AKADEMIA TARNOWSKA
Wydział Politechniczny

Kierunek: ***Informatyka***

Specjalność: Informatyka Stosowana

Specjalizacja: Inżynieria Oprogramowania

2023/2024

Katarzyna Bączek

PRACA INŻYNIERSKA

Testowanie kodu z wykorzystaniem frameworka JUnit

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, 2024

Spis treści

1. Wstęp

Testowanie oprogramowania stanowi ważny element procedury wytwarzania każdej aplikacji, pozwalając nie tylko potwierdzić poprawność działania kodu, ale także jego trwałość i jakość. W końcu nieważne czy tworzony jest najprostszy kalkulator czy skomplikowany program do zarządzania magazynem sklepowym - zarówno dla jednego jak i drugiego niezbędne jest upewnienie się, że wszystko jest w pełni funkcjonalne, działa zgodnie z wymaganiami i nie ulegnie awarii, nawet jeśli użytkownik będzie korzystać z niego w sposób sprzeczny z pierwotnym przeznaczeniem.

W celu zoptymalizowania tego procesu, tworzone są coraz to nowsze narzędzia umożliwiające łatwiejsze zarządzanie testami, dostosowane do konkretnych potrzeb. Nie ma jednego uniwersalnego, który byłby w stanie obsłużyć wszystkie możliwe przypadki - każdy jest nastawiony na określony zakres czy sposób działania. Dzięki ich różnorodności programista może odpowiednio skonfigurować strategię testową do charakterystyki danego projektu.

1.1. Motywacja

Duży wybór narzędzi do testowania oferuje szeroką gamę możliwości, dlatego warto dokładnie przemyśleć, które z nich okaże się najlepszym wyborem, aby skorzystać z pełnego potencjału dostępnych funkcji oraz dostosować je do konkretnych potrzeb.

1.2. Cel

Jednym z wyraźnych przykładów takich innowacyjnych rozwiązań jest framework JUnit, który od lat zyskuje aprobatę wśród programistów jako nieodłączne narzędzie do tworzenia, zarządzania i monitorowania testów jednostkowych w języku Java.

Głównym celem niniejszej pracy było przeprowadzenie jego szczegółowej analizy, obejmującej zarówno aspekt teoretyczny, jak i praktyczny.

1.3. Zakres pracy

W celu łatwiejszego zrozumienia działania JUnit najpierw została przedstawiona ogólna definicja testowania oprogramowania, dlaczego i jak powinno się testować, a także prezentacja jednej z technik która opiera swój proces na pisaniu testów. Kolejne rozdziały

obejmują zagadnienia związane z frameworkiem oraz jego praktyczne zastosowanie w kontekście przykładowych projektów.

2. Testowanie oprogramowania

"Testować" to inaczej "ewaluować" - wynika z tego, że testowanie oprogramowania polega przede wszystkim na weryfikowaniu i wycenianiu [1]. Pisanie scenariuszy testowych musi być przemyślane i choć czasem jest czasochłonne, długoterminowo z pewnością się opłaca [3]. W kodzie powinno być testowane wszystko, co może się uszkodzić - zarówno "happy path" (najbardziej podstawowe i oczekiwane przypadki) jak i skrajne sytuacje, które mogą wystąpić np. po udostępnieniu aplikacji większemu gronu odbiorców [18] . Wymóg ten sprawia, że oprogramowanie należy pisać w sposób umożliwiający jego weryfikację - przypadek, w którym nie da się zapewnić o odpowiednim wykonywaniu się operacji w aplikacji znacząco wpływa na jej trwałość oraz obniża jej jakość.

Dobrze napisane testy mogą również stanowić przydatną dokumentację danych funkcji (w końcu zawierają przykłady wykonywania się metod, co pozwala na łatwiejsze wyobrażenie sobie ich działania) oraz, jeśli napisało się je na początku, pomoc w implementacji trudniejszych funkcjonalności. Prościej trzymać się wytycznych, które są jasno określone i zapisane. Oprócz tego ich największa korzyścią jest ta najoczywistsza - jeśli wszystkie testy dla danego systemu przechodzą, oznacza to że działa on poprawnie oraz spełnia postawione wymagania [3].

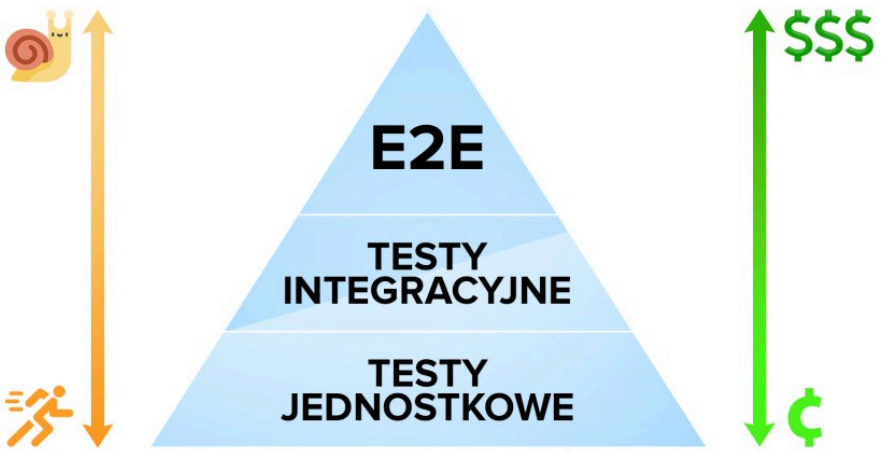
2.1. Rodzaje testów

Najogólniej można wyróżnić dwa rodzaje testowania oprogramowania podzielone ze względu na sposób ich wykonywania.

Pierwszym jest testowanie manualne polegające na ręcznym przechodzeniu przez konkretne scenariusze w celu sprawdzenia poprawnego działania aplikacji. Zaletą tego rozwiązania jest brak wymogu znajomości języka programowania przez testera, dzięki czemu może on patrzeć na funkcjonalności bardziej z perspektywy potencjalnego użytkownika niż programisty. Testy przeprowadzane ręcznie mają jednak więcej minusów niż plusów - wykonuje je człowiek, który może się pomylić, a sama procedura zajmuje bardzo dużo czasu.

Drugim typem testów są te automatyczne, realizowane maszynowo za pomocą dostosowanych do tego narzędzi. Testowanie tym sposobem jest znacznie szybsze, sprawniejsze i daje konkretne rezultaty, a przez fakt że są one generowane przez komputer - przeprowadzenie ich nie generuje ryzyka pomyłki w procesie “przeklikiwania” się przez aplikację. Siłą rzeczy na ich rzeczywistą użyteczność i skuteczność wpływa przede wszystkim jakość napisanych skryptów.

Wyróżnia się trzy główne poziomy testowania aplikacji w sposób automatyczny. Przedstawiane są one w formie piramidy (rysunek 2.1.), w której im wyżej dany typ się znajduje, tym więcej potrzebuje czasu i zasobów a także jest bardziej zintegrowany od poprzedniego [10, 21].



Rys. 2.1. Piramida testów [9]

- testy jednostkowe - ich zadaniem jest sprawdzenie działania pojedynczych modułów lub małych funkcjonalności, wykonują się szybko i z reguły jest ich najwięcej
- testy integracyjne - weryfikują poprawne funkcjonowanie i połączenie między komponentami, przez korzystanie z większej ilości klas mogą trwać dłużej

- testy E2E (End to End) - sprawdzają działanie aplikacji “od końca do końca”, tj. od samego wejścia danych, poprzez logikę biznesową, aż do zakończenia programu, skupiają się na całości systemu zamiast na fragmentach [10, 21]

