

Notatki - Testowanie - Intro

Spis treści

Czym jest testowanie? Po co się testuje?	1
Rodzaje testów	1

Czym jest testowanie? Po co się testuje?

Testy to nic innego jak oprogramowanie (można to rozumieć jako program testujący nasz program), który wywołuje jakiś fragment programu i oczekuje, że zachowa się on w określony sposób, przy podaniu określonych parametrów wejściowych. Testy istnieją po to, aby móc automatycznie zapewnić nas, że napisany przez nas program zachowuje się zgodnie z oczekiwaniami.

Testy są zazwyczaj wykonywane na etapie budowania paczki z aplikacją. Dzięki temu możemy dosyć szybko dowiedzieć się, że albo napisany przez nas kod nie spełnia oczekiwań, albo napisaliśmy kod, który popsuł wcześniej działającą funkcjonalność. W takim przypadku istniejące już testy zaczną nam sygnalizować, że coś przestało działać. W praktyce jest to częsta sytuacja i automatyczna możliwość weryfikacji czy nic nie popsuliśmy jest bardzo przydatna.



Pokrycie kodu testami oznacza, że kod naszej aplikacji został sprawdzony przez testy, które zostały wykonane. Jeżeli powiemy, że chcemy aby nasza aplikacja miała pokrycie (**code coverage**) na poziomie 80%, oznacza to, że 80% linijek napisanego przez nas kodu zostało uruchomionych/wykonanych podczas uruchamiania testów naszej aplikacji.

W praktyce wysokie pokrycie kodu testami jest bardzo przydatne, bo dzięki temu nie boimy się wprowadzać zmian w aplikacji. Nie boimy się bo wiemy, że coś nad nami czuwa i automatycznie sprawdzi czy nic nie popsuliśmy swoimi zmianami 😊. Oczywiście jest to sytuacja idealna i w praktyce nie zawsze spotkamy wysokie pokrycie kodu testami.

Panuje przekonanie, że testy powinny być pisane dla krytycznych i złożonych funkcjonalności. Piszę o tym dlatego, że przyjmuje się, że można zignorować pisanie testów dla getterów i setterów, a przecież nadal są to linijki kodu. Dlatego wspominałem wcześniej o pokryciu kodu testami na poziomie 80% (który w praktyce i tak jest bardzo wysokim wynikiem). Oznacza to, że zakładamy jakiś bufor na fragmenty kodu, które nie są wywoływane w żadnym z testów.

Jeżeli podchodzimy do aplikacji, która nie ma napisanych testów, dobrze jest zacząć od krytycznych funkcjonalności, czyli takich, które generują dla nas "największy biznes" lub są krytyczne z punktu widzenia użytkownika, np. proces zakupu produktu. Przykładowo dlatego, że np. więcej osób kupuje w sklepie niż reklamuje zakupiony towar.

Rodzaje testów

Jeżeli będziemy rozmawiać o testach, w pewnym momencie zaczniemy też wyróżniać rodzaje testów. W tym materiale skupimy się tylko na najprostszych, natomiast rodzajów testów jest dużo. Wymieńmy

parę z nich:

- testy **wydajnościowe** (performance tests) - ich podstawowym celem jest zapewnienie, że testowany kod będzie działał wystarczająco szybko nawet jak będziemy go uruchamiać pod bardzo dużym obciążeniem, np. aplikacja zachowa się inaczej gdy będzie z niej korzystało 1_000 a 1_000_000 użytkowników.
- testy **penetracyjne** (penetration tests) - celowe przeprowadzenie cyberataku na naszą aplikację celem sprawdzenia stopnia zabezpieczeń aplikacji.
- testy **jednostkowe** (unit tests) - testy pisane przez developerów, które wykonują określony fragment kodu z określonymi parametrami wejściowymi i sprawdzają czy zachowanie końcowe jest zgodne z oczekiwanym. Testy jednostkowe mają to do siebie, że sprawdzają bardzo małe fragmenty kodu, pojedyncze klasy lub metody. Jeżeli dany fragment kodu wykonuje operację, która jest zależna od interakcji z systemem zewnętrznym (np. baza danych, albo inna aplikacja), wtedy taki system jest zastępowany zaślepką (**mock** - powiemy o tym później). Zaślepka taka może nam odpowiedzieć w określony sposób, pozytywnie lub negatywnie, konfigurujemy to sami. Robimy to w tym celu, żeby osiągnąć swojego rodzaju izolację. Kod, który testujemy ma być niezależny od niczego z zewnątrz. Jeżeli natomiast nasze testy uwzględniają interakcję z systemami zewnętrznymi zaczynamy wtedy mówić o testach integracyjnych.
- testy **integracyjne** (integration tests) - mają sprawdzać zachowanie komponentów, na styku których następuje integracja. Testujemy wtedy komponenty, które integrują się ze sobą i są testowane w grupie. Z racji szerokości tej definicji, w życiu codziennym najprawdopodobniej spotkasz się ze stwierdzeniem "testy integracyjne" w poniższych sytuacjach:
 - testy, w których testujemy jednocześnie kilka/kilkanaście klas. Jeżeli przyjmimy, że testy jednostkowe testują tylko pojedynczą metodę w klasie, to w przypadku, gdy testujemy kod, który integruje ze sobą kilka/kilkanaście naszych klas lub modułów, to mówimy o testach integracyjnych,
 - testy, w których zaślepiamy komunikację z systemami zewnętrznymi, ale nadal testujemy kilka/kilkanaście naszych klas lub modułów. Testy integracyjne mają sprawdzić integrację między komponentami, więc sprawdzamy integrację między naszymi komponentami, ale zaślepiamy systemy zewnętrzne,
 - testy, w których faktycznie komunikujemy się z systemami zewnętrznymi, np. z bazą danych, albo innymi systemami. Może być to np. system, w którym składamy zamówienia. Wtedy jedynie zostaje kwestia utrzymania porządku z danymi w takim systemie, np. po wykonaniu takiego testu możemy wtedy takie dane czyścić.

Rodzajów testów jest więcej, ale na ten moment chcę się skupić tylko na **testach jednostkowych**. Wspominam też cały czas o tych zaślepkach, ale w ramach tego warsztatu nie będziemy poruszać tej tematyki. Przejdziemy do niej w dalszych częściach materiału.

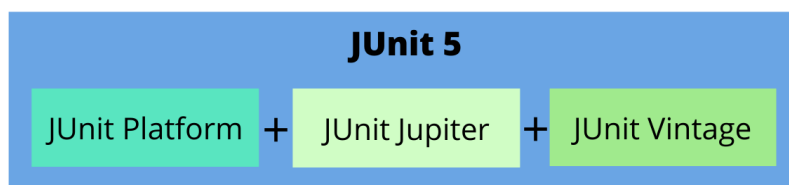
Notatki - Testowanie - JUnit

Spis treści

JUnit.....	1
Dodanie zależności	2
Konfiguracja dla Maven	2
Konfiguracja dla Gradle	3
Stworzenie klasy testowej.....	3
Metoda testowa.....	4
Maven	6
Gradle	7
Konwencja given, when, then	7
@DisplayName	8
Metody before i after	9
Asercje	11
Jakie mamy dostępne podstawowe asercje?	12
AssertAll	13
Sprawdzanie wyjątków	14
AssertJ	14
Wyłączenie testów	15
Testy parametrized.....	15
Arguments	18
Błędy	19
Logowanie.....	19

JUnit

Bardzo popularnym (są inne, ale o dziwo, zawsze w praktyce widziałem tylko ten) frameworkiem, który służy do tworzenia testów jednostkowych jest JUnit. Do przykładów posłuży nam JUnit w wersji 5. JUnit 5 składa się z 3 modułów, które wymieniam poniżej, a [tutaj](#) dokumentacja.



Obraz 1. JUnit 5

- **JUnit Platform** - jest platformą służącą do uruchamiania różnych frameworków testowych na maszynie wirtualnej Javy. Oznacza to, że przy wykorzystaniu platformy JUnit Platform możemy również uruchamiać inne biblioteki testujące takie jak Spock, Cucumber lub FitNesse, ale o nich nie będziemy rozmawiać. JUnit Platform zapewnia również interface pomiędzy JUnit a np. narzędziami

do budowania.

- **JUnit Jupiter** - moduł zawierający nowy model (np. nowe adnotacje), który został wprowadzony w JUnit w wersji 5.
- **JUnit Vintage** - moduł umożliwiający uruchomienie testów napisanych w JUnit 3 lub 4.

Dodanie zależności

Jeżeli chodzi o paczki, to będziemy mieli do czynienia z 2 nazwami:

- **junit-jupiter-api** - zależność, która umożliwia dodanie do projektu adnotacji oznaczających, które metody w kodzie mają zostać wykorzystane jako metody testowe.
- **junit-jupiter-engine** - zależność służąca do uruchomienia testów.

