



# Test Driven Development

Testowanie oprogramowania

Autor: Katarzyna Musiol

Prawa do korzystania z materiałów posiada Software Development Academy



# Czym jest testowanie oprogramowania?

*To jeden z procesów zapewnienia jakości oprogramowania. Ma na celu kontrolę zgodności oprogramowania ze specyfikacją (tzw. **weryfikacja**) i oczekiwaniami użytkownika (tzw. **walidacja**).*

# Kiedy tworzyć testy?



Testy najlepiej tworzyć **od razu** po napisaniu małej spójnej jednostki oprogramowania lub jeszcze **przed**

*(podstawowe założenie Test Driven Development, o czym później).*

Dzięki temu

nie ma potrzeby testowania programu ręcznie na wczesnym etapie jego tworzenia – dzieje się to automatycznie,

w kolejnych etapach nie narasta też **koszt błędów/dług techniczny**,  
testy stanowią swego rodzaju dokumentację.

Złą praktyką jest tworzenie testów na późnym etapie rozwoju oprogramowania, gdy coraz trudniej może być określić szczegóły działania np. metod.



# Rodzaje testów

Koszt i złożoność

Ilość

Jednostkowe

Weryfikują działanie pojedynczych elementów programu, np. poszczególnych metod.

Integracyjne

Testują błędy w interfejsach i komunikacji między modułami.

Funkcjonalne

Tzw. testy czarnej skrzynki. Osoba nie znająca budowy programu testuje go jako jego użytkownik.

Systemowe

Tzw. testy białej skrzynki. Do programu wprowadzane są takie dane testowe, aby przejść każdą możliwą ścieżkę testową (np. sprawdzić każdy warunek).

Akceptacyjne

Służą uzyskaniu formalnego potwierdzenia wykonania oprogramowania odpowiedniej jakości (testy alfa i beta).

W fazie utrzymywania systemu

Testy na istniejącym systemie, wyzwalane modyfikacjami, rozszerzaniem systemu i jego starzeniem się.



