UNIWERSYTETY WSB MERITO

Programowanie aplikacji Java

Maciej Gowin

Zjazd 10 - dzień 1

Testowanie aplikacji

- **Ręczne**
  - Łatwe w przypadku małych aplikacji
  - Przy częstych zmianach kodu staje się uciążliwe
  - Bardzo wolne
  - Łatwo popełnić błąd
- **Automatyczne**
  - Wymaga stworzenia kodu testującego kod
  - Pomaga w programowaniu
  - Szybkie (w uruchomieniu / otrzymaniu wyników)
  - Rzetelne

Rodzaje testowania

- **Testowanie statyczne**
  Analiza napisanego kodu poprzez **code review** albo użycie aplikacji do statycznej analizy kodu, np. **Sonar**.
- **Testowanie dynamiczne**
  Testy uruchamiane na działającej aplikacji. Sprawdzają, czy program działa tak jak się tego spodziewamy.

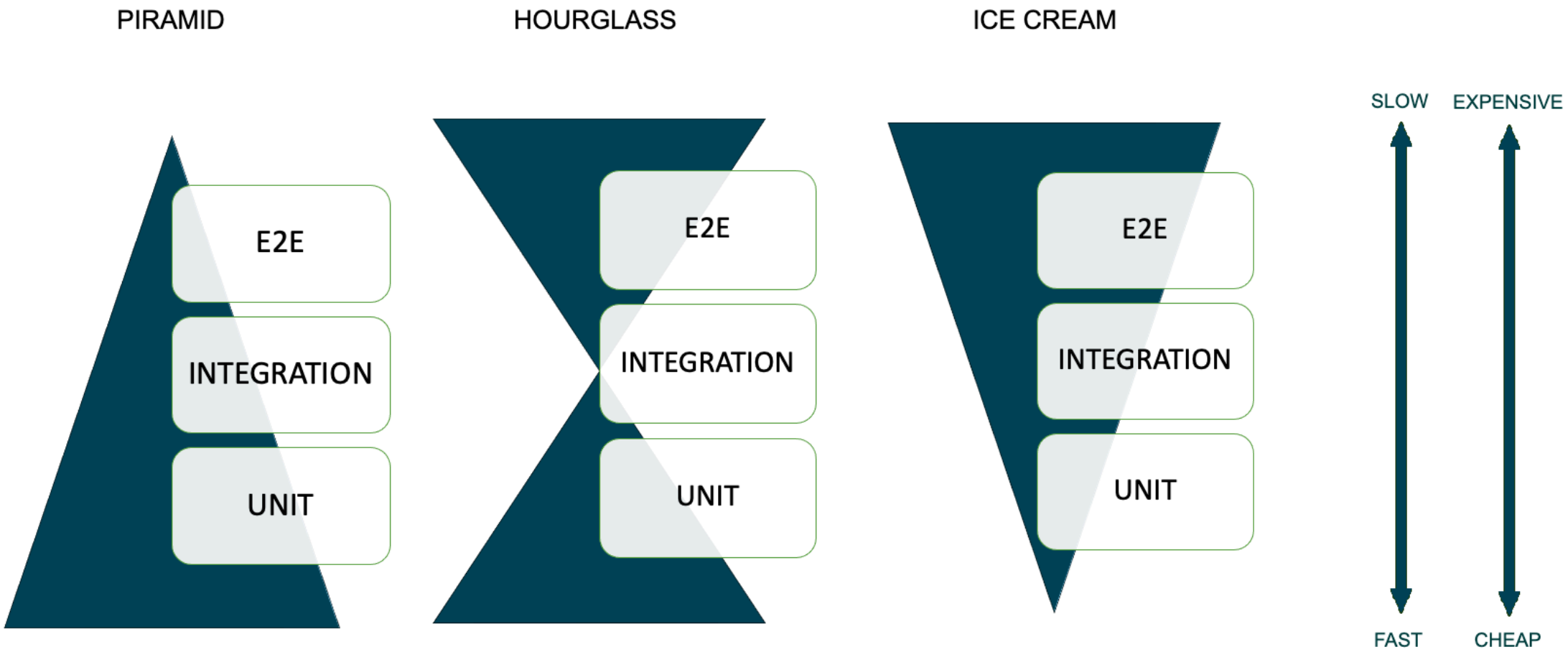
Rodzaje testowania

- **Testy funkcjonalne** (ang. black-box testing)
  Tester wie, jak program ma się zachować, nie zna szczegółów implementacji.
- **Testy strukturalne** (ang. while-box testing)
  Testy skupiające się na wewnętrznej pracy pojedynczego modułu.