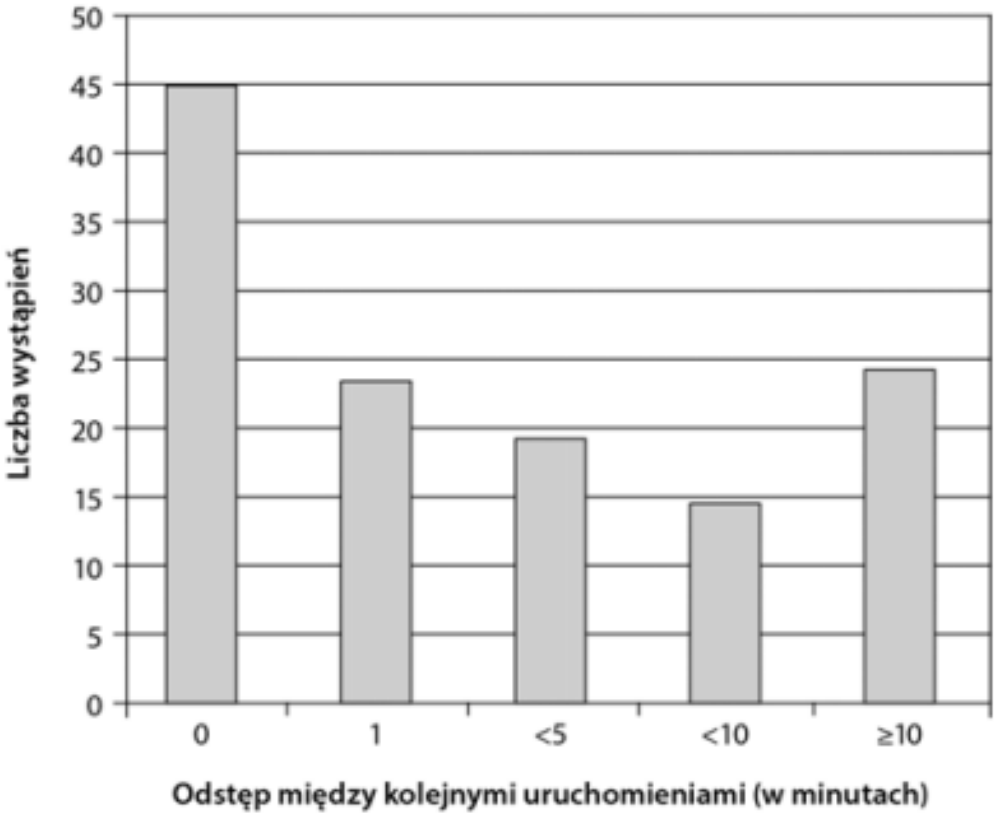
2.2. Zasada F.I.R.S.T.

Funkcjonują ogólnie przyjęte zasady, które stanowią jasne wskazówki na co należy zwrócić uwagę podczas pisania testów, aby były one skuteczne i użyteczne w projekcie. Zyskały one akronim F.I.R.S.T., z czego każda litera oznacza inną regułę - przestrzeganie ich sprawi, że testowanie programu stanie się rzeczywistą pomocą i ułatwieniem dla programisty, zamiast tylko uciążliwym obowiązkiem.

(F) jak **Fast** *(szybki)*

Przede wszystkim testy nie powinny zajmować dużo czasu. Jeśli na ich uruchomienie trzeba poświęcić więcej niż kilka sekund, to jasny znak, że należy je poprawić - w końcu każdy moment oczekiwania może przyczynić się do rozkojarzenia, skupienia uwagi programisty na czymś innym niż kod. A skoro wystartowanie wolnych testów wiąże się z takim ryzykiem, nierozsądnym byłoby się na nie narażać [17].

Na rysunku 2.2. został przedstawiony histogram z odstępami czasu pomiędzy sesjami testowania przykładowej aplikacji finansowej. Jasno wynika z niego, że programista sprawdzał swój kod bardzo często - w ponad połowie przypadków przerwa nie była dłuższa niż minuta. Nie mógłby sobie na to pozwolić, gdyby uruchamianie testów zabierało zbyt dużo czasu [1].



Rys. 2.2. Przykładowa statystyka przerw między kolejnymi sesjami testowania [1]

Nie zawsze jednak uda się zoptymalizować testy tak, by po wystartowaniu otrzymać ich wyniki niemal od razu. Zależy to przede wszystkim od złożoności i ilości testowanych komponentów - oczywiste jest, że małe funkcjonalności można zbadać szybciej niż działanie całych modułów. Aby przyspieszyć proces, należy skupić się na testowaniu konkretnie tego, co jest wymagane w danym scenariuszu. W miejscach, gdzie jest to wykonalne, można użyć atrap lub zaślepić interakcje, co znacząco wpłynie na czas budowania obiektów potrzebnych do testu.

(I) jak **Isolated** *(izolowany)*

Wskazane jest, aby testy były od siebie niezależne. Nieważne, czy są uruchamiane osobno czy w pakiecie - przy takiej samej implementacji powinny dać identyczne wyniki dla każdego ich wykonania. Do niepożądanej sytuacji może dojść przede wszystkim w wypadku, gdy jeden obiekt jest współdzielony, np. baza przechowująca dane tworzona poza testem. Aby zachować izolację i integralność, konieczne jest przywrócenie takich

udostępnionych zasobów do ich pierwotnego stanu przed lub po zakończeniu każdego przypadku testowego.

(R) jak **Repeatable** *(powtarzalny)*

Rezultat testu nie może być też uzależniony od czegoś zewnętrznego (np. dostępności do baz danych, sieci, plików) lub czynników losowych. Jeśli testowanie danego przypadku daje różne efekty bez żadnych zmian w kodzie pomiędzy uruchomieniami, jest to tzw. kruchy test *(flaky test)* - łatwo go zepsuć, a przez brak pewności co do poprawności wyniku (pozytywne przejście może być skutkiem przypadku, niekoniecznie dobrze działającej funkcjonalności) nie ma on żadnej wartości dla programisty.

(S) jak **Self-validating** *(samosprawdzający się)*

Test powinien dawać jasny, zrozumiały wynik:

- POZYTYWNY - dany scenariusz testowy wykonuje się w sposób poprawny dając oczekiwany, określony wcześniej wynik
- NEGATYWNY - określony przypadek nie działa tak jak powinien i implementacja wymaga zmiany

Ważne jest odpowiednie i jednoznaczne nazewnictwo testów oraz ich asercji, tak aby w wypadku ich niepowodzenia łatwo można było stwierdzić, co doprowadziło do błędu i w jaki sposób jest możliwe jego naprawienie. Adekwatne opisy pozwolą szybko odtworzyć przebieg funkcji i namierzyć usterkę bez potrzeby prześledzenia każdej linijki kodu.

(T) jak **Timely** *(aktualny)*

Ostatnia zasada tyczy się nie bezpośrednio treści testów, a tego, kiedy powinny być one pisane - razem z kodem. Nie może dojść do sytuacji, kiedy funkcjonalność jest wdrażana bez jej sprawdzenia, a brakujące testy dopisywane później. Taki przypadek nie tylko obniża jakość oprogramowania, a i również naraża system na usterki których wykrycie mogło być możliwe wcześniej.

Nie ma nic złego w pisaniu testów po implementacji kodu, o ile są one wraz z tym kodem traktowane jako “zestaw” i nie wypuszczane oddzielnie po jakimś czasie. Istnieją

koncepcje wspomagające przestrzeganie tej reguły, np. technika TDD opisana w kolejnym podrozdziale [17, 24].

2.3. Technika TDD

Metoda, która przykłada dużą wagę do testowania to technika TDD. Test-Driven Development (“rozwijanie oprogramowania sterowane testami”) zakłada powtarzanie w kółko trzech prostych kroków przedstawionych na rysunku 2.3 [2].

I. **RED** - napisanie testu z wynikiem negatywnym

Przed przystąpieniem do implementacji kodu konieczne jest napisanie testów, które pokryją tworzone funkcjonalności. Sprawdzane funkcje nie są jeszcze napisane, więc testy nie przechodzą.

II. **GREEN** - przejście testu

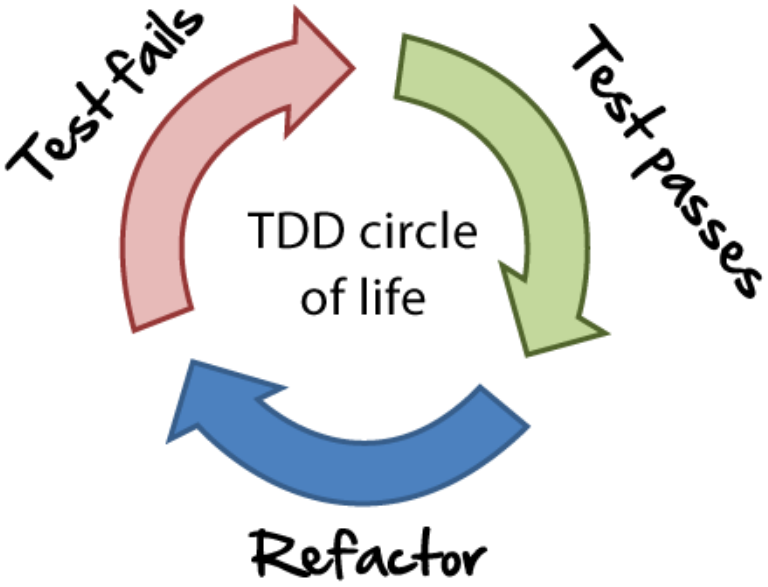
Następnym krokiem w cyklu TDD jest doprowadzenie do zaliczenia wszystkich testów - kod nie musi być “ładny”, liczy się tylko pozytywny rezultat na końcu.

