

TESTY JEDNOSTKOWE

JUnit & AssertJ



Hello

Mariusz Szymański

Java Developer, Software Trainer



TESTY

*Programming is like sex:
one mistake and you're providing support for a lifetime.*

Michael Sinz

Materiały do warsztatów

<https://github.com/infoshareacademy/jjdz8-materialy-junit.git>

Jakość oprogramowania

- ↻ skutki błędów zazwyczaj pojawiają się w czasie użytkowania
- ↻ często błędy są bardzo trudne do odtworzenia
- ↻ ręczne sprawdzenie każdej funkcjonalności przy każdej zmianie jest praktycznie niemożliwe i ekonomicznie nieopłacalne

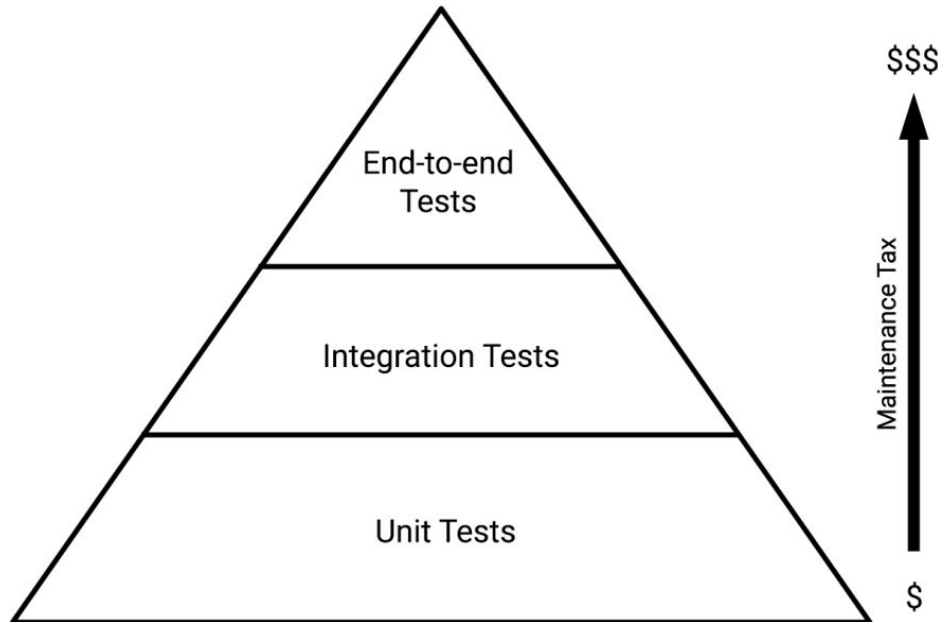
| F.I.R.S.T.

- **F**ast
- **I**ndependent (**I**solated)
- **R**epeatable
- **S**elf-checking
- **T**imely