Zależność **junit-jupiter-engine** jest zależna pod spodem od **junit-jupiter-api**, więc wystarczy dodać tylko tę pierwszą.

Konfiguracja dla Maven

Aby wykorzystać zależność JUnit w projekcie przy użyciu Maven, należy dodać następującą konfigurację w pliku **pom.xml**:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.8.2</version>
  <scope>test</scope>
</dependency>
```

Pamiętasz **scope** w Maven? Teraz dopiero przyda nam się **scope: test**. Oznaczało to, że dependencja nie jest wymagana podczas normalnego działania programu, potrzebujemy jej natomiast jeżeli będziemy pisać testy do naszego programu. Najczęściej stosujemy to do bibliotek, które są używane tylko do testów. Czyli jeżeli dodamy dependencję jako **test**, to będzie ona używana w testach, ale nie będziemy mieli dostępu do tej zależności z poziomu normalnego kodu.

Ponadto, w przypadku Maven, jeżeli chcemy, aby testy uruchamiały się podczas buildu aplikacji, musimy dodać **Maven Surefire Plugin**. Traktuje on jako testy wszystkie klasy, które są umieszczone w ścieżce **src/test/java** oraz spełniają warunki:

- Ich nazwa zaczyna się lub kończy Stringiem "Test",
- Ich nazwa kończy się Stringiem "Tests",
- Ich nazwa kończy się Stringiem "TestCase".

Z kolei pod ścieżką **src/test/resources** możemy umieścić pliki z konfiguracją, która ma być używana w testach.

Konfiguracja pluginu Maven Surefire Plugin

```
<build>
```

```

<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M5</version>
  </plugin>
</plugins>
</build>

```

Konfiguracja dla Gradle

Aby wykorzystać zależność JUnit w projekcie przy użyciu Gradle, należy dodać następującą konfigurację w pliku **build.gradle**:

```

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.2'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.2'
}
test {
    useJUnitPlatform()
    testLogging {
        events "passed", "skipped", "failed"
    }
}

```

Ponownie, oznaczenie **test** oznacza, że zależność będzie używana tylko w testach, nie będziemy mieli do niej dostępu w normalnym kodzie.

Pomimo, że Gradle w wersji wyższej niż **4.6** ma wbudowaną obsługę JUnit5, nie jest ona domyślnie włączona. Aby to zrobić, należy dodać dopisek:

```

test {
    useJUnitPlatform()
}

```

Natomiast fragment poniżej odpowiada za logowanie przebiegu wykonania testów. Bez niego Gradle nie loguje informacji, które testy zostały wykonane.

```

test {
    testLogging {
        events "passed", "skipped", "failed"
    }
}

```

Stworzenie klasy testowej

Klasy z testami powinny być umieszczone w ścieżce **src/test/java**, zatem możemy tam stworzyć taką ścieżkę ręcznie, lub wykorzystać do tego IntelliJ.

Stwórzmy najpierw taką klasę:

```
package pl.zajavka;

public class Calculator {

    public static Integer add(int left, int right) {
        return left + right;
    }
}
```

Jeżeli teraz użyjemy skrótu **CTRL + SHIFT + T**, IntelliJ podpowie nam, że możemy utworzyć test sprawdzający tę klasę. Jeżeli taki test już istnieje, zostanie nam to pokazane w oknie dialogowym (jeżeli klas testowych jest więcej niż jedna) lub zostaniemy do tego testu bezpośrednio przekierowani.



Dobłą praktyką jest, aby klasy testowe były umieszczone w tej samej paczce, co kod źródłowy. Czyli dla klasy `pl.zajavka.Calculator`, test powinien być umieszczony w paczce `pl.zajavka`, czyli pełna ścieżka to `pl.zajavka.CalculatorTest`.

Metoda testowa

Jako test rozumiemy metodę w klasie testowej, która jest oznaczona adnotacją `@Test`. Klasa taka jest używana tylko do testów, nie odnosimy się do niej w kodzie źródłowym (czyli tym w `src/main/java`). Jeżeli oznaczymy metodę adnotacją `@Test`, możemy ją wywołać tak samo jak wywoływaliśmy metodę `main()`. Co więcej, w jednej klasie testowej możemy mieć wiele metod testowych, zatem możemy uruchomić jednocześnie całą klasę testową, ze wszystkimi metodami w niej zawartymi.



Pamiętasz, jak kiedyś wspomniałem, że oprócz metody `main()`, możemy uruchomić aplikację jeszcze w inny sposób? Miałem na myśli to, że wcale nie musimy mieć w kodzie metody `main()`. Możemy też program uruchomić z poziomu testów. Należy jednak pamiętać o tym, że to zastosowanie ma sens tylko gdy piszemy mały program, który zostanie uruchomiony, coś policzy i zakończy swoje działanie. Wtedy możemy określić kilka sposobów uruchomienia takiego programu poprzez różne metody testowe z różnymi parametrami i w ten sposób go uruchamiać. W praktyce natomiast, gdy będziemy pisali aplikacje uruchamiane na serwerze (o czym jeszcze będzie), ten sposób nie zda egzaminu.

Poniżej przykład klasy testowej, z metodą testową `testAdd()`, która testuje metodę `Calculator.add()`:

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

class CalculatorTest {

    @Test
    void testAdd() {
        // given
        int left = 5;
        int right = 7;
        Integer expected = 12;
    }
}
```

```

    // when
    Integer result = Calculator.add(left, right);

    // then
    Assertions.assertEquals(expected, result);
}
}

```

Zwróć uwagę, że metoda testowa `testAdd()`, nic nie zwraca. Klasa `CalculatorTest` wcale nie musi być publiczna, tak samo jak metoda `testAdd()`. Jeżeli w tym momencie uruchomimy test z IntelliJ, zostanie wykonana metoda `Calculator.add()` z parametrami `int left = 5;` oraz `int right = 7;`, a na koniec nastąpi sprawdzenie, czy wynik metody `Calculator.add()` jest równy `Integer expected = 12;`.

Dodajmy teraz jeszcze 3 metody do klasy `Calculator`. Metodę odejmującą, mnożącą i dzielącą:

```

package pl.zajavka;

public class Calculator {

    public static Integer add(int left, int right) {
        return left + right;
    }

    public static Integer subtract(int left, int right) {
        return left - right;
    }

    public static Integer multiply(int left, int right) {
        return left * right;
    }

    public static Integer divide(int left, int right) {
        return left / right;
    }
}

```

Do tego, dodajmy metody testujące każdą z dopisanych metod:

```

package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

class CalculatorTest {

    // test metody testAdd()

    @Test
    void testSubtract() {
        // given
        int left = 5;
        int right = 7;
        Integer expected = -2;

        // when
        Integer result = Calculator.subtract(left, right);
    }
}

```

```

        // then
        Assertions.assertEquals(expected, result);
    }

    @Test
    void testMultiply() {
        // given
        int left = 5;
        int right = 7;
        Integer expected = 35;

        // when
        Integer result = Calculator.multiply(left, right);

        // then
        Assertions.assertEquals(expected, result);
    }

    @Test
    void testDivide() {
        // given
        int left = 15;
        int right = 7;
        Integer expected = 2;

        // when
        Integer result = Calculator.divide(left, right);

        // then
        Assertions.assertEquals(expected, result);
    }
}

```

Możemy teraz uruchomić wszystkie metody testowe z poziomu klasy `CalculatorTest`.

Maven

Testy te będą teraz również uwzględnione przy uruchamianiu budowania przez Maven i Gradle. Możemy zatem uruchomić poniższą komendę (pamiętając, że musimy wcześniej skonfigurować [Maven Surefire plugin](#)):

```
mvn verify
```

I na ekranie zobaczymy taki wydruk:

```

[INFO]
[INFO] -----
[INFO]  T E S T S
[INFO] -----
[INFO] Running pl.zajavka.CalculatorTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.017 s - in
pl.zajavka.CalculatorTest
[INFO]
[INFO] Results:

```



```
[INFO]  
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
```

Możemy również przekazać Mavenowi, że chcemy wywołać build, ale ma nie uruchamiać testów, w tym celu wykonamy komendę:

```
mvn verify -DskipTests
```

Gradle

Korzystając z Gradle, wykonamy komendę:

```
gradle build
```

I na ekranie zobaczymy taki wydruk:

```
> Task :test  
  
CalculatorTest > testAdd() PASSED  
  
CalculatorTest > testSubtract() PASSED  
  
CalculatorTest > testDivide() PASSED  
  
CalculatorTest > testMultiply() PASSED  
  
BUILD SUCCESSFUL in 1s  
4 actionable tasks: 2 executed, 2 up-to-date
```

Możemy również przekazać Gradlowi, że chcemy wywołać build, ale ma nie uruchamiać testów, w tym celu wykonamy poniższą komendę. Skrót **-x** oznacza exclude task.

```
gradle build -x test
```

Konwencja given, when, then

O co chodzi z tym dodaniem komentarzy **given, when, then**? Po prostu - konwencja.



