



TESTOWANIE OPROGRAMOWANIA | TEST DRIVEN DEVELOPMENT

Autor: Michał Bojanowski
Prawa do korzystania z materiałów posiada Software
Development Academy

Kilka słów o mnie



Software Developer @ Onegini

Główny język programowania: JAVA

Dodatkowo: JavaScript, Python, C++

5+ lat na rynku IT

Wejście na rynek IT:

- Podobna ścieżka jak Wasza!
- Studia nietechniczne
- Kursy

Kontakt: bojanowski.michal89@gmail.com

Agenda



1. Ogólnie o testowaniu oprogramowania
2. Jak testować oprogramowanie
3. Testowanie za pomocą Junit
4. Używanie biblioteki AssertJ
5. Testy parametryzowane
6. Testowanie wyjątków
7. Czym jest TDD
8. Mockowanie
9. Zawartość dodatkowa

Testowanie Oprogramowania – czyli co?

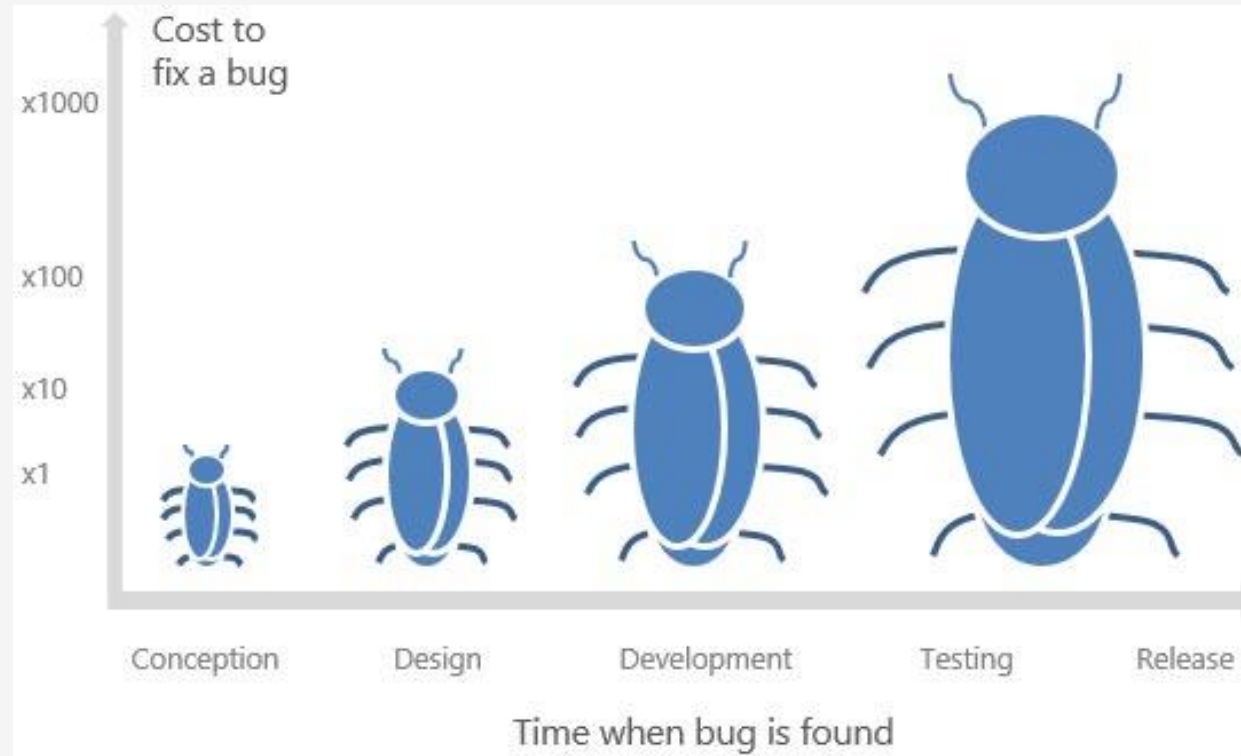


- jeden z **wielu** procesów tworzenia oprogramowania
- proces, który ma na celu weryfikację czy oprogramowanie działa tak jak się tego spodziewamy, np.
 - czy wynik jest zgodny z oczekiwaniami
 - czy funkcjonalność jaką stworzyliśmy jest zgodna ze specyfikacją

Wynikiem testowania oprogramowania często są...



Koszt naprawy buga, a cykl tworzenia oprogramowania



Autor: Michał Bojanowski

Prawa do korzystania z materiałów posiada Software Development Academy

Podział testów



- Testy Manualne

- Za **każdym razem** wymaga ręcznego wyklikania scenariusza
- Wymaga ręcznego sprawdzenia czy oczekiwany wynik jest zgodny z oczekiwaniami



- Testy Automatyczne

- Wymagają **RAZ** utworzenia skryptu/kawałka kodu
- Są łatwe i szybkie do powtórzenia
- Wynik testu pewniejszy niż manualny odpowiednik.



Rodzaje testów



- **Jednostkowe** (unit tests)
- **Integracyjne** (integration tests)
- Funkcjonalne (functional tests)
- ...
- Wiele innych (performance tests, stress tests, system tests, regression tests)

Testy jednostkowe vs integracyjne



Jednostkowe	Integracyjne
Testujemy izolowany element (np. metoda, klasa) bez zależności	Testujemy element z rzeczywistymi zewnętrznymi zależnościami (np. baza danych, zewnętrzny serwis google)
Tylko jedno potencjalne miejsce awarii	Wiele potencjalnych awarii
Szybkie wykonanie testów (< 1s)	Możliwe dłuższe wykonanie testów (< 1min)
Nie wymagają jakiegokolwiek konfiguracji	Możliwa konieczna konfiguracja zależna od środowiska

Testy jednostkowe i integracyjne są pisane przez **developerów**

Testy Jednostkowe:



Inne cechy:

- są integralną częścią kodu
- pozwalają poeksperymentować z daną funkcjonalnością (klasą, metodą) w wyizolowanym środowisku
- unit test to faktyczne pierwsze użycie kodu przed mergem kodu do master brancha
- potrafi być świetną dokumentacją
- znacznie ułatwiają refactoring kodu