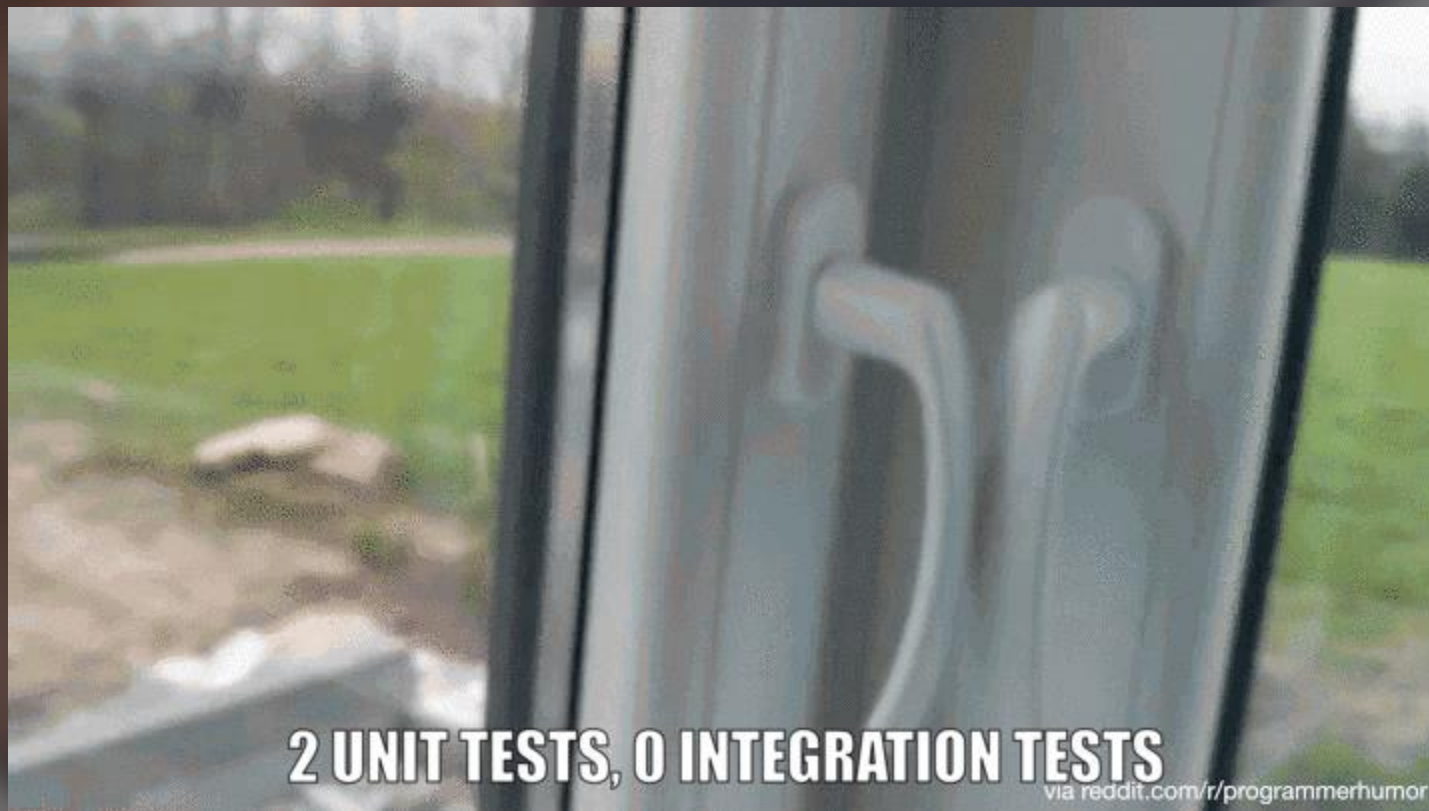


# Porównanie testów jednostkowych i integracyjnych

Cecha	Testy jednostkowe	Testy integracyjne
Zależności	Testowanie pojedynczych elementów w izolacji (metod, klas itd.).	Testowanie zależności modułów (wewnętrznych, zewnętrznych), interfejsów.
Punkty awarii	Jeden potencjalny punkt awarii.	Wiele potencjalnych punktów awarii.
Szybkość	Bardzo szybkie działanie, czas liczony w milisekundach.	Często powolne działanie (operacje dostępu do bazy danych, I/O, operacje na sesji).



# Porównanie testów jednostkowych i integracyjnych







# Czym jest test jednostkowy?

*To fragment kodu sprawdzający działanie niewielkiego obszaru (jednostki) funkcjonalności testowanego kodu.*



Dla testów jednostkowych został sformułowany zbiór 5 zasad (FIRST).

Testy powinny być:

**F**ast – szybkie (*czas uruchomienia < 1s*)

**I**ndependent – niezależne (*można je wywołać w dowolnej kolejności*)

**R**epeatable – powtarzalne (*ten sam rezultat za każdym razem*)

**S**elf-checking – ich rezultat ma być określany automatycznie

**T**imely – pisane w tym samym czasie, co kod produkcyjny





Jeden test testuje jedną jednostkę *(np. metodę, może zawierać kilka metod sprawdzających, ale muszą być one spójne.*

Popularne jest podejście **Given-When-Then**, gdzie najpierw przygotowujemy dane, następnie wykonujemy akcję, którą chcemy testować, a wynik sprawdzamy przy pomocy asercji.

Kod testu powinien być tak samo **przejrzysty** jak kod aplikacji – nie może być pod tym względem gorszy, *dzięki czemu będzie można posługiwać się nim jak dokumentacją.*

Test powinien testować **każdy możliwy przypadek** wywołania metody *(z różnymi danymi testowymi, tak, żeby przetestować cały kod metody).*



# Czym jest jUnit?

*To framework przeznaczony do tworzenia testów jednostkowych w języku Java, zintegrowany z tak popularnymi środowiskami jak IntelliJ, Eclipse czy NetBeans. Korzysta z mechanizmu adnotacji.*



# Przykładowy test jednostkowy

**Klasa testowa** w pakiecie testowym.

Nazewnictwo: **KlasaTest**.

Tworzona z klasy testowanej skrótem **CRTL+SHIFT+T** w IntelliJ.

Metoda z adnotacją **@Before** **wykona się przed każdym testem** i przygotowuje obiekty.

Testy spełnią warunek **FIRST – Independent** (niezależności).

**Metoda testowa** oznaczona adnotacją **@Test**, pojedynczy test.

Nazewnictwo: **testMetoda**.

**Asercja** sprawdzająca czy wartość otrzymana jest zgodna z oczekiwaną.

```
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Before  
    public void setup() {  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void testAdd() {  
        double result = calculator.add(a: 2, b: 3);  
  
        assertThat(result).isEqualTo(5);  
    }  
}
```



# Adnotacje w JUnit

@Before

@After

Metoda, nad którą się znajdują wywoływana jest przed lub po każdym teście.

@Test

Metoda, nad którą się znajduje jest metodą testową.

@BeforeClass

@AfterClass

Metoda, nad którą się znajdują wywoływana raz przed lub po wszystkich testach.

@Ignore

Metoda testowa, nad którą się znajduje jest wyłączana.



# Przykładowe asercje junit

## Asercja

## Opis

[assertEquals](#)(expected, actual)

Sprawdzenie czy wartości – oczekiwana i otrzymana są równe czy różne.

[assertNotEquals](#)(expected, actual)

[assertFalse](#)(value)

Sprawdzenie czy zmienna lub metoda zwracają wartość false lub true.

[assertTrue](#)(value)

[assertNotNull](#)(object)

Sprawdzenie czy obiekt został zainicjalizowany (nie jest nullem) lub nie (jest nullem).

[assertNotNull](#)(object)

[assertArrayEquals](#)([]expected, []actual)

Sprawdzenie czy tablice posiadają taką samą zawartość.

[assertSame](#)(object, object)

Sprawdzenie czy zmienne odwołują lub nie odwołują się do tego samego obiektu.

[assertNotSame](#)(object, object)

[fail](#)(message)

Test nie przechodzi i wyrzuca daną wiadomość.



# Zadania

## Adnotacje

**@Test, @Ignore**  
**@Before, @After, @BeforeClass, @AfterClass**

## Asercje

**assertEquals**(expected, actual)  
**assertNotEquals**(expected, actual)

**assertFalse**(value)  
**assertTrue**(value)

**assertNull**(object)  
**assertNotNull**(object)

**assertArrayEquals**([]expected, []actual)

**assertSame**(object, object)  
**assertNotSame**(object, object)

**fail**(message)

Autor: Katarzyna Musioł

Prawa do korzystania z materiałów posiada Software  
Development Academy

Pobierz projekt *JUnitEx* i zaimportuj go w swoim IDE.

