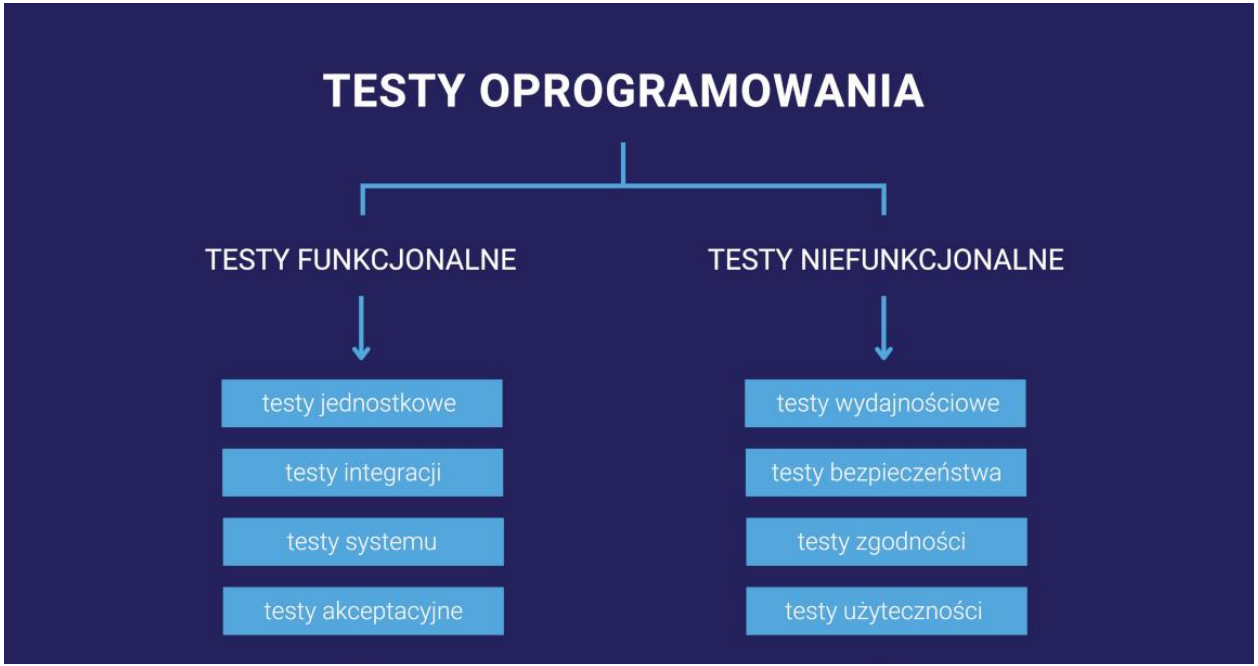
JUnit

Robert Jacak, Kacper Gałek

Proces testowania aplikacji



Testy funkcjonalne

Sprawdzają czy oprogramowanie działa zgodnie z założeniami. Można je podzielić na:

- testy jednostkowe - sprawdzają najmniejsze elementy programu (pojedyncze metody, klasy)
- testy integracji - sprawdzają jak większe części aplikacji współdziałają ze sobą
- testy systemu - ogólne sprawdzenie aplikacji pod kątem oceny specyfikacji
- testy akceptacyjne - przeprowadzane są zazwyczaj przez klienta. Sprawdzają, czy aplikacja spełnia ich oczekiwania i wpisuje się w przewidziany model biznesowy

Testy niefunkcjonalne

Sprawdzają bardziej techniczne części aplikacji. Można je podzielić m.in na:

- testy wydajności - sprawdzają jak aplikacja sprawdza się pod dużym obciążeniem
- testy bezpieczeństwa - sprawdzają czy dane są dobrze przechowywane i nie ma możliwości ich wycieku
- testy użyteczności - sprawdzają czy aplikacja jest prosta i intuicyjna do użytku
- testy zgodności - sprawdzają jak aplikacja działa pomiędzy różnymi systemami operacyjnym, przeglądarkami, urządzeniami itd.