Poziomy testowania

- **Testy jednostkowe**
  Jest to najniższy poziom testów. Ich zadaniem jest sprawdzenie poszczególnych funkcjonalności aplikacji. Testowane są zwykle małe fragmenty kodu, po czym wynik porównywany jest z wartością oczekiwaną. Najczęściej pisane są przez programistę w trakcie tworzenia implementacji.
- **Testy integracyjne**
  Testy sprawdzające działanie poszczególnych interface'ów aplikacji i ich wzajemne oddziaływanie.
- **Testy end-to-end**
  Testy sprawdzające działanie aplikacji jako całości od początku do końca (stąd nazwa end-to-end). Mają na celu znalezienie błędów wpływających na użytkownika.

Poziomy testowania



JUnit 5

JUnit - framework (biblioteka) służący do pisania testów w języku Java. Wspomaga nas w pisaniu oraz uruchamianiu testów jednostkowych.

Dostarcza:

- adnotacje wspomagające pisanie testów
- mechanizmy do uruchamiania testów
- mechanizmy do grupowania testów
- raportowanie

JUnit 5

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.8.1</version>
    <scope>test</scope>
</dependency>
```

> Uwaga: Będziemy korzystać z wersji 5. Dość często używana jest też wersja 4, której idea i zasada działania jest podobna.

JUnit 5: przykład testu

```
import org.junit.jupiter.api.Test;
import org.springframework.util.StringUtils;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class ExampleTest {

    @Test
    void shouldCapitalizeWord() {
        // given
        String lowercase = "wsb";

        // when
        String result = StringUtils.capitalize(lowercase);

        // then
        assertEquals("Wsb", result);
    }
}
```

JUnit 5: podstawowe zasady

1. Każda metoda testująca opatrzona jest adnotacją `@Test`.
2. Metoda testująca nic nie zwraca (void).
3. Klasa testowa powinna nazywać się tak jak klasa, dla której pisany jest test z suffixem `Test`.
4. Klasa testowa może zawierać wiele testów (dotyczących jednej bądź wielu metod dostępnych w klasie głównej).

Test jednostkowy: podział

given

Część ustawiająca wartości na potrzeby danego testu.
Inicjalizacja obiektów przekazywanych jako argumenty metody, która jest poddana testowi.

when

Wywołanie metody poddawanej testowi.
Najczęściej w tej części znajduje się jedna linijka kodu, czyli wywołanie metody.

then

Asercje, sprawdzenie, czy wartości oczekiwane odpowiadają tym otrzymanym podczas wykonywania testu.

Maven: struktura testów

```
src
  |- main
  |  |- java
  |  |  |- pl.wsb
  |  |  |  |- ExampleClass
  |  |- resources
  |- test
  |  |- java
  |  |  |- pl.wsb
  |  |  |  |- ExampleClassTest
  |  |- resources
```

- Wszystkie testy należy umieszczać w katalogu `src/test/java`
- Uruchomienie testów - `mvn test`

Maven: uruchomienie testów

Wykonanie komendy `mvn test` spowoduje uruchomienie testów programu.

Przykładowy rezultat:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running pl.maciejgowin.GeometricalUtilsTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.023 s - in pl.maciejgowin.GeometricalUtilsTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.789 s
[INFO] Finished at: 2024-06-18T22:32:31+01:00
[INFO] ------------------------------------------------------------------------
```

Maven: cykl testów

Lifecycle Reference

W lifecycle Maven (default) istnieją 2 fazy, które umożliwiają uruchomienie testów.

- test
- integration-test

Istnieje możliwość pogrupowania testów (separacji testów unitowych oraz integracyjnych), np.

- poprzez użycie innych suffixów: **Test** oraz **IntegrationTest**
- poprzez oznaczenie poszczególnych testów poprzez adnotację `@Tag`

Maven: cykl testów

Wybór grupowania testów należy skonfigurować w opisie plugina w `pom.xml` np.

`@Tag(junit5)`

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <configuration>
                <properties>
                    <includeTags>junit5</includeTags>
                </properties>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Testowanie: przyklad-junit5-basic-test

Przykładowy test JUnit5 wraz z prostą konfiguracją Maven.