## 1/

1. W pakiecie *com.sda.calc* znajduje się klasa *Calculator* zawierająca podstawowe operacje matematyczne.
2. Stwórz klasę testową i uzupełnij brakujące testy.
3. W każdym z testów zaimplementuj 2-3 asercje.

## 2/

1. W pakiecie *com.sda.air* znajduje się klasa *Airplane*. Obiektowi *Airplane* można przypisać wysokość, którą zmieniać można przy użyciu metod *ascent* i *descent*.
2. Stwórz klasę testową i napisz testy sprawdzające możliwe scenariusze zmiany wysokości.
3. Pamiętaj, żeby odpowiednio przygotować dane.

\*Przetestuj asercje na kolekcjach.





# Czym jest AssertJ?

*To biblioteka matcherów i asercji. Posługuje się składnią zbliżoną do języka naturalnego.*



# Składnia

```
assertThat(frodo)
    .isNotEqualTo(sauron)
    .isIn(fellowshipOfTheRing);

assertThat(frodo.getName())
    .startsWith("Fro")
    .endsWith("do")
    .isEqualToIgnoringCase("frodo");

assertThat(fellowshipOfTheRing)
    .hasSize(9)
    .contains(frodo, sam)
    .doesNotContain(sauron);
```

Składnia wygląda zawsze w następujący sposób:

```
assertThat(value)
    .matcher()
    .matcher()
    ...
```

Możliwe jest łańcuchowanie (chaining) metod.

Składnia zbliżona jest do języka naturalnego, co poprawia czytelność kodu.

```
assertThat(person.getAge())
    .as("%s's age should be equal to 100", person.getName())
    .isEqualTo(100);
```

Wypisywanie komunikatów przy asercjach przy użyciu

```
.as(message, value)
```

```
[Alex's age should be equal to 100] expected:<100> but was:<34>
```




# Przykładowe matchery obiektów

```
public class Dog {  
    private String name;  
    private Float weight;  
  
    // standard getters and setters  
}  
  
Dog fido = new Dog("Fido", 5.25);  
  
Dog fidosClone = new Dog("Fido", 5.25);
```



```
assertThat(fido).isEqualTo(fidosClone);
```



```
assertThat(fido).isEqualToComparingFieldByFieldRecursively(fidosClone);
```

W podanym przykładzie pierwsza asercja nie powiedzie się, gdyż `.isEqualTo()` porównuje referencje do obiektów.

W drugim przypadku problem został rozwiązany poprzez porównanie kolejnych pól obiektów.



# Przykładowe matchery

Klasa	Asercje	Matchery
AbstractObjectAssert	assertThat(object)	<code>.isEqualTo(object)</code> <code>.isEqualToComparingFieldByFieldRecursively(object)</code>
AbstractIterableAssert	assertThat(list)	<code>.isEmpty()</code> <code>.contains(value)</code> <code>.startsWith(value)</code> <code>.doesNotContainNull()</code> <code>.containsSequence(value1, value2)</code>
AbstractBooleanAssert	assertThat(boolean)	<code>.isTrue()</code> <code>.isFalse()</code>
AbstractCharacterAssert	assertThat(char)	<code>.isNotEqualTo(char)</code> <code>.inUnicode()</code> <code>.isGreaterThanOrEqualTo(char)</code> <code>.isLowerCase()</code>
AbstractClassAssert	assertThat(class)	<code>.isInterface()</code> <code>.isAssignableFrom(class)</code>



# Przykładowe matchery

Klasa	Asercje	Matchery
AbstractFileAssert	assertThat(file)	<code>.exists()</code> <code>.isFile()</code> <code>.canRead()</code> <code>.canWrite()</code>
AbstractDoubleAssert	assertThat(number)	<code>.isEqualTo(value, withPrecision(1d))</code>
AbstractInputStreamAssert	assertThat(inputStream)	<code>.hasSameContentAs(expected)</code>
AbstractThrowableAssert	assertThat(throwable)	<code>.hasNoCause()</code> <code>.hasMessageEndingWith(value)</code>
AbstractMapAssert	assertThat(map)	<code>.isEmpty()</code> <code>.containsKey(key)</code> <code>.doesNotContainKeys(key)</code> <code>.contains(entry(key, value))</code> <code>.size().isGreaterThan(value)</code>
	assertThat(string)	<code>.startsWith(string)</code> <code>.endsWith(string)</code> <code>.isEqualToIgnoringCase(string)</code>



# Tworzenie własnych asercji

Rozszerzenie klasy asercji, w tym przypadku `AbstractAssert`.

Konstruktor wywołujący konstruktor nadklasy.

Metoda statyczna asercji, jak zawsze `AssertThat` konstruuje obiekt, na którym będziemy operować.

Matcher, w tym przypadku `.hasName()`.

Zwraca obiekt, na którym operujemy przy pomocy

`return this`,

dzięki czemu mamy możliwość łańcuchowania matcherów.

```
public class TolkienCharacterAssert extends AbstractAssert<TolkienCharacterAssert, TolkienCharacter> {

    public TolkienCharacterAssert(TolkienCharacter actual) {
        super(actual, TolkienCharacterAssert.class);
    }

    public static TolkienCharacterAssert assertThat(TolkienCharacter actual) {
        return new TolkienCharacterAssert(actual);
    }

    public TolkienCharacterAssert hasName(String name) {

        isNotNull();

        if (!Objects.equals(actual.getName(), name)) {
            failWithMessage("Expected character's name to be <%s> but was <%s>", name, actual.getName());
        }

        return this;
    }
}
```





## Zadania

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

*Składnia*

```
assertThat(value)
    .matcher()
    ...
```

```
assertThat(actual)
    .as(message, value)
    .isEqualTo(expected);
```

```
assertThat(list).size().isEqualTo(value);
```

Autor: Katarzyna Musioł

Prawa do korzystania z materiałów posiada Software  
Development Academy

Pobierz projekt *JUnitEx* i zaimportuj go w swoim IDE.

