają się nawet do zmiany dostępu w testach wykorzystując refleksję.

Przy dobrze skonstruowanej aplikacji metody prywatne będą automatycznie przetestowane przy okazji testów metod, do których mamy dostęp.

Biblioteki

- // JUnit – podstawowa biblioteka do definiowania, uruchamiania, realizowania testów
- // AssertJ – rozszerzenie biblioteki JUnit, pozwala na zapisywanie wyrażeń oceniających poprawność działania kodu, zwiększa czytelność testów

Testy jednostkowe

Testy jednostkowe weryfikują poprawność jednej operacji realizowanej przez jeden moduł programu.

Założmy istnienie modułu z funkcjonalnościami:

- Utworzenie użytkownika z podanym loginem i domyślnym hasłem
- Zmiana hasła
- Pobranie loginu użytkownika

Każda z tych pojedynczych funkcjonalności powinna być z osobna pokryta testami jednostkowymi.

Zabezpieczenie przed błędami

Bardzo często, dodawanie, zmiana, usuwanie innych funkcjonalności zależnych jak również refaktoryzacja, nawet ta błaha, mogą doprowadzić to błędnego działania innych funkcjonalności. Przygotowanie testów jednostkowych powinno zapewnić ochronę przed wdrożeniem na produkcję kodu, który nie spełnia dotychczasowych założeń.

Zadanie

Zapoznaj się z działaniem klas:

- `com.isa.user.User`
- `com.isa.UserMain`



Biblioteka testów
JUnit

Wprowadzenie

Jedną z czołowych bibliotek do obsługi testów w Javie jest Junit.

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.4.1</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.4.1</version>
  <scope>test</scope>
</dependency>
```

Wprowadzenie

Aby testy poprawnie działały z goali mavena, potrzeba jest dodatkowy plugin:

```
<plugin>  
  <artifactId>maven-surefire-plugin</artifactId>  
  <version>2.22.1</version>  
</plugin>
```

Klasy testów

Testy jednostkowe zapisujemy w zwykłych klasach, takich samych w jakich realizujemy implementację właściwej aplikacji.

- Klasy testów umieszczone są w niezależnym katalogu, o ile implementacja znajduje się w katalogu **main** tak testy znajdują się w katalogu **test**
- Metody wykonujące test oznaczane są adnotacją **@Test**

Szkielet klasy testów

Przyjmuje się, że klasa testowa powinna zostać umieszczona dokładnie w takim samym pakiecie w jakim jest umieszczona klasa testowana, z tą jedną różnicą, że w katalogu **test**.

Dodatkowo, nazwa klasy testowej powinna nosić tę samą nazwę co klasa testowana z suffixem **Test**.

Klasa testów jednostkowych musi być publiczna, nie może być abstrakcyjna. Klasy testów mogą dziedziczyć po innych klasach abstrakcyjnych.

Zadanie

Utwórz klasę testową dla klasy:

- `com.isa.user.User`

Pozostaw tę klasę pustą, bez żadnego testu.

Pierwsze testy

Każdy test musi być metodą **publiczną** oznaczoną adnotacją **@Test**. Nazwa tej metody nie ma znaczenia ale powinna być intuicyjna i w prosty sposób wyjaśniać co dokładnie testujemy.

```
@Test  
public void testIfUserCreatedForLegalArgument () {  
  
}
```





@DisplayName

Alternatywnie możemy użyć adnotacji **@DisplayName** co pozwoli nam na użycie czytelnego opisu metody w raporcie testów.

