TESTOWANIE OPROGRAMOWANIA

BARTOSZ ANDREATTO

Programista w Banku Pekao S.A z bogatym doświadczeniem w zakresie technologii back-end'owych wykorzystujących wirtualną maszynę Javy. Pasjonat rozwiązań opartych na ekosystemie Spring oraz rozwiązań Oracle. Certyfikowany programista Java. Uwielbia nauczać oraz dzielić się wiedzą.

LinkedIn: https://www.linkedin.com/in/bartosz-andreatto-02b03413a/

e-mail: bandreatto@gmail.com

ZAKRES SZKOLENIA

- **Wprowadzenie do testowania oprogramowania**
- Cechy dobrych testów (np. zasada **FIRST**)
- **JUnit**: struktura testu, metody cyklu życia, asercje, własne asercje (asercje biznesowe)
- Biblioteka matcherów (**AssertJ**)
- Wprowadzenie do **TDD**
- **Testy parametryzowane**
- **Testowanie wyjątków**
- Mockito*

ORGANIZACYJNIE

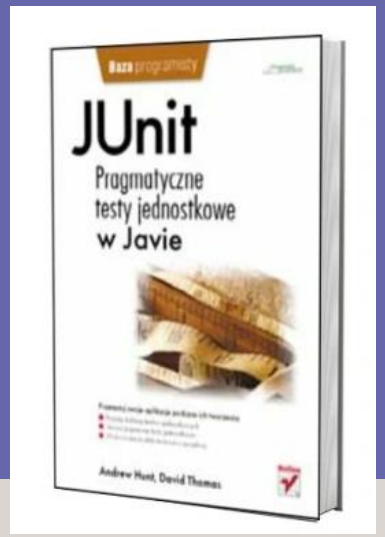
- Repozytorium GIT:

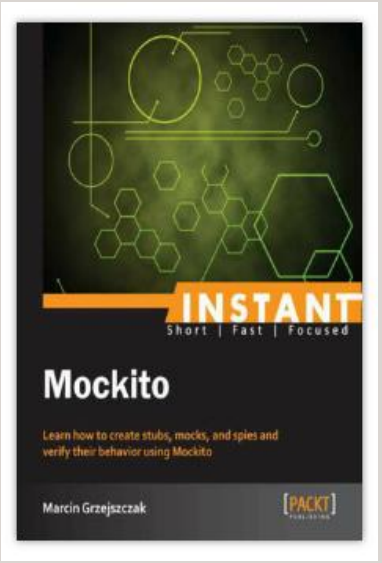
  https://github.com/bandreatto/sda/tree/ZDJAVApol140
- 14 (**2 x 7**) godzin
- Przerwy:

  - … ?

LITERATURA

- **JUnit. Pragmatyczne testy jednostkowe w Javie**, Andy Hunt, Dave Thomas
- **Testowanie aplikacji Java za pomocą Junit**, Radosław Sokół
- **Instant Mockito**, Marcin Grzejszczak





TESTOWANIE

Testowanie wiąże się głównie ze sprawdzeniem, czy dana „rzecz" działa zgodnie z założeniami. Można powiedzieć, że testowanie jest nieodłączną formą naszego życia.

Nie testujemy tylko rzeczywistej aplikacji, ale **testujemy** również czysty „**kod**", aby sprawdzić, czy **przestrzegamy standardów kodowania**, czy nie wprowadzamy kodu regresji. Daje nam to **szybką informację zwrotną** i szansę na reakcję, zanim oddamy kod do gałęzi deweloperskiej.

POTRZEBA TESTÓW

Testowanie daje nam szansę aby:

- **oszczędzić czas** programisty, testera i całego projektu,
- **oszczędzić budżet** projektu,
- nasz kod był **zgodny ze standardami** kodowania,
- dobrze napisane testy mogą **opisywać historyjkę użytkownika** i będą **łatwe do zrozumienia** dla programisty, testera i innych interesariuszy projektu.

RODZAJE TESTÓW

Testy jednostkowe:

- Weryfikują działanie pojedynczych części kodu (najczęściej metody).
- Szybkie i łatwe do zautomatyzowania.
- Zazwyczaj uruchamiane przy każdej zmianie kodu źródłowego

Testy integracyjne:

- Weryfikują działanie kilku modułów aplikacji na raz.
- Stosunkowo długi czas wykonywania.
- Wiele możliwych punktów awarii.

RODZAJE TESTÓW

Zagadnienie	Test jednostkowy	Test integracyjny
Zależności	Testowany pojedynczy element kodu	Testowane wiele zależności
Punkty awarii	Jeden potencjalny punkt awarii (metoda)	Wiele potencjalnych punktów awarii
Szybkość działania	Bardzo duża	Potencjalnie długi czas działania ze względu na np. dostęp do bazy danych
Konfiguracja	Brak dodatkowej konfiguracji	Test może wymagać konfiguracji (np. połączenie do bazy danych)

RODZAJE TESTÓW

Testy systemowe:

Testy systemowe przeprowadzamy, gdy elementy systemu zostały ze sobą zintegrowane. Na tym etapie można wykonać test **e2e** (end to end)
- czyli **przejście całościowe procesu**. Na przykład funkcjonalność rejestracji konta powinna zapisać użytkownika w bazie danych oraz wysłać e-mail z potwierdzeniem rejestracji. **Przetestowanie całego takiego procesu to test e2e**.

RODZAJE TESTÓW

Testy akceptacyjne:

Testy oprogramowania, których celem **nie jest wykrycie błędów**, a jedynie uzyskanie formalnego potwierdzenia wykonania oprogramowania **odpowiedniej jakości** (zgodnie ze specyfikacją).