### 1/

1. W pakiecie *com.sda.calc* znajduje się klasa *Calculator*.
2. Korzystając z napisanych wcześniej testów, zmodyfikuj je tak, aby korzystać z matcherów AssertJ (*.isEqualTo()*, *.isNotEqualTo()*).
3. Wykorzystaj metodę *.as()*, żeby wyświetlić komunikat o błędzie w przypadku niepowodzenia.

### 2/

1. W pakiecie *com.sda.db* znajdują się klasy *Database* i *User*. Symulują one prostą bazę danych.
2. Dodaj użytkowników do *Database* i sprawdź czy ich liczba się zgadza.
3. Dodaj nowego użytkownika, a następnie wyszukaj go i sprawdź czy dane zapisały się poprawnie oraz jak zmienił się rozmiar listy.
- 4.\* Przeprowadź taki test dla usuwania.



# Czym są testy parametryzowane?

*To testy służące do testowania tego samego kodu przy użyciu różnych parametrów (danych wejściowych).*

# Zalety testów parametryzowanych



Pozwalają zmniejszyć ilość pisanego kodu (liczbę asercji), dzięki czemu zwiększają jego czytelność.

Kod testowy nie jest zbędnie powielany.

W przypadku niepowodzenia testu wiemy dokładnie, dla których danych występuje błąd.



# Narzędzia do testów parametryzowanych

## Parametrized

- Wbudowane narzędzie JUnit.
- Nie potrzeba dodatkowych bibliotek.

### Dokumentacja

<http://junit.org/junit4/javadoc/4.12/org/junit/runners/Parameterized.html>

## JUnitParams

- Projekt polskiej firmy Pragmatics.
- Łatwa integracja z JUnit.
- Mniej kodu w porównaniu do Parameterized.
- Można mieszać testy parametryzowane z nieparametryzowanymi.
- Parametry można pobrać z pliku CSV lub klasy dostawcy parametrów.

### Dokumentacja

<https://github.com/Pragmatists/JUnitParams>



# Parameterized

Zdefiniowanie testu parametryzowanego poprzez `@RunWith(Parameterized.class)`.

Wartości parametrów przechowywane przez zmienne.

Konstruktor przypisujący kolejne wartości testowe.

Metoda statyczna z kolejnymi wartościami parametrów oznaczona przez `@Parameters`.

Test z wartościami przekazanymi z `getParameters()`.

```
@RunWith(Parameterized.class)
public class CalculatorTest {

    private Calculator calculator;
    private Double valueA;
    private Double valueB;

    public CalculatorTest(Double valueA, Double valueB) {
        this.valueA = valueA;
        this.valueB = valueB;
    }

    @Before
    public void setup() {
        calculator = new Calculator();
    }

    @Parameterized.Parameters
    public static Collection getParameters() {
        return Arrays.asList(new Double[][]{{1.0, 2.5}, {3.3, 5.0}, {2d, 4d}});
    }

    @Test
    public void testAdd() {
        double result = calculator.add(valueA, valueB);
        assertThat(result).isEqualTo(valueA+valueB);
    }
}
```

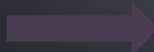


# JUnitParams

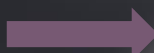
Zdefiniowanie testu parametryzowanego poprzez  
`@RunWith(JUnitParamsRunner.class)`.



Metoda statyczna z kolejnymi wartościami parametrów oznaczona  
przez `@Parameters`.



Test z wartościami przekazanymi listy parametrów.



```
@RunWith(JUnitParamsRunner.class)
public class CalculatorTestParams {

    private Calculator calculator;

    @Before
    public void setup() {
        calculator = new Calculator();
    }

    @Test
    @Parameters({"1d,2d", "8.5,3.5"})
    public void testAdd(double valueA, double valueB) {
        assertThat(calculator.add(valueA, valueB)).isEqualTo(valueA+valueB);
    }
}
```





# Zadania

## Składnia

```
@RunWith(Parameterized.class)
```

```
+ konstruktor
```

```
+ @Parameters
```

```
    getParameters()
```

```
@RunWith(JUnitParamsRunner.class)
```

```
+ @Test
```

```
+ @Parameters({})
```