III. **REFACTOR** - poprawienie kodu

Finałowym etapem cyklu jest refaktoryzacja - należy po sobie “posprzątać” i zwiększyć czytelność implementacji. Można osiągnąć to przez np.

- użycie wzorców projektowych,
- wyodrębnienie podobnych/takich samych fragmentów do osobnych funkcji,
- usunięcie niepotrzebnych importów i zmiennych,
- zmianę nazewnictwa [13].

Napisane wcześniej testy w każdej chwili można uruchomić, co daje gwarancję na to, że w trakcie modyfikacji kodu nie została uszkodzona żadna funkcjonalność - powstała zarówno w tym konkretnym cyklu, jak i wcześniejszych.



Rys 2.3. Technika TDD [25]

Stosowanie techniki TDD nosi ze sobą wiele korzyści, z czego największymi są ciągła kontrola nad działaniem programu i jasne granice, jakie funkcje są wymagane na danym etapie tworzenia - wszystkie powinny być pokryte testami, które dostępne są w jednym miejscu i odpowiednio opisane. Poprzez ciągłe testowanie można mieć pewność, że implementując nową funkcjonalność nie uszkodzi się innej [1].

3. Tworzenie testów przy użyciu JUnit

Początki JUnit sięgają 1997 roku, kiedy to miejsce miało jego pierwsze wydanie. Wcześniej testy opierały się bardziej na interfejsie graficznym i polegały na próbie odtworzenia konkretnego stanu systemu, co było przede wszystkim czasochłonne. JUnit pozwolił zaś na testowanie kodu za pomocą skryptów, przez co błyskawicznie zyskał uznanie wśród programistów - sposób ten był szybszy, bardziej uniwersalny i dawał możliwość weryfikacji działania małych funkcjonalności zamiast komponentów GUI [4].

Powstał na podstawie SUnit - frameworka przeznaczonego do testowania języka Smalltalk. Używając tego samego modelu opracowano również narzędzia do testów w innych językach (np. Python, PHP, Ruby, C), które funkcjonują pod zbiorczą nazwą *xUnit* (odpowiednio np. PyUnit, PHPUnit, RUnit, CUnit) [6].

Najnowsza wersja, JUnit 5, to framework składający się z 3 niezależnych modułów:

- **JUnit Platform** - dostarcza fundament do uruchamiania testów na różnych środowiskach, w tym w różnych środowiskach programistycznych (np. IntelliJ IDEA lub Eclipse), narzędziach do budowy projektów oraz systemach ciągłej integracji [15]
- **JUnit Jupiter** - API za pomocą którego można pisać testy z użyciem przyjętego modelu oraz asercji, wprowadza również nowe (w porównaniu do JUnit 4) adnotacje, takie jak np. @Tag lub @DisplayName
- **JUnit Vintage** - służy do uruchamiania testów z wcześniejszych wersji JUnit, zapewniając zgodność wsteczną [7]

3.1. Struktura testu

Do napisania najprostszego testu nie potrzeba wiele - wystarczy odpowiednia adnotacja przed metodą testującą, a w samej metodzie - jakakolwiek asercja. Wszystko to powinno się znajdować w osobnej klasie wraz z potrzebnymi importami [15].

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import example.util.Calculator;
import org.junit.jupiter.api.Test;

class ExampleTest {

    private final Calculator calculator = new Calculator();

    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }

}
```

Listing 3.1. Przykładowy test w JUnit [15]

Aby zwiększyć czytelność testów, przy ich pisaniu powinno stosować się konwencję GWT (Given-When-Then) lub działającą na tej samej zasadzie AAA (Arrange-Act-Assert). Pozwalają one łatwo na podstawie testu odtworzyć konkretny scenariusz, który jest poddawany weryfikacji. Kolejno w każdym "bloku" powinno znajdować się:

- **Given / Arrange** - założenia początkowe potrzebne do testu (co na pewno jest prawdziwe? co musi być spełnione, aby funkcjonalność zadziałała?)
- **When / Act** - wywołanie sprawdzanej funkcjonalności (co konkretnie jest poddawane testowi?)
- **Then / Assert** - wywołanie asercji, weryfikacja wyniku (co powinno się stać? jaki powinien być rezultat?) [4]

```
 @Test
  void checkBorrowAvailableBook() {

    //given:
    Library library = new Library();
    Book newBook = new Book("King", "Stephen", "The
Shining");
    library.addBook(newBook);

    //when:
    result = library.isBookAvailable(newBook.getId());

    //then:
    assertTrue(result);
  }
```

Listing 3.2. Przykładowy test w JUnit z użyciem GWT [opracowanie własne]

3.2. Adnotacje

Adnotacja w Javie to szczególny rodzaj interfejsu, pozwalający na oznaczenie danego elementu kodu (np. klasy, parametru, zmiennej) i przekazanie na jego temat dodatkowych informacji dla kompilatora. Umieszcza się ją bezpośrednio nad fragmentem, który chcemy oznaczyć. W celu rozróżnienia od zwykłych interfejsów, jej nazwę zaczyna się znakiem "@" [8, 20].

W JUnit adnotacje są używane do oznaczania metod.

- **@Test** - podstawowa adnotacja w JUnit, po oznaczeniu metody kompilator traktuje ją jako test:

```
@Test
void exampleTest() {
    assertTrue(2 + 2 == 4);
}
```

Listing 3.3. Przykładowe użycie adnotacji @Test [opracowanie własne]

- **@ParametrizedTest** - służy do oznaczania testów sparametryzowanych, tj. uruchamianych z różnymi wartościami, które podane są za pomocą kolejnej adnotacji, np. **@ValueSource**:

```
@ParametrizedTest
@ValueSource(strings = {"kajak", "radar"})
void palindrome(String word) {
    assertTrue(isPalindrome(word));
}
```

Listing 3.4. Przykładowe użycie adnotacji @ParametrizedTest [15]

Więcej informacji na temat sparametryzowanych testów znajduje się w rozdziale 3.4.

- **@RepeatedTest** - służy do oznaczenia testu, który ma zostać powtórzony daną ilość razy (wartość w parametrze *value*), z czego każdy jest traktowany osobno, jeśli więc w kodzie znajdują się metody uruchamiające się po lub przed każdym testem, będą one brane pod uwagę i wykonywane za każdym ponowieniem;

```
    @RepeatedTest(value = 3)
    void exampleTest() {
        assertTrue(isOdd(5));
    }
```

Listing 3.5. Przykładowe użycie adnotacji @RepeatedTest [opracowanie własne]

W metodzie jest możliwe użycie informacji o obecnym teście - numeru aktualnej iteracji *(currentRepetition)* oraz ich ogólnej liczby *(totalRepetitions)*. Dostępne są poprzez przekazanie do testu parametru typu *RepetitionInfo*.

```
    @RepeatedTest(value = 5, name = "Test no.
{currentRepetition} out of {totalRepetitions}")
    void exampleRepeatedTest(RepetitionInfo repetitionInfo)
{
        assertTrue(repetitionInfo.getCurrentRepetition() <=
repetitionInfo.getTotalRepetitions());
    }
```

Listing 3.6. Przykładowe użycie adnotacji @RepeatedTest z zastosowaniem RepetitionInfo [7]

- **@DisplayName** - pozwala nadać klasie lub metodzie testowej spersonalizowaną nazwę, która będzie wyświetlana w raportach; można użyć zarówno liter czy cyfr, jak i również znaków specjalnych, a nawet emotikonów:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("A special test case 🤔")
class DisplayNameDemo {

    @Test
    @DisplayName("╯°□°）╯")
    void testWithDisplayNameContainingSpecialCharacters() {
    }

}
```

Listing 3.7. Przykładowe użycie adnotacji @DisplayName w klasie z metodą testową [15]

- **@DisplayNameGeneration** - podobnie jak adnotacja *@DisplayName* daje możliwość manipulacji nad nazwami testów w raportach, jednak skupia się na przekształcaniu nazw metod w oparciu o wybrany typ generowania, np. **ReplaceUnderscores** zastępujący znak "_" spacją:

```
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscor
es.class)
class DisplayNameGenerationDemo {

    @Test
    void test_for_display_name() {
    }

}
```

Listing 3.8. Przykładowe użycie adnotacji @DisplayNameGeneration [15]

Test przedstawiony w listingu 3.8. będzie więc raportowany jako “test for display name”.

Zamiast używać adnotacji w każdej klasie, można również ustalić domyślny generator nazw testów w pliku *junit-platform.properties* [15]. Możliwa jest również implementacja własnych generatorów i używanie ich w ten sam sposób [7].