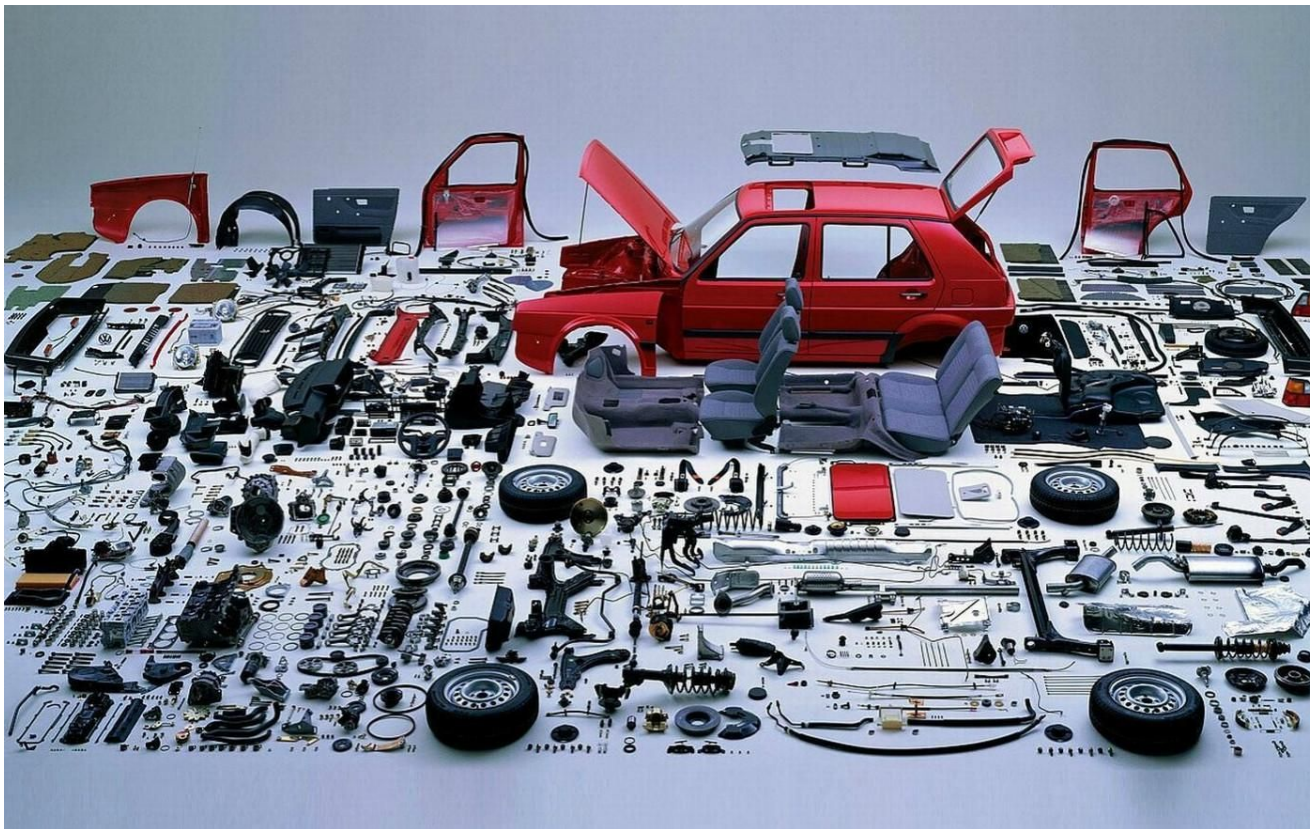
Principles of Unit Testing

Rodzaje testów

- Testy jednostkowe
- Testy integracyjne
- Testy End-to-End (E2E)
- Testy manualne
- Inne



Testy jednostkowe



Testy jednostkowe

Dobre praktyki

- testujemy nie tylko przypadki optymistyczne, ale również brzegowe i wyjątkowe
- dbamy o czytelność i zwięzłość (clean code)
- pojedyncza odpowiedzialność
- nie umieszczamy w testach warunków i pętli

Testy jednostkowe

Dobre praktyki

- testujemy tylko istotne elementy logiki biznesowej
- nie testujemy trywialnego kodu typu getter/setter
- nie testujemy zewnętrznych bibliotek
- nie testujemy metod prywatnych
- realne pokrycie testami to min. 60%-70%

Testy jednostkowe

Metody Prywatne

Niektórzy deweloperzy rozluźniają atrybuty dostępu do metod aby być w stanie je testować. Inne przypadki prowadzą 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.

Testy jednostkowe

Zabezpieczenie przed błędami

Bardzo często, dodawanie, zmiana, usuwanie innych funkcjonalności zależnych jak również refaktoryzacja, nawet ta błaha, mogą doprowadzić do błędnego działania innych funkcjonalności.

Przygotowanie testów jednostkowych powinno zapewnić ochronę przed wdrożeniem na produkcję kodu, który nie spełnia dotychczasowych założeń.

Biblioteki do testów

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

JUnit

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

Mitch Ratcliffe

| JUnit

- JUnit Platform
- JUnit Jupiter
- JUnit Vintage

JUnit 5 = Platform + Jupiter + Vintage



JUnit 5

Platform

- Platforma do uruchamiania framework'ów testowych na JVM
- TestEngine API używane do tworzenia narzędzi testowych uruchamianych na platformie
- W skład platformy wchodzi m.in. Console Launcher oraz pluginy do budowania w Maven oraz Gradle

JUnit 5

Jupiter

- Nowy model pisania testów i rozszerzeń w JUnit 5
- Implementacja TestEngine do uruchamiania testów na JUnit Platform

JUnit 5

Vintage

- Implementacja TestEngine pozwalającą uruchamiać testy napisane w JUnit 3 i JUnit 4 na JUnit Platform (kompatybilność wsteczna)

<https://junit.org/junit5/docs/current/user-guide/>

Zadanie

Zapoznaj się z działaniem klas:

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

JUnit 5

Maven dependency

```
<dependency>
  <groupId>org.junit.jupiter</ groupId>
  <artifactId>junit-jupiter-api</ artifactId>
  <version>5.6.2</ version>
  <scope>test</ scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</ groupId>
  <artifactId>junit-jupiter-engine</ artifactId>
  <version>5.6.2</ version>
  <scope>test</ scope>
</dependency>
```

JUnit 5

Maven dependency

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

```
<plugin>  
  <artifactId>maven-surefire-plugin</artifactId>  
  <version>2.22.2</version>  
</plugin>
```

Kompilacja i uruchomienie testów:

```
mvn clean test
```

Testy jednostkowe

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**

Testy jednostkowe

Szkielet klasy testów

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

Testy jednostkowe

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 () {  
  
}
```

JUnit 5

@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() {



}
```

Testy w IntelliJ

Uruchomienie

Testy w IntelliJ możemy uruchomić na różne sposoby:

Ctrl + Shift + F10

- Jeden test – symbol  na linii sygnatury metody testowej
- Jeden test – `mvn test -Dtest=UserTest#testIfUserCreated`
- Wszystkie testy w klasie – symbol  w linii nazwy klasy testowej
- Wszystkie testy w klasie – `mvn test -Dtest=UserTest`
- Wszystkie testy – Run 'all tests' z menu kontekstowe katalogu testów
- Wszystkie testy – `mvn clean test`