Autor: Katarzyna Musioł

Prawa do korzystania z materiałów posiada Software  
Development Academy

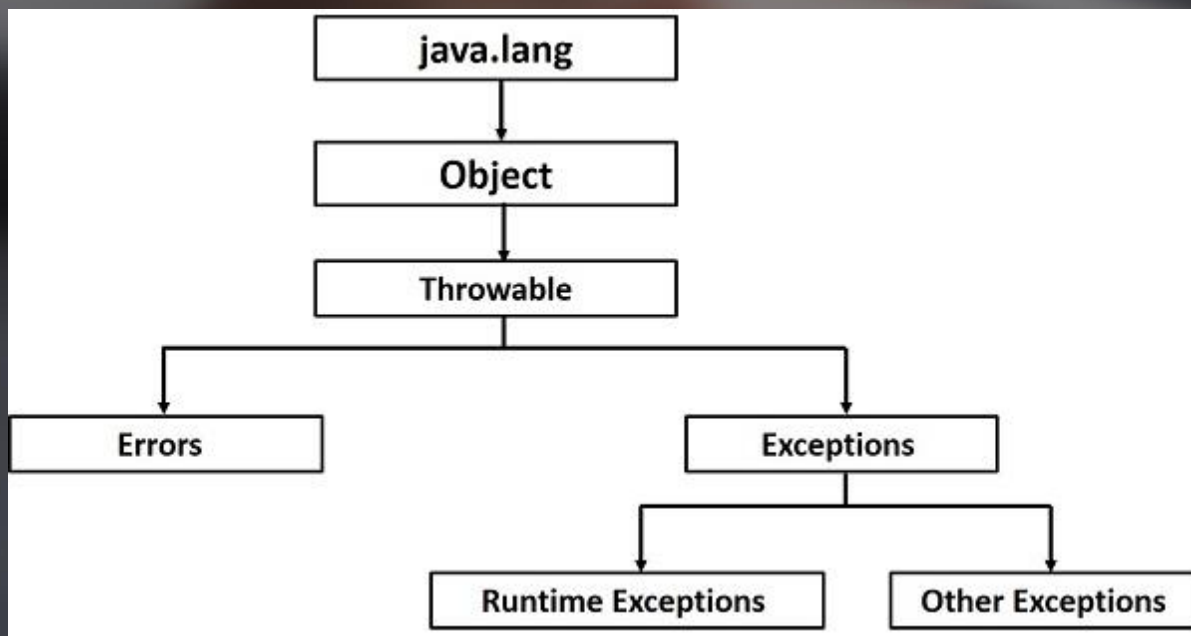
Pobierz projekt *JUnitEx* i zaimportuj go w swoim IDE.

## 1/

1. W pakiecie *com.sda.calc* znajduje się klasa *Calculator*.
2. Zmodyfikuj wcześniej napisane testy tak, aby korzystały z JUnitParams. Dla porównania możesz skorzystać z Parameterized.
3. Korzystając z <https://github.com/Pragmatists/junitparams/wiki/Quickstart> wypróbuj jeden ze sposobów na przekazywanie parametrów z zewnątrz, np. z klasy, metody.



# Wyjątki w Java



## Runtime Exceptions (Unchecked) :

`NullPointerException`  
`NumberFormatException`  
`ClassCastException`  
`IndexOutOfBoundsException`  
`ArithmeticException`

## Other Exceptions (Checked) :

`SQLException`  
`IOException`  
`MalformedURLException`



# Po co testować wyjątki?

Gdy chcemy mieć pewność, że

system **poprawnie reaguje na wprowadzenie błędnych wartości** (*np. dzielenie przez zero*),

w przypadku wystąpienia określonych problemów **system wyrzuci odpowiedni zdefiniowany przez nas wyjątek**,

**informacja** zawarta w rzuconym wyjątku **jest poprawna**.



# Sposoby testowania wyjątków

## Parametr expected

```
@Test(expected = NullPointerException.class)
public void testAddUserException() {
    database.addUser(null);
}
```

## try - catch

```
@Test
public void testAddUserException() {
    try {
        database.addUser(null);
    } catch (Exception e) {
        assertThat(e).assertInstanceOf(NullPointerException.class);
    }
}
```

## Adnotacja @Rule

```
public class DatabaseTest {

    private Database database;

    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Before
    public void setup() { ... }

    @Test
    public void testAddUserException() {
        thrown.expect(NullPointerException.class);
        //thrown.expectMessage();

        database.addUser(null);
    }
}
```

Gdy testowana metoda zwraca wynik, warto też przetestować przy użyciu asercji czy w przypadku wystąpienia wyjątku zwróci null.



## Zadania

*Sposoby:*

1. `@Test(expected = ... )`
2. `try - catch`
3. `@Rule`  
`ExpectedException ...`

Pobierz projekt *JUnitEx* i zaimportuj go w swoim IDE.

### 1/

1. W pakiecie *com.sda.db* znajdują się klasy *Database* i *User*. Symulują one prostą bazę danych.
2. Spróbuj dodać do bazy wartość null zamiast użytkownika, zobacz, co się stanie i napisz test wyłapujący odpowiedni wyjątek.
3. Powtórz czynności z punktu 2, tym razem dodając użytkownika z istniejącym już w bazie loginem.

### 2/

1. W pakiecie *com.sda.calc* znajduje się klasa *Calculator*.
2. Sprawdź, które z metod rzucają wyjątki i przetestuj je. Pamiętaj o upewnieniu się, że w przypadku podania niewłaściwych danych wejściowych, metoda zwraca null.



# Czym jest TDD?

*Inaczej Test Driven Development – technika tworzenia oprogramowania, w skrócie polegająca na uprzednim pisaniu testów, a później kodu programu.*





# Cykl TDD

W TDD obowiązuje cykl zwany w skrócie **cyklem RGR**, czyli **Red-Green-Refactor**  
(inaczej *TDD Mantra*),

w jego skład wchodzi następujące kroki:

1. Tworzenie testów.
2. Implementowanie funkcjonalności.
3. Refaktoryzacja kodu.

Głównym **celem TDD**, wbrew nazwie **nie jest pisanie testów** ani też próba wyeliminowania pracy testera.

TDD **zwiększa efektywność** pisania i **zwięzłość kodu**, a przy okazji również **pokrycie kodu testami**.