Obraz 2. Konwencja *given, when, then*

Jedną ze stosowanych konwencji przy pisaniu testów jest nazywanie klas testowych tak aby kończyły się na `*Test.java`. Jednocześnie wspomnieliśmy już, że narzędzia takie jak np. Maven rozpoznają w ten sposób, gdzie mogą znaleźć testy.

Kolejną konwencją przy pisaniu testów jest dokładne określenie co dana metoda testowa sprawdza. Możemy to określić nazywając metodę w odpowiedni sposób np. `shouldCalculateAdditionCorrectly()`. W praktyce jednak nie jest to tak często stosowane i każdy nazywa testy tak aby dało się zrozumieć co one sprawdzają. JUnit5 daje również możliwość dodania adnotacji `@DisplayName`, która może być umieszczona nad metodą testową i która pozwala opisać co dany test robi.

Często stosowaną konwencją jest *given, when, then*. Dzięki stosowaniu tej konwencji możemy podzielić naszą metodę testową na 3 logiczne fragmenty:

- **given** - w którym definiujemy co jest parametrem wejściowym naszego testu,
- **when** - w którym określamy jaka logika jest faktycznie testowana,
- **then** - w który definiujemy co jest spodziewanym wynikiem.

Stąd komentarze w kodzie i stąd nazwa *given, when, then*.

Wrócimy jeszcze do tej konwencji na etapie omawiania **Mockito** w przyszłych materiałach.

@DisplayName

Adnotacja `@DisplayName` może być zastosowana jak w przykładzie poniżej:

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

class CalculatorTest {
```

```

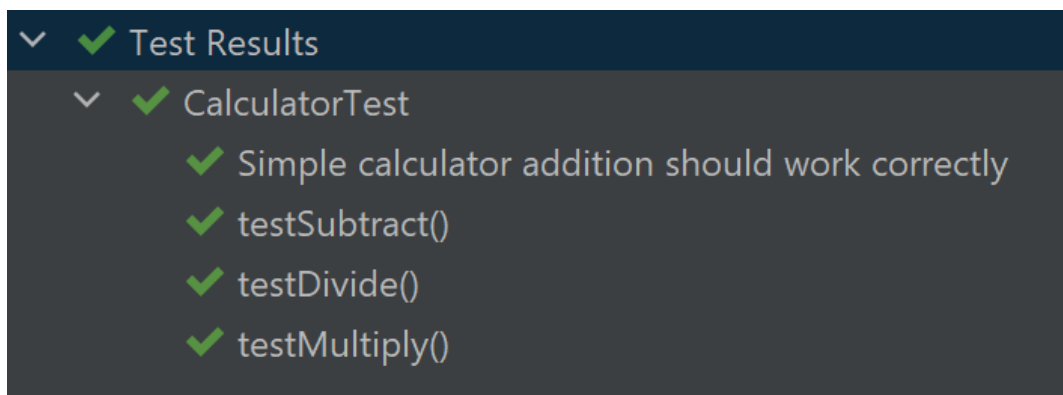
@Test
@DisplayName("Simple calculator addition should work correctly")
void testAdd() {
    // given
    int left = 5;
    int right = 7;
    Integer expected = 12;

    // when
    Integer result = Calculator.add(left, right);

    // then
    Assertions.assertEquals(expected, result);
}
}

```

Jeżeli teraz uruchomimy ten test, to na ekranie zobaczymy taki zapis:



Obraz 3. Rezultat zastosowania adnotacji `@DisplayName`

Widać, że jest to czytelniejsze niż określanie co dany test sprawdza poprzez nazywanie w ten sposób metody. Jeżeli nazywamy w metodzie co dany test robi to dostaniemy szlaczek pokroju `shouldCalculateAdditionCorrectly()`, które w praktyce potrafią być jeszcze dłuższe. Stosując `@DisplayName` łatwiej jest odczytać intencję autora.

Metody before i after

JUnit daje nam możliwość uruchomienia kodu jednokrotnie przed wszystkimi metodami testowymi w klasie lub za każdym razem przed każdą z nich. Jednocześnie możemy zrobić to samo po każdej metodzie testowej lub jednokrotnie po wszystkich.

Zdefiniujmy klasę testową w ten sposób:

```

package pl.zajavka;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

class CalculatorTest {

```

```

@BeforeEach
void methodSetup() {
    System.out.println("Calling @BeforeEach");
}

@AfterEach
void methodTearDown() {
    System.out.println("Calling @AfterEach");
}

@BeforeAll
static void classSetup() {
    System.out.println("Calling @BeforeAll");
}

@AfterAll
static void classTearDown() {
    System.out.println("Calling @AfterAll");
}

@Test
@DisplayName("Simple calculator addition should work correctly")
void testAdd() {
    System.out.println("Calling testAdd");
}

@Test
void testSubtract() {
    System.out.println("Calling testSubtract");
}

@Test
void testMultiply() {
    System.out.println("Calling testMultiply");
}

@Test
void testDivide() {
    System.out.println("Calling testDivide");
}
}

```

Jeżeli teraz uruchomimy wszystkie testy z poziomu klasy, na ekranie wydrukuje się przebieg wykonania testów z całej klasy:

```

Calling @BeforeAll
Calling @BeforeEach
Calling testAdd
Calling @AfterEach
Calling @BeforeEach
Calling testSubtract
Calling @AfterEach
Calling @BeforeEach
Calling testDivide
Calling @AfterEach
Calling @BeforeEach
Calling testMultiply
Calling @AfterEach

```

Zwróć uwagę, że zachowanie jest takie jak napisałem. `@BeforeAll` i `@AfterAll` uruchamia się tylko na początku i na końcu klasy testowej. Natomiast `@BeforeEach` i `@AfterEach` uruchamiają się za każdym razem przed i po uruchomieniu metody testowej.



Zwróć uwagę, że metody oznaczone adnotacjami `@BeforeAll` i `@AfterAll` są statyczne!

Po co to? W naszym przykładzie testujemy metody statyczne (metody w klasie `Calculator` są statyczne). Ale jeżeli musielibyśmy testować metody na obiekcie, który trzeba utworzyć, to po co za każdym razem tworzyć obiekt w każdej metodzie testowej na nowo, jeżeli można zapisać to tak:

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

class CalculatorTest {

    private Calculator calculator;

    @BeforeEach
    void methodSetup() {
        calculator = new Calculator();
    }

    @Test
    @DisplayName("Simple calculator addition should work correctly")
    void testAdd() {
        calculator.calculate();
        // reszta testu
    }
}
```

W ten sposób, możemy przy każdym teście operować na nowym obiekcie, ale nie musimy za każdym razem zapisywać jego utworzenia na nowo. `@BeforeAll` i `@AfterAll` mogą być natomiast użyte jeżeli chcemy w ramach całego testu zainicjować np. połączenie do bazy danych. Przecież nie będziemy go inicjować za każdym razem, przed uruchomieniem metody testowej.

Asercje

Teraz możemy przejść do zapisu na końcu całego testu, czyli `Assertions.assertEquals(expected, result);`. Asercje są sposobem na określenie, co jest spodziewanym wynikiem naszego testu. Asercja sprawdza wtedy, czy wynik działania testu zgadza się ze stanem oczekiwanym. W poprzednim przypadku została użyta metoda `assertEquals()`, która przyjmuje parametry: spodziewany i faktyczny wynik. Następnie sprawdza czy faktycznie jest tak jak oczekiwaliśmy. Jeżeli wartość oczekiwana i faktyczny rezultat są zgodne, test zostaje zaliczony (`passed`). Jeżeli nie, asercja przerywa wykonanie testu i oznacza go jako nieudany (`failed`).