```
@Test
@DisplayName("Check if creating user with valid data provided
ends with success.")
public void testIfUserCreatedForLegalArgument() {
```

Uruchomienie

Testy w IntelliJ możemy uruchomić na różne sposoby.

- Jeden test – symbol  /  na linii sygnatury metody testowej
- Wszystkie testy w klasie – symbol  /  w linii nazwy klasy testowej
- Wszystkie testy – goal *package* lub menu kontekstowe katalogu testów. Uwaga! W goalach mavena brane pod uwagę są tylko klasy, których nazwa kończy się słowem ***Test**

Podstawowy raport

✓ ✕ Test Results	144 ms
▼ ✕ CircleTest	144 ms
✓ testIfCircumferenceCalculatedProperlyForValidIn	122 ms
✓ testIfAreaCalculatedProperlyForValidInput()	3 ms
✕ testIfCircumferenceCalculatedProperlyForInvalidIn	19 ms

TESTS

Running com.isa.geometry.CircleTest

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.093 sec

Running com.isa.geometry.PointTest

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec

Running com.isa.operator.BasicIntOperatorTest

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec

Running com.isa.user.UserTest

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec

Results :

Tests run: 15, Failures: 0, Errors: 0, Skipped: 0

Zadanie

W klasie testowej

- `com.isa.user.UserTest`

Utwórz metodę testową, która będzie weryfikowała czy dla poprawnych danych wejściowych tworzony jest obiekt klasy

- `com.isa.user.User`

Uruchom testy.

Asercje

Czyli „załóżmy, że...”. Zbiór metod, które pozwalają na deklarację założenia jakie ma zostać spełnione przez test. Jeśli założenie nie jest zgodne z wynikiem, test się wyklada.

Dostępne asercje znajdziemy w pakiecie:

```
import static org.junit.jupiter.api.Assertions.*;
```

Zwróć uwagę na pakiet, z którego pochodzą metody.

Asercje: assertEquals

Jako pierwszy argument podajemy wartość oczekiwaną, jako drugi, wartość aktualną.

```
assertEquals("cool_login", user.getLogin());  
assertNotEquals("cool_login", user.getLogin());
```

Odwrócenie kolejności nie wpłynie na działanie testu jednak wprowadzi istotne zaciemnienie w kodzie.

Asercje: pozostałe metody

Wspomniany pakiet zawiera jeszcze kilka innych metod, które pozwalają na porównania tablic, obiektów, **rzucanych wyjątków**, itp.

Asercje: *Equals

- `assertEquals(expected, actual)` – założenie równości dwóch wartości, w przypadku typów prostych wykonuje `==`, w przypadku obiektów wykonuje `.equals()`. Dwa nulle są sobie równe.
- `assertEquals(expected, actual, delta)` – założenie równości dwóch wartości z uwzględnieniem zaokrągleń liczb zmiennie-przecinkowych. Jeśli różnica między wartościami będzie mniejsza od delty, warunek będzie spełniony
- `assertArrayEquals(expected, actual)` – założenie równości dwóch tablic, tablice muszą mieć ten sam rozmiar oraz na poszczególnych indeksach muszą znajdować równe sobie elementy

Zadanie

W klasie testowej

- `com.isa.user.UserTest`

Dokończ metodę testującą tworzenie nowego użytkownika, weryfikując czy dane użytkownika są poprawne.

Uruchom testy.

Asercje: *Null

- `assertNull(actual)` – założenie, że uzyskana wartość jest nullem
- `assertNotNull(actual)` – założenie, że uzyskana wartość nie jest nullem

Asercje: *Same

- `assertSame(expected, actual)` – założenie, że uzyskana oraz spodziewana referencja wskazują na ten sam obiekt
- `assertNotSame(expected, actual)` – założenie, że uzyskana oraz spodziewana referencja nie wskazują na ten sam obiekt

Asercje: *True|False

- `assertTrue(actual)` – założenie, że uzyskana wartość jest true
- `assertFalse(actual)` – założenie, że uzyskana wartość jest false

Asercje: fail

- Jeśli chcemy doprowadzić test do statusu niepowodzenia, nie wykonujemy celowo niepoprawnego założenia `assertEquals` lub innej metody. Używamy wówczas:

`Assertions.fail()` – test automatycznie zostanie zakończony niepowodzeniem

Testowanie wyjątków

Bywają sytuacje, kiedy spodziewamy się, że dana metoda rzuci konkretny wyjątek i jest to celowe działanie. Wówczas musimy napisać test, który będzie sprawdzał, czy dla błędnie podanych danych oczekiwany wyjątek jest rzucony.