RODZAJE TESTÓW - PYTANIA

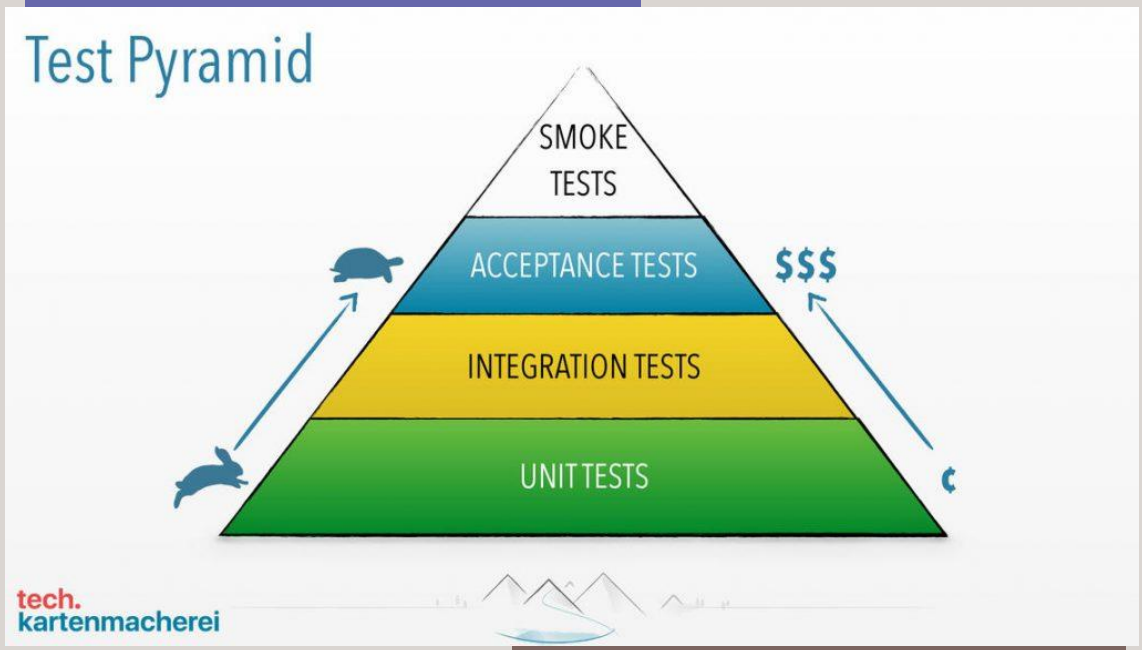
- Jakie są rodzaje testów automatycznych?
- Który rodzaj jest najszybszy?
- Który rodzaj najlepiej bada wymagania biznesowe?
- Kto pisze które testy?
- Które rodzaje testów pozwalają najdokładniej zbadać warunki brzegowe i logikę kodu?

PIRAMIDA TESTÓW

Piramida testów to rodzaj układu współrzędnych, w którym oś pozioma to **liczba przypadków testowych** (testów), a oś pionowa to **czas wykonania i koszt utrzymania** (im wyższy, tym wolniej i drożej).

Piramida testów mówi nam, że:

- **najwięcej jest testów jednostkowych** (są najtańsze i najszybsze)
- **jest coraz mniej testów na wyższych poziomach** (są droższe w utrzymaniu i wolniejsze)



CECHY DOBRYCH TESTÓW

Testy jednostkowe, **tak samo jak implementowane oprogramowanie, również tworzymy zachowując dobre praktyki**. Kolejno zostaną przedstawione najważniejsze **cechy dobrych testów**, które powinniśmy stosować.

FIRST

Szybkość (**Fast**) - powinny wykonywać się szybko, aby nie wydłużać zbytnio czasu wykonania procesu budowania oraz nie czekać zbyt długo na wyniki testów.

FIRST

Niezależne (**Isolated** / **Independent**) - testy powinny być od siebie niezależne (odizolowane). Uruchomienie jednego testu, nie powinno mieć wpływu na stan innych testów w tzw. **test suite** (oznaczającym pakiet testowy, np. testy w obrębie jednej klasy testującej).

FIRST

Powtarzalne (**Repeatable**) - powinny być powtarzalne na każdym środowisku. To znaczy, że nie powinny mieć zależności z innymi systemami np. bazą danych. Konfiguracja innych systemów nie powinna mieć wpływu na powtarzalność wykonywania testów na różnych środowiskach. Jeśli test zostanie uruchomiony np. 10 razy, to tyle samo razy powinniśmy otrzymać ten sam wynik. Wykorzystywanie klasy Random, poleganie na kolejności elementów w kolekcjach nieuporządkowanych, czy korzystanie z rzeczywistych timestamp'ów może powodować problemy ze stałością wyników testów.

FIRST

Samosprawdzające (**Self-checking**) - stwierdzające czy test przeszedł lub nie (brak ręcznej interpretacji).

FIRST

Kompletne (**Timely** / **Thorough**) — pisane razem z testem produkcyjnym. Powinniśmy sprawdzać przypadki testowe podając wartości dające wyniki pozytywne, negatywne oraz korzystając z wartości granicznych. Dzięki czemu testy będą dokładniej sprawdzać napisany kod.

POZOSTAŁE DOBRE PRAKTYKI

- **Pojedyncza odpowiedzialność** – jeden test bada jedną funkcjonalność
- **Second Class Citizens** – kod testowy nie jest kodem drugiej kategorii. Należy o niego dbać tak samo, jak o kod produkcyjny.

JUNIT

JUnit jest najpopularniejszym narzędziem służącym do **tworzenia testów jednostkowych** oprogramowania pisanego w języku Java.

JUNIT - WPROWADZENIE

Testy najczęściej umieszczamy w ścieżce **src/test/java**. Klasa testująca powinna mieć nazwę **NazwaKlasyTest.java** Na przykład, jeśli w naszej aplikacji, w paczce **pl.sdacademy.calculations**, znajduje się klasa **Calculator.java**, to klasa testująca powinna nazywać się **CalculatorTest.java** i również znajdować się (choć nie jest to wymagane) w paczce o tej samej nazwie, tzn. **pl.sdacademy.calculations**.

JUNIT - ZALEŻNOŚCI

Aby móc korzystać z biblioteki JUnit w naszych testach, musimy do naszego projektu opartego o **Mavena**, w pliku pom.xml w sekcji **dependencies** dodać następujące zależności:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.9.2</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.9.2</version>
    <scope>test</scope>
</dependency>
```

JUNIT – STRUKTURA TESTU

```java
package pl.sdacademy.calculations;

public class Calculator {

    public double add(double first, double second) {
        return first + second;
    }
}


package pl.sdacademy.calculations;

import org.junit.jupiter.api.Test;

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

class CalculatorTest {

    @Test
    void shouldAddTwoNumbers() {
        // given
        double valueA = 5.2;
        double valueB = 3.1;
        Calculator calculator = new Calculator();

        // when
        double actualResult = calculator.add(valueA, valueB);

