



# Mockito



**Michał Nowakowski**

Lead Software Engineer @EPAM

[michal@nowakowski.me.uk](mailto:michal@nowakowski.me.uk)

# Mockito

// Podstawowe pojęcia

// Tworzenie mocka

// Mockowanie metod

// Weryfikacja wywołania metody (i sprawdzenie parametrów)

// Q&A

# Materiały do zajęć

// <https://github.com/infoshareacademy/jbdd6-materiały-mockito>



# Mockito

Podstawowe pojęcia

# Mockito

- // **Mockito** to framework, który znacznie ułatwia testowanie kodu
- // Najpopularniejszy framework do mockowania (ponad 2 mln użytkowników)
- // Dzięki mockowaniu testy mogą być małe i sprawdzać wyizolowaną logikę
- // Możemy testować kod, który zależy od niedostępnych funkcjonalności (np. jeszcze niedziałających poprawnie) lub czasochłonnych (np. połączenie z bazą danych)

# Mockito

- // Pozwala tworzyć **obiekty zastępcze (mocki)**, co ułatwia testowanie
- // Umożliwia pisanie testów bez wykorzystywania zewnętrznych systemów – np. usług sieciowych
- // Testy są bardzo czytelne, wyizolowane od pozostałej logiki
- // Pozwala sprawdzać zachowanie metod, np. ilość wywołań danej metody, przekazane argumenty
- // Używany z JUnit

# Podstawowe pojęcia

// **fake – uproszczona implementacja** obiektu, który chcemy zastąpić, np. baza danych w pamięci RAM

// **stub – „prawdziwy obiekt”**, który posiada implementację, ale można mu zmienić **konkretne zachowanie**, np. wybraną metodę

// **mock** – obiekt, który **nie posiada implementacji**, ale wymaga skonfigurowania specjalnie dla testu

# Podstawowe pojęcia

// **spy** – działa jak proxy (pośrednik), część metod wywołuje na „prawdziwym obiekcie”, a część **symuluje**

// **dummy** – obiekt, który jest wykorzystywany jako argument tylko na potrzeby poprawnej kompilacji (**nie jest używany w teście**)



## Mockito

Tworzenie mocka

# Konfiguracja

// Do zależności projektu należy dodać JUnit, Mockito i wsparcie Mockito w JUnit (**wystarczy scope test – dlaczego?**)

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.25.1</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.4.1</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>2.25.1</version>
    <scope>test</scope>
</dependency>
```

# Tworzenie mocka

Metoda statyczna mock()

// Najprościej utworzyć mock przez wywołanie statycznej metody mock() w ciele metody

// Parametr to klasa, która chcemy zamockować (np. Currency.class)

```
private UserDao userDao;
private UserService userService;

@BeforeEach
public void setUp() {
    userDao = mock(UserDao.class);
    userService = new UserService(userDao);
}
```

# Tworzenie mocka

Adnotacja @Mock

// Możemy także użyć adnotacji `@Mock` do utworzenia mocka i `@InjectMock` do wstrzyknięcia mocka do klasy, która potrzebuje go jako zależności

// Należy pamiętać o adnotacji `@ExtendWith` na klasie z testami!!!

```
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {
    @Mock
    private UserDao userDao;

    @InjectMocks
    private UserService userService;
```

# Brak zdefiniowanego zachowania

// Co się stanie, jeśli wywołamy na mocku metody bez wcześniejszego zdefiniowania ich zachowania?

```
@Test
public void returnFromMock() {
    User user = userDao.findUser("admin");
    System.out.println(user);

    List<User> users = userService.getAllUsers();
    System.out.println(users);
}
```

# Brak zdefiniowanego zachowania

// Co się stanie, jeśli wywołamy na mocku metody bez wcześniejszego zdefiniowania ich zachowania?

Domyślny  
wartość dla  
obiektu: null

```
public void returnFromMock() {  
    User user = userDao.findUser(1);  
    System.out.println(user);  
  
    List<User> users = userService.getAllUsers();  
    System.out.println(users);  
}
```

Domyślna wartość dla  
kolekcji: pusta  
kolekcja []



## Mockito

Mockowanie metod – definiowanie zachowania

# Definiowanie zachowania

// Dla każdej metody mocka możemy zdefiniować zachowanie – tj.  
co dana metoda zwraca (wartość)

```
// given
List<User> users = Arrays.asList(
    new User("admin"), new User("michall")
);

when (userDao.getAllUsers()).thenReturn(users);

// when
List<User> result = userService.getAllUsers();

// then
assertThat(result).hasSize(2);
```

# Definiowanie zachowania

// when(**mock.metoda()**).thenReturn(**zwracanaWartość**):

```
// given
List<User> users = Arrays.asList(
    new User("admin"), new User("michall")
);

when (userDao.getAllUsers()).thenReturn(users);

// when
List<User> result = userService.getAllUsers();

// then
assertThat(result).hasSize(2);
```

# Zadanie 1

// Uzupełnij test `shouldGetAllUsers` tak, aby metoda `userService.getAllUsers()` zwracała 4 różnych użytkowników

# Zadanie 2

- // Uzupełnij test *shouldReturnAllUsersMatchingPattern* tak, aby metoda *userService.findUsers()* zwracał 2 użytkowników zawierających „ow” w nazwie
- // Metoda *userDao.getAllUsers()* powinna zwracać kolekcję użytkowników (dwóch z „ow” w nazwie i kilku innych celem przetestowania logiki metody *userService.findUsers()*)

# Definiowanie zachowania dla parametrów

// Ta sama metoda może zwracać różne wartości w zależności od przyjmowanych argumentów

// W tym celu stosuje się tzw. ArgumentMatcher