# Red

Red

W pierwszym etapie, tzw. Red

1. **Piszemy test**, który ma opisywać pożądanie zachowanie nowej metody.
2. **Uruchamiamy test.**
3. **Test kończy się niepowodzeniem**, gdyż nie ma jeszcze kodu żądanej funkcjonalności.



# Green

Red

Green

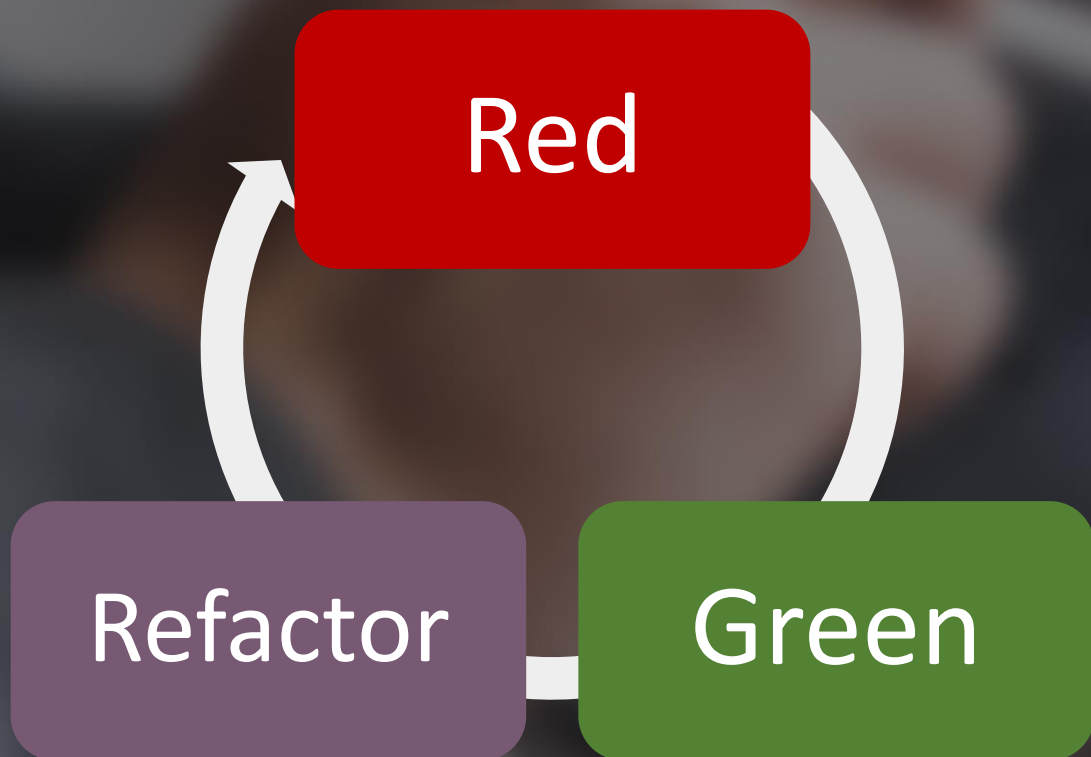
W drugim etapie, tzw. Green

1. **Implementujemy funkcjonalność**, do której wcześniej utworzyliśmy test.
2. **Piszemy kod w taki sposób, żeby test się powiódł.**
3. Nie implementujemy od razu wszystkich możliwych scenariuszy działania metody (skrajnych przypadków itd.).



# Refactor

W ostatnim etapie, tzw. Refactor



1. **Refaktoryzujemy kod**, czyli poprawiamy jego jakość np. poprzez usunięcie duplikacji i nieużywanych zmiennych, skracamy.
2. **Ponownie uruchamiamy test.**
3. **Jeśli test kończy się powodzeniem**, wracamy do pierwszego kroku i piszemy kolejny test.



# Zadania

Kroki:

1. Red (pisanie testów)
2. Green (pisanie kodu programu i sprawdzenie)
3. Refactor (refaktoryzacja)

Pobierz projekt *JUnitEx* i zaimportuj go w swoim IDE.

1/

1. W pakiecie *com.sda.db* znajdują się klasy *Database* i *User*. Symulują one prostą bazę danych.
2. Stosując technikę TDD dopisz następujące funkcje:
  - a. Wyszukiwanie użytkowników po urywku nazwy.
  - b. Modyfikacja użytkownika wybranego przy pomocy jego loginu.
  - c. Dodaj odpowiedni wyjątek dla przypadku nie odnalezienia użytkownika w metodzie przeznaczonej do modyfikacji. Zastosuj wyjątek w pozostałych metodach i uaktualnij testy.
3. Stosując technikę TDD stwórz klasę *Authentication*, która:
  - a. Pozwoli użytkownikowi zalogować się przy użyciu loginu i hasła (utwórz odpowiedni wyjątek dla niepoprawnego hasła),
  - b. Będzie pamiętać, jaki użytkownik jest obecnie zalogowany (jego login),
  - c. Umożliwi wylogowanie się.

2/

1. W pakiecie *com.sda.calc* znajduje się klasa *Calculator*.
2. Stosując technikę TDD dopisz następujące funkcje:
  - a. Obliczanie dowolnej potęgi liczby.
  - b. Sprawdzanie czy liczba jest podzielna przez drugą liczbę.
  - c. Sumowanie tablicy liczb.



# Czym jest mockowanie?

Autor: Katarzyna Musioł  
Prawa do korzystania z materiałów posiada Software  
Development Academy



# Mockowanie – po co?

Wraz z rozwojem programu wykorzystuje się zewnętrzne biblioteki, bazy danych itp..