        //then
        assertEquals(8.3, actualResult);
    }
}
```

JUNIT - ZALEŻNOŚCI

Na podstawie przykładu zauważmy, że:

- **test jest metodą o dowolnej nazwie**, nad którą musimy użyć adnotacji Test
- **test nigdy nic nie zwraca** (wykorzystujemy słowo kluczowe void)
- test, o ile nie jest testem parametryzowanym**, nie posiada żadnych argumentów wejściowych**
- test zawsze **testuje tylko jeden przypadek**
- **test składa się trzech sekcji**, które najczęściej są od siebie oddzielone pustymi liniami.

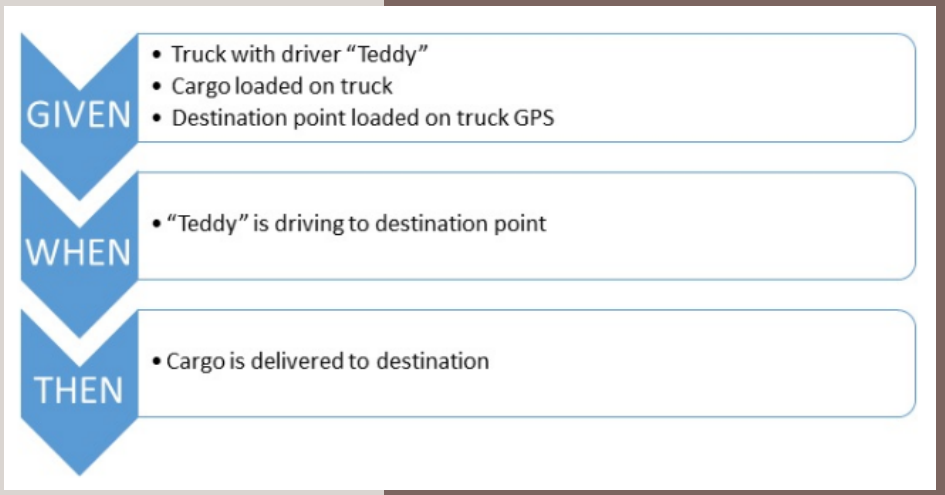
JUNIT - SEKCJE

- **given**, w której definiujemy warunki początkowe testu (tzn. przygotowujemy wszystkie potrzebne obiekty do testów)
- **when**, która najczęściej składa się z jednej linijki - wywołania testowanej metody
- **then**, w której wykorzystujemy tzw. asercje, które sprawdzają oczekiwania z rzeczywistymi wynikami.

```java
@Test
public void shouldDeliverCargoToDestination() {
    // given
    Driver driver = new Driver("Teddy");
    Cargo cargo = new Cargo();
    Position destinationPosition = new Position("52.229676", "21.012228");
    Truck truck = new Truck();
    truck.setDriver(driver);
    truck.load(cargo);
    truck.setDestination(destinationPosition);

    // when
    truck.driveToDestination();

    // then
    assertEquals(destinationPosition, truck.getCurrentPosition());
}
```



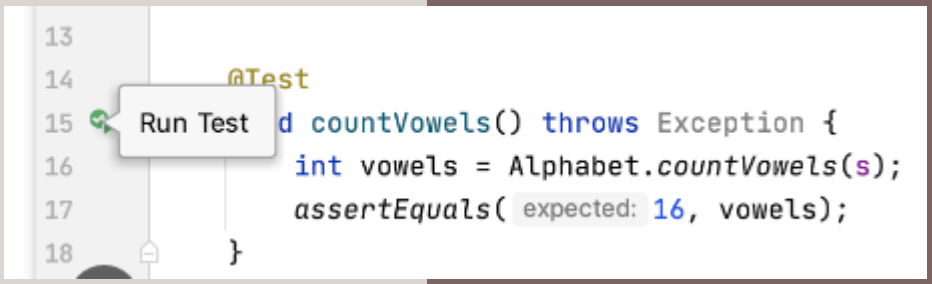
JUNIT – STRUKTURA TESTU

UWAGA: Klasy testowe, metody reprezentujące testy oraz metody cyklu życia testów mogą mieć **dowolny modyfikator dostępu**, jednak najczęściej jest on w ogóle pomijany (tzn. wykorzystywany jest **package-private**).
W starszej wersji JUnit (np. 4.13), wszystkie te elementy muszą być publiczne.

JUNIT – URUCHAMIANIE TESTU

Stworzone testy można uruchamiać na kilka sposobów. Zazwyczaj podczas pisania kodu, test taki możemy **uruchomić bezpośrednio z naszego IDE**. IntelliJ, obok każdego z testów i definicji klasy testowej, posiada menu kontekstowe pozwalające je uruchomić.

Jeżeli nasz projekt oparty jest o **Mavena**, to testy możemy uruchomić w fazie test. Testy będą również uruchamiane, jeżeli wykonamy komendę **mvn clean install**.



JUNIT – ASERCJE

Asercje są instrukcjami, których celem jest **potwierdzenie, że uzyskane obiekty w teście spełniają pewne założenia**, czyli ostatecznie sprawdzają poprawność wykonania testu. Asercje są **statycznymi metodami z klasy org.junit.jupiter.api.Assertions**, które pozwalają na porównanie wyników działania określonej funkcji z oczekiwaną wartością. Ich wywołania powinny znaleźć się na końcu testu, tzn. w sekcji **then**.

JUNIT – ASERCJE PODSTAWOWE

- **assertEquals(expected, actual)** - porównuje obiekty za pomocą metody equals. Oczekiwana wartość zawsze powinna być pierwszym argumentem.
- **assertNotNull(object)** - sprawdza, czy dany obiekt nie jest równy null.
- **assertNull(object)** - sprawdza, że obiekt jest równy null.
- **assertTrue(value)** - oczekuje, że wartość boolean przekazana jako argument jest równa true.
- **assertFalse(value)** - podobnie jak wyżej, oczekuje, że argument będzie równy false.

JUNIT – ASERCJE PODSTAWOWE

- **assertSame(o1, o2)** - oczekuje, że referencje są takie same (tzn. porównuje obiekty za pomocą ==).
- **assertNotsame(o1, o2)** - sprawdza, że referencje są różne.
- **assertFail()** - która natychmiast kończy test niepowodzeniem. Wykorzystywana do np. testowania wyjątków metodą try catch.

JUNIT – ASERCJE ZAAWANSOWANE

- **assertArrayEquals(expectedArray, actualArray)** - sprawdza równość i kolejność wszystkich elementów w obu tablicach.
- **assertIterableEquals(expectedIterable, actualIterable)** - sprawdza równość i kolejność wszystkich elementów w obiektach implementujących interfejs Iterable. Interfejs ten jest implementowany przez interfejs Collection, stąd asercję tę możemy używać z dowolnymi kolekcjami.

JUNIT – ASERCJE ZAAWANSOWANE

- **assertLinesMatch(expected, actual)** - sprawdza, za pomocą pewnego algorytmu i ewentualnie wyrażeń regularnych, czy dwie listy Stringów zawierają takie same elementy.
- **assertTimeout(timeout, executable)** - sprawdza, czy kod wykonuje się szybciej, niż wartość argumentu timeout. Zadanie do wykonania, zawsze musi się zakończyć zanim przejdziemy do kolejnej linii kodu.
- **assertTimeoutPreemptively()** - działa podobnie jak assertTimeout, ale zadanie executable nie zostanie zakończone w przypadku, gdy nastąpi timeout.

JUNIT – ASERCJE GRUPUJĄCE

Test w sekcji **then** może **wykonać dowolną ilość asercji**. W przypadku, gdy takich asercji będzie kilka i niepowodzeniem zakończy się jedna z nich, **kolejne asercje nie zostaną wykonane**. Jeżeli chcemy, aby w teście zawsze wykonały się wszystkie asercje, musimy je wykonać wewnątrz asercji **assertAll(...)**. Asercja ta wykorzystuje **varargs**, oczekując obiekt typu **Executable**. Obiekt ten możemy zaimplementować za pomocą **bezargumentowej lambdy**.

JUNIT – ASERCJE GRUPUJĄCE