- **@Tag** - dzięki niej można nadać metodzie testowej jedną lub więcej etykiet, które potem mogą zostać użyte do ich filtrowania; może to być dowolny ciąg znaków, jednak przyjęte jest używanie krótkich wyrażeń, np. “integration” czy “security”:

```
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("calculator")
@Tag("basic")
class TaggingDemo {

    @Test
    @Tag("add")
    void testingAddingNumbers() {
    }

}
```

Listing 3.9. Przykładowe użycie adnotacji @Tag [15]

- **@BeforeAll i @BeforeEach** - pierwsza adnotacja sprawia, że metoda nią oznaczona wykonuje się przed wszystkimi testami w danej klasie, zaś druga - przed każdym testem z osobna; są one przydatne przede wszystkim w przypadku współdzielenia obiektu w różnych scenariuszach:

```
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

class CountingTests {

    private Counter counter;

    @BeforeAll
    void startup() {
     LOG.info("Counting tests starting…");
    }

    @BeforeEach
    void counterRestart() {
     counter.restart();
    }

}
```

Listing 3.10. Przykładowe użycie adnotacji @BeforeAll i @BeforeEach [7]

- **@AfterAll i @AfterEach** - działają na tej samej zasadzie, co *@BeforeAll* i *@BeforeEach*, z tą różnicą, że wykonują się po testach/teście:

```
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

class ReservationTest{

    private Hotel hotel;

    @BeforeAll
```

```
    void cleanup() {
     hotel.addUser(1, "John", "Doe");
    }

    @AfterEach
    void cleanup() {
     hotel.clearReservations();
    }

    @AfterAll
    void cleanAll() {
     hotel.removeUser(1, "John", "Doe");
    }

}
```

Listing 3.11. Przykładowe użycie adnotacji @AfterAll i @AfterEach [opracowanie własne]

3.3. Asercje

JUnit oferuje wiele metod sprawdzających poprawność kodu - asercje. Zapewniają one, że dany warunek jest spełniony, a w przeciwnym wypadku zatrzymują test zwracając negatywny wynik [22].

Wszystkie mają podobną konstrukcję - nazwa sugerująca, co jest w niej weryfikowane oraz od jednego do kilku argumentów, które będę tej weryfikacji poddane. Do każdej z wymienionych w tym podrozdziale funkcji można również dodać dodatkowy parametr *message*, czyli ciąg tekstowy wyświetlany w wypadku niepowodzenia asercji [15].

Przykładowe użycia podanych asercji mają charakter poglądowy, mające na celu przedstawić ich działanie. W prawdziwym programie takie testy nie miałyby wartości i praktycznego zastosowania.

- **assertTrue** - zakłada, że dostarczony warunek (boolean) to prawda:

```
@Test
 void checkEvenForNegativeNumber() {
   assertTrue(isEven(-32));
 }
```

Listing 3.12. Przykładowe użycie asercji assertTrue [opracowanie własne]

- **assertFalse** - zakłada, że dostarczony warunek (boolean) to fałsz:

```
@Test
 void checkNotEvenForPositiveNumber() {
   assertFalse(isEven(15));
 }
```

Listing 3.13. Przykładowe użycie asercji assertFalse [opracowanie własne]

- **assertEquals** - zakłada, że dostarczone obiekty tego samego typu (Object, byte, char, double, float, int, long, short) mają taką samą wartość; przy porównywaniu liczb zmiennoprzecinkowych możliwe jest dodanie kolejnego parametru tego samego typu *delta*, czyli margines błędu:

```
@Test
 void checkMultiplyNumbers() {
   int expected = 6;
   int actual = multiplyNumbers(2,3);

   assertEquals(expected, actual);
 }
```

```
@Test
 void checkDivideNumbers() {
   double expected = 3.3;
   double actual = divideNumbers(10.0, 3.0);
   double delta = 0.01;

   assertEquals(expected, actual, delta);
 }
```

Listing 3.14. Przykładowe użycie asercji assertEquals bez i z parametrem delty [opracowanie własne]

- **assertNotEquals** - działa na takich samych zasadach co asercja assertEquals, ale gwarantuje że obiekty *nie* mają tej samej wartości:

```
@Test
 void checkHash() {
   String password = "password";
   String hashedPassword = hash(password);

   assertNotEquals(password, hashedPassword);
 }
```

Listing 3.15. Przykładowe użycie asercji assertNotEquals [opracowanie własne]

- **assertSame** - zakłada, że obydwa dostarczone argumenty tego samego typu (Object, byte, char, double, float, int, long, short) to ten sam obiekt; w przeciwieństwie do assertEquals ta sama wartość nie wystarczy, by asercja się powiodła:

```
@Test
```

```
    void testEqualsVsSame() {
        String example1 = new String("abc");
        String example2 = example1;

        //asercja się powiedzie - ta sama wartość
        assertEquals(example1, new String("abc"));

        //asercja się nie powiedzie - inne obiekty
        assertSame(example1, new String("abc"));

        //asercje się powiodą - ten sam obiekt z taką samą
wartością
        assertEquals(example1, example2);
        assertSame(example1, example2);
    }
```

Listing 3.16. Porównanie assertSame i assertEquals [opracowanie własne]

- **assertNotSame** - działa na takich samych zasadach co asercja assertSame, ale gwarantuje że argumenty to *nie* ten sam obiekt:

```
@Test
 void checkNotSame() {
   assertNotSame(new String("abc"), new String("abc"));
 }
```

Listing 3.17. Przykładowe użycie asercji assertSame [opracowanie własne]

- **assertArrayEquals** - zakłada, że dostarczone tablice tego samego typu (Object, byte, char, double, float, int, long, short) mają taką samą zawartość - zarówno długość, jak i kolejność:

```
@Test
 void checkSortAsc() {
   int[] arrayToSort = new int[]{4, 20, -3, 100};
   int[] actual = sortAsc(arrayToSort);
   int[] expected = new int[]{-3, 4, 20, 100};

   assertArrayEquals(actual, expected, "The array is not
sorted ascending");
 }
```

Listing 3.18. Przykładowe użycie asercji assertArrayEquals z opcjonalną wiadomością przy niepowodzeniu asercji [opracowanie własne]

- **assertArrayNotEquals** - działa na takich samych zasadach, co asercja assertArrayEquals, ale gwarantuje, że tablice się różnią - długością czy też kolejnością lub wartością elementów:

```
@Test
 void checkArrays() {
   int[] firstArray = new int[]{1, 2, 3};
   int[] secondArray = new int[]{1, 3, 2};

   assertArrayNotEquals(firstArray, secondArray);
 }
```

Listing 3.19. Przykładowe użycie asercji assertArrayNotEquals [opracowanie własne]

- **assertThrows** - zakłada, że podany kod (przekazany za pomocą interfejsu funkcyjnego *Executable* czyli np. jako wyrażenie lambda) zwraca instancję określonej klasy wyjątku:

```
@Test
 void checkDividingByZero() {

    assertThrows(WrongCalculatorArgumentsException.class, ()
-> {Calculator.divide(15,0);});

 }
```

Listing 3.20. Przykładowe użycie asercji assertThrows [11]

3.4. Parametryzowanie

W przypadku, gdy scenariusz testowy może zawierać różne zestawy danych, wygodne jest użycie parametryzacji. Ułatwia to zarządzanie - dzięki temu klasy nie są zaśmiecane wieloma testami, które mają weryfikować to samo zachowanie i posiadają wspólną logikę.

Aby napisać test parametryzowany, należy oznaczyć metodę adnotacją **@ParameterizedTest** i dodać do niej parametr, który potem będzie można wykorzystać. Do poprawnego działania konieczne jest również przekazanie danych, które będą pod niego "podstawiane". Do tego celu służą specjalne adnotacje:

- **@NullSource** - dostarcza pojedynczy argument *null.*
- **@EmptySource -** dostarcza pojedynczy pusty argument typów takich jak String, Collection, List, Set i Map, jak i również puste tablice [19].

```
@ParameterizedTest
@NullSource
@EmptySource
 void checkWordValidator(String text) {
    assertFalse(validateWord(text));
}
```

Listing 3.21. Przykładowe użycie adnotacji @NullSource i @Empty Source [opracowanie własne]

Zamiast użycia adnotacji *@NullSource* i *@EmptySource* do tej samej metody, można użyć **@NullAndEmptySource** która jest ich połączeniem.