*Uwaga! W goalach mavena brane pod uwagę są tylko klasy, których nazwa kończy się słowem ***Test***



Testy jednostkowe

Podstawowy raport

```
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.053 s - in com.isa.geometry.PointTest
[INFO] Running com.isa.geometry.CircleTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 s - in com.isa.geometry.CircleTest
[INFO] Running com.isa.operator.BasicIntOperatorTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.005 s - in com.isa.operator.BasicIntOperatorTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 20, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 9.145 s
[INFO] Finished at: 2020-05-16T09:52:53+02:00
[INFO] -----
```

▼ ✘ geometry (com.isa)	115 ms
▼ ✔ PointTest	28 ms
✔ testIfPointCreated()	17 ms
✔ testIfDistancelsCorrect()	11 ms
▼ ✘ CircleTest	87 ms
✔ Check if circumference is calculated properly when correc	63 ms
✔ Check if area is not calculated when radius is null.	4 ms
✔ Check if circumference is not calculated when radius is null	2 ms
✔ Check if area is calculated properly when correct input is pi	3 ms
✘ testIfCircumferenceNotCalculatedForRadiusLess0()	15 ms

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.

JUnit 5

Asercje

Asercje, czyli „założmy, 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 (expected), jako drugi, wartość aktualną (actual).

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

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

Asercje

Pakiet `org.junit.jupiter.api.Assertions`

Wspomniany pakiet `Assertions` zawiera wiele innych metod, które pozwalają, m.in. na:

- porównanie obiektów,
- porównanie tablic,
- sprawdzanie rzucanych wyjątków,
- itd.

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

*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

assertThrows

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)

Utwórz metody testujące tworzenie użytkownika z hasłem:

- `null`
- `„(empty)”`

W sumie 5 metod testowych.

Przynajmniej jedna metoda powinna mieć adnotację **@DisplayName**

Uruchom testy.

Asercje

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

Warunkowe wykonanie testu

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.

Testy jednostkowe

Cykl życia testu

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.

JUnit 5

Sterowanie testami

@BeforeAll – metoda wykonywana raz przed wszystkimi testami w klasie

@BeforeEach – metoda wykonywana raz przed każdym testem

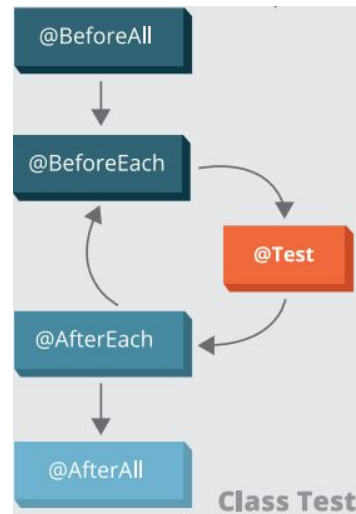
@AfterAll – metoda wykonywana raz po wszystkich testach w klasie

@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.

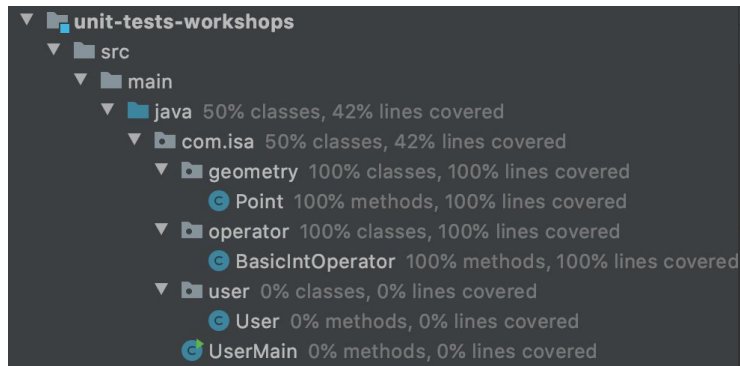
Uruchom testy. Przeanalizuj log.

Testy jednostkowe

Pokrycie kodu testami

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**



Coverage: All in unit-tests-workshops

50% classes, 42% lines covered in package 'com.isa'

Element	Class, %	Method, %	Line, %
geometry	100% (1/1)	100% (4/4)	100% (7/7)
operator	100% (1/1)	100% (5/5)	100% (8/8)
user	0% (0/1)	0% (0/6)	0% (0/15)
UserMain	0% (0/1)	0% (0/1)	0% (0/5)

Retrospekcja

Czyli czego się nauczyliśmy o testach jednostkowych?

- wiemy czym jest test jednostkowy
- znamy lokalizację testów w projekcie
- znamy podstawowe biblioteki do testów
- potrafimy nadać testom odpowiednie nazwy i adnotacje
- znamy podstawowe asercje z pakietu Assertions
- wiemy jak testować wyjątki
- potrafimy sterować testami
- wiemy jak sprawdzić test coverage



Dzięki



mszymanski500@gmail.com



linkedin.com/in/mariuszszymanskipl