Metody mogą zwracać wynik niedeterministyczny  
(godzinę, temperaturę itd.).

Obiekt może posiadać stany trudne do zreplikowania lub potrzebuje do działania wielu innych obiektów.

Podczas normalnego działania, program może korzystać z ogromnych zasobów danych  
(np. bazy danych).

Implementacja danej klasy jest niekompletna, aktualnie modyfikowana lub jeszcze nie istnieje.





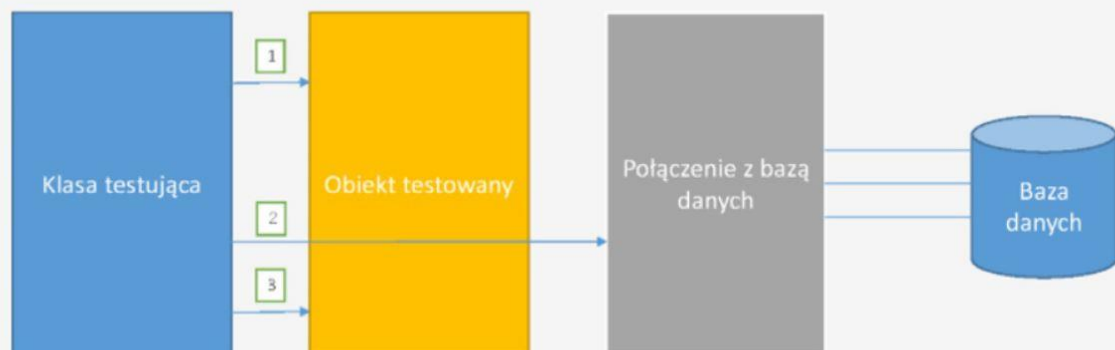
# Czym jest mock?

*Inaczej **Atrapa obiektu (ang. mock object)** – symulowany obiekt, który w kontrolowany sposób naśladuje zachowanie rzeczywistego obiektu. Programista tworzy atrapy obiektów w celu przetestowania zachowania jakiegoś innego obiektu, podobnie jak projektanci samochodów wykorzystują manekiny do symulacji zachowania ludzkiego ciała podczas zderzenia pojazdów.”*

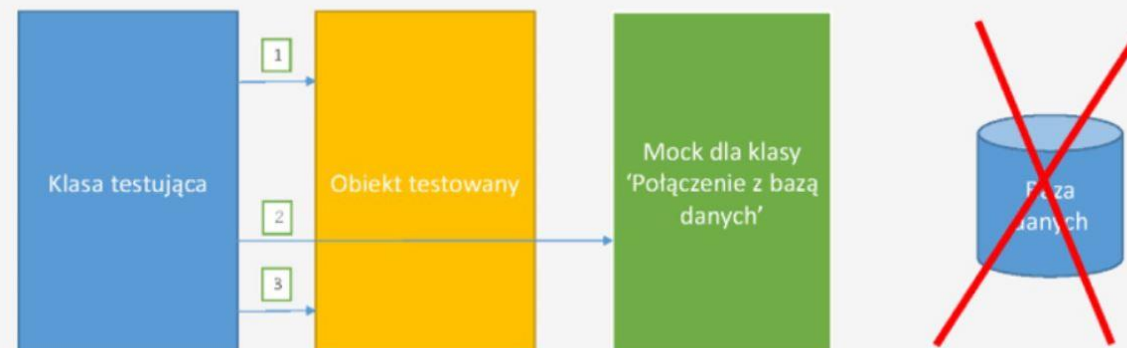


# Konfiguracja środowiska testowego bez i z użyciem mocka

Bez użycia mocka, konieczny dostęp do bazy danych.



Z użyciem mocka, brak konieczności dostępu do bazy danych.





# Zalety i wady mockowania

## Zalety

- Atrapy posiadają **identyczny interfejs** jak obiekty, które naśladują.
- Można w prosty sposób zasymulować **wiele scenariuszy zachowań**.
- Atrapa może być **wielokrotnie uruchamiana**, dzięki czemu można sprawdzić czy metody zawsze zwracają ten sam wynik *(bez konieczności operowania np. na bazie danych)*.
- Testowana klasa może być uruchamiana **w wielu konfiguracjach**.

## Wady

- W przypadku złożonych obiektów konfiguracja mocka może być pracochłonna.
- **Nie można** mockować klas finalnych i finalnych lub statycznych metod.
- **Nie można** mockować metod equals() i hashCode().
- **Nie można** mockować konstruktorów.



# mockito



<https://site.mockito.org/>

<https://www.baeldung.com/mockito-annotations>

Autor: Katarzyna Musioł  
Prawa do korzystania z materiałów posiada Software  
Development Academy



# Verify

Zdefiniowanie mocka poprzez `mock()`, zamiennie można używać adnotacji `@Mock`.

Metoda sprawdzająca wywołanie żądanej funkcji, `verify()`.

Metoda sprawdzająca czy funkcja `add()` z argumentem „once” została wywołana 1 raz.