Wspomniałem, że asercja `failed` przerywa wykonanie testu. Możemy zdefiniować kilka asercji pod sobą,

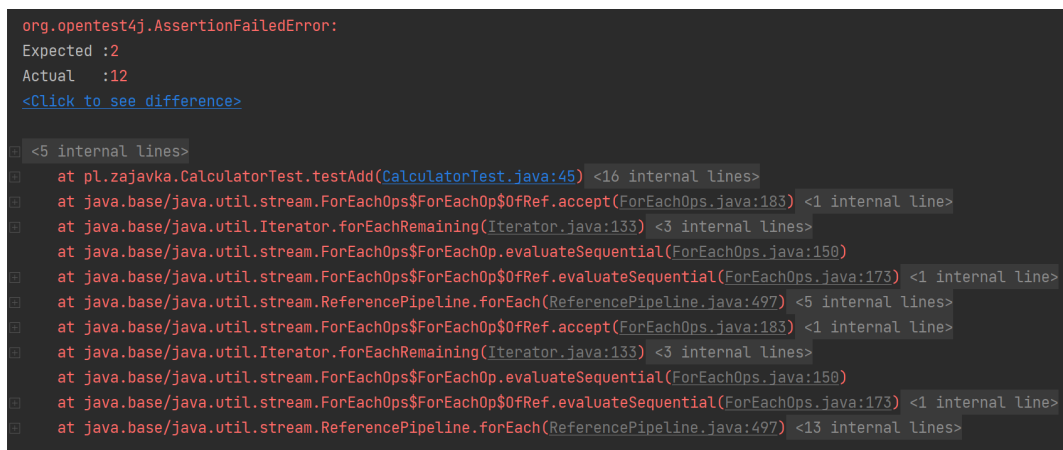
w zależności od tego jakie są nasze potrzeby. Jeżeli tylko pierwsza z nich failuje (w sumie to na co dzień tak się to określa), pozostałe nawet nie są sprawdzane. Przykład failującego testu:

```
@Test
@DisplayName("Simple calculator addition should work correctly")
void testAdd() {
    // given
    int left = 5;
    int right = 7;
    Integer expected = 2;

    // when
    Integer result = Calculator.add(left, right);

    // then
    Assertions.assertEquals(expected, result);
}
```

Wynik $5 + 7 \neq 2$, zatem test failuje. Na ekranie zobaczymy wtedy coś takiego:



Obraz 4. Wynik testu, który zakończył się niepowodzeniem

Jakie mamy dostępne podstawowe asercje?

Przykład podstawowych asercji umieszczam w tabeli poniżej:

Tabela 1. Podstawowe asercje

Metoda	Przykład
<code>assertEquals</code>	<code>assertEquals(2, 12, "Opcjonalna wiadomość przy niepowodzeniu");</code>
<code>assertTrue</code>	<code>assertTrue(5 != 7, 0 → "Opcjonalna wiadomość przy niepowodzeniu");</code>
<code>assertFalse</code>	<code>assertFalse(2 == 4, "Opcjonalna wiadomość przy niepowodzeniu");</code>
<code>assertNotNull</code>	<code>assertNotNull(jakaśReferencja, "Opcjonalna wiadomość przy niepowodzeniu");</code>
<code>assertNull</code>	<code>assertNull(jakaśReferencja, "Opcjonalna wiadomość przy niepowodzeniu");</code>

Zauważ, że w jednym przykładzie, wiadomość została przekazana przez **lambdę**. Napiszmy taki przykład:

```
class CalculatorTest {

    @Test
    void someTest() {
        Assertions.assertEquals(1, 1, createMessage(1));
        Assertions.assertEquals(1, 1, () -> createMessage(2));
        Assertions.assertEquals(1, 2, () -> createMessage(3));
    }

    private String createMessage(int i) {
        System.out.println("Evaluating failure message: " + i);
        return "failure message";
    }
}
```

Po wywołaniu metody testowej `someTest()`, na ekranie zostanie wydrukowane:

```
Evaluating failure message: 1
Evaluating failure message: 3

org.opentest4j.AssertionFailedError: failure message ==>
Expected :1
Actual   :2
```

Zauważ, że jeżeli przekazaliśmy wiadomość bezpośrednio (oczywiście w formie metody, aby widzieć, poprzez `System.out.println()`, że metoda została wykonana), to nawet jeżeli asercja nie failuje, to i tak na ekranie wydrukowała się wiadomość `Evaluating failure message: 1`.

W przypadku wiadomości `Evaluating failure message: 2`, została ona przekazana w formie lambdy, zatem w przypadku gdy asercja przechodzi poprawnie, wiadomość ta nie jest drukowana na ekranie. Zostanie ona wydrukowana dopiero gdy asercja failuje, dlatego na ekranie widzimy wiadomość `Evaluating failure message: 3`.

Powyższy przykład pokazuje, że jeżeli chcemy przekazywać wiadomość jako parametr asercji, należy to robić przy wykorzystaniu **Supplier**, wtedy implementujemy lambdę jako argument wywołania. Oszczędzamy dzięki temu zasoby, jeżeli określenie wiadomości, która jest przekazana byłoby kosztowne.

AssertAll

Wspomnieliśmy, że jeżeli mamy kilka asercji po sobie i którakolwiek z nich zakończy się niepowodzeniem, to pozostałe nie zostaną sprawdzone. Jeżeli natomiast chcemy w takim przypadku wymusić sprawdzenie wszystkich, należy zastosować metodę `assertAll()`. Porównaj ze sobą 2 poniższe wywołania:

Wywołanie bez `assertAll()`

```
Assertions.assertEquals("test1", "test1", createMessage(1));
Assertions.assertEquals("test1", "test2", () -> createMessage(2));
Assertions.assertEquals("test1", "test4", () -> createMessage(3));
```

I jego wynik na ekranie:

```
org.opentest4j.AssertionFailedError: failure message ==>
Expected :test1
Actual   :test2
```

Wywołanie z `assertAll()`

```
Assertions.assertAll(
    () -> Assertions.assertEquals("test1", "test1", createMessage(1)),
    () -> Assertions.assertEquals("test1", "test2", () -> createMessage(2)),
    () -> Assertions.assertEquals("test1", "test4", () -> createMessage(3))
);
```

I jego wynik na ekranie:

```
failure message ==> expected: <test1> but was: <test2>
Comparison Failure:
Expected :test1
Actual   :test2

failure message ==> expected: <test1> but was: <test4>
Comparison Failure:
Expected :test1
Actual   :test4
```

Widać, że jeżeli stosujemy `assertAll()`, to wywołanie nie zostaje przerwane i wykonywane są wszystkie asercje.

Sprawdzanie wyjątków

Możemy oczekiwać, że kod, który wywołujemy w teście zakończy się wyrzuceniem wyjątku. Jak zapisać taką asercję?

```
Throwable exception =
    Assertions.assertThrows(NumberFormatException.class, () -> Integer.parseInt("MyInput"));
Assertions.assertEquals("For input string: \"MyInput\"", exception.getMessage());
```

W pierwszej linijce określamy, że oczekujemy, że fragment kodu `Integer.parseInt("MyInput")` wyrzuci wyjątek `NumberFormatException`. W drugiej linijce natomiast określamy, że wiadomość jaka ma być zawarta w tym wyjątku to: `For input string: "MyInput"`.

AssertJ

Jeżeli powyżej pokazane asercje uznamy za niewystarczające, możliwe jest korzystanie z bibliotek, które dają nam więcej możliwości. Przykładem, w którym takie asercje okażą się niewystarczające będzie sytuacja, gdy chcemy porównać ze sobą 2 obiekty, które składają się z 15 pól, ale chcemy pominąć przy tym porównaniu 2 wybrane. Wtedy sięga się np. po **AssertJ**, którego dokumentację zamieszczam [tutaj](#). Nie chcę natomiast w tym momencie poruszać kwestii wykorzystania tych asercji. Dodam tylko, że są one intuicyjne i w miarę analogiczne do tych domyślnie dostępnych w JUnit.

Wyłączenie testów

Poszczególne testy można wyłączyć przy wykorzystaniu adnotacji. W poprzednich wersjach Junit, było to **@Ignore**, która mogła zostać umieszczona nad klasą z testami lub nad poszczególnymi metodami. W JUnit5 adnotacja ta została zastąpiona adnotacją **@Disabled**. Adnotacja ta daje również możliwość opisanie jako parametr, dlaczego dany test jest wyłączony.

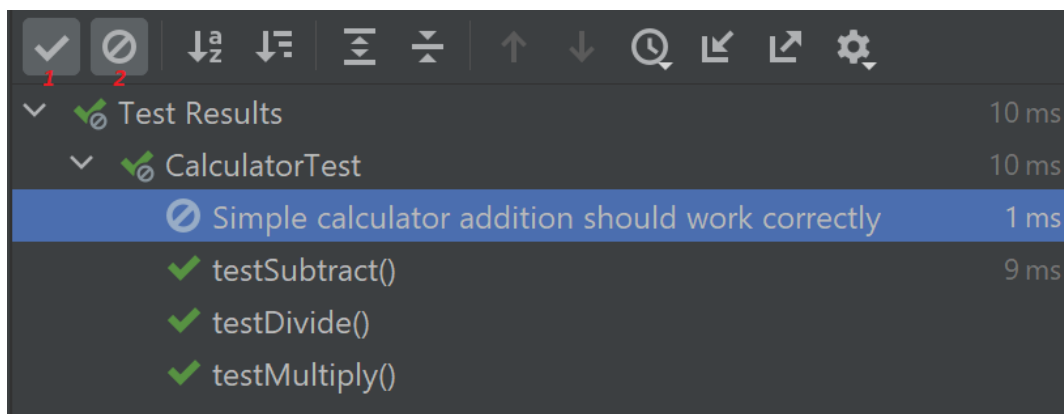


Czy powinno się oznaczać testy jako **@Disabled**? Moim osobistym zdaniem **NIE**. Jeżeli test failuje to jest to oznaka, że albo mamy błąd w kodzie, albo mamy błąd w testach. Niezależnie od przyczyny błędu trzeba poprawić jedno albo drugie, a nie to ignorować 😊.