```
assertThrows (IllegalArgumentException.class, () -> new User());
```


Zadanie

W klasie testowej

- `com.isa.user.UserTest`

Utwórz metody testujące tworzenie użytkownika z loginami:
null, „(empty)”, bez podawania loginu (domyślny konstruktor)
oraz hasłem null, „(empty)”.

W sumie 5 metod testowych. Przynajmniej jedna metoda
powinna mieć adnotację `@DisplayName`

Uruchom testy.

Grupowanie asercji

Domyślnie, wykonując kilka asercji w ramach jednego testu, w przypadku wystąpienia błędu test jest przerywany. Grupowanie asercji pozwala na zalogowanie wszystkich wyników z danej grupy.

```
assertAll("numbers",  
    () -> assertEquals(numbers[0], 1),  
    () -> assertEquals(numbers[3], 3),  
    () -> assertEquals(numbers[4], 1)  
);
```

Assumptions

Istnieje możliwość warunkowego wykonania testów poprzez założenie. Czyli, wykonaj test jeśli dany warunek jest spełniony.

```
assumeFalse (1 < 0) ;  
assertEquals (1 + 2, 3) ;
```

Zadanie

Przeanalizuj kod klasy:

- `com.isa.JunitAssertions`

Uruchom testy.

Cykl życia testów

- Kolejność testów jest nieistotna – działanie konkretnego testu w żadnym wypadku nie może być uzależniona od działania innych. Każdy test jednostkowy powinien działać niezależnie.
- Testy nie mają efektów ubocznych – wszystkie zmiany w zasobach w czasie działania testu powinny zostać przywrócone po zakończeniu jego działania. Działanie testu nie powinno pozostawiać po sobie żadnych trwałych śladów.

Sterowanie testami

@BeforeAll – metoda wykonywana raz przed wszystkimi testami w klasie

@BeforeEach – metoda wykonywana raz przed każdym testem

@AfterAll – metoda wykonywana raz po wszystkich testach w klasie

@AfterEach – metoda wykonywana raz po każdym teście

@RepeatedTest – powtórzenie testu n-razy

@Disabled – wyłączenie testu

@Tag – opisanie testu tagiem, możliwość profilowania

Zadanie

W klasie testowej

- `com.isa.operator.BasicIntOperatorTest`

Zaproponuj rozwiązanie, które pozwoli uniknąć powtarzania linii:

```
BasicIntOperator basicIntOperator = new BasicIntOperator(4, 2);
```

Uruchom testy.

Zadanie

W klasie testowej

- `com.isa.geometry.PointTest`

Przygotuj metody dla wszystkich adnotacji: `@BeforeAll`, `@AfterAll`, `@BeforeEach`, `@AfterEach`.

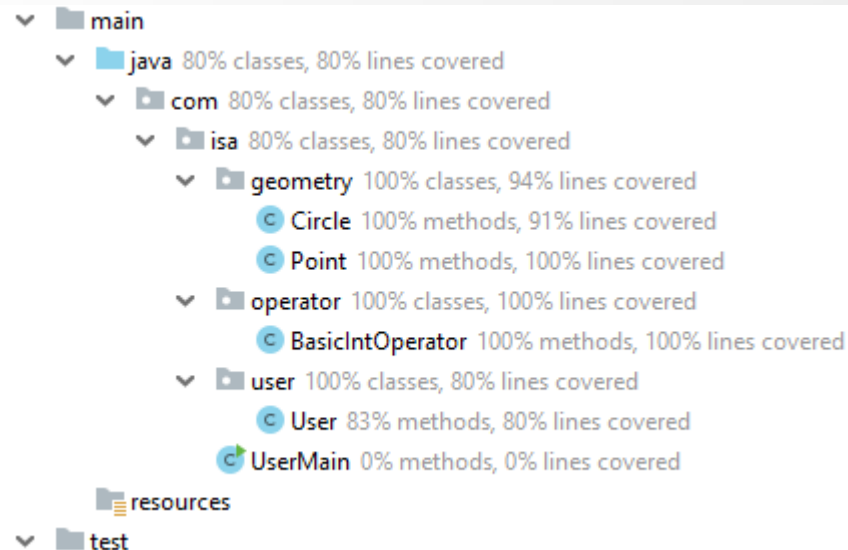
W każdej z powyższych metod oraz metod testowych umieść logowanie informacji jaka metoda została uruchomiona.

Uruchom testy. Przeanalizuj log.

Pokrycie

Istnieje możliwość wygenerowania raportu z informacją o skali pokrycia kodu testami.

Aby w IntelliJ wygenerować raport, należy z menu kontekstowego wybrać **Run All Tests with Coverage**





Biblioteka
AssertJ

Ale po co?

Czytelność warunków oraz konieczność używania uogólnionych, statycznych metod może często zniechęcić programistę do tworzenia bardziej rozbudowanych, skomplikowanych testów.

Bardzo częstym problemem jest mieszanie kolejności wartości oczekiwanej z aktualnie występującą.

Co nam to daje?

AssertJ dla odmiany, realizuje model zwany *fluent programming interface* który ma na celu wywoływania potoku metod, które zwracają kolejno referencję do obiektu, co pozwala na płynne realizowanie testów bez oddzielnych wywołań statycznych metod.

Dodatkowo, na każdym etapie mamy dostęp tylko do tych metod, które są adekwatne do przetwarzanych danych.

Zależności