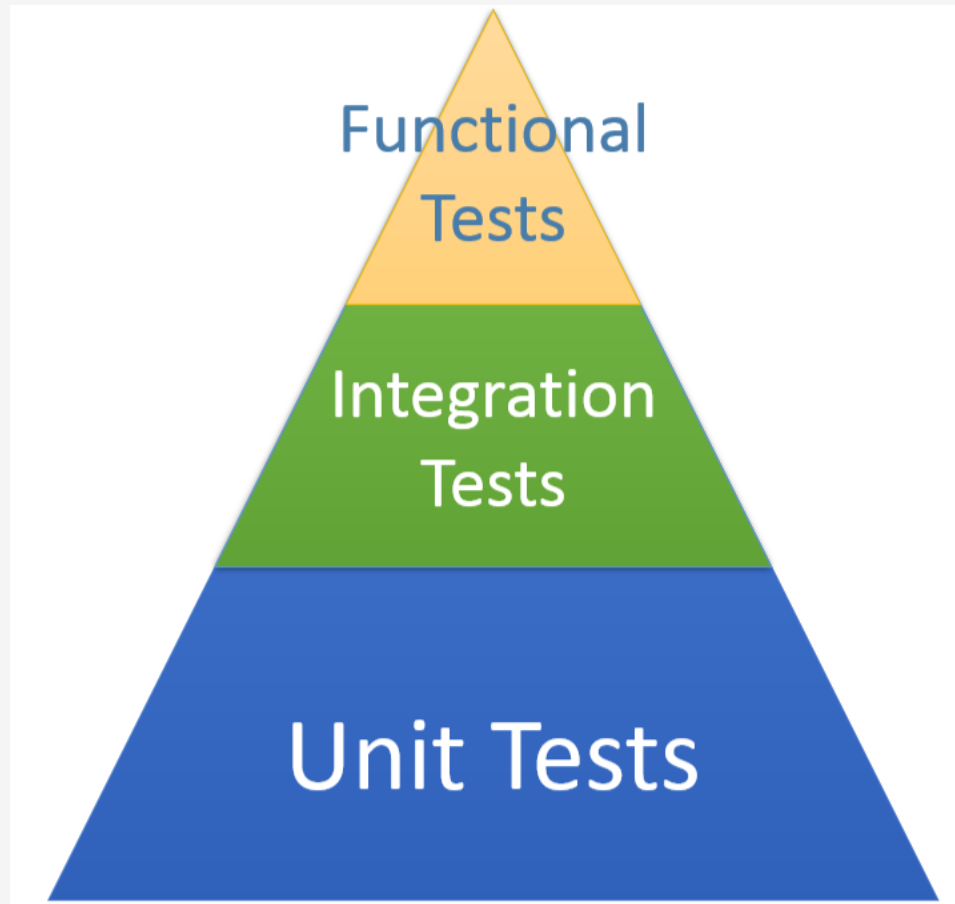
Testy funkcjonalne



Testują wytworzone oprogramowanie poprzez weryfikację zachowania kompletnej aplikacji pod kątem wymagań biznesowych za pomocą różnego rodzaju interfejsu udostępnianego użytkownikom.

- pisane często z wykorzystaniem np. selenium lub j-meter
- tworzone aby symulować zachowanie aplikacji w rzeczywistym środowisku
- tworzone najczęściej przez osoby dedykowane (QA)

Typ vs ilość



Dobre testy jednostkowe – FIRST Unit Tests



- Fast – szybki czas wykonania
- Isolated/Independent – niezależne
- Repeatable – powtarzalne (nie używamy np. dat, czasu, losowych liczb)
- Self-Validating – samo sprawdzające
- Thorough - dokładne

Thorough - przypadki testowe



QA Engineer walks into a bar. Orders a beer.
Orders 0 beers. Orders 9999999999 beers.
Orders a lizard. Orders -1 beers. Orders a
sfdeljknesv.

- rozważamy przypadki pozytywne
- rozważamy przypadki graniczne (np. skrajnie małe/duże wartości)
- rozważamy przypadki negatywne (np. niepoprawne dane wejściowe, wyjątki)

Po co więc testować?



- podnieść jakość tworzonego przez nas kodu
- udowodnić, że kod robi to co myślimy, że robi
- nie naprawiać jednej funkcjonalności psując drugą
- nie bać się zmieniać już istniejącego kodu
- nie iść na przekór dobrym standardom

Pytania



- Jakie są rodzaje testów?
- Ile powinny trwać Unit Testy?
- Które testy trwają najdłużej?
- Czy testy integracyjne wymagają specjalnej konfiguracji?
- Jakie rodzaje testów tworzą programiści?



- Jakie są rodzaje testów? (główny podział: jednostkowe, integracyjne, funkcjonalne)
- Ile powinny trwać Unit Testy?(mniej niż 1sec)
- Które testy trwają najdłużej? (funkcjonalne, ale czasem integracyjne)
- Czy testy integracyjne wymagają specjalnej konfiguracji? (raczej nie ale niekiedy tak)
- Jakie rodzaje testów tworzą programiści? (jednostkowe, integracyjne, rzadziej funkcjonalne)

Podsumowanie



- Zdefiniowaliśmy czym jest testowanie i po co to robić
- Poznaliśmy główne rodzaje testów (jednostkowe, integracyjne, funkcjonalne)
- Poznaliśmy różnice między typami testów
- Dowiedzieliśmy się jak budować przypadki testowe.

Java i biblioteki do testowania



JUnit



mockito



cucumber




Autor: Michał Bojanowski

Prawa do korzystania z materiałów posiada Software Development Academy



Biblioteka do tworzenia testów (m. in. jednostkowych) kodu pisanego w języku Java.

- Najpopularniejsza biblioteka do testowania dostępna na rynku
- Aktualnie używane wersje: 4 i 5.
- Wersja 4 wciąż popularniejsza niż 5.
- Skupimy się na wersji .
- ALE zobaczymy różnice między wersją 4 i 5.

Junit - zależności



```
<properties>
  <junit-platform.version>5.3.2</junit-platform.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>${junit-platform.version}</version>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>${junit-platform.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Junit – nasz pierwszy test



```
package pl.sdacademy;

import org.junit.jupiter.api.Test;

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

public class OurFirstUnitTest { // klasa testowa

    @Test // adnotacja oznaczająca, że metoda jest testem
    void shouldMultiplyTwoNumbers() { // sygnatura metody testowej w przeciwieństwie do JUnit4
        // nie musi być publiczna
        // given // warunki początkowe
        final double firstNumber = 2;
        final double secondNumber = 3;

        // when // wykonanie testu
        final double multiplicationResult = firstNumber * secondNumber;