Uruchom testy:

- poprzez komendę `mvn test` (pamiętaj o przejściu do poprawnego folderu)
- z poziomu Intellij (na folder **java** w **src/test**, opcja **Run 'all tests'**)

Przeanalizuj wyniki.

JUnit 5: adnotacje

Podstawowe adnotacje dostarczone przez bibliotekę:

- `@Test` - metoda testująca
- `@BeforeAll` – metoda uruchamiana przed wszystkimi metodami testującymi
- `@AfterAll` – metoda uruchamiana po wszystkich metodach testujących
- `@BeforeEach` – metoda uruchamiana przed każdym testem
- `@AfterEach` – metoda uruchamiana po każdym teście
- `@Disabled` – metoda nie zostanie wywołana

> Wyżej wymienione adnotacje różnią się nieco między poprzednią wersją biblioteki (JUnit4) a obecną (JUnit5)

Testowanie: zadanie

Na podstawie `przyklad-junit5-basic-test` przetestuj działanie przestawionych adnotacji `JUnit`.

Asercje

- Asercje są to warunki, których spełnienie jest wymagane do zaliczenia testu.
- Porównują wartość otrzymaną w wyniku wykonania kawałka kodu z wartością oczekiwaną
- Pojedynczy test zawiera **co najmniej** jedną asercję.
- Niepowodzenie którejkolwiek z asercji powoduje przerwanie testu z wynikiem negatywnym.

```
assertEquals(expected, actual);          // org.junit.jupiter.api.Assertions
assertThat(actual).isEqualTo(expected); // org.assertj.core.api.Assertions
```

Asercje: dostępne biblioteki

- wbudowane: dostępne w bibliotece JUnit

```
assertEquals(expected, actual);
assertArrayEquals(expected, actual);
assertNull(object);
```

- Hamcrest

```
assertThat(array, hasItemInArray("text"));
assertThat("text", isOneOf(array));
assertThat(5, greaterThanOrEqualTo(5));
```

Asercje: dostępne biblioteki

- Truth

```
assertThat(text).contains("wsb");
assertThat(projectsByTeam()).valuesForKey("field1").containsExactly("w","s","b");
```

- AssertJ

```
assertThat(stringVariable).isEqualTo("Frodo");
assertThat(array).hasSize(9).contains(a1, a2).doesNotContain(a3);
assertThatThrownBy(() -> { throw new Exception("boom!"); }).hasMessage("boom!");
```

AssertJ

Bogata biblioteka dostarczająca zbiór asercji.

Świetnie współpracuje z JUnit.

Umożliwia pisanie bardzo czytelnych asercji, co ułatwia pracę i przyspiesza detekcję błędów.

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

AssertJ: zależność Maven

```
<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.27.3</version>
    <scope>test</scope>
</dependency>
```

AssertJ: podstawowe asercje

Podstawowa forma:

```
assertThat(referenceOrValue).<ASERCJA>
```

W przypadku potrzeby przetestowania przypadku wystąpienia wyjątku forma ma nieco inny wygląd:

```
assertThatThrownBy(() -> {} }
```

AssertJ: assertThat()

- obiekty
  - `isEqualTo() / isNotEqualTo()` - porównuje obiekty
  - `isEqualToComparingFieldByFieldRecursively()` - porównuje pola dwóch obiektów tego samego typu
- typ boolean
  - `isTrue() / isFalse()` - sprawdza, czy wartość jest równa true albo false
- kolekcje
  - `hasSize()` - sprawdza wielkość kolekcji
  - `isEmpty() / isNotEmpty()` - sprawdza, czy kolekcja jest pusta / nie jest pusta
  - `containsAll() / containsExactly()` - sprawdza, czy kolekcja zawiera podane wartości

AssertJ: assertThat()

Powyższe przykłady to jedynie ułamek możliwości tej biblioteki.

W sytuacji, w której chcemy sprawdzić kilka warunków, możemy połączyć wywołania metod, używając tzw. **method chaining**.

```
assertThat(list)
        .isNotNull()
        .isNotEmpty()
        .hasSize(2)
        .containsExactly(1, 2);
```

Testowanie: zadanie

Na podstawie kodu z przykładu `przyklad-junit5-simple` wykonaj zadania przy użyciu JUnit5 oraz biblioteki AssertJ.

- Klasa `CustomStringUtils`
  - Napisz testy jednostkowe. Spróbuj przeanalizować wszystkie możliwe przypadki użycia danej metody i do każdego z nich napisać odpowiedni test.