```
<dependency>  
  <groupId>org.assertj</groupId>  
  <artifactId>assertj-core</artifactId>  
  <version>3.12.2</version>  
</dependency>
```

AssertJ vs JUnit

Nie! AssertJ + JUnit. Pamiętajmy, że biblioteki te w żadnym wypadku się nie wykluczają ani nie zamieniają.

Biblioteki te się **uzupełniają**.

Budowanie kryteriów

Każdy test rozpoczynamy od wskazania wartości rzeczywistej (aktualnej) jako argumentu metody *assertThat*.

```
Assertions.assertThat(value) ;
```

Dostępnej w pakiecie:

```
import org.assertj.core.api.Assertions;
```

Zwróć uwagę na pakiet, z którego pochodzą metody.

Łańcuch warunków

Każdy kolejny warunek zwraca obiekt stanowiący testowaną podstawę. Dlatego też możliwy jest zapis:

```
Assertions.assertThat(circle.calculateCircumference())  
    .isNotNaN()  
    .isEqualTo(12.566370614359172);
```


Zadanie

Wykorzystując AssertJ uzupełnij klasę testów:

- `com.isa.geometry.CircleTest`

O test weryfikujący poprawność wyliczania pola powierzchni koła.

Test powinien sprawdzać:

- Czy wynik jest liczbą
- Czy wynik jest liczbą dodatnią
- Czy wynik jest bliski z dokładnością 4-go miejsca po przecinku

Zadanie

Wykorzystując AssertJ uzupełnij klasę testów:

- `com.isa.geometry.CircleTest`

Napisz test, który wykonuje obliczenia obwodu dla ujemnej wartości promienia.

Wnioski? 😊

Kryterium wyjątków

```
Assertions.assertThatThrownBy();
```

```
Assertions.assertThatExceptionOfType();
```

Zadanie

Wykorzystując AssertJ uzupełnij klasę testów:

- `com.isa.geometry.CircleTest`

O test weryfikujący wystąpienie wyjątku *IllegalArgumentException* dla pola powierzchni w przypadku kiedy wartość promienia nie została ustawiona.

Zadanie

Dla klasy:

- `com.isa.sorter.MapSorter`

Napisz test sprawdzający czy sortowanie po wartości działa poprawnie.

Zadanie

Dla klasy:

- `com.isa.sorter.MapSorter`

Dopisz metodę sortującą po kluczu oraz uzupełnij testy.

Zdebuguj test weryfikując czy działa poprawnie?

Zadanie

Dla klasy:

- `com.isa.geometry.CircleTest`

Dopisz metodę testującą przypadek ujemnej wartości promienia. Jakie są Twoje wnioski?

Zadanie

Dla klas:

- `com.isa.user.AdminUserTest`
- `com.isa.user.User`

Dopisz metodę testującą przypadek, że nowo utworzony użytkownik należy do grupy administratorów.



Thanks!

Q?



Contact

Rafał Misiak

Slack: #rafalmisiak

rafalmisiak@gmail.com