```
assertAll(
  () -> assertEquals(expectedObject, actualObject, "Assertion 1 failed"),
  () -> assertTrue(actualResult, "Assertion 2 failed")
);
```

W powyższym przykładzie, jeżeli obie asercje zakończą się niepowodzeniem, to na ekranie zobaczymy:

```
Assertion 1 failed
Assertion 2 failed
```

JUNIT – ADNOTACJE

JUnit pozwala na pisanie testów, za pomocą dodania adnotacji **@Test** do metody. Tworząc wiele testów dla jednej klasy, bardzo możliwe, że nasz kod testowy będzie zawierał **jakieś powtórzenia**.

W celu uniknięcia takich powtórzeń JUnit pozwala zdefiniować tzw. **metody cyklu życia testu**. Metody te, nad swoją sygnaturą muszą mieć pewną adnotację:

@BeforeEach

@AfterEach

@BeforeAll

@AfterAll

JUNIT – ADNOTACJE

Wszystkie metody cyklu życia testu:

- **nie zwracają żadnego obiektu**
- **mogą mieć dowolną nazwę***
- **nie przyjmują argumentów***
- **nie muszą mieć zdefiniowanego modyfikatora dostępu**
- są **opcjonalne** i możemy je zdefiniować **wielokrotnie** (tzn. jedna klasa testowa może nie mieć metody z adnotacją @AfterEach i mieć kilka metod z adnotacją @BeforeEach)
- mogą znaleźć się w **dowolnym miejscu w klasie***

@BEFOREEACH

Metoda z adnotacją **@BeforeEach** zostanie wywołana **przed każdą metodą testową**. Celem tej metody, jest przygotowanie obiektów dla każdego z testów.

@AFTEREACH

Metoda z adnotacją **@AfterEach** zostanie wywołana **po każdej metodzie testowej**. Służy głównie do usuwania danych lub przywracania stanu początkowego po każdym teście.

@BEFOREALL

Metoda cyklu życia testów, oznaczona adnotacją **@BeforeAll** musi być metodą statyczną. Jest ona **wywołana raz, na początku wykonywania testów w danej klasie**. Służy głównie do ustawiania wartości obiektów, które będą wykorzystywane w testach, ale nie będą w nich modyfikowane lub są kosztowne (np. czas tworzenia w kontekście czasu), np. nawiązywania połączenie z bazą danych.

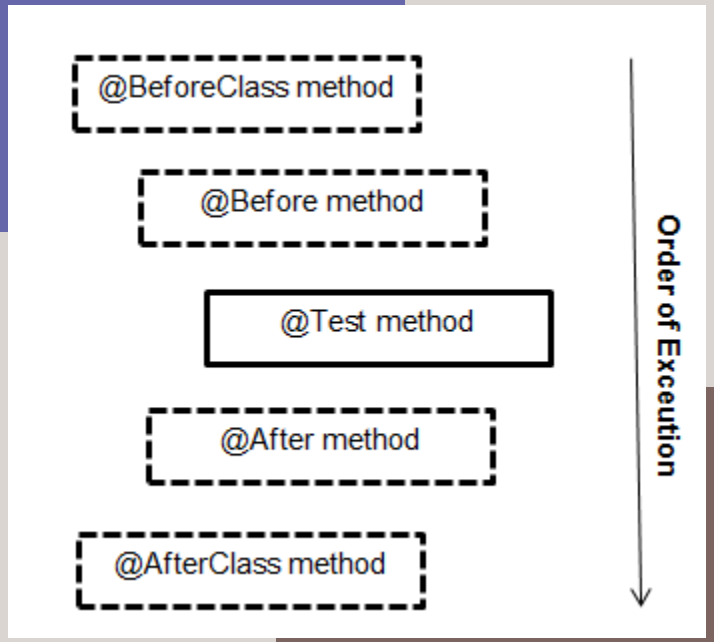
@AFTEREALL

Podobnie jak @BeforeAll, metoda oznaczona adnotacją **@AfterAll musi być statyczna**. Zostanie ona wywołana **raz, po wykonaniu testów w danej klasie**. Służy m.in., do usuwania obiektów, które były wspólne dla wszystkich testów, jak np. zamykanie połączenia z bazą danych.

JUNIT - ADNOTACJE

UWAGA: Jeżeli masz do czynienia z **JUnitem 4**, adnotacje cyklu życia testów mają nieco inne nazwy. Są to:

- **@Before**
- **@After**
- **@AfterClass**
- **@BeforeClass**.



JUNIT – ADNOTACJE DODATKOWE

@DisplayName - do klasy testującej oraz metody testującej możemy dodać własną nazwę, która będzie wyświetlana podczas uruchamiania testu oraz w raportach z testów.

@Disabled - oznacza metodę lub klasę, która nie powinna być uruchamiana w trakcie testów. Przyjmuje ona argument, w którym możemy dodać informację, dlaczego test ten jest wyłączony. Powodów wykorzystania tej adnotacji może być wiele.

ASSERTJ

AssertJ to biblioteka (tzw. **biblioteka matcherów**), której głównym celem jest **poprawienie czytelności oraz ułatwienie utrzymywania testów** w języku Java. Zwiększa czytelność i w wielu przypadkach dba o zwięzłość kodu. **AssertJ nie zastępuje biblioteki JUnit**, a jest jej uzupełnieniem. W sekcji **then** testów, zamiast wykorzystywać asercje z biblioteki JUnit, wykorzystamy te **z biblioteki AssertJ**.

ASSERTJ

Aby móc korzystać z biblioteki AssertJ w testach, musimy do projektu opartego o Mavena, w pliku **pom.xml** w sekcji **dependencies** dodać następującą zależność:

```
<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.24.2</version>
    <scope>test</scope>
</dependency>
```

ASSERTJ - SKŁADNIA

Dostęp do matcherów z biblioteki AssertJ daje nam metoda statyczna **assertThat** z pakietu **org.assertj.core.api.** W zależności od typu obiektu, jaki zostanie do niej przekazany, **otrzymujemy inny zestaw matcherów**, jaki możemy wykorzystać.