- **@ValueSource -** umożliwia przekazanie dowolnej ilości argumentów typów short, byte, int, long, float, double, char, boolean, String lub Class:

```
@ParameterizedTest
@ValueSource(ints = {-1, 0, 1, 4, 10000})
 void checkIsNumberNotPrime(int number) {

   assertFalse(isPrime(number));

}
```

Listing 3.22. Przykładowe użycie adnotacji @ValueSource [15]

- **@EnumSource -** pozwala na przekazanie argumentów typu wyliczeniowego, domyślnie przekazywane są wszystkie elementy klasy:

```
@ParameterizedTest
@EnumSource(Month.class)
 void checkNumberOfDaysInMonth(Month month) {

   assertTrue(getDays(month) > 0);

}
```

Listing 3.23. Przykładowe użycie adnotacji @EnumSource [7]

Można również wybrać konkretne elementy potrzebne do testów za pomocą takich atrybutów jak *names* (gdzie można przekazać nazwy lub wyrażenia regularne

zarówno jako listę, jak i pojedyncze wartości) oraz *mode* (*INCLUDE* - wymienione wartości enum będą uwzględnione w zestawie testowym, *EXCLUDE* - wszystkie wartości oprócz wymienionych, *MATCH_ALL* - spełniające wszystkie kryteria, *MATCH_ANY* - spełniające minimum jedno kryterium) [15]:

```
 @ParameterizedTest
 @EnumSource(value = Day.class, mode = INCLUDE, names =
{"MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY",})
  void checkWeekday(Day day) {

    assertTrue(isWeekday(day));
 }

 @ParameterizedTest
 @EnumSource(value = Day.class, mode = MATCH_ALL, names =
{".*DAY$", "^S.*"})
  void checkWeekend(Day day) {

    assertTrue(isWeekend(day));
 }
```

Listing 3.24. Przykładowe użycia adnotacji @EnumSource z atrybutami names i mode [opracowanie własne]

- **@MethodSource -** umożliwia przekazanie argumentów za pomocą metody; musi być ona statyczna (warunku tego nie muszą spełniać funkcje użyte w testach oznaczonych dodatkowo adnotacją *@TestInstance(Lifecycle.PER_CLASS)* oraz napisane w tej samej klasie) oraz zwracać typ, który JUnit może przetworzyć na Stream, np. Stream, IntStream, Collection, Iterator czy tablice:

```
 @ParameterizedTest
```

```
@MethodSource("dataProvider")
 void squaredTest(int number) {

    assertEquals(number * number, squared(number));

}

static Stream<Integer> dataProvider() {
        return Stream.of(-2, 0, 4, 100);
    }
```

Listing 3.25. Przykładowe użycie adnotacji @MethodSource [opracowanie własne]

W przypadku, gdy w adnotacji nie zostanie podana nazwa metody, brana jest pod uwagę funkcja o tej samej nazwie co metoda testowa.

Aby podać więcej parametrów do testu, możliwe jest użycie typu Stream składającego się z instancji *arguments*:

```
@ParameterizedTest
@MethodSource
 void multiplyTest(int firstNumber, int secondNumber, int
expectedResult) {

    int actualResult = multiply(firstNumber, secondNumber);
    assertEquals(expectedResult, actualResult);

}

static Stream<Arguments> multiplyTest() {
         return Stream.of(
        arguments(3, 4, 12),
```

```
        arguments(-1, 0, 0),
        arguments(5, 1000, 5000),
    );

    }
```

Listing 3.26. Przykładowe użycie adnotacji @MethodSource z wykorzystaniem Stream<Arguments> [15]

3.5. Mock, Stub i Spy

Ważne jest, by podczas pisania testów jednostkowych skupić się na testowaniu konkretnej "jednostki", np. komponentu. W przypadku, gdy korzysta ona z innego modułu, warto jest użyć mocka, czyli "atrapy" - obiektu który zastępuje realną implementację klasy. Pozwala to nie tylko na uniezależnienie działania testowanego fragmentu od innych części aplikacji, ale również na przyspieszenie wykonywania testów.

W celu zamockowania klasy można użyć adnotacji @Mock przed inicjalizacją obiektu lub zrobić to ręcznie, poprzez użycie funkcji mock() - np. mock([nazwa klasy].class), co zostało przedstawione w listingu 3.27 [16].

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

class ReservationTest{

    private UserFacade = mock(UserFacade.class);

    @Mock
    private RoomFacade roomFacade;
```

```
    private ReservationFacade = new
ReservationFacade(userFacade, roomFacade);

}
```

Listing 3.27. Przykładowe tworzenie mocków za pomocą funkcji lub adnotacji [opracowanie własne]

Metody takiej atrapy nie posiadają żadnej implementacji czy zachowania, a w razie wywołania dla typu innego niż *void* zwracają "puste" obiekty (np. cyfrę 0 dla typów liczbowych, puste kolekcje) [16]. Jest jednak możliwe sterowanie (zaślepianie - *"stubowanie"*) oraz kontrola takich funkcji na potrzeby testów. Służą do tego metody takie jak:

- **doThrow()** - rzucanie wyjątków
- **doReturn()** - zwracanie konkretnej wartości
- **doNothing()** - narzucenie braku zachowania
- **doAnswer()** - narzucenie konkretnego zachowania
- **doCallRealMethod()** - narzucenie zachowania z rzeczywistej implementacji

Oprócz tego należy również określić, do której funkcji mocka dane zachowanie ma się odnosić, za pomocą *when()*. Możliwe jest sprecyzowanie, dla których konkretnie parametrów ma ono działać lub zdefiniowanie ich jako *any()* - jakikolwiek, każdy.

Ostatecznie stubowanie ma strukturę podobną do given-when-then, co pozwala na łatwe odczytanie "kiedy (coś) zrób (coś)" [12]:

```
Calculator calculator = mock(Calculator.class)

// rzucenie wyjątku przy próbie podzielenia jakiejkolwiek
liczby przez 0

doThrow(new
BadValuesException()).when(calculator).divide(any(), 0);
```

```
// zwrócenie liczby 5 przy każdym dodaniu

doReturn(5).when(calculator).add(any(), any());

// narzucenie braku zachowania przy metodzie clear

doNothing().when(calculator).clear();
```

Listing 3.28. Przykładowe stubowanie [opracowanie własne]

Poza kontrolą nad tym, w jaki sposób metody mają się zachowywać, można również zweryfikować liczbę ich wywołań za pomocą *verify()* oraz takich określeń jak:

- **times(x)** - wywołano x razy
- **never()** - wywołano 0 razy
- **atLeast(x)** - wywołano maksymalnie x razy
- **atMost(x)** - wywołano minimalnie x razy
- **only()** - nie wywołano żadnej innej metody
- **timeout(x)** - wywołanie nie zajęło więcej niż x ms od uruchomienia testu
- **after(x)** - wywołanie nie zajęło mniej niż x ms od uruchomienia testu [12]

```
MyClass mockedClass = mock(MyClass.class)

// metoda firstMethod była wywołana 5 razy

verify(mockedClass, times(5)).firstMethod(any());

// metoda secondMethod z argumentem "5" była wywołana
minimum 2 razy

verify(mockedClass, atLeast(2)).secondMethod(5);
```

```
// metoda thirdMethod była wywołana dokładnie jeden raz,
metoda fourthMethod nie była wywoływana wcale, oprócz tego
mockedClass nie wywołało żadnej innej funkcji

verify(mockedClass).thirdMethod();
verify(mockedClass, never()).fourthMethod();
verifyNoMoreInteractions(mockedClass);
```

Listing 3.29. Przykładowe weryfikowanie wywołania funkcji [12]

Wywołanie funkcji liczy się przez cały cykl życia mocka, więc może być ono brane pod uwagę nie tylko w konkretnym teście, ale również w np. metodach z oznaczeniem *@BeforeAll*. Aby klasa zapomniała o jakichkolwiek wywołaniach funkcji, można użyć funkcji *reset()*, jest to jednak uważane za code smell i zaleca się tworzyć nowe mocki zamiast ich resetowania.

W momencie, gdy potrzebna jest zmiana zachowania tylko dla niektórych, konkretnych metod klasy, możliwe jest utworzenie obiektu *spy()* - wszystkie funkcje działają "normalnie" pod warunkiem że nie zostały zmodyfikowane w taki sam sposób, jak stubowało się mocki [12].