Testowanie: zadanie

Na podstawie kodu z przykładu `przyklad-junit5-simple` wykonaj zadania przy użyciu JUnit5 oraz biblioteki AssertJ.

- Klasa `Calculator`
  - Zaimplementuj metodę klasy tak, aby test opisany w `CalculatorTest` zaczął działać.
  - Rozszerz klasę o kolejne funkcjonalności: odejmowanie, mnożenia i dzielenia.
  - Zaimplementuj testy jednostkowe. Pamiętaj o wszystkich możliwościach użycia metod.

Testowanie: zadanie

Na podstawie kodu z przykładu `przyklad-junit5-simple` wykonaj zadania przy użyciu JUnit5 oraz biblioteki AssertJ.

- Klasa `InternalCache`
  - Napisz testy jednostkowe dla metod klasy.
  - Sprawdzaj, czy ilość elementów w `cache` zgadza się z Twoimi założeniami. Pamiętaj o inicjalizacji i czyszczeniu cache przed i po każdym teście.

JUnit 5: testy sparametryzowane

Załóżmy przykładową logikę aplikacji.

```
public class NumberUtils {

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

JUnit 5: testy sparametryzowane

Powtarzalne testy możemy sparametryzować, przekazując prostą wartość. Pozwoli to nam na wykonanie tego samego testu dla zadanego zbioru wartości.

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

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

public class NumberUtilsParametrizedTest {

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

JUnit 5: testy sparametryzowane

Załóżmy przykładową logikę aplikacji.

```
import java.math.BigDecimal;
import java.util.Map;

public class CurrencyConversion {

    private static final Map<String, BigDecimal> CURRENCY_PLN_RATIO = Map.of(
            "USD", BigDecimal.valueOf(4.5123),
            "EUR", BigDecimal.valueOf(4.1989)
    );

    public static BigDecimal convertToPln(BigDecimal value, String currencyCode) {
        return value.multiply(CURRENCY_PLN_RATIO.get(currencyCode));
    }
}
```

JUnit 5: testy sparametryzowane

Powtarzalne testy możemy sparametryzować, przekazując złożone wartości. Pozwoli to nam na wykonanie tego samego testu dla zadanego zbioru wartości.

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

import java.math.BigDecimal;
import java.util.stream.Stream;

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

public class CurrencyConversionParametrizedTest {

    private static Stream<Arguments> shouldCalculatePriceInGivenCurrency() {
        return Stream.of(
                Arguments.of(BigDecimal.valueOf(10), "USD", BigDecimal.valueOf(45.123)),
                Arguments.of(BigDecimal.valueOf(10.34), "EUR", BigDecimal.valueOf(43.416626)));
    }

    @ParameterizedTest
    @MethodSource
    void shouldCalculatePriceInGivenCurrency(BigDecimal value, String currencyCode, BigDecimal expected) {
        BigDecimal actual = CurrencyConversion.convertToPln(value, currencyCode);
        assertThat(actual.doubleValue()).isEqualTo(expected.doubleValue());
    }
}
```

Testowanie: przyklad-junit5-parameterized

Uruchom przykładowe testy sparametryzowane. Sprawdź biblioteki wymagane do uruchomienia testów.

Testowanie: zadanie

Na podstawie kodu z przykładu `przyklad-junit5-parameterized` dokonaj następujących zmian.

1. Popraw funkcję `CurrencyConversion.convert` tak, aby zaokrąglała rezultat konwersji w górę, do 2 miejsc po przecinku (np. 1.687 -> 1.69, 2.344 -> 2.35).
2. Popraw testy oraz dodaj więcej przypadków testowych dla `CurrencyConversion.convert`. Pamiętaj o scenariuszach negatywnych takich jak: niepoprawne dane, błędy.
3. Dodaj testy funkcji `NumberUtils.isOdd` dla liczb parzystych.

Mockowanie

Uruchomienie aplikacji wiąże się z dostarczeniem wszystkich wymaganych zależności do klasy poddawanej testowi.

Zamiast używać rzeczywistych implementacji obiektów (np. repozytorium), można zastąpić je obiektami imitującymi ich działanie.

Taki sposób ułatwia pisanie testów oraz umożliwia skupienie się na testowaniu funkcjonalności danej klasy.