        //then // sprawdzenie wyniku testu
        assertEquals(6, multiplicationResult);
    }
}
```

JUnit – jak pisać testy



Dzielenie testów na sekcje: given, when, then

Given:

- Tworzymy założenia początkowe, tworzymy instancje obiektów

When:

- Wykonujemy wywołanie metody, którą chcemy przetestować

Then:

- Sprawdzamy nasze oczekiwania z rzeczywistymi rezultatami, najczęściej wykorzystując asercje.

JUnit – jak nazywać metody testowe



Istnieją różne konwencje: <https://dzone.com/articles/7-popular-unit-test-naming>, np.

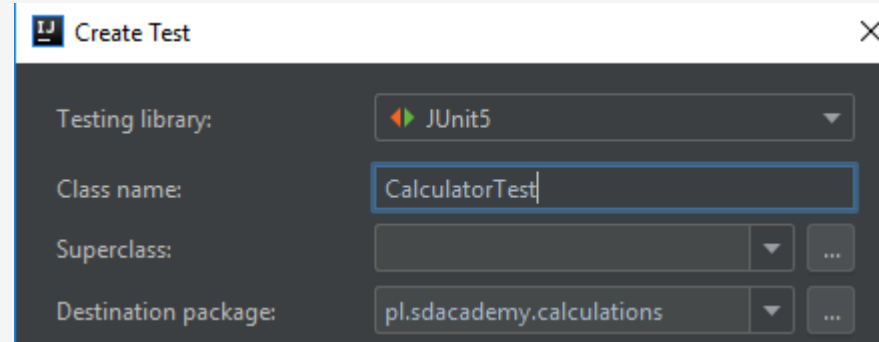
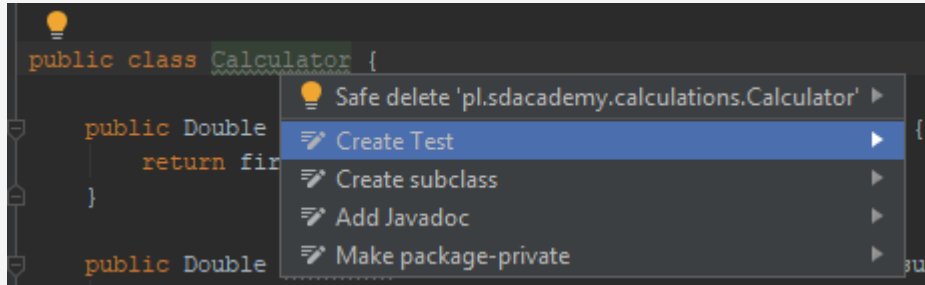
- test[Feature being tested]
- should[ExpectedBehavior]When[StateOccurs]
- should[ExpectedBehavior]When[StateOccurs] będziemy używać podczas kursu

- ## NOT available in JUnit4 (5 only)
- `assertArrayEquals()`
 - `assertIterableEquals()`
 - `assertLinesMatch()`
 - `assertTimeout()`
 - `assertTimeoutPreemptively()`
 - `assertAll()`

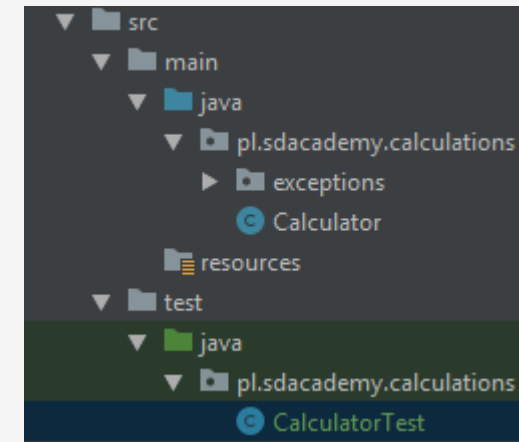
JUnit – tworzenie klas testowych



Z poziomu intelliJ [alt + enter] (windows/ubuntu) [command + enter] (macOS)



Klasa testowa powinna znajdować się w paczce o tej samej nazwie co klasa testowana, w katalogu **test/java**



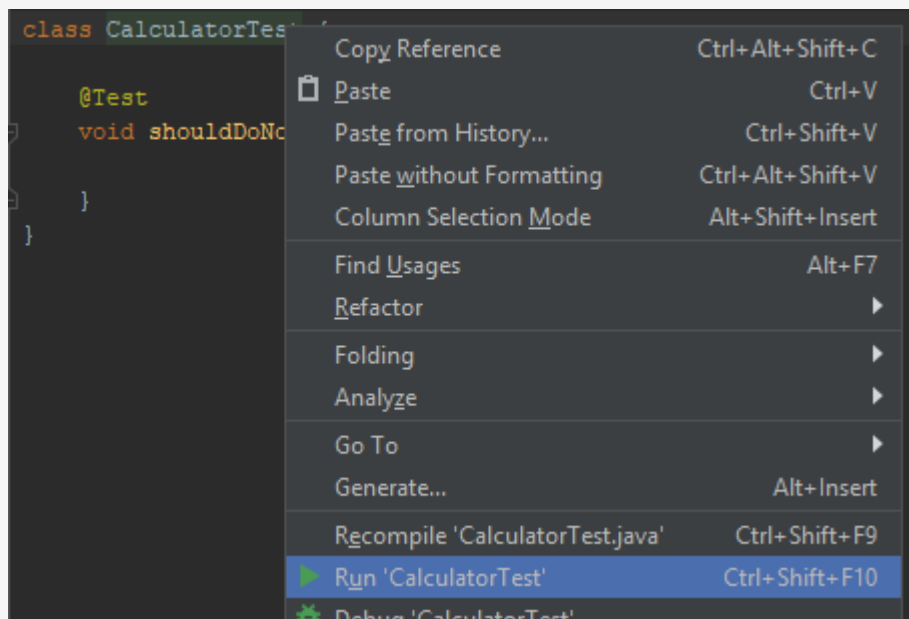
JUnit – uruchamianie testów



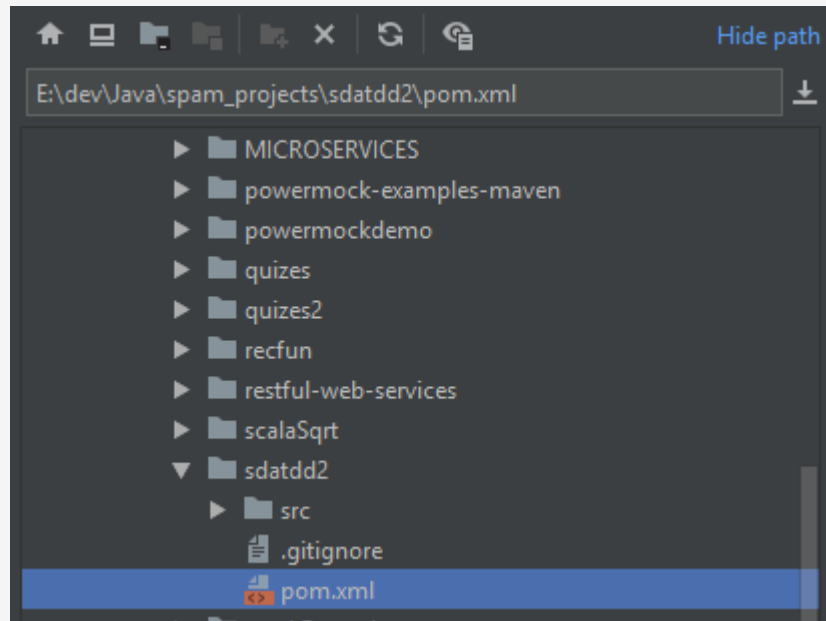
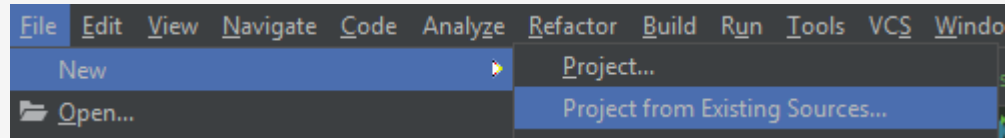
Sposób pierwszy: cmd line za pomocą mavena

- mvn clean install

Sposób drugi (preferowany w trakcie developmentu): z poziomu intellij



IDEA – poprawne importowanie projektów



IDEA – poprawne importowanie projektów



Import Project from Maven

×

Root directory

E:\dev\Java\spam_projects\sdatdd2

...

☐ Search for projects recursively

Project format:

.idea (directory based)

▼

▼

Synchronize Maven project model and IDEA project model each time when pom.xml is changed

▼

☒ Import Maven projects automatically

☒ Create IntelliJ IDEA modules for aggregator projects (with 'pom' packaging)

☐ Create module groups for multi-module Maven projects

☒ Keep source and test folders on reimport

☒ Exclude build directory (%PROJECT_ROOT%/target)

☒ Use Maven output directories

Generated sources folders:

Detect automatically

▼

Phase to be used for folders update:

process-resources

▼

IDEA needs to execute one of the listed phases in order to discover all source folders that are configured via Maven plugins.
Note that all test-* phases firstly generate and compile production sources.

Automatically download:

☐ Sources

☐ Documentation

Dependency types:

jar, test-jar, maven-plugin, ejb, ejb-client, jboss-har, jboss-sar, war, ear, bundle

Comma separated list of dependency types that should be imported

Environment settings...

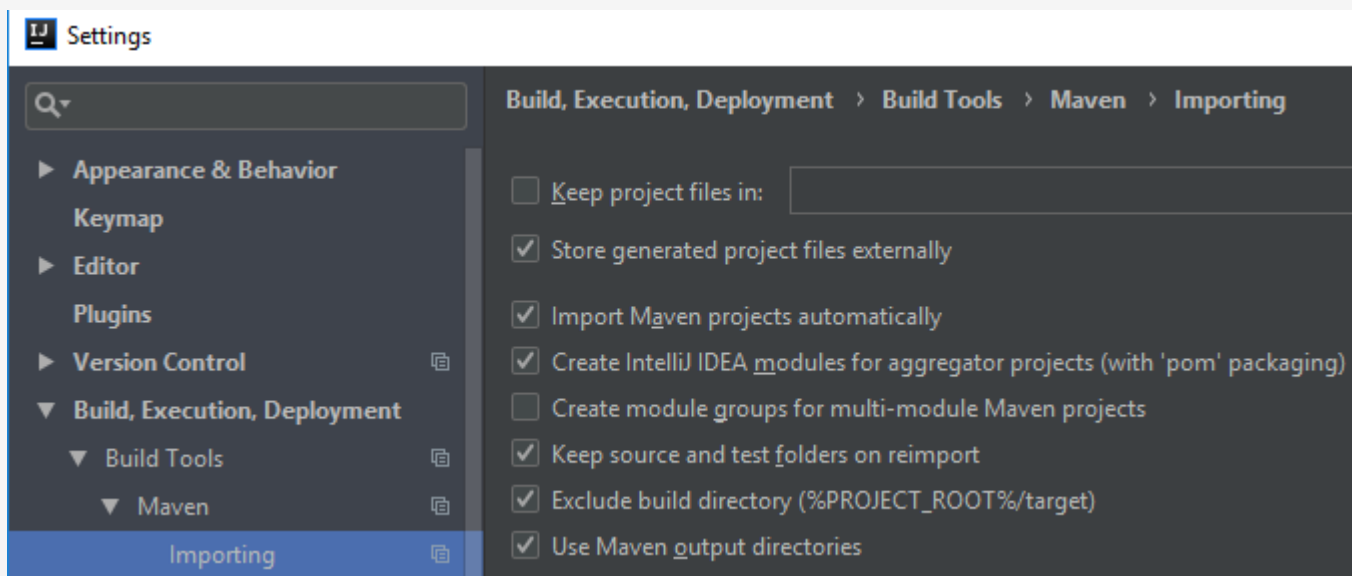
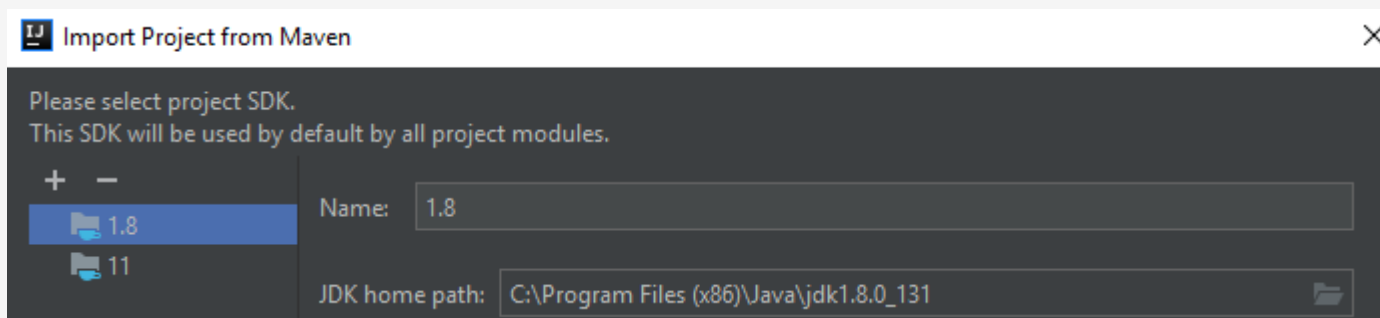
Previous

Next

Cancel

Help

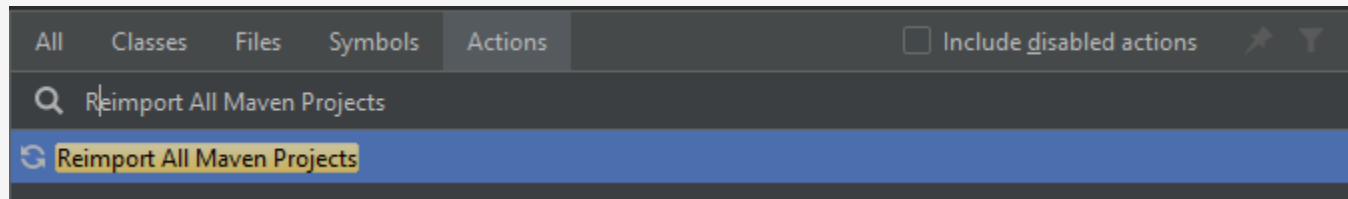
IDEA – poprawne importowanie projektów



IDEA + maven – wymuszenie reimportu



CTRL + SHIFT + A



Ćwiczenie - 1



1. Zimportuj do IDE projekt "sdatdd"
<https://github.com/Mbojanow/sdatdd>
1. Stwórz klasę testową dla klasy Calculator
2. Stwórz trzy testy. Przetestuj metody add, subtract, multiply. Testując metodę subtract odejmij 3.8 od 2.5.
4. Stwórz czwarty test, w którym asercja zawiedzie, wyświetl w tej asercji swój własny komunikat.