Testy automatyczne, a ręczne

- zawsze warto mieć oba w swojej aplikacji
- ręczne pozwalają na rzeczywistą interakcje człowieka z gotowym produktem
- automatyczne pozwalają oszczędzić czas ponieważ testerzy korzystają z gotowych scenariuszy lub sami tworzą skrypty
- automatyczne można wykorzystywanie wielokrotnie i są one bardziej niezawodne

"Automatyzacja testów oprogramowania może zaoszczędzić czas i pieniądze, ale nigdy nie zastąpi cząstki ludzkiej w znajdowaniu błędów. Zwłaszcza, jeśli chodzi o testowanie użyteczności oprogramowania, które ma być przeznaczone dla prawdziwych użytkowników końcowych."

JUnit

JUnit to biblioteka Javy do pisania i uruchamiania testów jednostkowych i integracyjnych

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

Asercje w JUnit

Assercje (ang. assertions) to narzędzia w JUnit, które sprawdzają poprawność wyników testowanych metod. Pozwalają porównać oczekiwane i rzeczywiste wartości w testach jednostkowych. Najczęściej używane asercje to:

- assertEquals(expected, actual) – sprawdza, czy dwie wartości są sobie równe.
- assertTrue(condition) – sprawdza, czy podany warunek jest prawdziwy.
- assertFalse(condition) – sprawdza, czy warunek jest fałszywy
- assertNull(object)– sprawdza, czy obiekt jest null.

CalculatorServiceAddTest.java

```
1  class CalculatorServiceAddTest {
2
3      @Test
4      void shouldAddTwoIntegers() {
5          int result = calculatorService.add("10", "20");
6
7          Assertions.assertEquals(30, result);
8      }
9  }
```

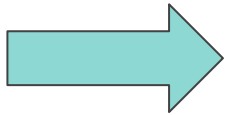
Cykl życia testów

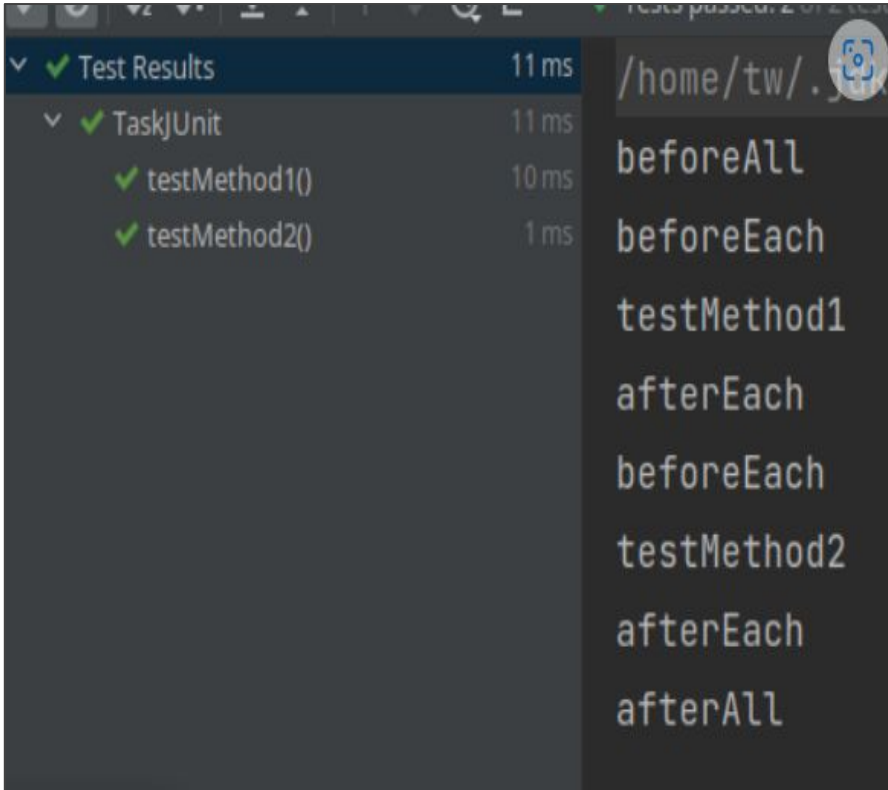
Cykl życia testów w JUnit odnosi się do sposobu, w jaki framework JUnit zarządza tworzeniem i niszczeniem instancji testowych oraz uruchamianiem metod testowych. Obejmuje kilka kroków:

- @BeforeEach – metoda oznaczona tą adnotacją jest uruchamiana przed każdym testem.
- @AfterEach – uruchamiana po każdym teście, pozwala np. na czyszczenie zasobów.
- @BeforeAll – metoda uruchamiana przed wszystkimi testami, służy do inicjalizacji zasobów współdzielonych.
- @AfterAll – uruchamiana po zakończeniu wszystkich testów, np. do zwolnienia zasobów.