Przykład oznaczenia metody testowej jako **@Disabled**:

```
@Test
@DisplayName("Simple calculator addition should work correctly")
@Disabled("Example why this test is disabled")
void testAdd() {
    // ...
}
```

Jeżeli teraz uruchomimy całą klasę testową, na ekranie będzie to wyglądało w ten sposób:



Obraz 5. Test oznaczony jako **@Disabled**

Co natomiast oznaczają ikonki określone jako 1 i 2?

- 1 - włącz/wyłącz pokazywanie testów, które zostały zaliczone
- 2 - włącz/wyłącz pokazywanie testów, które są ignorowane

Testy parametrized

Wyobraźmy sobie teraz sytuację, w której mamy kilka testów, które testują ten sam fragment kodu, ale chcemy wywołać je z innymi parametrami i spodziewamy się wtedy innego wyniku. ~~W takiej sytuacji należałoby napisać dla każdego wariantu oddzielną metodę testową.~~ Oczywiście, że nie! Po to są testy parametrized (parametryzowane).

W tym celu musimy dodać zależność do naszego projektu:

Maven

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-params</artifactId>
  <version>5.8.2</version>
  <scope>test</scope>
</dependency>
```

Gradle

```
dependencies {
    // .. pozostałe dependencje
    testImplementation 'org.junit.jupiter:junit-jupiter-params:5.8.2'
}
```

Przykład kodu, który obrazuje w jaki sposób można napisać test **parametrized**:

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;

class CalculatorTest {

    @ParameterizedTest
    @MethodSource(value = "testData")
    void testParametrizedAdding(int[] testData) {
        // given
        int left = testData[0];
        int right = testData[1];
        Integer expected = testData[2];

        // when
        Integer result = Calculator.add(left, right);

        // then
        Assertions.assertEquals(expected, result);
    }

    @SuppressWarnings("unused")
    public static int[][] testData() {
        return new int[][]{{1, 1, 2}, {2, 3, 5}, {9, 8, 17}, {2, 19, 21}};
    }
}
```

Co zostało zmienione? Zamiast adnotacji **@Test** mamy teraz **@ParameterizedTest**. Dodana została adnotacja **@MethodSource(value = "testData")**, która określa nam nazwę metody, która zwraca źródło danych do testu. W tym źródle danych mamy nie tylko parametry wejściowe, ale też spodziewany wynik. Metoda **testData()** zwraca tablicę dwuwymiarową, w której wymiar "zewnętrzny" określa kolejne grupy danych testowych, czyli możemy rozumieć to w ten sposób:

```
public static int[][] testData() {
```

```

return new int[][]{
    {1, 1, 2}, // pierwsza grupa parametrów
    {2, 3, 5}, // druga grupa parametrów
    {9, 8, 17}, // trzecia grupa parametrów
    {2, 19, 21} // czwarta grupa parametrów
};
}

```

Na tej podstawie widać, że test uruchomi się 4 razy, z 4 różnymi grupami parametrów.

W metodzie testowej `testParametrizedAdding(int[] testData)` umieszczona już jest tablica jednowymiarowa. Wynika to z tego, że każda z grup wspomnianych wcześniej reprezentuje już tablicę jednowymiarową i to z nią dany test jest wykonywany. Czyli mamy czterokrotne wykonanie testu z parametrami, które są przekazane w postaci tablicy jednowymiarowej.

W teście następnie określamy, że indeksy 0 i 1 tablicy są naszymi danymi wejściowymi, natomiast indeks 2 oznacza wartość oczekiwaną.

Dodałem też adnotację `@SuppressWarnings("unused")` nad metodą `testData()`, gdyż IntelliJ może pokazywać warning, że metoda ta nie jest używana. W ten sposób oznaczam, że jest i wiem o tym - jest to moje świadome działanie.

Zapis `@MethodSource(value = "testData")` może zostać uproszczony. Zamiast wpisywać ręcznie nazwę metody, która zawiera dane testowe możemy nazwać metodę z danymi testowymi identycznie jak nazywa się nasza metoda testowa. W taki przypadku testy zostaną uruchomione z prawidłowymi danymi testowymi. Przykład:

```

package pl.zajavka;

// importy

class CalculatorTest {

    @ParameterizedTest
    @MethodSource
    void testParametrizedAdding(int[] testData) {
        // ciało testu
    }

    @SuppressWarnings("unused")
    public static int[][] testParametrizedAdding() {
        return new int[][]{{1, 1, 2}, {2, 3, 5}, {9, 8, 17}, {2, 19, 21}};
    }
}

```

Jeżeli teraz uruchomimy test `testParametrizedAdding()`, to na ekranie zobaczymy jego czterokrotne wywołanie z różnymi parametrami.

✓	Test Results	61 ms
✓	CalculatorTest	61 ms
✓	testParametrizedAdding(int[])	61 ms
✓	[1] [1, 1, 2]	58 ms
✓	[2] [2, 3, 5]	1 ms
✓	[3] [9, 8, 17]	1 ms
✓	[4] [2, 19, 21]	1 ms

Obraz 6. Wynik wywołania testu parametrized

Arguments

Przykład poniżej zostanie pokazany dopiero w ćwiczeniu praktycznym, ale w kontekście notatek chciałbym umieścić to tutaj.

W testach parametryzowanych mogą wystąpić sytuacje, gdzie dostarczenie danych testowych w postaci dwuwymiarowej tablicy typu `Object` nie zadziała i będziemy dostawali błędy. W takiej sytuacji należy wykorzystać inną konstrukcję. Przykład zapisu poniżej.

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

import java.util.stream.Stream;

class CalculatorTest {

    @ParameterizedTest
    @MethodSource ①
    void testParametrizedAdding(int left, int right, int expected) { ②
        // given, when
        Integer result = Calculator.add(left, right);

        // then
        Assertions.assertEquals(expected, result);
    }

    @SuppressWarnings("unused")
    public static Stream<Arguments> testParametrizedAdding() { ③
        return Stream.of(
            Arguments.of(1, 1, 2),
            Arguments.of(2, 3, 5),
            Arguments.of(9, 8, 17),
            Arguments.of(2, 19, 21)
        );
    }
}
```

- ① Nadal nie podajemy nazwy metody z danymi testowymi jawnie, tylko musimy pamiętać, że nazwa metody z danymi musi być identyczna jak nazwa metody testowej.
- ② Każdy z argumentów, który parametryzujemy został tutaj rozdzielony jako parametry wywołania testu i będą one przypisywane zgodnie z kolejnością argumentów określoną w metodzie z danymi.
- ③ Wykorzystujemy `Stream`, gdzie typem jest interfejs `Arguments`. Możemy wtedy wykorzystać albo metodę `Arguments.of()` albo `Arguments.arguments()`.

W zapisie powyżej kod nadal zachowa się tak samo, czyli uruchomią nam się testy z parametrami. Który zapis jest czytelniejszy - zostawiam to do Twojej oceny ☺. Ja osobiście najczęściej korzystam ze `Stream<Arguments>`.

Tak jak zostało wspomniane wcześniej, ten przykład zostanie pokazany w rozwiązaniu ćwiczenia praktycznego.



Jeżeli potrzebujesz lepiej zobaczyć jak się to zachowa, pamiętaj zawsze możesz taki Debugować.

Błędy



Jeżeli ktoś dostaje błędy w stylu:

```
java.lang.NoSuchMethodError:
org.junit.jupiter.api.extension.ExtensionContext
    .getConfigurationParameter(java.lang.String, java.util.function.Function)
```

Może to wynikać z niezgodności wersji bibliotek. Niestety, biblioteki mają do siebie, że jeżeli używamy kilku to muszą być ze sobą kompatybilne. Jeżeli natomiast otrzymujesz takie błędy to jak je rozwiązać? Googlować, w praktyce zawsze się to tak kończy.

Logowanie

Wróćmy jeszcze na moment do tematu logowania w zestawieniu z uruchamianiem testów. Stwórz katalog `src/test/resources` i dodaj w nim plik `logback-test.xml`, w którym określisz następującą konfigurację:

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>

    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <root level="WARN">
        <appender-ref ref="CONSOLE"/>
    </root>
```

```
</configuration>
```

Do tego stwórzmy plik `logback.xml` w katalogu `src/main/resources` i dodajmy w nim identyczną konfigurację jak powyższa, tyle, że w `root` zamiast `WARN` napiszmy `INFO`.