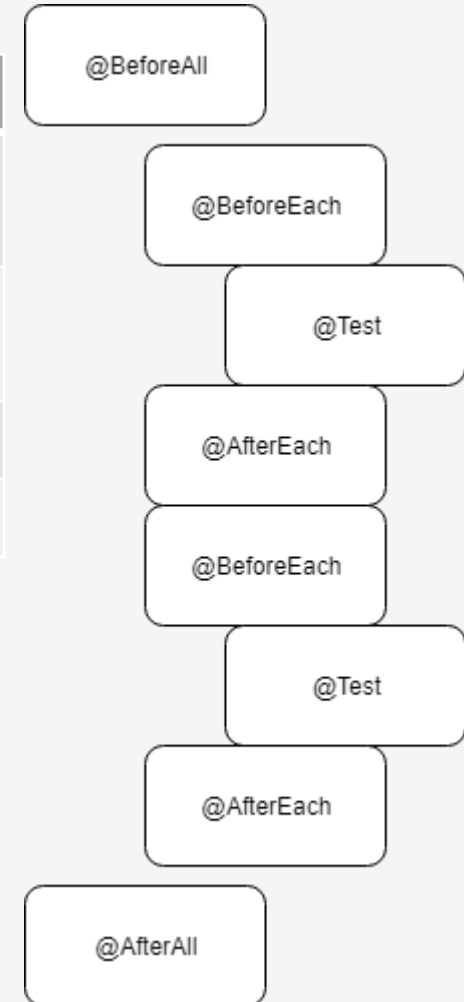
Uwagi

- Stwórz osobną instancję klasy Calculator w każdym teście
- Pamiętaj o podzieleniu testu na sekcje given/when/then
- Pamiętaj o zasadach FIRST
- Rozważ czy użycie asercji assertEquals ma sens w powyższych testach

JUnit – cykl życia testów



JUnit5	JUnit4	Czas uruchamiania
@BeforeAll	@BeforeClass	metoda statyczna, przed wszystkimi testami
@AfterAll	@AfterClass	metoda statyczna, po wykonaniu wszystkich testów
@BeforeEach	@Before	przed każdym testem
@AfterEach	@After	po każdym teście



Ćwiczenie - 2



1. Zaimportuj do IDE projekt "sdatdd"
2. Dodaj użycie BeforeEach w klasie CalculatorTest
3. Stwórz klasę testową dla klasy DatabaseStore
4. Stwórz przynajmniej po 1 teście dla metod `addData` i `removeData`.



Uwagi

- Wykorzystaj wszystkie możliwe adnotacje cyklu życia testów w celu:
 - stworzenia połączenia z bazą danych przed wszystkimi testami (użyj `DatabaseConnection` jako statyczne pole),
 - zamknięcia połączenia z bazą danych po wszystkich testach
 - wyczyszczenia zawartości testowanego `DatabaseStore` po każdym teście
 - wyświetlenia na ekran dowolnej informacji przed każdym z testów

JUnit – dodatkowe możliwości



@DisplayName – kiedy łatwiej nam nazwać test za pomocą Stringa zamiast za pomocą nazwy metody

@Disabled – wyłącza dany test. Używamy kiedy np. test nam nie przechodzi i chcemy implementację naprawić później.

Ćwiczenie - 3



1. W klasie testowej DatabaseStoreTest zmień nazwę testów za pomocą @DisplayName i wyłącz jeden z nich z użyciem @Disabled.
2. Napisz testy dla metody doubleValues i doubleInputValues w klasie ListUtil.



Wykorzystaj odpowiednie asercje. W których testach możesz użyć assertEquals?

Uwagi

- Potwierdź, że test z adnotacją @Disabled rzeczywiście jest ignorowany.
- Potwierdź, że @DisplayName rzeczywiście zmienia nazwę testu.

JUnit – część pierwsza - pytania



1. Jakie adnotacje cyklu życia testów możemy użyć na statycznych metodach?
2. Jak mogę zmienić wyświetlaną nazwę testu?
3. Czym różni się assertEquals i assertEquals?
4. Jakie dependencje są potrzebne w pliku pom.xml aby korzystać z biblioteki JUnit5?
5. Jaki poziom dostępu musi mieć metoda oznaczona adnotacją @Test w JUnit5, a jaki w JUnit4?

JUnit – część pierwsza - pytania



1. Jakie adnotacje cyklu życia testów możemy użyć na statycznych metodach? (@BeforeAll, @AfterAll)
2. Jak mogę zmienić wyświetlaną nazwę testu? (za pomocą @DisplayName)
3. Czym różni się assertEquals i assertEquals? (assertEquals porównuje elementy metodą equals, assertEquals za pomocą ==)
4. Jakie dependencje są potrzebne w pliku pom.xml aby korzystać z biblioteki JUnit5? (junit-jupiter-api, junit-jupiter-engine)
5. Jaki poziom dostępu musi mieć metoda oznaczona adnotacją @Test w JUnit5, a jaki w JUnit4? (5 - package private, 4 public)

JUnit - podsumowanie



1. Metody testowe oznaczamy adnotacją `@Test`
2. `@BeforeAll`, `@BeforeEach`, `@AfterAll`, `@AfterEach` to adnotacje dla oznaczenia metod cyklu życia testów.
3. Najczęściej wykorzystywaną asercją w JUnit jest `assertEquals`, ale pamiętajmy, że JUnit daje nam wiele więcej możliwości.

Asercje zaawansowane - assertAll



JUnit – assertAll

- Asercja dostępna w JUnit 5
- Pozwala na grupowanie wielu asercji w jedną
- Wykona zawsze wszystkie asercje wewnątrz