Cykl życia testów

```java
@BeforeAll
static void beforeAll() {
    System.out.println("beforeAll");
}

@BeforeEach
void beforeEach() {
    System.out.println("beforeEach");
}

@Test
void testMethod1() {
    System.out.println("testMethod1");
}

@Test
void testMethod2() {
    System.out.println("testMethod2");
}

@AfterEach
void afterEach() {
    System.out.println("afterEach");
}

@AfterAll
static void afterAll() {
    System.out.println("afterAll");
}
```





Warunkowe testy i ręczne wyłączanie

JUnit pozwala na warunkowe wykonywanie testów. Używa się do tego adnotacji:

- @Disabled – pozwala na wyłączenie testu ręcznie (np. gdy jest w fazie rozwoju lub poprawy).
- @EnabledIf / @DisabledIf – umożliwiają włączenie lub wyłączenie testów w zależności od spełnienia określonych warunków (np. wersji systemu operacyjnego, zmiennych środowiskowych).

Pozwala to na dynamiczne dostosowanie testów do różnych środowisk.

```
@Test
@DisabledIf("isJavaVersionLowerThan11")
public void testDisabledForOldJavaVersions() {
    System.out.println("Ten test uruchomi się tylko na Java 11 lub wyższej.");
}
```

Powtórzenia Testów

JUnit umożliwia wielokrotne uruchamianie tego samego testu, co jest przydatne np. do testowania stabilności i zachowania kodu w różnych powtarzalnych sytuacjach. Można to zrobić przy użyciu adnotacji @RepeatedTest(n), gdzie n to liczba powtórzeń testu. Dzięki temu można łatwo sprawdzić, czy kod działa poprawnie przy wielu uruchomieniach, co jest przydatne np. przy testach wydajnościowych lub w testach wielowątkowości.

```
    @RepeatedTest(5)  // Test będzie powtórzony 5 razy
    public void repeatedAdditionTest() {
        CalculatorTest calculator = new CalculatorTest();
        int result = calculator.add(2, 3);

        // Sprawdzamy, czy wynik dodawania 2 + 3 zawsze wynosi 5
        assertEquals(5, result, "2 + 3 should always equal 5");
    }
}
```

Parameterized tests

Pozwalają na uruchamianie testów dla wielu różnych wartości które możemy podawać w wielu różnych postaciach np. @ValueSource podajemy wartości po przecinku

Do każdego testu trzeba podać źródło z którego będziemy brali argumenty i jakiś sposób żeby z nich korzystać

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

Źródła danych

Sources:

- *@ValueSource - podajemy kilka wartości po przecinku*
- *@EnumSource - podajemy jakaś zmienną typu enum*
- *@CsvSource - podajemy wartości w formacie csv (możemy mieć kilka argumentów*
- *@CsvFileSource - podajemy nazwę pliku w formacie csv*
- *@MethodSource - możemy podać metodę*
- *@FieldSource - możemy podać jakieś utworzone wcześniej pole klasy (nazwę tego pola)*
- *@ArgumentsSource - interfejs do tworzenia własnych źródeł danych*

```
@EnumSource(Month.class) // passing all 12 months
```

```
@CsvSource({"test,TEST", "tEst,TEST", "Java,JAVA"})
void toUpperCase_ShouldGenerateTheExpectedUppercaseValue(String input, String expected) {
```

```
@CsvFileSource(resources = "/data.csv", numLinesToSkip = 1)
```

Dodatkowe funkcje JUnita

```
@Timeout(5)
void setUp() {
    // fails if execution time exceeds 5 seconds
}
```