```
public class Calculator {
    public static int add(int x, int y) {
      return x + y;
    }
}
```

```
@Test
void shouldAddTwoNumbers() {
    // given
    int a = 1;
    int b = 3;

    // when
    int result = Calculator.add(a, b);

    // then
    assertThat(result).isEqualTo(4);
}
```

ASSERTJ – SKŁADNIA ŁAŃCUCHOWA

AssertJ zapewnia **składnię łańcuchową**, dzięki czemu możemy **dla pojedynczego obiektu sprawdzać kilka warunków jednocześnie**. Składnia ta jest zazwyczaj czytelniejsza niż wykorzystanie wielu asercji z biblioteki JUnit. Ponadto **komunikaty o błędach z matcherów, są zazwyczaj dokładniejsze niż te z wbudowanych w JUnit asercji.**

```
@Test
void shouldCreateMalePerson() {
    final Person person = personFactory.createPerson(MALE_NAME, SURNAME);

    assertThat(person).isNotNull()
        .extracting(Person::getPrefix)
        .isNotNull()
        .isEqualTo(MALE_PREFIX);
}
```

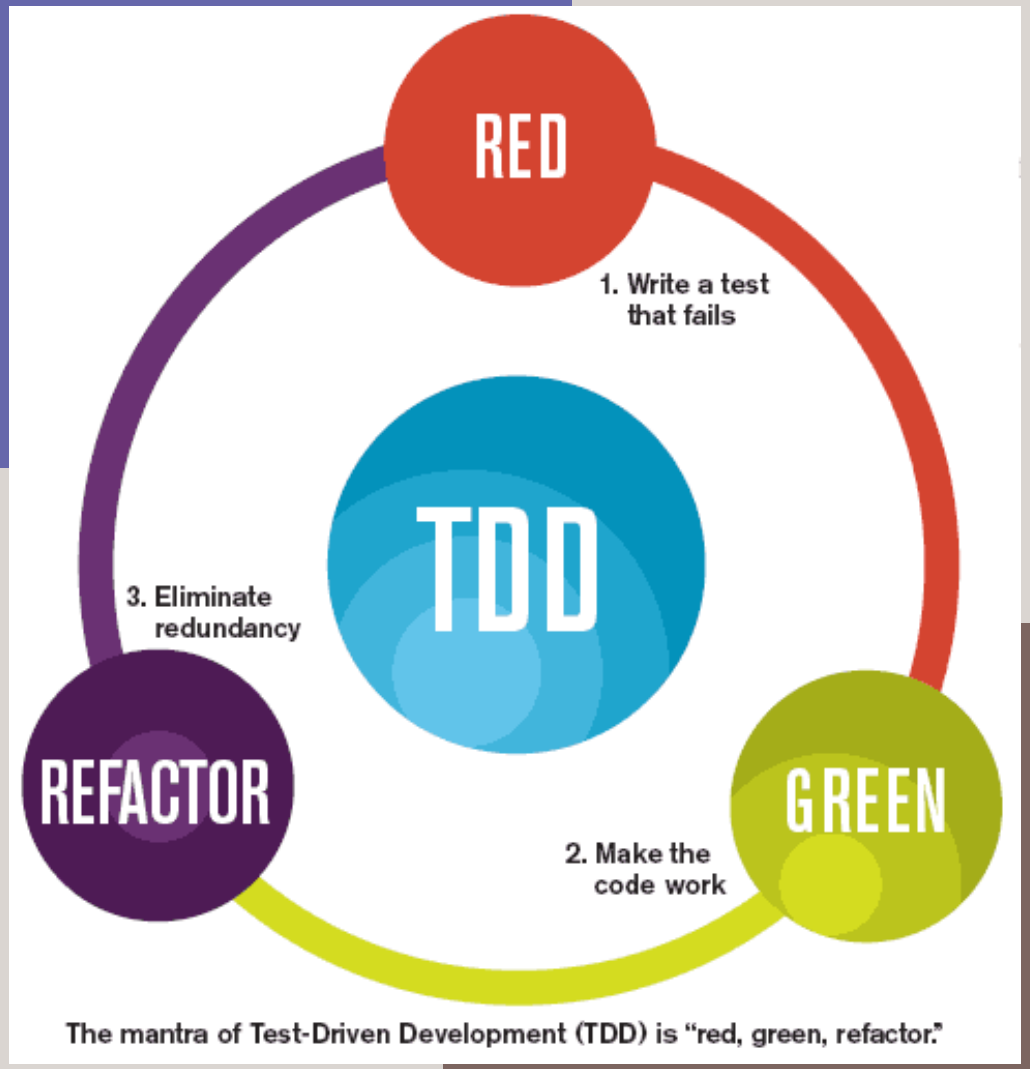
PODEJŚCIE DO TESTOWANIA

TEST-DRIVEN DEVELOPMENT

TDD (**Test-Driven Development**) to technika tworzenia oprogramowania, w której **najpierw piszemy testy dla nieistniejącej funkcjonalności**, a następnie ją **wdrażamy**. TDD nie jest jedyną prawidłową techniką tworzenia oprogramowania. Możesz pisać testy równolegle z logiką biznesową lub po wdrożeniu kodu, ale wtedy nie jest to już TDD. W Test-Driven Development **zawsze piszemy testy przed wdrożeniem**. TDD było pierwotnie częścią **eXtreme Programming**. Teraz jest to samodzielna technika. Jego twórcą jest **Kent Beck** (twórca programowania ekstremalnego i jeden z twórców Manifestu Agile).

TDD – CYKL ŻYCIA

- **Pisanie i uruchamianie testu (faza czerwona)** - pierwszym krokiem jest napisanie testu.
- **Napisanie minimalnego kodu do zdania testu (faza zielona)** - w tej fazie piszemy kod odpowiedzialny za logikę biznesową.
- **Refaktoryzacja kodu (faza refaktoryzacji)** - w tej fazie refaktoryzujemy kod tak, aby był zgodny ze wszystkimi standardami dobrego programowania. Dotyczy to również testów.



TDD – ZALETY

Dla programisty:

- wyższy komfort pracy - dotyczy również istniejącego kodu,
- dokumentacja funkcjonalności wysokiego poziomu,
- lepsza znajomość biznesu i produktów,
- większa szansa na napisanie czystego kodu,
- łatwiejszy start podczas pracy nad nowym projektem,

TDD – ZALETY

Dla klienta:

- wyższy poziom zapewnienia jakości aplikacji i mniej błędów,
- umiejętność przewidywania kosztów utrzymania aplikacji i uwzględniania modyfikacji istniejących rozwiązań,
- do projektu trzeba przydzielić mniej programistów,
- łatwiej jest zlokalizować i naprawić błędy,
- lepsze zrozumienie wdrożonych rozwiązań, co może prowadzić do sugestii ulepszeń przez zespół programistów,

TDD – WADY

- czas i wysiłek potrzebny na szkolenie i przygotowanie programistów,
- potrzeba dyscypliny, tj. testy muszą być zarządzane i ulepszane w czasie w taki sam sposób, jak cały inny kod,
- początkowa inwestycja o dłuższym czasie developmentu,
- nie wszystkich menedżerów przekonuje powyższy argument. Całkowity czas developmentu (w tym znajdowanie błędów, naprawianie i pisanie kodu) jest krótszy w TDD niż w innych.

TESTY PARAMETRYZOWANE

Niejednokrotnie istnieje potrzeba wykonania jednego testu jednostkowego wielokrotnie, z różnymi danymi wejściowymi. Problem taki może zostać rozwiązany poprzez powielenie istniejącego testu lub wyodrębnienie osobnej metody, którą wywołujemy w osobnych testach. Rozwiązanie takie powoduje jednak, że liczba testów jednostkowych rośnie. **Problem ten rozwiązują testy parametryzowane.**

JUNIT – TESTY PARAMETRYZOWANE

Testy parametryzowane wymagają od nas **zaimplementowania kodu testowego tylko raz**. Kod ten może być wykonany wielokrotnie z różnymi zestawami danych wejściowych. JUnit w wersji 5 udostępnia zestaw narzędzi umożliwiających implementację testów parametryzowanych. W tym celu do projektu należy wprowadzić następującą zależność:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.9.2</version>
    <scope>test</scope>
</dependency>
```