```
@Test
void shouldCreatePersonCorrectly() {
    final Person person = Person.create(FIRST_NAME, LAST_NAME);

    assertAll(
        () -> assertEquals(FIRST_NAME, person.getFirstName()),
        () -> assertEquals(LAST_NAME, person.getLastName())
    );
}
```

Asercje zaawansowane – biblioteka assertJ



- Biblioteka tzw. płynnych asercji
- Daje dostęp do wielu nowych asercji
- Daje dostęp do pisania asercji w taki sposób aby były one czytelniejsze

```
@Test
void junitAssertions() {
    final String actual = "I_love_sda";

    assertEquals(EXPECTED, actual);
}
```

```
@Test
void assertjAssertions() {
    final String actual = "I_love_sda";

    assertThat(actual).isEqualTo(EXPECTED);
}
```




Co po za czytelnością?

- Szybsza analiza testów
- Lepsza dokładność testów
- Lepsze komunikaty o błędach
- Działa świetnie w parze z JUnit.
- Pozwala łączyć matchery w ciągi. (np. Zapewnij, że ciąg znaków zaczyna się od słowa „Start”, kończy się na „koniec” i ma długość przynajmniej 30.

Co potrafi AssertJ?



Dużo!

- W zależności od typu testowanego obiektu, mamy dostępne inne matchery
- Pozwala łączyć wiele asercji w jeden ciąg

Kilka przykładów dla String:

- `doesNotContainAnyWhitespaces`
- `isEqualTo`
- `containsPattern`
- `endsWith`
- `doesNotStartWith`
- `isBetween`
- kilkanaście innych matcherów.

I kilka przykładów dla Listy:

- `isEqualTo`
- `contains`
- `containsAnyOf`
- `containsExactlyInAnyOrder`
- `isNotSameAs`

AssertJ - przykłady



```
@Test
void shouldGetFullName() {
    final Person person = Person.create(FIRST_NAME, LAST_NAME);

    assertThat(person.getFullName())
        .startsWith(FIRST_NAME)
        .endsWith(LAST_NAME)
        .contains(" ");
}
```

```
@Test
void shouldReverseOrderAndCopyList() {
    final List<String> testedList = Arrays.asList("raz", "dwa", "trzy");

    final List<String> reversed = ListUtil.getReversedList(testedList);

    assertThat(reversed)
        .isNotEmpty()
        .isNotSameAs(testedList)
        .containsExactly("trzy", "dwa", "raz");
}
```

Ćwiczenie - 4



- Stwórz klasę testową IntegerUtilsTest
- Przetestuj z wykorzystaniem assertJ dostępne metody publiczne klasy IntegerUtils.
- Wykorzystaj assertAll z JUnit5 aby sprawdzić czy otrzymana lista z metody filter jest nie pusta, ma odpowiednią długość i ma odpowiednie elementy
- Wykorzystaj assertJ aby sprawdzić czy otrzymana lista z metody filterDigitsGreaterThan jest niepusta, ma odpowiednią długość i posiada oczekiwane elementy
- Wykorzystaj assertJ aby sprawdzić czy otrzymana lista z metody getLastEvenDigit zwraca poprawną wartość, upewnij się że wartość istnieje zanim ją odczytasz
- Dopisz kolejne przypadki testowe, aby znaleźć bug w klasie którą testujesz. Popraw znalezione bugi w implementacji.

Asercje zaawansowane - pytania



1. Jaką asercję użyć w JUnit5 aby sprawdzić wiele rzeczy na raz?
2. Jaką asercją z assertJ sprawdzić zawartość kolekcji, która nie gwarantuje zachowania kolejności?

Asercje zaawansowane - pytania



1. Jaką asercję użyć w JUnit5 aby sprawdzić wiele rzeczy na raz? (assertAll)
2. Jaką asercją z assertJ sprawdzić zawartość kolekcji, która nie gwarantuje zachowania kolejności? (containsExactlyInAnyOrder)

Duplikacja testów



```
@Test
void zeroShouldBeEven() {
    assertThat(integerUtils.isEven(0)).isTrue();
}

@Test
void fourShouldBeEven() {
    assertThat(integerUtils.isEven(4)).isTrue();
}

@Test
void bigNumberShouldBeEven() {
    assertThat(integerUtils.isEven(432311316)).isTrue();
}
```

Wygląda ok ale...

... zdecydowanie można tu coś poprawić.

JUnit – testy parametryzowane



Testy parametryzowane to testy, które pozwalają nam testować dany scenariusz na podstawie wielu danych testowych.

Co nam to daje? Możemy, np.

- sprawdzić zachowanie scenariusza na wielu poprawnych danych
- sprawdzić scenariusze negatywne (np. oczekujemy takiego samego zachowania przy nullu, pustym optionalu, pustej liście, zerze itd.)
- sprawdzić zachowanie na przypadkach skrajnych

JUnit – testy parametryzowane



Ponadto:

- wszystkie dane wejściowe są najczęściej w jednym miejscu
- mamy tylko jeden kod testowy
- łatwo znaleźć wadliwy parametr gdy test failuje
- lepsza czytelność testów, niż korzystanie z metody 1 test - 1 parametr

JUnit – co potrzebujemy?



```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-params</artifactId>
  <version>${junit-platform.version}</version>
</dependency>
```

- junit-jupiter-params – wsparcie do testów parametryzowanych w JUnit
- adnotację @ParametrizedTest do oznaczenia testu
- źródła argumentów

Uwaga: @ParametrizedTest zastępuje @Test

Testy parametryzowane - źródła



- `@ValueSource` – dane bezpośrednio z listy adnotacji, dostępne typy proste + `String`
- `@EnumSource` – dane z możliwych wartości `Enum`
- `@CsvSource` – dane z listy oddzielonej przecinkami
- `@CsvFileSource` – dane z listy oddzielonej przecinkami z pliku w `classpath`
- `@MethodSource` – dane z metody o danej nazwie
- `@ArgumentsSource` – dane z klasy implementującej interfejs `ArgumentsProvider`

Źródła - przykłady



```
@DisplayName("Value Source Test Example")
@ParameterizedTest(name = "{displayName} - [{index}] {arguments}")
@ValueSource(strings = {"Ela", "Kasia", "Ula"})
void shouldBePolishFemaleName(final String nameArg) {
    final boolean result = StringUtil.isPolishFemaleName(nameArg);

    assertThat(result).isTrue();
}
```

```
▼ ✓ Test Results
  ▼ ✓ StringUtilTest
    ▼ ✓ Value Source Test Example
      ✓ Value Source Test Example - [1] Ela
      ✓ Value Source Test Example - [2] Kasia
      ✓ Value Source Test Example - [3] Ula
```