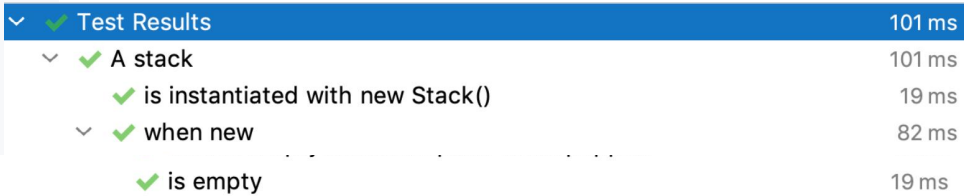
Pozwala na tworzenie testów z ograniczonym czasem wykonania. Po przekroczeniu określonej liczby sekund test zostanie porzucony i otrzymamy negatywny wynik testu

```
@Tag("fast")
@Tag("model")
class TaggingDemo {

    @Test
    @Tag("taxes")
    void testingTaxCalculation() {
    }

}
```

Pozwala na tagowanie testów w celu późniejszego uruchamiania tylko testów z określonym tagiem

Nested tests

```java
@DisplayName("A stack")
class TestingAStackDemo {

    Stack<Object> stack;

    @Test
    @DisplayName("is instantiated with new Stack()")
    void isInstantiatedWithNew() {
        new Stack<>();
    }

    @Nested
    @DisplayName("when new")
    class WhenNew {

        @BeforeEach
        void createNewStack() {
            stack = new Stack<>();
        }

        @Test
        @DisplayName("is empty")
        void isEmpty() {
            assertTrue(stack.isEmpty());
        }
```



Pozwala na grupowanie testów. Należy pamiętać, że w nested tests nie działa @BeforeAll i @AfterAll

Dynamic tests

Pozwalają na tworzenie testów, które są generowane podczas wykonywania programu

JUnit z IntelliJ/Maven

JUnit w IntelliJ IDEA: IntelliJ IDEA jest zintegrowanym środowiskiem programistycznym (IDE), które doskonale wspiera tworzenie testów z wykorzystaniem JUnit.

JUnit z Mavenem: Maven to popularne narzędzie do zarządzania zależnościami i budowania projektów w Javie. W projektach korzystających z Mavena, JUnit można zintegrować, dodając zależność do pliku pom.xml

Praca z Maven w IntelliJ IDEA jest bardzo intuicyjna, ponieważ to środowisko programistyczne (IDE) zapewnia pełne wsparcie dla Mavena.

Korzyści z użycia JUnit z Maven i IntelliJ:

- **Automatyczne zarządzanie zależnościami** (Maven)
- **Intuicyjna nawigacja i tworzenie testów** (IntelliJ)

Dobre praktyki w pisaniu testów jednostkowych

1. **Pisz testy niezależne** - Każdy test powinien być niezależny od innych testów i nie powinien polegać na stanie globalnym
2. **Zasada “AAA” (Arrange, Act, Assert)** - **Arrange:** Przygotuj dane wejściowe i środowisko do testu.**Act:** Wykonaj operację, którą testujesz. **Assert:** Sprawdź, czy wyniki są zgodne z oczekiwaniami za pomocą asercji
3. **Testuj tylko jedną rzecz na raz** - Każdy test powinien sprawdzać jedną funkcjonalność lub jeden scenariusz.
4. **Testuj scenariusze negatywne i pozytywne** - Oprócz testowania poprawnych przypadków użycia, warto sprawdzić, jak metoda reaguje na niepoprawne dane, błędy i nieoczekiwane sytuacje
5. **nazewnictwo testów** - Nazwy metod testowych powinny być opisowe
6. **testuj krawędziowe przypadki** - Upewnij się, że testy obejmują wartości graniczne i skrajne przypadki
7. **Testuj na bieżąco**: Twórz testy jednostkowe w momencie pisania kodu produkcyjnego.

Dziękujemy za uwagę

źródła:

https://stormit.pl/testy-jednostkowe-junit/

https://udigroup.pl/blog/testowanie-oprogramowania-typy-rodzaje-poziomy/

https://www.baeldung.com/parameterized-tests-junit-5

https://www.jetbrains.com/help/idea/junit.html

https://www.samouczekprogramisty.pl/testy-jednostkowe-z-junit5/

https://www.samouczekprogramisty.pl/testy-jednostkowe-z-junit/

https://junit.org/junit5/docs/current/user-guide/#overview