```
// given
when(userDao.findUser(any()))
    .thenReturn(new User("kowalski"));

// when
boolean result = userService.doesUserExist("kowalski");

// then
assertThat(result).isEqualTo(true);
```

# Definiowanie zachowania dla parametrów

```
// when(mock.metoda(ArgumentMatcher))  
.thenReturn(zwracanaWartość);
```

```
// given  
when(userDao.findUser(any()))  
    .thenReturn(new User("kowalski"));  
  
// when  
boolean result = userService.doesUserExist("kowalski");  
  
// then  
assertThat(result).isEqualTo(true);
```

# ArgumentMatchers

// Pozwalają na zdefiniowanie zachowania dla mockowanej metody w zależności od parametrów wejściowych – przekazywanych przy wywołaniu metody

// Zastępują parametry w wywołaniu metody  
`when(mock.metoda(...))`

// `any()`, `anyBoolean()`, `anyString()`, `anyInt()`, ... - wartości dowolne (metoda przyjmie wszystko)

// `eq(T value)` – konkretna wartość zdefiniowania przez programistę

# Definiowanie zachowania dla parametrów

// Zwrócenie konkretnej wartości w zależności od parametru (matcher eq()):

```
// given
when(userDao.findUser(eq("kowalski")))
    .thenReturn(new User("kowalski"));

// when
boolean result = userService.doesUserExist("kowalski");

// then
assertThat(result).isEqualTo(true);
```

# Zadanie 3

// Uzupełnij test *shouldReturnTrueIfUserExists* tak, aby metoda *userService.doesUserExist()* zwracała prawdę dla użytkownika o nazwie *admin*

# Zadanie 4

// Uzupełnij test *shouldReturnFalseIfUserDoesNotExist* tak, aby metoda *userService.doesUserExist()* zwracała fałsz dla użytkownika o nazwie *janek1*

# Rzucanie wyjątków

// Jeśli dana metoda ma rzucić wyjątek, musimy użyć konstrukcji  
`doThrow(wyjątek).when(mock).nazwaMetody(parametry):`

```
// given
doThrow(new RuntimeException()).when(userDao).deleteUser(any());

// when, then
assertThrows(RuntimeException.class,
    () -> userService.deleteUser(new User("test")));
```

# Zadanie 5

// Uzupełnij test

*shouldThrowExceptionWhenRemovingNonExistingUser* tak, aby metoda *userDao.deleteUser()* rzuciła wyjątek *RuntimeException* przy próbie usunięcia użytkownika *ania13*



# Mockito

Weryfikacja wywołania

# Weryfikacja wywołania

- // Często zdarza się, że nasz kod nie zwraca konkretnych wyników tylko wywołuje kolejne metody z określona liczbą parametrów (np. z innych klas)
  
- // W celu przetestowania takiego kodu należy sprawdzić, czy parametry wywołania zewnętrznych metod są zgodne z oczekiwaniami

# Sprawdzenie parametrów (I)

- // Aby sprawdzić wartości parametrów przekazanych do mocka, należy skorzystać z klasy **ArgumentCaptor**
- // Obiekt **ArgumentCaptor** tworzymy poprzez wywołanie statycznej metody `ArgumentCaptor.forClass(klasa)`
- // W wywołaniu `mock()` w miejscu przekazania parametrów wywołujemy metodę `ArgumentCaptor.capture()`
- // Po użyciu mocka należy sprawdzić wartość poprzez wywołanie `ArgumentCaptor.getValue()`

# Sprawdzenie parametrów (I)

```
// given
ArgumentCaptor<String> nameCaptor = ArgumentCaptor.forClass(String.class);
when(userDao.findUser(nameCaptor.capture()))
    .thenReturn(new User("kowalski"));

// when
boolean result = userService.doesUserExist("kowalski");

// then
assertThat(result).isTrue();
assertThat(nameCaptor.getValue()).isEqualTo("kowalski");
```

# Zadanie 6

// Rozszerz test `shouldReturnTrueIfUserExists` o sprawdzenie parametrów przekazanych do metody `userDao.findUser()`

// Wykorzystaj klasę **ArgumentCaptor**

# Sprawdzenie parametrów (II)

// Wywołanie metody (a także przekazane parametry) możemy sprawdzić za pomocą metody `verify()`

```
verify(mock, times(ileRazy)).nazwaMetody(parametry)
```

// Aby zweryfikować, że na mocku nie wykonano żadnych innych akcji, możemy wywołać metodę `verifyNoMoreInteractions(mock)`

# Sprawdzenie parametrów (II)

```
// given
when(userDao.findUser("kowalski"))
    .thenReturn(new User("kowalski"));

// when
boolean result = userService.doesUserExist("kowalski");

// then
assertThat(result).isEqualTo(true);
verify(userDao, times(1))
    .findUser("kowalski");
verifyNoMoreInteractions(userDao);
```

# Zadanie 7

// Skopiuj test `shouldReturnTrueIfUserExists` i zmień sposób sprawdzania parametrów przekazanych do metody `userDao.findUser()`

// Wykorzystaj metodę **verify()**

# Q&A

// THANKS!