Następnie w klasie `Calculator` dodaj logowanie (np. w metodzie `add()`), żebyśmy mieli tam przykładowo taki kod:

```
package pl.zajavka;

import lombok.extern.slf4j.Slf4j;

@Slf4j
public class Calculator {

    public static Integer add(int left, int right) {
        log.debug("DEBUG message");
        log.info("INFO message");
        log.warn("WARN message");
        log.error("ERROR message");
        return left + right;
    }
}
```

Teraz uruchom test `testAdd()` a następnie uruchom metodę `Calculator.add()` normalnie, np. z metody `main()`. Zobacz jaka będzie różnica w logach pokazywanych na ekranie.

Jeżeli przy powyższej konfiguracji uruchomimy kod normalnie (nie z testu), to na ekranie logują nam się wiadomości `INFO`, `WARN` i `ERROR`. Jeżeli uruchomimy natomiast metodę testową to na ekranie logują się tylko wiadomości `WARN` i `ERROR`. Wynika to z tego, że rozdzieliliśmy konfigurację logowania na dwie różne - jedna ma być używana normalnie w kodzie, a druga tylko w testach. Możemy to rozumieć jak 2 classpathy, jeden classpath jest używany tylko w kodzie, a drugi w testach. Tak samo zachowywałyby się inne pliki, które umieścimy w katalogu `src/test/resources` - będą one dostępne w kontekście testów.

Taki podział często jest przydatny w praktyce w sytuacjach, gdy chcemy ustawić inną konfigurację (czy to logowania czy czego innego) podczas normalnego wykonania programu, a inną w przypadku uruchamiania tego samego kodu aplikacji w testach.

Notatki - Testowanie - Zasady

Spis treści

Zasady pisania dobrych testów jednostkowych	1
Nie używaj infrastruktury	1
Testy jednostkowe powinny być wyizolowane	1
Testy jednostkowe powinny być małe	1
Testujmy nie tylko przypadki pozytywne	2
Testy powinny być niezależne od siebie	2
Testy powinny być deterministyczne	2

Zasady pisania dobrych testów jednostkowych

Poniżej chciałbym umieścić kilka reguł, których powinniśmy się trzymać pisząc testy jednostkowe. Oczywiście jest to kwestia filozoficzna, więc prezentuję tutaj moje zdanie.

Nie używaj infrastruktury

Jeżeli będziemy pisali testy jednostkowe i natkniemy się na klasę, która korzysta z bazy danych to zaślepmy ją. W sensie zastąpmy ją zaślepką. Testy jednostkowe mają testować jednostkę, a nie jednostkę i bazę danych. O zaślepieniu (**mockowaniu**) dowiemy się później.

Dodam również, że testy jednostkowe mają być szybkie, a jeżeli zaczniemy uwzględniać bazę danych w testach, to ani nie będą one jednostkowe, ani nie będą szybkie bo dojdzie nam cała komunikacja z bazą danych. Nie będzie to widoczne przy jednym teście, ale przy kilkuset już tak.

Testy jednostkowe powinny być wyizolowane

Trochę wiąże się to z poprzednim punktem. Jednostkowy, oznacza, że testujemy tę konkretną klasę. Czyli jeżeli nasza klasa odwołuje się do innych klas, to te inne klasy powinny zostać zaślepione (**mock**). Oczywiście w praktyce często jest tak, że piszemy testy jednostkowe, które wykorzystują kilka klas ☺ - czyli w sumie są to takie testy integracyjne, ale nazywane są nadal jednostkowymi. Wiadomo, teoria i praktyka.

Testy jednostkowe powinny być małe

Test powinien testować tylko jedną funkcję. Wpływa to później na strukturę klasy testowej, bo jeżeli dobrze opiszemy testy, to będzie wiadomo co konkretnie jest testowane, która metoda lub wycinek kodu. Nie będziemy mieli jednej metody testującej pół naszej aplikacji. Oczywiście cały czas mówię o testach jednostkowych. W przypadku testów integracyjnych, czasem jest naszym założeniem, żeby przetestować pół naszej aplikacji, by zobaczyć jak klasy/moduły integrują się ze sobą.

Testujmy nie tylko przypadki pozytywne

To jest pułapka w którą często wpadają programiści. Jeżeli będziemy testować tylko przypadki pozytywne (innymi słowy nie będziemy uwzględniać żadnych **corner case'ów**), może się okazać, że przy wprowadzaniu zmian, nasza ścieżka pozytywna będzie nadal super działać, ale kod popsuje nam sytuacje brzegowe. Takie które zdarzają się rzadko i są trudne do znalezienia. Czyli nie piszmy testów tylko wprowadzając dane pozytywne i oczekując, że zostanie wszystko dobrze policzone. Stwórzmy też test, który przyjmuje dane dziwne/złe/niewłaściwe i określmy jak wtedy aplikacja ma się zachować.

Testy powinny być niezależne od siebie

Dobłą praktyką jest pisanie testów w taki sposób żeby były one całkowicie niezależne od siebie. Jeżeli przestawimy losowo kolejność wykonania testów, mają one nadal działać prawidłowo. Jeżeli zaczniemy uzależniać testy od siebie (tzn, test kolejny oczekuje, że poprzedni coś zmieni), może to doprowadzić do bardzo dziwnych i nieprzewidywalnych błędów.

Testy powinny być deterministyczne

Zagadnienie to nie zostało omówione na nagraniu, natomiast odniesienie do tego zagadnienia znajdziesz w nagraniach w części praktycznej.

Bardzo dużym problemem w praktyce okazuje się sytuacja gdy testy zachowują się w sposób niedeterministyczny. Czyli jeżeli uruchomimy test i na zmianę otrzymujemy wynik **passed** i **failed**. Takie błędy są często bardzo trudne w diagnozie, więc musimy zawsze się upewnić, czy testy zachowują się w sposób deterministyczny, czyli zawsze tak samo. Inaczej takie zjawisko określa się stwierdzeniem **migotanie testów**. Jeżeli test na przemian przechodzi bądź nie, określa się to stwierdzeniem, że **test migocze**.

Notatki - Testowanie - Coverage

Spis treści

Pokrycie kodu testami	1
Zapewnienie minimalnego pokrycia testami	3
Maven	3
Gradle	5

Pokrycie kodu testami

Wróćmy do definicji klasy klasy `Calculator` jak poniżej:

```
package pl.zajavka;

public class Calculator {

    public Integer add(int left, int right) {
        return left + right;
    }

    public Integer subtract(int left, int right) {
        return left - right;
    }

    public Integer multiply(int left, int right) {
        return left * right;
    }

    public Integer divide(int left, int right) {
        return left / right;
    }
}
```

Jeżeli teraz dodamy metody testowe, tylko do pierwszych 2 metod z klasy `Calculate` (jak poniżej):

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class CalculatorTest {

    private Calculator calculator;

    @BeforeEach
    void beforeEach() {
        calculator = new Calculator();
    }
}
```

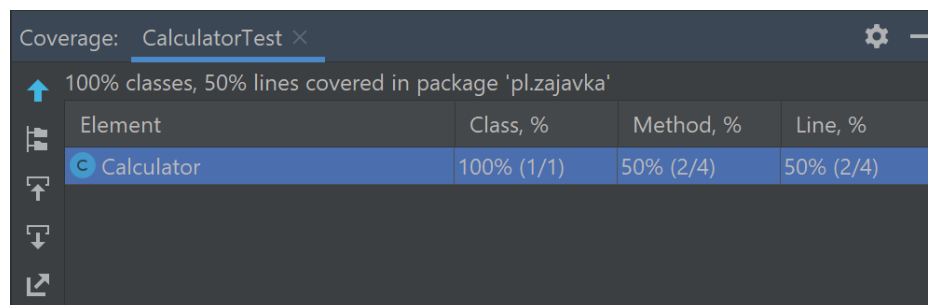
```