```
List list = new LinkedList();
List spy = spy(list);

when(spy.size()).thenReturn(100);

spy.add("abc");

// wyświetli "abc"
System.out.println(spy.get(0));
```

```
//wyświetli "100"
System.out.println(spy.size());

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

Listing 3.30. Przykładowe użycie spy() [12]

4. Zarządzanie testami w środowisku JUnit

Aby móc pisać testy w JUnit w wybranym projekcie, należy najpierw zaimportować odpowiednią bibliotekę. W przypadku korzystania z Mavena wystarczy dopisać zależność w pliku *pom.xml* [7]:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>(najnowsza wersja)</version>
    <scope>test</scope>
</dependency>
```

Listing 4.1. Wymagana zależność do działania JUnit [7]

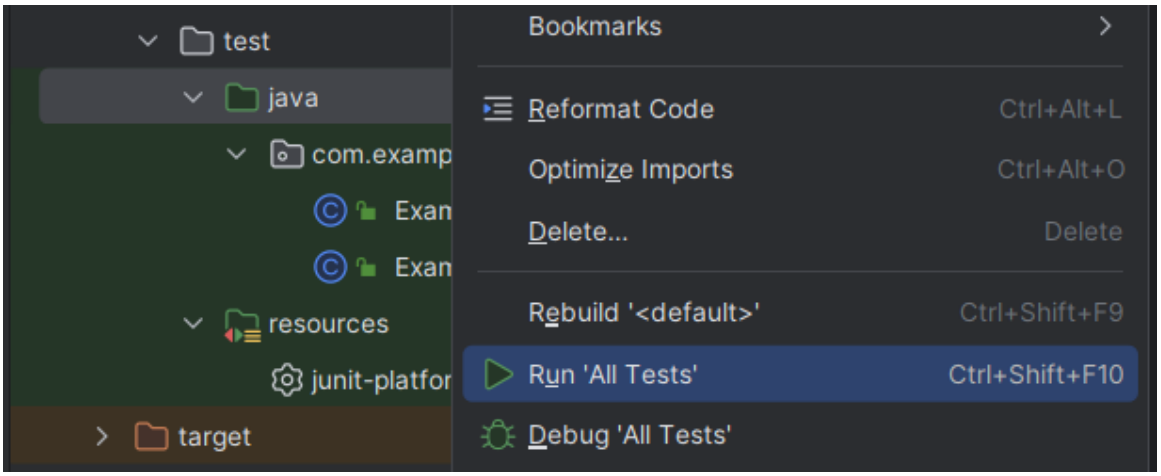
Następnie w celu ułatwienia zarządzania testami i ich prawidłowego działania, należy oznaczyć folder z nimi jako *Test Sources Root*. Opcjonalnie można dodać do niego katalog *resources*, a w nim plik *junit-platform.properties* zawierający dodatkowe opcje [14] :

```
junit.jupiter.displayname.generator.default=org.junit.jupite
r.api.DisplayNameGenerator$ReplaceUnderscores
```

Listing 4.2. Przykładowy plik junit-platform.properties [opracowanie własne]

4.1. Uruchamianie testów (Intellij Idea)

Zależnie od potrzeby można uruchomić wszystkie testy w projekcie, w konkretnym folderze czy też w konkretnej klasie.



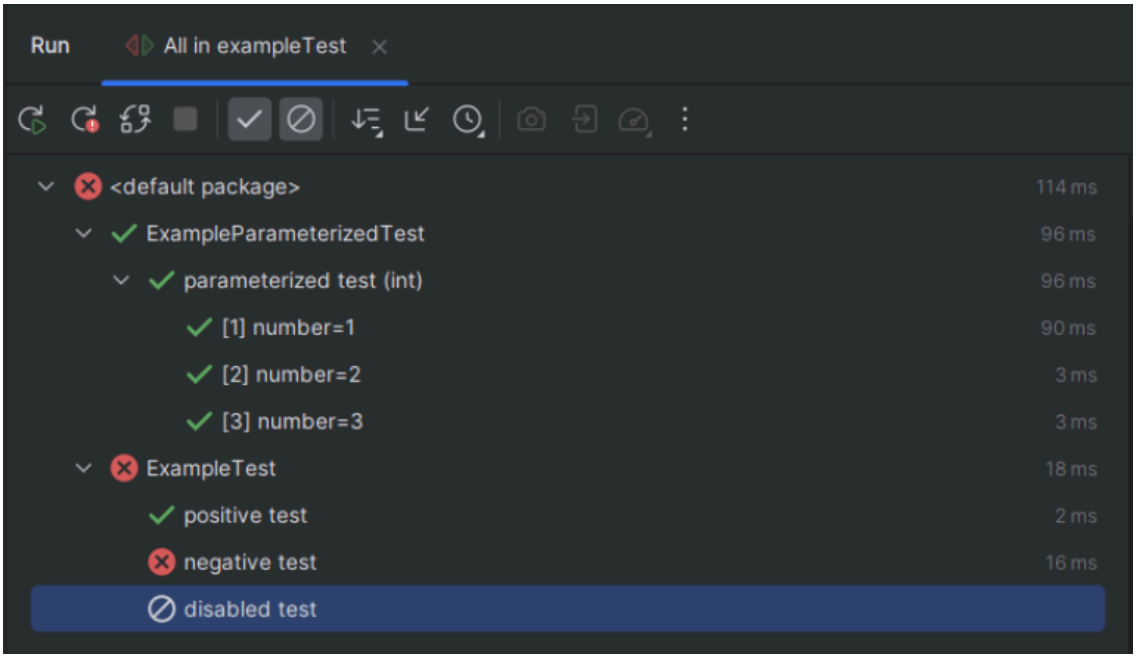
Rys 4.1. Uruchomienie wszystkich testów w projekcie [opracowanie własne]

Możliwe jest również uruchomienie pojedynczego, wybranego testu.

Rys 4.2. Uruchomienie wybranego testu [opracowanie własne]

4.2. Raportowanie testów (Intellij Idea)

Po uruchomieniu testów można w przejrzysty sposób wyświetlić ich wyniki:



Rys 4.3. Wyświetlanie wyników testów [opracowanie własne]

Użytkownik dostaje również jasny komunikat, ile testów zostało uruchomionych i z jakim rezultatem. W przypadku, gdy wynik nie jest pozytywny, wyświetlona zostaje informacja, które asercje się nie powiodły:

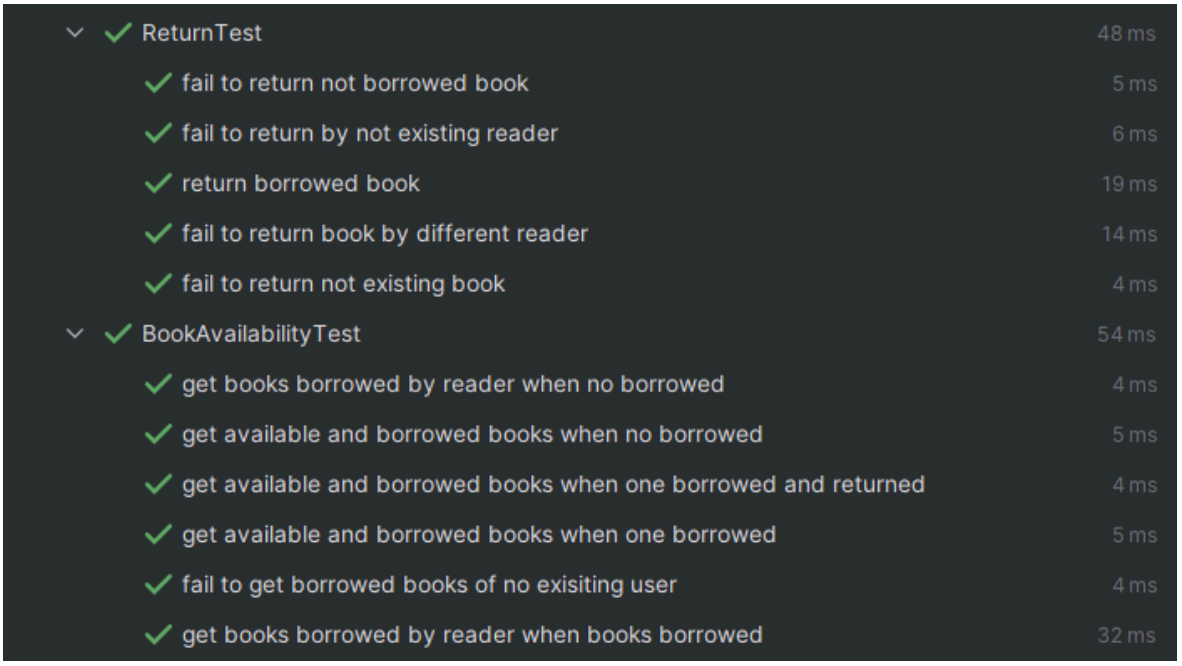
Rys 4.4. Informacja o niepowodzeniu testów [opracowanie własne]

5. Praktyczne zastosowanie JUnit

W ramach pracy zaimplementowano dwie aplikacje serwerowe, które korzystają z frameworka JUnit.

5.1. Aplikacja “Biblioteka”

Aplikacja jest skierowana dla pracowników biblioteki. Pozwala na zarządzanie książkami, czytelnikami, jak i również wykonywanie oraz śledzenie operacji takich jak wypożyczanie i zwracanie wybranych egzemplarzy.



Rys 5.1. Testy aplikacji “Biblioteka” (fragment) [opracowanie własne]

Program stosuje repozytoria, które stanowią warstwę pośrednią pomiędzy logiką biznesową a bazą danych. W testach, które z nich korzystają, wykorzystano ich wersje *InMemory* - klasy, które implementują metody “rzeczywistych” interfejsów (ich odpowiedników), jednak nie są w żaden sposób połączone z prawdziwymi danymi w pamięci. Sprawia to, że można łatwo nimi manipulować, a inicjalizacja nie zajmuje dużo czasu.

Na początku na potrzeby testowania utworzono uniwersalną klasę *InMemoryRepository*. Zamiast działać na rzeczywistej bazie danych, operuje ona na wartościach zawartych w mapie. Implementuje ona *JpaRepository*, a więc wszystkie potrzebne metody są przeładowane i dostosowane do działania klasy.