@PARAMETERIZEDTEST

Testy parametryzowane implementowane są **niemal identycznie jak zwykłe testy jednostkowe**, wyjątkiem jest sposób ich oznaczania. Zamiast adnotacji **@Test**, wykorzystujemy **@ParameterizedTest**.

Parametryzowane metody muszą specyfikować źródła parametrów (np. poprzez adnotacje: **@ArgumentSource**, **@ValueSource**, **@CsvSource**).

Osobą odpowiedzialną za dopasowanie ilości i typów argumentów to źródła jest programista!

@ValueSource definiuje zestaw danych wejściowych, które będą każdorazowo przekazywane do metody testującej jako jej argument.

```java
import org.junit.jupiter.api.*;

class SSTest {

    @ParameterizedTest
    @ValueSource(ints = {1, 2, 3})
    void testSlowSort(int arg) {

    }

    @Test
    void testSort() {

    }

}
```

@PARAMETERIZEDTEST

```java
public class NumbersHelper {
    public static boolean isOdd(int number) {
        return number % 2 != 0;
    }
}
```

```java
@ParameterizedTest
@ValueSource(ints = {1, 3, 5, -3, 15, Integer.MAX_VALUE})
void shouldReturnTrueForOddNumbers(int number) {
    assertTrue(NumbersHelper.isOdd(number));
}
```

ŹRÓDŁA ARGUMENTÓW

JUnit oferuje **kilka sposóbów definiowania zestawu argumentów** przekazywanych do testu parametryzowanego. Różnią się one możliwościami, a także sposobem definicji parametrów. Te adnotacje to:

- **@ValueSource**
- **@EnumSource**
- **@CsvSource**
- **@CsvFileSource**
- **@MethodSource**
- **@ArgumentsSource**

@VALUESOURCE

Adnotacja **@ValueSource** pozwala przekazać pojedynczy parametr do testu. Parametr ten może być jednym z następujących typów.

UWAGA! Jednym z ograniczeń danej adnotacji jest brak możliwości przekazywania wartości null!

Typ	Atrybut adnotacji @ValueSource
short	shorts
byte	bytes
int	ints
long	longs
float	floats
double	doubles
char	chars
java.lang.String	strings
java.lang.Class	classes

@VALUESOURCE - PRZYKŁADY

```
class ValueSourceExamplesTest {

  @ParameterizedTest
  @ValueSource(doubles = {1, 2.3, 4.1})
  void shouldPassDoubleToParam(double param) {
  }

  @ParameterizedTest
  @ValueSource(strings = {"Ala", "ma", "kota"})
  void shouldPassStringToTest(String word) {
  }

  @ParameterizedTest
  @ValueSource(classes = {String.class, Integer.class, Double.class})
  void shouldPassClassTypeAsParam(Class<?> clazz) {

  }
}
```

@ENUMSOURCE

Adnotacja ta umożliwia wywołanie testów parametryzowanych dla argumentów typów wyliczeniowych.

```
public enum TemperatureConverter {
  CELSIUS_KELVIN(cTemp -> cTemp + 273.15f),
  KELVIN_CELSIUS(kTemp -> kTemp - 273.15f),
  CELSIUS_FAHRENHEIT(cTemp -> cTemp * 9 / 5f + 32);

  private Function<Float, Float> converter;

  TemperatureConverter(Function<Float, Float> converter) {
    this.converter = converter;
  }

  public float convertTemp(float temp) {
    return converter.apply(temp);
  }
}
```

```
@ParameterizedTest
@EnumSource(TemperatureConverter.class)
void shouldConvertToValueHigherThanMinInteger(TemperatureConverter converter) {
  assertTrue(converter.convertTemp(10) > Integer.MIN_VALUE);
}
```

@ENUMSOURCE

Istnieje również możliwość **określenia konkretnych obiektów danego typu wyliczeniowego** za pomocą atrybutu **names**, np.:

```
@ParameterizedTest
@EnumSource(value = TemperatureConverter.class, names = {"CELSIUS_KELVIN",
"CELSIUS_FAHRENHEIT"})
void shouldConvertToTemperatureLowerThanMaxInteger(TemperatureConverter converter)
{
  assertTrue(converter.convertTemp(10) < Integer.MAX_VALUE);
}
```

@ENUMSOURCE

W ramach omawianej adnotacji istnieje również możliwość **definiowania trybu**, który określa czy wartości podane w atrybucie **names** są wartościami **wykluczonymi**, czy **uwzględnianymi**.

tryb	wartości w atrybucie names
`EnumSource.Mode.INCLUDE`	uwzgędnia, jest to domyślna wartość
`EnumSource.Mode.EXCLUDE`	wyklucza
`EnumSource.Mode.MATCH_ALL`	uzględnia te, które zawierają *wszystkie* podane ciągi znaków
`EnumSource.Mode.MATCH_ANY`	uwzględnia te, które zawierają *jakikolwiek* z podanych ciągów znaków

```
@ParameterizedTest
@EnumSource(value = TemperatureConverter.class,
    names = {"KELVIN_CELSIUS"},
    mode = EnumSource.Mode.EXCLUDE)
void shouldConvertTemperatureToPositiveValue(TemperatureConverter converter) {
  assertTrue(converter.convertTemp(10) > 0); // test zostanie uruchomiony dla
wartości CELSIUS_KELVIN i CELSIUS_FAHRENHEIT
}
```

@CSVSOURCE

Adnotacja **@CsvSource** umożliwia definiowanie parametrów testu za pomocą literałów **CSV** (comma separated value). Ponadto:

- Łańcuchy danych są **oddzielone pewnymi znakami** (domyślnie przecinkami).
- Każdy oddzielony element stanowi **osobny parametr** przyjmowany w teście.
- Mechanizm ten może mieć zastosowanie w przypadku, gdy na potrzeby danego testu jednostkowego **chcemy przekazać parametry wejściowe, jak i wartość oczekiwaną.**

@CSVSOURCE

- **Limitem adnotacji @CsvSource jest ograniczona ilość typów**, jakie możemy wykorzystać w teście. Wszystkie typy parametrów jakie chcemy użyć w teście muszą dać się przekonwertować z obiektu String.

```
public class Strings {
  public static String toUpperCase(String input) {
    return input.trim().toUpperCase();
  }
}

@ParameterizedTest
@CsvSource({"  test  ,TEST", "tEst ,TEST", "   Java,JAVA"})
void shouldTrimAndUppercaseInput(String input, String expected) {
  String actualValue = Strings.toUpperCase(input);
  assertEquals(expected, actualValue);
}
```

@CSVSOURCE

W ramach @CsvSource **istnieje możliwość zmiany znaku oddzielającego za pomocą atrybutu delimiter**, np.:

```
@ParameterizedTest
@CsvSource(value = {"  test  ;TEST", "tEst ;TEST", "   Java;JAVA"}, delimiter = ';'
)
void shouldTrimAndUppercaseInput(String input, String expected) {
  String actualValue = Strings.toUpperCase(input);
  assertEquals(expected, actualValue);
}
```

@CSVFILESOURCE

Adnotacja **@CsvFileSource** działa bardzo podobnie jak @CsvSource z tą różnicą, że **dane ładowane są bezpośrednio z pliku**. Ma ona możliwość zdefiniowania następujących parametrów:

- **numLinesToSkip** - określa ilość ignorowanych linii w pliku źródłowym. Jest to przydatne jeżeli dane pochodzą z tabeli, która oprócz wartości posiada dodatkowo nagłówki.
- **delimeter** - oznacza separator pomiędzy poszczególnymi elementami.

@CSVFILESOURCE

- **lineSeparator** - oznacza separator pomiędzy poszczególnymi zestawami parametrów.
- **encoding** - oznacza sposób kodowania zawartości pliku.

```
@ParameterizedTest
@CsvFileSource(resources = "/data.csv", numLinesToSkip = 1) // plik data.csv musi
znaleźć się w roocie classpath, pomijamy pierwszą linię w pliku
void shouldUppercaseAndBeEqualToExpected(String input, String expected) {
  String actualValue = Strings.toUpperCase(input);
  assertEquals(expected, actualValue);
}
```

@METHODSOURCE

Adnotacje **@ValueSource**, **@EnumSource**, **@CsvSource** i **@CsvFileSource** mają ograniczenia - **ilość parametrów lub ich typy**. Problem ten rozwiązuje źródło argumentów definiowane przez adnotację **@MethodSource**. Jedynym parametrem ten adnotacji jest **nazwa metody statycznej** znajdującej się **w tej samej klasie***. Metoda ta powinna być **bezargumentowa** i zwracać **strumień**:

- **obiektów dowolnego typu** w przypadku testów z pojedynczym parametrem,
- **obiektów typu Arguments** w przypadku testów z wieloma parametrami.

@METHODSOURCE

Oba sposoby wykorzystania adnotacji @MethodSource pokazują kolejne przykłady:

```java
@ParameterizedTest
@MethodSource("provideNumbers")
void shouldBeOdd(final Integer number) {
  assertThat(number % 2).isEqualTo(1);
}

static Stream<Integer> provideNumbers() {
  return Stream.of(1, 13, 101, 11, 121);
}
```

```java
@ParameterizedTest
@MethodSource("provideNumbersWithInfoAboutParity")
void shouldReturnExpectedValue(int number, boolean expected) {
  assertEquals(expected, number % 2 == 1);
}

private static Stream<Arguments> provideNumbersWithInfoAboutParity() {
  return Stream.of(Arguments.of(1, true),
      Arguments.of(2, false),
      Arguments.of(10, false),
      Arguments.of(11, true));
}
```

@ARGUMENTSSOURCE

@ArgumentsSource jest najbardziej uniwersalnym źródłem definiowania argumentów. Podobnie jak @MethodSource pozwala zdefiniować **dowolną ilość argumentów dowolnego typu**. Główną różnicą między tymi dwoma adnotacjami jest fakt, że wewnątrz adnotacji **@ArgumentsSource** podajemy typ, który musi implementować interfejs **ArgumentsProvider.**

Metoda jaką musimy zaimplementować, **provideArguments** powinna zwracać **strumień obiektów typu Arguments**.

@ARGUMENTSSOURCE

```java
public class NumberWithParityArgumentsProvider implements ArgumentsProvider {

  @Override
  public Stream<? extends Arguments> provideArguments(ExtensionContext context) {
    return Stream.of(
        Arguments.of(1, true),
        Arguments.of(100, false),
        Arguments.of(101, true)
    );
  }
}
```

```java
@ParameterizedTest
@ArgumentsSource(NumberWithParityArgumentsProvider.class)
void shouldReturnExpectedValue(int number, boolean expectedResult) {
  assertEquals(expectedResult, number % 2 == 1);
}
```

AGREGOWANIE ARGUMENTÓW*

- Domyślnie, **każdy argument** dostarczany do parametryzowanego testu **odpowiada pojedynczemu parametrowi**.
- W konsekwencji, **przekazywanie wielu parametrów do danej metody może być nieczytelne.**
- Jednym z podejść jest **enkapsulacja wszystkich przekazywanych argumentów w jednej instancji obiektu ArgumentAccessor** i odwoływania się do nich poprzez **indeks** w metodzie testowej.