```
@ParameterizedTest
@ValueSource(ints = {1, 2, 5, 22})
void shouldBeNaturalNumber(final int arg) {
    assertTrue(arg > 0);
}
```

Źródła - przykłady



```
public enum Operation {  
    ADD,  
    SUBTRACT,  
    MULTIPLY,  
    DIVIDE  
}
```

```
@ParameterizedTest  
@EnumSource(Operation.class)  
void shouldDisplayOperationName(final  
    Operation operation) {  
    System.out.println(operation.name());  
}
```

```
▼ ✓ Test Results  
  ▼ ✓ EnumSourceTest  
    ▼ ✓ shouldDisplayOperationName(Operation)  
      ✓ [1] ADD  
      ✓ [2] SUBTRACT  
      ✓ [3] MULTIPLY  
      ✓ [4] DIVIDE
```

```
@ParameterizedTest(name = "{displayName} - [{index}] {arguments}")  
@CsvSource({  
    "PL, 38, 0",  
    "EN, 22, 1",  
    "FR, 59, 2"  
})  
void csvInputTest(String countryCode, int numOfResidentsInMillions, int index) {  
    System.out.println(index + ". In " + countryCode + " lives " + numOfResidentsInMillions + " millions of  
people");  
}
```

```
0. In PL lives 38 millions of people  
1. In EN lives 22 millions of people  
2. In FR lives 59 millions of people
```

Źródła – przykłady - @CsvFileSource



- tworzymy katalog resources w podkatalogu test
- oznaczamy go jako Test Sources Root z poziomu intellij
- tworzymy plik np. source.csv z zawartością *PL, 38, 0*

```
@ParameterizedTest
@CsvFileSource(resources = "/source.csv", numLinesToSkip = 0)
void csvFromFileTest(final String countryCode, final int numResidentsInMillions, final int index)
{
    System.out.println(index + ". In " + countryCode + " lives " + numResidentsInMillions + "
millions of people");
}
```

Ćwiczenie



- Napisz test parametryzacyjny dla metody *isTypicalPolishSurname* w klasie *PolishPersonUtil*.
- Napisz test parametryzacyjny dla metody *isWomanWithTypicalPolishSurname* w klasie *PolishPersonUtil*. Wykorzystaj *CsvSource* i następnie *CsvFileSource* jako źródło argumentów.

Źródła – przykłady - @MethodSource



- stworzymy metodę która zwraca argumenty

```
@ParameterizedTest
@MethodSource("getTestArgs")
void fromMethodTest(final String name, final int age) {
    final boolean isAdult = polishPersonUtil.isPolishFemaleAdult(name, age);
    assertThat(isAdult).isTrue();
}

static Stream<Arguments> getTestArgs() {
    return Stream.of(
        Arguments.of("Marcin", 33),
        Arguments.of("Andrzej", 28),
        Arguments.of("Ula", 44));
}
```


Źródła – przykłady - @ArgumentsSource



```
public class CustomArgumentsProvider implements ArgumentsProvider {

    @Override
    public Stream<? extends Arguments> provideArguments(final ExtensionContext extensionContext) throws
Exception {
        return Stream.of(
            Arguments.of(2.5),
            Arguments.of(4.1)
        );
    }
}
```

```
@ParameterizedTest
@ArgumentsSource(CustomArgumentsProvider.class)
void argumentsSourceTest(final double arg) {
    System.out.println(arg);
}
```

Ćwiczenie - 5



- Stwórz klasę Months, a w niej statyczną mapę Map<Month, Integer> która dla danego miesiąca przechowuje ilość dni w miesiącu
- Stwórz w klasie metodę: boolean has31days(Month month);
- Przetestuj tę metodę testem parametryzacyjnym z wybranym przez Ciebie typem źródła.

Dla chętnych / do pracy w domu:

- stwórz metodę w klasie Months: Map<Month, Integer> getByIndexes(Integer... monthIndexes); i przetestuj ją wykorzystując AssertJ.
- bonus: wykorzystaj klasę Table z biblioteki guava aby stworzyć tabelę w której do indeksu i nazwy miesiąca przypiszesz ilość dni

<groupId>com.google.guava</groupId>

<artifactId>guava</artifactId>

Ćwiczenie - 6



- Przetesuj metode compute() w klasie FibonacciSeries korzystając z testów parametryzowanych wykorzystując klasę implementacją interfejs ArgumentsProvider

Dzień 1 - pytania



1. Na jakie 3 sekcje powinniśmy dzielić testy jednostkowe?
2. Jakich typów obiektów powinniśmy unikać pisząc testy jednostkowe?
3. Co oprócz adnotacji ParametrizedTest jest potrzebne do napisania testu parametryzacyjnego?
4. Jaki typ źródła użyć do testów parametryzacyjnych, powinniśmy wybrać ale użyć to źródło w wielu klasach testowych?
5. Jaki typ powinniśmy zwracać w metodzie użytej do źródła MethodSource?
6. Jakie problemy możemy napotkać w testach sprawdzając zawartość takich kontenerów jak HashSet czy HashMap?
7. Ile powinien wynosić Code Coverage naszego kodu?
8. Czym jest blok statyczny w javie?

JUnit – testowanie wyjątków



Co się stanie z testem z poprzedniego zadania dla wartości ujemnych?



Autor: Michał Bojanowski

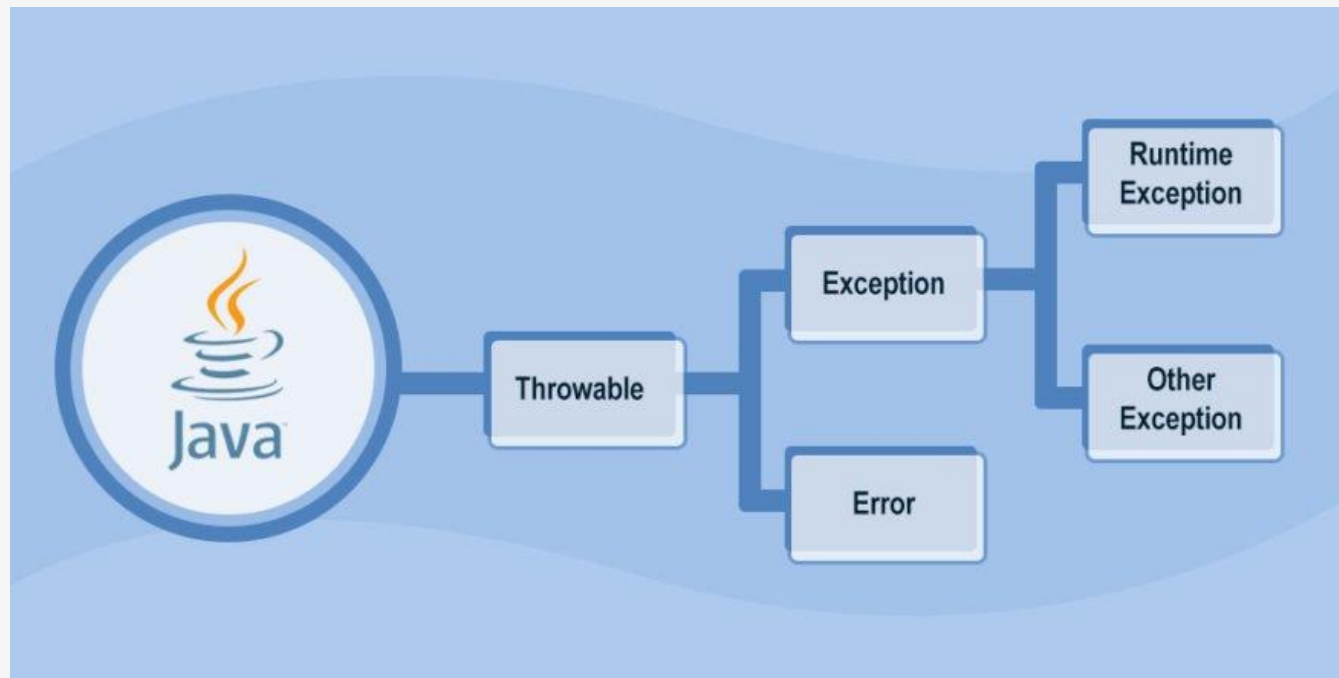
Prawa do korzystania z materiałów posiada Software Development Academy

JUnit – testowanie wyjątków



Wyjątek jest również często „warunkiem brzegowym”

Powinniśmy testować przypadki, w których występują wyjątki!



Autor: Michał Bojanowski

Prawa do korzystania z materiałów posiada Software Development Academy

JUnit – testowanie wyjątków



Jakich wyjątków nie testujemy? Zbyt ogólnych np.

- Exception,
- RuntimeException,
- IOException
- Throwable

Takich, których raczej nie powinniśmy spodziewać się w kodzie produkcyjnym, np.:

- IndexOutOfBoundsException
- NullPointerException
- ArithmeticException
- NumberFormatException
- InterruptedException

JUnit – sposoby testowania wyjątków



- try catch

JUnit5:

- assertThrows

AssertJ:

- assertThatThrownBy
- assertThatExceptionOfType

JUnit4:

- @Test(expected = ExceptionType.class)
- @Rule i ExpectedException

JUnit – try catch