```
public abstract class InMemoryRepository<T, ID> implements
JpaRepository<T, ID> {
    Map<ID, T> table = new ConcurrentHashMap<>();

[...]

    @Override
    public <S extends T> List<S> saveAll(Iterable<S>
entities) {
        entities.forEach(this::save);
        return (List<S>) entities;
    }

    @Override
    public Optional<T> findById(ID id) {
        return Optional.ofNullable(table.get(id));
    }

    @Override
    public boolean existsById(ID id) {
        return table.containsKey(id);
    }

    @Override
    public List<T> findAll() {
        return new ArrayList<>(table.values());
    }
```

```
    @Override
    public List<T> findAllById(Iterable<ID> ids) {
        throw new RuntimeException("method not
implemented");
    }

    @Override
    public long count() {
        return table.size();
    }

    @Override
    public void deleteById(ID id) {
        table.remove(id);
    }

[...]

}
```

Listing 5.1. Fragment klasy InMemoryRepository [opracowanie własne]

Następnie dla każdego repozytorium utworzono odpowiadającą mu wersję *InMemory* - implementującą właściwy interfejs oraz rozszerzającą *InMemoryRepository*. Przykładem jest *BookOperationRepository* z listingu 5.2, które oprócz metod JpaRepository posiada również własną - *getLastBookOperation*.

```
@Repository
public interface BookOperationRepository extends
JpaRepository<BookOperation, UUID> {
```

```
    @Query("SELECT bo FROM BookOperation bo " +
            "WHERE bo.bookId = :bookId " +
            "ORDER BY bo.time DESC LIMIT 1")
    BookOperation getLastBookOperation(@Param("bookId") UUID
bookId);

}

public class InMemoryBookOperationRepository extends
InMemoryRepository<BookOperation, UUID> implements
BookOperationRepository {

     @Override
    public BookOperation getLastBookOperation(UUID bookId) {

        return table.values().stream()
          .filter(bo -> bo.getBookId() == bookId)
          .max(Comparator.comparing(BookOperation::getTime))
          .orElse(null);

    }

[...]

}
```

Listing 5.2. Interfejs BookOperationRepository oraz odpowiadająca mu klasa InMemoryBookOperationRepository (fragment) [opracowanie własne]

W celu dalszego ułatwienia i przyspieszenia testowania, została wydzielona klasa *BookOperationSetup*, która zajmuje się przygotowaniem środowiska pod testy sprawdzające operacje na książkach (wypożyczenie, zwrot, sprawdzenie dostępności) oraz posiadające te same warunki początkowe, czyli: w bibliotece istnieje jeden egzemplarz

książki i dwóch czytelników. Aby uniezależnić testowany moduł od komponentu odpowiadającego za czytelników, wszystkie interakcje z nim zostały zmockowane.

```
public class BookOperationSetup {
    BookRepository bookRepository = new
InMemoryBookRepository();
    BookOperationRepository bookOperationRepository = new
InMemoryBookOperationRepository();
    ReaderRepository readerRepository =
mock(InMemoryReaderRepository.class);

    BookFacade bookFacade = new BookFacade(bookRepository,
bookOperationRepository, readerRepository);

    BookToAddDto SHINING = BookToAddDto.builder()
            .title("The Shining")
            .author("Stephen King")
            .releaseDate(1977)
            .build();

    UUID READER_ONE_ID = UUID.randomUUID();
    UUID READER_TWO_ID = UUID.randomUUID();
    UUID bookId;
    BookDto book;

    @BeforeEach
    void setUp() {
        book = bookFacade.addBook(SHINING);
        bookId = book.getId();

when(readerRepository.existsById(READER_ONE_ID)).thenReturn(
true);
```

```
when(readerRepository.existsById(READER_TWO_ID)).thenReturn(
true);
    }

    @AfterEach
    void cleanUp() {
        bookRepository.deleteAll();
        bookOperationRepository.deleteAll();
    }

    [...]

}
```

Listing 5.3. Klasa BookOperationSetup (fragment) [opracowanie własne]

Przedstawiona w listingu 5.3. klasa *BookOperationSetup* jest rozszerzana przez m. in. *BorrowTest* - testy wypożyczania książek.

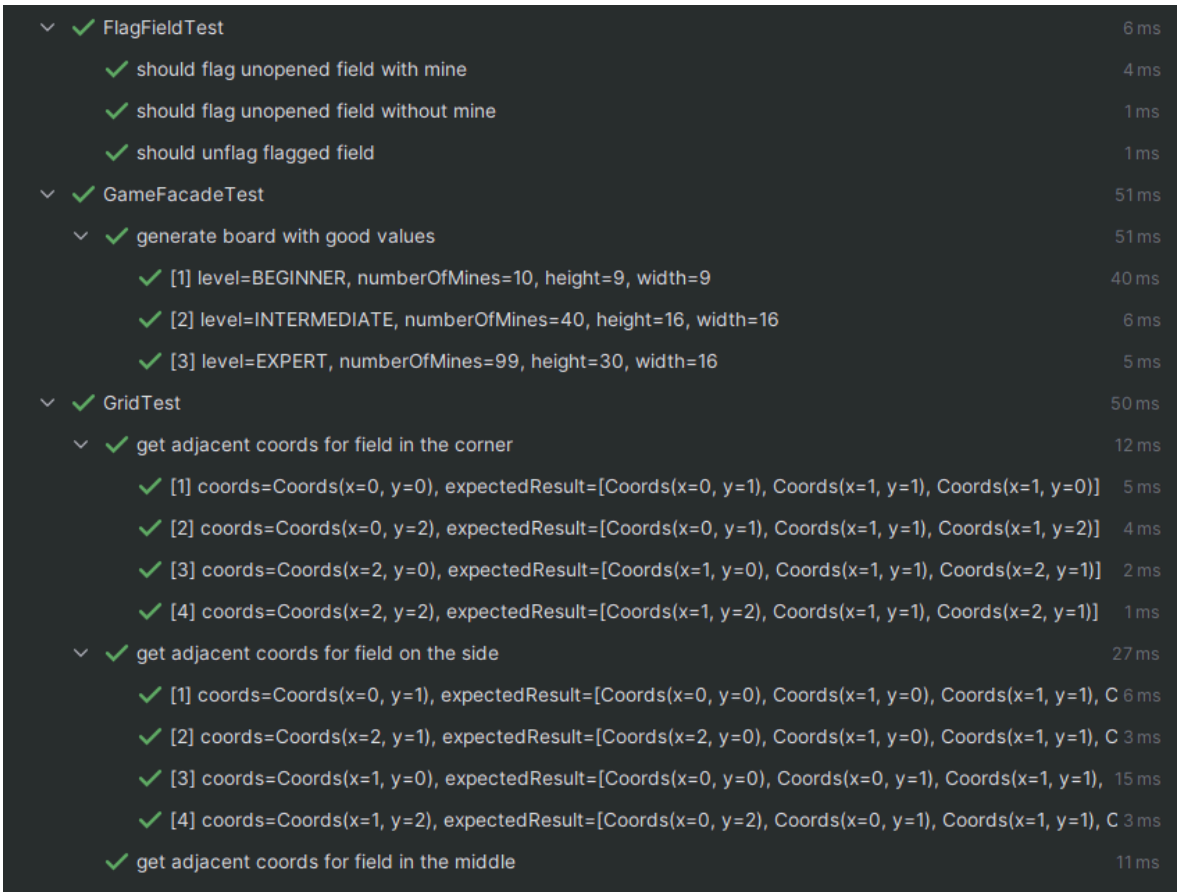
```
class BorrowTest extends BookOperationSetup{

    @Test
    void borrow_available_book() {
        BookOperationDto operation =
bookFacade.borrowBook(book.getId(), READER_ONE_ID);

        assertEquals(READER_ONE_ID,
operation.getReaderId());
        assertEquals(bookId, operation.getBookId());
        assertEquals(OperationType.BORROW,
operation.getType());
```

```
        assertEquals(List.of(operation),
bookFacade.getAllOperations());
    }