@Test
void testAdd() {
    // ciało testu
}

@Test
void testSubtract() {
    // ciało testu
}
}

```

Uruchom teraz te testy z poziomu klasy testowej, ale tym razem wykorzystaj ikonkę tarczy, która jest obok ikonki uruchamiania i ikonki debuggowania. Obok tej ikonki jest napisane **Run 'CalculatorTest' with coverage**. Na ekranie pojawi się wtedy okno, które będzie pokazywało stan pokrycia naszego kodu testami. Jeżeli teraz przeklikamy się przez paczki do `pl.zajavka` to zobaczymy klasę `Calculator` wraz z procentowymi wskazaniem pokrycia kodu testami.



Element	Class, %	Method, %	Line, %
Calculator	100% (1/1)	50% (2/4)	50% (2/4)

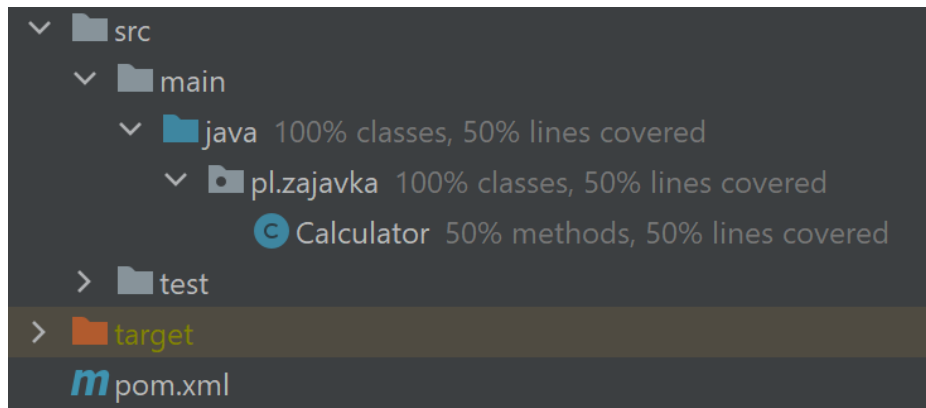
Obraz 1. Pokrycie kodu testami w IntelliJ

Mamy tutaj podział w tabelce na **class**, **method** i **line**. Wartości te oznaczają kolejno:

- **class** - ile klas zostało pokrytych testami - ze wszystkich możliwych klas z wykonywalnym kodem w danej paczce,
- **method** - ile metod zostało pokrytych testami z możliwych metod do wywołania,
- **line** - ile linijek kodu (z wykonywalnych) zostało pokrytych testami (nie są liczone linijki, których nie da się uruchomić). Zobacz, że wypisane są możliwe 4 linijki, które mogą być uruchomione i faktycznie w klasie `Calculator` kod "uruchamialny" jest tylko w 4 linijkach.

Jeżeli w tym momencie chcemy przejść do klasy, która nas interesuje, możemy kliknąć na niej w oknie z pokryciem testami i wcisnąć **F4**.

Ta sama informacja pojawia się teraz na drzewie projektu (po lewej stronie).



Obraz 2. Pokrycie kodu testami na drzewie projektu

Jednocześnie też w samej klasie `Calculator` pojawiają się kolorki mówiące, które linijki kodu zostały wykonane w trakcie działania testu (zielony), a które nie (czerwony).

```

1      package pl.zajavka;
2
3      public class Calcula
4
5      @      public static In
6
7
8
9      @      public static In
10
11
12
13     @      public static In
14
15
16
17     @      public static In
18
19
20     }
21

```

Obraz 3. Pokrycie kodu testami w kodzie źródłowym

IntelliJ na podstawie tego, które linijki kodu zostały wykonane podczas uruchomionych testów, a które nie jest w stanie zliczyć ile faktycznie uruchamialnych linii kodu zostało uruchomionych podczas wykonywanych testów i zliczyć stosunek linii kodu wykonanych do wszystkich możliwych. W ten sposób wyliczane jest pokrycie kodu testami.

Zapewnienie minimalnego pokrycia testami

Maven

Narzędzia takie jak **Maven** albo **Gradle** pomagają nam wymusić minimalny poziom pokrycia kodu testami. Takie sprawdzenie może być uruchomione na etapie buildu, co uniemożliwi nam jego ukończenie, jeżeli nie są spełnione narzucone wymagania. Aby skonfigurować taki minimalny poziom pokrycia kodu testami, możemy to zrobić w poniższy sposób.

```

<project>

  <properties>
    <min.code.coverage>0.50</min.code.coverage>
  </properties>

  <!-- RESZTA PLIKU -->

  <plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.6</version>
    <executions>
      <execution>
        <goals>
          <goal>prepare-agent</goal>
        </goals>
      </execution>
      <execution>
        <id>jacoco-report</id>
        <phase>test</phase>
        <goals>
          <goal>report</goal>
        </goals>
      </execution>
      <execution>
        <id>jacoco-check</id>
        <goals>
          <goal>check</goal>
        </goals>
        <configuration>
          <rules>
            <rule implementation="org.jacoco.maven.RuleConfiguration">
              <element>BUNDLE</element>
              <limits>
                <limit implementation="org.jacoco.report.check.Limit">
                  <counter>LINE</counter>
                  <value>COVEREDRATIO</value>
                  <minimum>${min.code.coverage}</minimum>
                </limit>
              </limits>
            </rule>
          </rules>
        </configuration>
      </execution>
    </executions>
  </plugin>
</project>

```

Dodaliśmy plugin **jacoco**, który umożliwia nam określenie minimalnego progu pokrycia kodu testami. Na ten moment najważniejsze co musimy pamiętać to, że potrzebujemy jednocześnie **surefire** oraz **jacoco**, aby móc uruchamiać testy z **Maven** oraz określić minimalny próg pokrycia testami. Resztę informacji oraz przykłady konfiguracji da się wyszukać w internecie wiedząc jakie pluginy są nam potrzebne.

Po uruchomieniu polecenia `mvn verify`, zostaną uruchomione testy i obliczone pokrycie kodu testami. Tylko, że tym razem robi to plugin, a nie IntelliJ i zlicza to już inaczej, dlatego na ekranie pojawia się

poniższa informacja i build nie przechodzi:

```
[WARNING] Rule violated for bundle testing-maven-examples: instructions covered ratio is 0.43, but
expected minimum is 0.50
```

Jeżeli chcemy zobaczyć dlaczego **jacoco** obliczył pokrycie na poziomie 0.43, a IntelliJ na poziomie 0.5 należy udać się do katalogu `target/site/jacoco` i otworzyć plik `index.html`. Możemy tutaj zobaczyć cały raport pokrycia kodu testami. Jeżeli wejdziemy do paczki `pl.zajavka` i do klasy `Calculator` zobaczymy, że uwzględniona została również linijka z konstruktorem, czego nie zrobił IntelliJ. Stąd też różnica w wynikach. Należy pamiętać o tym, że każde narzędzie jest skonfigurowane w inny sposób. Możemy w tym momencie albo dopisać więcej testów (usunęliśmy 2 z 4, więc ciągle mamy 2) aby przebić próg 50%, lub obniżyć próg. Ale już wiesz, co jest lepszym pomysłem 😊.

Oczywiście możliwych ustawień jest więcej, w tym celu możemy odnieść się do [dokumentacji](#).

Gradle

Stosując **Gradle**, również możemy skorzystać z pluginu **jacoco**. Analogiczna konfiguracja dla **Gradle**:

```
plugins {
    id 'java'
    id 'jacoco'
}

group = 'pl.zajavka'
version = '1.0.0'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}

dependencies {
    // zależności
}

test {
    useJUnitPlatform()
    testLogging {
        events "skipped", "failed"
    }
    finalizedBy jacocoTestReport
}

jacoco {
    toolVersion = "0.8.7"
}

jacocoTestReport {
    dependsOn test
    reports {
        xml.required = false
        csv.required = false
        html.outputLocation = layout.buildDirectory.dir('jacocoHtmlCustom')
    }
}
```

```
}

jacocoTestCoverageVerification {
    violationRules {
        rule {
            limit {
                counter = 'LINE'
                value = 'COVEREDRATIO'
                minimum = 0.5
            }
        }
    }
}