```
@Test
void shouldThrowSomeException() {
    try {
        methodThrowingSomeException();
        fail("Exception not thrown");
    } catch (final DatabaseConnectionException exp) {
        assertEquals("Is it message I excepted?", exp.getMessage());
    }
}
```

- Działa? Tak!
- Czytelne? Można to zrobić lepiej!



JUnit – assertThrows

```
@Test
void shouldThrowSomeException() {
    assertThrows(DatabaseConnectionException.class, () ->
methodThrowingSomeException());
}
```

Mamy typ!

Czego nie wiemy a chcemy sprawdzić?

```
final Throwable throwable =
assertThrows(DatabaseConnectionException.class, () ->
methodThrowingSomeException());
```

Ćwiczenie – 7 – testowanie wyjątków



- Dodaj test dla metody divide w klasie Calculator – użyj metody try catch
- Dodaj test dla metody compute w klasie FibonacciSeries – użyj assertThrows w JUnit5
- Dodaj test dla metody addData w klasie DatabaseStore – użyj assertThatThrownBy
- Dodaj test dla metody removeData w klasie DatabaseConnection – użyj assertThatExceptionOfType
- Dodaj test dla metody setPersonDetails w klasie Person.
- Dodaj testy dla klasy PersonService.

Dla chętnych:

Stwórz nowy project i dorzuć JUnit 4 jako dependencję testową. Stwórz klasę która otwiera plik (path jako input) i zwraca mapę Map<String, Integer> która odpowiada słowom z pliku do liczby ich wystąpień.

Dodaj testy dla tej klasy korzystając z JUnit 4. Przetestuj wystąpienie wyjątku np. FileNotFoundException za pomocą expected w adnotacji @Test i następnie za pomocą @Rule.

Pytania



1. Jakich typów wyjątków nie powinniśmy testować?
2. Jakie typy powinniśmy testować?
3. Jak przetestować wystąpienie wyjątku w JUnit5?
4. Jak przetestować cause i message tego wyjątku?
5. Jak najlepiej testować wyjątki w JUnit4?



1. Jakich typów wyjątków nie powinniśmy testować? (zbyt ogólnych)
2. Jakie typy powinniśmy testować? (przychodzące z zewn. bibliotek, zdefiniowanych przez nas)
3. Jak przetestować wystąpienie wyjątku w JUnit5? (assertThrows)
4. Jak przetestować cause i message tego wyjątku? (za pomocą metod getCause() i getMessage())
5. Jak najlepiej testować wyjątki w JUnit4? za pomocą @Rule i ExpectedException

Podsumowanie



- zawsze powinniśmy testować scenariusze w których występują wyjątki!
- wyjątki, ich typ, message, cause testujemy odpowiednimi asercjami dostępnymi w JUnit5 lub AssertJ
- jeżeli korzystamy z JUnit4 korzystamy z adnotacji @Rule i klasy ExpectedException

TDD – test driven development



TDD – technika wytwarzania oprogramowania, polegająca na

1. Stworzeniu testu przed implementacją funkcjonalności
2. Napisaniu kodu w sposób aby test jak najszybciej był zielony
3. Uporządkowaniu kodu, poprawienie implementacji



TDD – test driven development

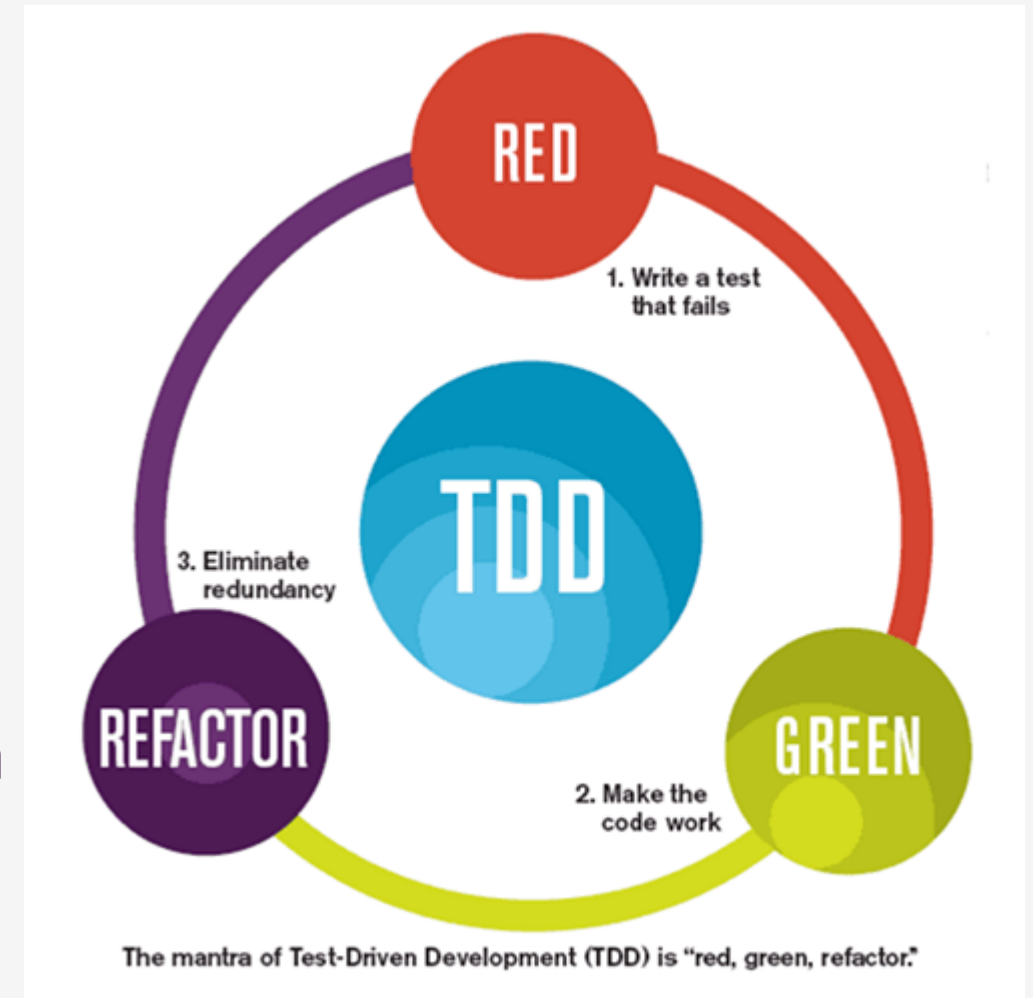


TDD – tzw. RED GREEN REFACTOR

red – napisanie testu, czasem nie kompilującego się

green – napisanie minimalnej implementacji, która powoduje, że test przechodzi

refactor – uporządkowanie i reorganizacja implementacji i testów



Autor: Michał Bojanowski

Prawa do korzystania z materiałów posiada Software Development Academy

TDD – test driven development



Czym TDD nie jest?

- sposobem pisania testów
- definicją jak powinien wyglądać kod
- sposobem pisania testów przez testerów

TDD – plusy?



- wymusza modularność kodu (łatwiejsza do testowania)
- wymusza dobrą architekturę (podział na małe funkcjonalności)
- daje dobre pokrycie kodu testami
- bezpieczny refactor
- mniej bugów! głupie błędy wyłapanie szybciej
- szybka identyfikacja dziwnych wymagań lub ich braku

TDD – wady?



- wymagana dobra dokumentacja, wymagania i architekt
- niektóre testy mogą być bardzo trudne do napisania (integracyjne)
- na początku, kod produkcyjny powstaje wolno
- zmiany w wymaganiach wymagają dużo zmian w testach
- developerzy, którzy długo pracowali w innym systemie pracy, mogą mieć problem przejść na nowy system



DEMO TIME



Zasada FIRST dla testów jednostkowych

- **Isolated, Independent**
- Jak zachować niezależność testu który musi, np. nazwować połączenie z bazą danych lub skontaktować się z zewnętrznym serwisem?
- Odpowiedź: zaMOCKować



mock – atrapa obiektu, najczęściej zależności, która posiada taki sam interfejs jak zależność i który pozwala na zaprogramowanie zachowania w danej sytuacji (np. zwrócenia konkretnego argumentu w trakcie wywołania metody)

spy – atrapa obiektu, często mylona z mockiem, opakowuje **prawdziwy** obiekt. Oprócz programowania zachowania, możemy wywołać prawdziwą implementację

captor – obiekt pozwalający przechwycić argument przekazany do zamockowanej metody

Mockito – mock bez adnotacji