AGREGOWANIE ARGUMENTÓW*

- **getString(index)** – pobiera wartość i konwertuje ją do typu java.lang.String
- **get(index)** – pobiera wartość w postaci klasy java.lang.Object
- **get(index, type)** – pobiera wartość i przypisuje do konkretnego typu

```
@ParameterizedTest
@CsvSource({"a+b,a,b", "+,,"})
public void shouldProperlySumLiterals(ArgumentsAccessor argumentsAccessor) {
    String result = argumentsAccessor.getString(0);
    String operand1 = defaultString(argumentsAccessor.getString(1));
    String operand2 = defaultString(argumentsAccessor.getString(2));

    assertEquals(result, operand1 + "+" + operand2);
}
```

AGREGOWANIE ARGUMENTÓW*

Bezpośrednie wykorzystywanie klasy **ArgumentAccessor** może powodować problemy w zrozumieniu kodu, jak również **zaburzyć jego czytelność** (ze względu na nieintuicyjne pobieranie poszczególnych parametrów wywołania metody).
W celu eliminacji tego problemu można posłużyć się interfejsem: **ArgumentsAggregator**

AGREGOWANIE ARGUMENTÓW*

```java
public class OperandsAggregator implements ArgumentsAggregator {

    @Override
    public Object aggregateArguments(ArgumentsAccessor accessor, ParameterContext context)
            throws ArgumentsAggregationException {
        return new Operands(defaultString(accessor.getString( index: 1)),
                defaultString(accessor.getString( index: 2)));
    }
}
```

```java
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class Operands {

    private String operand1;

    private String operand2;
}
```

```java
    @ParameterizedTest
    @CsvSource({"a+b,a,b", "+,,"})
    public void shouldProperlySumLiterals(String result,
            @AggregateWith(OperandsAggregator.class) Operands operands) {
        assertEquals(result,  actual: operands.getOperand1() + "+" + operands.getOperand2());
    }
}
```

ŁĄCZENIE ŹRÓDEŁ

JUnit pozwala na **łączenie wielu źródeł argumentów**. Ponieważ omówione adnotacje oznaczające źródła argumentów nie posiadają adnotacji **@Repeatable**, **możemy wykorzystywać konkretny typ źródła tylko raz**, np. następujący test parametryzowany zostanie uruchomiony 5 razy:

```java
class CombinedSourcesTest {

  @ParameterizedTest
  @CsvSource(value = "1, true")
  @MethodSource("provideNumbersWithInfoAboutParity")
  void shouldReturnExpectedValue(int number, boolean expected) {
    assertEquals(expected, number % 2 == 1);
  }

  private static Stream<Arguments> provideNumbersWithInfoAboutParity() {
    return Stream.of(Arguments.of(1, true),
        Arguments.of(2, false),
        Arguments.of(10, false),
        Arguments.of(11, true)
    );
  }
}
```

ŹRÓDŁA POMOCNICZE

Oprócz standardowych źródeł dostępne mamy kilka **pomocniczych adnotacji**, które umożliwiają dodanie do istniejących testów, **dodatkowych wartości parametrów**. Są to:

- **@NullSource**, która umożliwia przekazanie parametru null do testu
- **@EmptySource**, która umożliwia przekazanie pustego obiektu do testu
- **@NullAndEmptySource**, która łączy funkcjonalności adnotacji @NullSource i @EmptySource.

ŹRÓDŁA POMOCNICZE

```
public class Strings {
    public static boolean isBlank(String input) {
        return input == null || input.trim().isEmpty();
    }
}

@ParameterizedTest
@NullSource
void shouldbeBlankForNull(String input) {
  assertTrue(Strings.isBlank(input));
}
```

```
public class Arrays {
  public static boolean isValid(List<String> values) {
    return values != null && !values.isEmpty();
  }
}

@ParameterizedTest
@EmptySource
void shouldNotBeValid(List<String> input) {
  assertFalse(Arrays.isValid(input));
}

@ParameterizedTest
@NullAndEmptySource
void nullAndEmptyShouldBeBlank(String input) {
  assertTrue(Strings.isBlank(input));
}
```

TESTOWANIE WYJĄTKÓW

Pisząc testy jednostkowe, **najczęściej zaczynamy od testowania tzw. "happy path".** Nie powinniśmy jednak zapominać o przetestowaniu scenariuszy, **które wyrzucają wyjątki.**

TRY-CATCH

Wyjątki możemy testować wykorzystując blok **try catch** oraz asercję **fail** z JUnit 5, np.:

```
@Test
void shouldThrowException() {
    try {
      numbers.findFirstDigit("");
      fail("Exception was not thrown");
    } catch (final IllegalArgumentException exp) {
      assertEquals("Input cannot be empty", exp.getMessage());
    }
}
```

ASSERTTHROWS

Junit 5 udostępnia **zestaw narzędzi umożliwiających łatwe testowanie wyjątków w Javie**. Za pomocą przystosowanych do tego celu asercji **można przechwytywać wyjątek i weryfikować jego potencjalne szczegóły**. Można wyróżnić trzy przeciążenia takiej asercji:

- `public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable)`
- `public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable, String message)`
- `public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable, Supplier<String> messageSupplier)`.

ASSERTTHROWS

Pierwszym argumentem tej metody jest oczekiwany typ wyjątku (lub typ, po którym ten wyjątek dziedziczy). Drugim argumentem jest **interfejs funkcyjny, w którym powinniśmy wywołać kod, który według nas powinien wyrzucić konkretny wyjątek**. Opcjonalnie, możemy dodatkowo spersonalizować wiadomość w przypadku, gdy taki wyjątek nie wystąpi.

Asercja **assertThrows** zwraca instancję wyrzuconego wyjątku. Dzięki temu mamy możliwość sprawdzenia, np. **wiadomość wyjątku czy powód jego wyrzucenia** (czyli cause).

ASSERTTHROWS

```
public static float divide(float a, float b) {
  if (b == 0) {
    throw new IllegalArgumentException("dividend can't be 0");
  }
  return a / b;
}


@Test
void shouldThrowIllegalArgumentException() {
  final IllegalArgumentException exception = assertThrows(IllegalArgumentException
.class, () -> divide(10, 0));

  assertThat(exception).hasMessage("dividend can't be 0")
      .hasNoCause();
}
```

ASSERTJ

Wybierając asercje do testów, mamy możliwość pozostać przy tych, jakie oferuje JUnit **lub wykorzystać bibliotekę AssertJ**. Biblioteka AssertJ oferuje dwie metody, które rozpoczynają łańcuch wywołań sprawdzających informacje o wyrzuconym wyjątku. Są to:

- **assertThatExceptionOfType**
- **assertThatThrownBy**

ASSERTJ

Ponadto, AssertJ udostępnia **wyspecjalizowane wersje** metody **assertThatExceptionOfType**, które oczekują konkretnego typu wyjątku. Wynika on z nazwy metody, np.:

- **assertThatNullPointerException**
- **assertThatIllegalArgumentException**

```
@Test
void shouldThrowIllegalArgumentException() {
  assertThatExceptionOfType(IllegalArgumentException.class)
        .isThrownBy(() -> divide(10, 0))
        .withMessage("dividend can't be 0")
        .withNoCause();
}

@Test
void shouldThrowIllegalArgumentException() {
  assertThatIllegalArgumentException()
        .isThrownBy(() -> divide(10, 0))
        .withMessage("dividend can't be 0")
        .withNoCause();
}

@Test
void shouldThrowIllegalArgumentException() {
  assertThatThrownBy(() -> divide(10, 0))
      .isExactlyInstanceOf(IllegalArgumentException.class)
      .hasMessage("dividend can't be 0")
      .hasNoCause();
}
```

DZIĘKUJĘ ZA UWAGĘ