check.dependsOn jacocoTestCoverageVerification
```

Zwróć uwagę, że specjalnie w powyższej konfiguracji wpisałem swój własny katalog, gdzie będzie generowany raport **jacoco**. Katalog ten nosi nazwę **jacocoHtmlCustom**. Możemy oczywiście zmienić o wiele więcej ustawień, niż jest to pokazane w powyższym przykładzie. W tym celu najlepiej jest odnieść się do [dokumentacji](#).

Notatki - Testowanie - TDD

Spis treści

TDD	1
Zalety stosowania podejścia TDD	2
Wady stosowania podejścia TDD	3
Filozofia	3
To co nam dają te testy	3
Refactor	3
Dokumentacja	3
Poprawność	4
Metody void	4
Podsumowanie	4

TDD

~~To Done-Do?~~ **Test Driven Development**. Jest podejściem do programowania, w którym definiujemy przypadki testowe określające co ma robić i jak ma się zachowywać kod, który piszemy, zanim go napiszemy. W praktyce sprowadza się to do tego, że zanim napiszemy kod, piszemy testy. Początkowo żaden z takich testów nie zakończy się sukcesem, ale stopniowo jak będziemy dopisywać kolejne funkcjonalności, testy zaczynają świecić się na zielono, aż osiągniemy stan, gdzie wszystkie testy zakończą się sukcesem.

W tym podejściu development zaczyna się od wymyślenia jakie funkcjonalności mają być dostarczone z perspektywy podziału na metody testowe. Inaczej na to patrząc, patrzymy jakie testy chcielibyśmy mieć w swoim kodzie, które potem mają działać. Jeszcze inaczej mówiąc, pisząc testy możemy zdefiniować wymagania, jakie ma spełniać nasz kod. Następnie implementujemy takie wymagania faktycznie ten kod pisząc. Inaczej używa się czasem określenia **Test First Development**, z racji, że testy są pisane najpierw.



Obraz 1. Cykl Test Driven Development

Czyli podchodząc do tematu krokowo, pisałibyśmy program w taki sposób:

- Dodaj test,
- Uruchom wszystkie testy i zobacz czy któryś nie przechodzi,
- Dopisz kod, aby test przechodził,
- Uruchom testy i jeżeli wszystkie przechodzą możesz refactorować swój kod,
- Powtórz.

Jeżeli chodzi o filozofię stosowania takiego podejścia, wyrób ją samodzielnie. Aby było prościej, postaraj się napisać projekt zaczynając od testów i zobacz jak będzie Ci to szło.

Poniżej umieszczam zalety i wady stosowania **TDD**, natomiast wiele z poniższych kwestii można poruszyć przy zadaniu sobie pytania: "Czy w ogóle pisać testy?" - Moja jednoznaczna odpowiedź brzmi **TAK**.

Zalety stosowania podejścia TDD

- z czasem jesteśmy w stanie osiągnąć stan, w którym mamy kod wraz z zestawem testów, które są w stanie automatycznie ten kod sprawdzać, oczywiście pisząc w sposób odwrotny (czyli najpierw kod, potem testy) w pewnym momencie również osiągniemy ten stan,
- pomaga zrozumieć kod zanim go napiszemy, gdyż myślimy o tym jak różne fragmenty kodu mają ze sobą rozmawiać,
- pomaga rozbić kod na mniejsze części, gdyż zależy nam na tym, aby metody testowe były jak najmniejsze. Wtedy każda metoda faktycznie realizuje swoją funkcję,
- daje nam pewność siebie w momencie gdy zaczynamy coś w kodzie zmieniać. W praktyce często zmiana kodu, modyfikacja czy refactor powoduje, że możemy bardzo łatwo wprowadzić błędy. Często jest niemożliwe sprawdzenie wszystkich przypadków, które mogliśmy popsuć. Dlatego lepiej jest mieć automat, który sprawdza popełniane przez nas błędy,
- w przypadku pracy zespołowej, jeżeli najpierw napiszemy wspólnie testy to łatwiej jest podzielić

pracę, ustalić kto potem ma implementować różne fragmenty systemu,

Wady stosowania podejścia TDD

- stosując podejście TDD zaczniemy mieć problem jeżeli często zmieniane są wymagania dotyczące tego co faktycznie kod przez nas pisany ma robić. Wyobraź sobie, napisaliśmy już kilka testów, podzieliliśmy to w głowie, zaczynamy pisać kod i przychodzi mail, że zaraz jest spotkanie, na którym się dowiesz, że połowa z tego co zostało przez Ciebie wymyślone jest do wyrzucenia. Więc znowu poprawiasz testy, chcesz zacząć pisać kod i dowiadujesz się, że zmienione zostały wymagania. W takim przypadku znacznie wydłuża to czas developmentu,
- możemy również wprowadzić błędy w samych testach, wtedy napiszemy błędny kod, który powoduje, że błędne testy przechodzą,
- takie podejście jest często ciężko obronić przez biznesem (mówię z praktyki). Bo z perspektywy biznesu, dowozimy funkcjonalność o wiele dłużej, niż byśmy to robili nie pisząc testów wcale. Natomiast jest to perspektywa bardzo krótkoterminowa, bo w długiej perspektywie zapewniamy sobie możliwość popełnienia mniejszej ilości błędów. Ten punkt w sumie jest niezależny od tego czy piszemy najpierw testy, czy kod. Pomaga tutaj jak cały zespół ma spójne podejście i wszyscy mocno naciskają na pisanie testów uzasadniając dlaczego ma to sens.

Filozofia

To może na koniec skupmy się jeszcze na filozofii dotyczącej testów. Zapoznaj się z poniższymi dywagacjami ☺.

To co nam dają te testy

Refactor

Bezpieczeństwo - to przede wszystkim. Gdy masz już napisane testy do swojej aplikacji (tutaj nie ma znaczenia, czy w podejściu TDD, czy inaczej) możesz bezpiecznie podejść do jakichkolwiek modyfikacji kodu źródłowego. Czemu bezpiecznie? **Bo "czuwa" nad Tobą magiczny sprawdzacz**, który weryfikuje czy zmiana jednej linijki, nie popsuje Ci zachowania całej aplikacji. Testy w ten sposób pomagają w refactoringu kodu. Jeżeli jakiś fragment kodu jest dobrze przetestowany i pokryte są przypadki użycia (najlepiej wszystkie), możesz spokojnie przejść do refactorowania. Co więcej, jeżeli czytasz jakiś fragment kodu i dochodzisz do wniosku, że przydałby się refactor, warto jest zacząć od napisania testów, jeżeli dany fragment kodu takich testów nie posiada. Ponownie podkreślę - daje Ci to **bezpieczeństwo w Twoich działaniach**.

Dokumentacja

Wyobraź sobie, że przychodzisz do nowego projektu i chcesz się zapoznać z kodem źródłowym. Czytasz i czytasz, ale dalej nie rozumiesz, jak to działa, bo nie wiesz jakiego rodzaju dane będą przepływały przez dany fragment kodu. Co w takiej sytuacji? **Szukasz testów, one pomogą Ci zrozumieć pełny obraz wywołania danego fragmentu kodu**. Dlatego właśnie jednym z najlepszych rodzajów dokumentacji są dobre testy. Z nich można wyczytać, jakie są dane wejściowe dla danego przypadku i jakie są oczekiwane rezultaty. Co więcej, możesz takie testy debuggować i zobaczyć jak dana linijka kodu

zachowuje się dla różnych danych, jakie wartości będzie przyjmowała zmienna itp.

Jeżeli chcesz pomóc innym i *sobie z przyszłości*, pisz dobre testy. Uwierz mi, że po czasie zapominasz, jak napisany przez Ciebie kod działa i co tak naprawdę miałeś/miałaś na myśli. Dobre testy są wtedy jak skarb.

Poprawność

Ten podpunkt zrozumiesz solidnie dopiero po zaliczeniu kilku takich sytuacji, ale i tak go napiszę ☺. Jeżeli zaczniesz pisać testy do istniejącego już kodu, to bardzo szybko nastąpi weryfikacja, czy dany fragment kodu jest dobrze zaprojektowany. Co mam na myśli? Jeżeli kod jest źle zaprojektowany, to może się okazać, że nie da się go łatwo przetestować, albo czasami w ogóle nie da się go przetestować. Wtedy najczęściej trzeba robić jakieś fikołki, żeby najpierw zmienić ten kod (tu jest ryzyko, że zmienisz logikę zawartą w tym kodzie, bo nie masz przecież testów), a dopiero potem dopisujesz testy. Opcja druga to zaczynasz od TDD, najpierw piszesz testy, a potem usuwasz całą implementację i piszesz ją na nowo. Takie kwestie to już jest kwestia przypadku, ale można podsumować ten podpunkt w taki sposób, że testowanie kodu pomaga nam trzymać się pewnych wzorców (o wzorcach będzie w dedykowanym warsztacie), bo testy w pewnym sensie to na nas wymuszają.

Metody void

A co z testami jednostkowymi metod zwracających `void`? Co wtedy?

Metody `void` (jak już wiesz) nic nie zwracają. Mogą natomiast robić inne rzeczy:

- zmienić stan obiektu przekazywanego jako argument metody,
- wyrzucić wyjątek w trakcie wykonania takiej metody,
- logować przebieg działania aplikacji - to już jest trochę bardziej skomplikowane, ale można w testach weryfikować, czy jakaś informacja została zalogowana na ekranie.

W takich przypadkach również można przeprowadzać testy takich metod.

Podsumowanie

Dodam jeszcze na koniec, że materiał jest tak podzielony, żeby ten warsztat dawał namiastkę testowania i był tylko wprowadzeniem. Do tematyki testów będziemy wracać jeszcze wielokrotnie w kolejnych warsztatach. Jeżeli chcesz sprawdzić gdzie, zapoznaj się z zakładką 'Lista warsztatów'.