Inne warianty `verify()`.

```
import static org.mockito.Mockito.*;
```

```
// mock creation
```

```
List mockedList = mock(List.class);
```

```
// using mock object - it does not throw any "unexpected interaction" exception
```

```
mockedList.add("one");
```

```
mockedList.clear();
```

```
// selective, explicit, highly readable verification
```

```
verify(mockedList).add("one");
```

```
verify(mockedList).clear();
```

```
verify(mockedList).add("once");
```

```
verify(mockedList, times(1)).add("once");
```

```
verify(mockedList, never()).add("never happened");
```

```
//verification using atLeast()/atMost()
```

```
verify(mockedList, atLeastOnce()).add("three times");
```

```
verify(mockedList, atLeast(2)).add("three times");
```

```
verify(mockedList, atMost(5)).add("three times");
```





# Stubbing

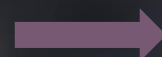
**Zdefiniowanie zachowania mocka** poprzez `when().thenReturn()`.



```
// stubbing appears before the actual execution  
when(mockedList.get(0)).thenReturn("first");
```

**Działanie zdefiniowanego zachowania.**

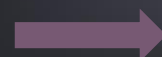
Zwrócenie sztywnej wartości zadziała tylko w pierwszym przypadku, gdyż zdefiniowano zwracaną wartość tylko dla przypadku `.get(0)`.



```
// the following prints "first"  
System.out.println(mockedList.get(0));  
  
// the following prints "null" because get(999) was not stubbed  
System.out.println(mockedList.get(999));
```

**Rozwiązanie problemu** przy pomocy `anyInt()`.

Teraz wartość zostanie zwrócona dla każdego inta podanego w `.get()`.



```
// stubbing using built-in anyInt() argument matcher  
when(mockedList.get(anyInt())).thenReturn("element");
```



# Adnotacje

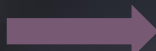
Adnotacja dostarczająca Runner pozwalający korzystać z Mockito

`@RunWith(mockitoJUnitRunner.class)`



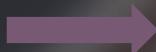
```
@RunWith(MockitoJUnitRunner.class)
public class MockitoAnnotationTest {
    ...
}
```

Alternatywny sposób poprzez wywołanie  
`MockitoAnnotations.initMocks(this);`



```
@Before
public void init() {
    MockitoAnnotations.initMocks(this);
}
```

Oznaczenie mocka przy pomocy `@Mock`.



```
@Mock
List<String> mockedList;
```





# Adnotacje

Wstrzyknięcie mocków przez  
`@InjectMocks`

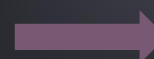
Wstrzykuje wszystkie atrapy do oznaczonego obiektu.



```
public class MyDictionary {  
    Map<String, String> wordMap;  
}
```

```
@Mock  
Map<String, String> wordMap;  
  
@InjectMocks  
MyDictionary dic = new MyDictionary();
```

Alternatywny sposób poprzez konstruktor.



```
public class Person {  
  
    private Car car;  
  
    public Person(Car car) {  
        this.car = car;  
    }  
}
```

```
@RunWith(MockitoJUnitRunner.class)  
public class SampleTest {  
  
    @Mock  
    Car car;  
  
    Person person;  
  
    @Before  
    public void setUp() {  
        person = new Person(car);  
    }  
}
```



# Adnotacje\*

**Spy\*** pozwala korzystać z prawdziwych obiektów i definiować na sztywno działanie wybranych metod.

Oznaczany przy użyciu:

**@Spy**

W podanym przykładzie wywołanie metody `.add()` jest prawdziwym wywołaniem tej metody, a wywołanie metody `.size()` nie.

```
@Spy
List<String> spiedList = new ArrayList<String>();

@Test
public void whenUseSpyAnnotation_thenSpyIsInjected() {
    spiedList.add("one");
    spiedList.add("two");

    Mockito.verify(spiedList).add("one");
    Mockito.verify(spiedList).add("two");

    assertEquals(2, spiedList.size());

    Mockito.doReturn(100).when(spiedList).size();
    assertEquals(100, spiedList.size());
}
```



## Zadania

Składnia:

```
@RunWith(MockitoJUnitRunner.class)

@Mock, @InjectMocks, @Spy

MockitoAnnotations.initMocks(this);

verify(object).objectMethod();
verify(object, times(int)).objectMethod();
    never()
    atLeastOnce()
    atLeast()
    atMost()

when().thenReturn()

anyInt()
```

Autor: Katarzyna Musioł

Prawa do korzystania z materiałów posiada Software  
Development Academy

Pobierz projekt *JUnitEx* i zaimportuj go w swoim IDE.

### 1/

1. W pakiecie *com.sda.db* znajdują się klasy *Database* i *User*. Symuluj one prostą bazę danych.
2. Utwórz nowy test i atrapę (mocka) klasy *Database*. Przeprowadź testy:
  - a. Zapisz użytkownika i sprawdź czy metoda wykonała się tylko jeden raz.
  - b. Stwórz obiekt klasy *User*, a następnie zwróć go na sztywno przy pomocy `when().thenReturn()` i odpowiedniej metody z *Database*. Sprawdź czy obiekt jest tym samym, który wcześniej utworzyłeś.
  - c. Jak powyżej zmodyfikuj metodę zwracającą wszystkich użytkowników, aby w liście wynikowej pod dowolnym indeksem zwracała stworzonego przez ciebie użytkownika.
  - d. Dodaj kilku użytkowników i sprawdź, ile razy została wywołana metoda przy użyciu różnych metod (`times()`, `atLeast()`).
3. Stwórz mocka klasy *Database* w teście klasy *Authentication*. Przeprowadź podobne testy.
- 4.\* Wypróbuj powyższe pomysły używając Spy'a (`@Spy`) i porównaj wyniki.