Mockowanie

Rodzaje mockowania:

- **Dummy** to obiekt w teście, który jest nam potrzebny jako wypełnienie. Najczęściej w formie pustej klasy.
- **Stub** to obiekt mający minimalną implementację interfejsu, bez skomplikowanej logiki.
- **Mock** to obiekt, któremu wskazujemy dokładne zachowania dla określonych metod. Najlepiej skorzystać już z dostępnych wchodzących w skład bibliotek (Mockito)

Mockito

Mockito to biblioteka dostarczająca mechanizmy służące do mockowania obiektów i definiowania ich zachowania.

Świetnie współgrająca z JUnit. Zależności są już dostarczone wraz ze Spring Boot Starters.

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.18.0</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>5.18.0</version>
    <scope>test</scope>
</dependency>
```

Mockito: przyklad użycia

```
@ExtendWith(MockitoExtension.class)
public class BookServiceTest {
    @InjectMocks
    private BookService bookService;
    @Mock
    private BookRepository bookRepository;

    @Test
    public void shouldGetTitle() {
        // given
        when(bookRepository.find(anyInt())).thenReturn(
                Book.builder().title("Dummy Title").build());

        // when
        String title = bookService.getTitle(0);

        // then
        assertThat(title).isEqualTo("Dummy Title");
    }
}
```

Mockito: zalety mockowania

Zalety użycia Mockito:

- chcemy napisać testy sprawdzający działanie pojedynczej metody, nie zależy nam na stawianiu całego kontekstu Springa
- testy wykonują się bardzo szybko
- jesteśmy w stanie sprawdzić wykonanie kodu linijka po linijce, śledząc stan obiektów na każdym kroku
- można zamockować wszystko lub jedynie pojedyncze beany (np. repozytoria, co eliminuje potrzebę stawiania bazy danych)

> Wspominamy tutaj o mock'owaniu w kontekście Spring'a, aczkolwiek biblioteka ta działa też na zwykłych obiektach i ich zależnościach.

Mockito: wady mockowania

Wady użycia Mockito:

- test będzie tak dobry, jak dobre będą mock'i (jeśli będziemy zwracać nierealne dane to taki test nie będzie pomocny)
- poprawne testy z użyciem Mockito nie gwarantują, że kontekst Springa zostanie poprawnie uruchomiony (nie sprawdzają poprawności konfiguracji)

Mockito: format mockowania

Mockowanie polega na ustaleniu odpowiedzi metody danej klasy w formie:

JEŚLI zostanie wykonana metoda **X**, **WTEDY** zwróć **Y**

```
when(object.method()).thenReturn(objectOrValue);
```

Podobna składnia jest w przypadku chęci zwrócenia wyjątku:

```
when(object.method()).thenThrow(ex);
```

Mockito: format mockowania

Mockowanie może zostać rozszerzone o `ArgumentMatchers`, które pozwalają na doprecyzowanie, dla jakich parametrów dany wynik powinien zostać zwrócony.

```
when(object.method(anyInt())).thenReturn(objectOrValue);
when(object.method(eq(5))).thenReturn(objectOrValue);
```

Mockito: weryfikacja wywołania metody

Oprócz mockowania rezultatu wywołania metody istnieje też możliwość weryfikacji czy dana metoda faktycznie została wywołana (oraz ile razy).

Aby sprawdzić, czy metoda `bookRepository.save()` została wywołana 1 raz:

```
verify(bookRepository, times(1)).save(any(Book.class));
```

Mockito: weryfikacja wywołania metody

Możemy również przechwycić dowolny obiekt przekazywany jako parametr metody oraz sprawdzić, czy jego stan jest poprawny.
Mechanizm ten pozwala na szczegółowe sprawdzenie implementacji danej sekcji.

```
    @Test
    public void shouldAddTitle() {
        // given
        when(bookRepository.save(any(Book.class))).thenReturn(5);

        // when
        int id = bookService.addTitle("Dummy Title");

        // then
        assertThat(id).isEqualTo(5);

        ArgumentCaptor<Book> argumentCaptor = ArgumentCaptor.forClass(Book.class);
        verify(bookRepository, times(1)).save(argumentCaptor.capture());
        Book book = argumentCaptor.getValue();
        assertThat(book).isNotNull();
    }
```

Testowanie: przyklad-mockito

Przeanalizuj przykład oraz sposoby mock'owania wywołań metod.

Testowanie: zadanie

Dodaj brakujące testy do metod klasy `BookService` z przykładu `przyklad-mockito`.

Testowanie: ciekawe linki

- A Guide to JUnit 5
- Intellij IDEA - uruchamianie testów
- Running a Single Test or Method With Maven
- AssertJ docs
- AssertJ - wprowadzenie