```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>2.24.0</version>
  <scope>test</scope>
</dependency>
```

```
@Test
void mockitoDemo() {
    final Person person = Mockito.mock(Person.class); // stworzenie mocka
    when(person.getFirstName()).thenReturn("HELLO"); // ustalenie zachowania

    System.out.println(person.getFirstName()); // co zostanie wypisane na konsoli?
}
```

Mockito – spy bez adnotacji



Co zostanie wypisane w konsoli?

```
@Test
void mockitoDemo() {
    final Person person = Person.create("Marcin", "Marciniak");
    final Person spiedPerson = Mockito.spy(person);

    when(spiedPerson.getFirstName()).thenReturn("Andrzej");
    when(spiedPerson.getLastName()).thenCallRealMethod();

    System.out.println(spiedPerson.getFirstName() + " " + spiedPerson.getLastName());
}
```


Mockito – adnotacje w JUnit5



```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-junit-jupiter</artifactId>
  <version>2.24.0</version>
  <scope>test</scope>
</dependency>
```

```
@ExtendWith(MockitoExtension.class)
public class MockitoDemo {

    @Mock
    private DatabaseConnection databaseConnection;

    private DatabaseStore databaseStore;

    @Test
    void shouldGetDatabaseStore() {
        when(databaseConnection.isOpen()).thenReturn(true);

        databaseStore = new DatabaseStore(databaseConnection);

        assertThat(databaseStore.getData()).isEmpty();
        verify(databaseConnection).isOpen();
    }
}
```

Mockito – adnotacje w JUnit5



- @Mock – stworzenie instancji mockowanego obiektu
- @Spy – stworzenie instancji szpiegowanego obiektu
- @ExtendWith – rozszerzenie klasy testowej o extension (niekoniecznie związaną z mockito)
- @Captor – instancja przechwytywana argumentu
- @InjectMock – automatyczne stworzenie obiektu z mockowanymi obiektami
- verify
- verifyNoMoreInteractions
- verifyZeroInteractions – weryfikacje sprawdzające czy oczekiwane zachowania zostały wywołane. Sprawdzamy zachowania w sekcji //then!

Mockito – łapanie argumentów



1. Tworzymy „łapacz”

```
@Captor  
private ArgumentCaptor<String> argumentCaptor;
```

2. Łapiemy argument i sprawdzamy przypuszczenia co do jego wartości

```
verify(dunderService).dunderValue(argumentCaptor.capture());  
assertThat(argumentCaptor.getValue()).isEqualTo(TEST_STR);
```

Mockito – bardziej złożony przykład



Wykorzystajmy wiedzę o mockito ze slajdów!
DEMO!

Mockito – argumenty ogólne



- Mockito pozwala definiować bardziej ogólne zachowanie dla mocków
- ArgumentsMatchers i statyczne metody takie jak:
 - `any()`
 - `anyDouble()`
 - `anyCollection()`
 - `eq()`
 - `matches()`
 - `same()`

Mockito – argumenty ogólne - zasady



- Nie mieszamy argumentów konkretnych i ArgumentMatcherów, tzn.

```
when(dunderService.dunderValue(any(String.class), TEST_STR)).thenReturn("ŹLE");
```

- Jeżeli już MUSIMY je wymieszać, używamy np. eq

```
when(dunderService.dunderValue(eq(TEST_STR), any())).thenReturn("DOBRA");
```

- ogólnie lepiej dobrać konkretne wartości do testów i nie używać any()!

Mockito – odpowiedzi



- `when(...).thenReturn()`
- `thenReturn` daje nam dostęp do `InvocationOnMock`
- Co daje `InvocationOnMock`?
 - argumenty metody!
 - i kilka innych:
- Dzięki temu możemy, np. zwrócić n-ty argument mockowanej metody

	<code>getArguments ()</code>	<code>Object []</code>
	<code>getArgument (int i)</code>	<code>T</code>
	<code>callRealMethod ()</code>	<code>Object</code>
	<code>getMethod ()</code>	<code>Method</code>
	<code>getMock ()</code>	<code>Object</code>

Ćwiczenie – 8 – Mockito



- w projekcie sdatdd dodaj test dla dowolnej metody publicznej klasy PersonService. Zamockuj zależności za pomocą mockito.
- dodaj testy dla klasy PersonVerifier. Użyj mockito do zamockowania zależności
- Dodaj testy dla klasy PersonVerifier. Użyj mockito do zamockowania 3 z 4 zależności i użyj spy do czwartej.

Ćwiczenia 9



Sklonuj i zaimportuj projekt: <https://github.com/Mbojanow/sdatdd2>

- Napisz testy jednostkowe za pomocą JUnit4 dla klasy `MessagesService`. Wykorzystaj do tego mockito.
- Napisz testy jednostkowe dla klasy `saveAndSend` w tej samej klasie. Wykorzystaj captor aby zweryfikować messages które będą wysyłane przez `messageSender`. Zamockuj `messagesIdGenerator` w taki sposób aby generować inne id przy każdym wywołaniu.



Sklonuj i zaimportuj projekt: <https://github.com/Mbojanow/sdatdd2>

- Dodaj testy jednostkowe dla klasy PasswordEncoder
- Dodaj testy jednostkowe dla klasy EmailValidator, użyj testów parametryzowanych
- Dodaj testy jednostkowe dla klasy UserRepository, użyj mockito aby zamockować zewnętrzne dependencje, pamiętaj o testowaniu możliwych scenariusz wyrzucających wyjątki
- Dodaj testy jednostkowe dla klasy AuthenticationService, ponownie użyj mockito do zamockowania zależności



Korzystając z <https://github.com/Mbojanow/sdatdd2>

- zmień implementację metod deleteUser, addRole, removeRole w klasie UserRepository aby szukały użytkownika po emailu, a nie po nazwie użytkownika
- spróbuj wykorzystać metodykę tdd do tej zmiany – napisz najpierw testy, a później zajmij się implementacją
- dopisz do klasy AuthenticationService obsługę listy loggedInUsers. Metoda signInUser powinna dodawać do niej użytkownika jeżeli ten się zaloguje po raz pierwszy. Dodaj metodę logoutUser, która będzie usuwała użytkownika z tej listy. Ponownie wykorzystaj metodykę TDD – dopisz najpierw testy.