    @Test
    void fail_to_borrow_not_available_book() {

        bookFacade.borrowBook(bookId, READER_ONE_ID);

        assertThrows(BookNotAvailable.class, () ->
bookFacade.borrowBook(bookId, READER_ONE_ID));
    }

    @Test
    void fail_to_borrow_not_existing_book() {

        assertThrows(BookNotExisting.class, () ->
bookFacade.borrowBook(UUID.randomUUID(), READER_ONE_ID));
    }

    @Test
    void fail_to_borrow_by_not_existing_reader() {

        assertThrows(ReaderNotExisting.class, () ->
bookFacade.borrowBook(bookId, UUID.randomUUID()));
    }
}
```

Listing 5.4. Testy funkcjonalności wypożyczania książek (fragment) [opracowanie własne]

5.2. Aplikacja “Saper”

“Saper” to bardzo znana, logiczna gra komputerowa, której celem jest odkrycie wszystkich bezpiecznych pól na planszy. W chwili, gdy gracz trafi na minę, rozgrywka kończy się porażką. Istnieje również opcja flagowania wybranych obszarów tak, by przez przypadek ich nie odsłonić.



Rys 5.2. Testy aplikacji “Saper” (fragment) [opracowanie własne]

Z każdą rozgrywką generowana jest nowa plansza, w sposób losowy rozmieszczająca miny. Funkcjonalność ta sprawia, że trudno jest bez żadnych modyfikacji przeprowadzać testy jakichkolwiek działań gracza oraz łamana jest zasada ich powtarzalności. Z pomocą wtedy przychodzi *mock*, a dokładniej jego odmiana *mockStatic* pozwalająca zmieniać zachowanie statycznych metod. W celu uniknięcia powtarzalności kodu, inicjalizacja mocka wraz z “czyszczeniem” jego zachowania, zostały przeniesione do osobnej klasy *MockedRandomBoardGenerator* (listing 5.5):

```
public class MockedRandomBoardGenerator {

    private static MockedStatic<RandomBoardGenerator>
mockedRandomBoardGenerator;

    @BeforeEach
    void setUp() {
        mockedRandomBoardGenerator =
mockStatic(RandomBoardGenerator.class);
    }

    @AfterEach
    void cleanUp() {
        mockedRandomBoardGenerator.close();
    }

}
```

Listing 5.5. Klasa MockedRandomBoardGenerator [opracowanie własne]

Następnie jest ona rozszerzana przez klasy testujące rozgrywkę. W zależności od potrzeby, jej zachowanie (tj. zwracana plansza) jest modyfikowane w każdym przypadku testowym.

```
class FlagFieldTest extends MockedRandomBoardGenerator{

    @Test
    void should_flag_unopened_field_without_mine() {

when(RandomBoardGenerator.generate(any())).thenReturn(new
Board(createGrid(SMALL), 1, ONGOING));
```

```
        GameFacade game = new GameFacade(Level.BEGINNER);

        Board board = game.flagField(new Coords(0, 0));

        assertEquals(new
Board(createGrid(SMALL_FLAG_AT_0_0), 0, ONGOING), board);

    }

    @Test
    void should_flag_unopened_field_with_mine() {

when(RandomBoardGenerator.generate(any())).thenReturn(new
Board(createGrid(SMALL), 1, ONGOING));

        GameFacade game = new GameFacade(Level.BEGINNER);

        Board board = game.flagField(new Coords(1, 1));

        assertEquals(new
Board(createGrid(SMALL_FLAG_AT_1_1), 0, ONGOING), board);

    }

    @Test
    void should_unflag_flagged_field() {

   when(RandomBoardGenerator.generate(any())).thenReturn(new
Board(createGrid(SMALL_FLAG_AT_0_0), 0, ONGOING));

        GameFacade game = new GameFacade(Level.BEGINNER);
```

```
        Board board = game.flagField(new Coords(0, 0));

        assertEquals(new Board(createGrid(SMALL), 1,
ONGOING), board);

    }

}
```

Listing 5.6. Testy funkcjonalności flagowania pól [opracowanie własne]

Niektóre z testów korzystają również z klasy *GridSamples*. Posiada ona kilka szablonów siatki pól potrzebnych do planszy, a zastosowanie jej w testach (widoczne np. na listingu 5.4) zwiększa ich czytelność.

```
public class GridSamples {

    public static Grid createGrid(GridName name) {

        Map<Coords, Field> fields = new HashMap<>() {
            {
                put(new Coords(0, 0), new
Field(FieldStatus.UNOPENED, false, 1));
                put(new Coords(0, 1), new
Field(FieldStatus.UNOPENED, false, 1));
                put(new Coords(0, 2), new
Field(FieldStatus.UNOPENED, false, 1));
                put(new Coords(1, 0), new
Field(FieldStatus.UNOPENED, false, 1));
                put(new Coords(1, 1), new
Field(FieldStatus.UNOPENED, true, 0));
```

```
                put(new Coords(1, 2), new
Field(FieldStatus.UNOPENED, false, 1));
                put(new Coords(2, 0), new
Field(FieldStatus.UNOPENED, false, 1));
                put(new Coords(2, 1), new
Field(FieldStatus.UNOPENED, false, 1));
                put(new Coords(2, 2), new
Field(FieldStatus.UNOPENED, false, 1));
            }
        };
        int width = 3;
        int height = 3;

        switch (name) {
            case SMALL:
                break;
            case SMALL_OPEN_AT_0_0:
                fields.put(new Coords(0, 0), new
Field(FieldStatus.OPENED, false, 1));
                break;
            case SMALL_FLAG_AT_0_0:
                fields.put(new Coords(0, 0), new
Field(FieldStatus.FLAGGED, false, 1));
                break;
            [...]

        }

        return new Grid(fields, width, height);
    }

[...]
}
```

Listing 5.7. Klasa GridSamples (fragment) [opracowanie własne]

6. Podsumowanie i wnioski

Głównym celem pracy była szczegółowa analiza frameworka JUnit. Założenia zostały spełnione.

W pierwszej kolejności, aby uzasadnić potrzebę korzystania ze wspomnianego narzędzia, przybliżono temat testowania oprogramowania - czym dokładnie jest oraz jakie niesie ze sobą korzyści. Następnie skoncentrowano się na JUnit, prezentując jego funkcje wraz z poglądowymi przykładami, które miały ułatwić zrozumienie ich zastosowań. Przedstawione zostały również dwie aplikacje, które używają opisywanego frameworka do testów, aby pokazać jego funkcjonalności również w praktyce.

W kontekście rozwinięcia pracy można poszerzyć analizę o inne narzędzia z rodziny xUnit - w czym są podobne, pod jakimi względami się różnią, jak często są używane czy też w jaki sposób ewoluowały. Ciekawym aspektem do zbadania jest, na ile prototyp pochodzący z SUnit jest uniwersalny i czy sprawdza się równie skutecznie w innych językach programowania, jak to ma miejsce w przypadku popularnego JUnit dla Javy.

Dodatkowo warto rozważyć również zestawienie JUnit z alternatywnymi frameworkami do testowania języka Java, takich jak np. Spock lub TestNG. Każdy z nich posiada gamę własnych, unikalnych funkcji, które w zależności od rodzaju implementowanej aplikacji mogą okazać się mniej lub bardziej efektywne. Przydatne może okazać się porównanie pod tym kątem różnych narzędzi i zadanie pytania “W czym JUnit wypada lepiej niż reszta oraz czego mu brakuje?”.

Podsumowując, zgodnie z wcześniej wyznaczonym celem, niniejsza praca stanowi rozbudowany opis frameworka JUnit. Przykłady dołączone do każdej z funkcjonalności pozwalają na łatwe zobrazowanie, w jakich konkretnych przypadkach może ona zostać użyta. Czytelnik może również zauważyć ich praktyczne zastosowanie w działających, realnych aplikacjach.

W trakcie pisania pracy wywnioskowano, że JUnit jest wzorowym narzędziem do testowania kodu w języku Java. Posiada dużo różnorodnych funkcji, które umożliwiają pisanie testów w optymalny sposób z wykorzystaniem dobrych praktyk oraz jest na tyle uniwersalny, że sprawdza się w różnego rodzaju aplikacjach. Ponadto ciągły rozwój frameworka sprawia, że z biegiem czasu może stać się jeszcze lepszy.

Bibliografia

[1] Kent Beck, *TDD. Sztuka tworzenia dobrego kodu*, Helion, 2014

[2] Victor Farcic, Alex Garcia, *TDD. Programowanie w Javie sterowane testami*, Helion, 2016

[3] Robert C. Martin, *Czysty kod. Podręcznik dobrego programisty*, Helion, 2009

[4] Gulati Shekhar; Sharma Rahul, *Java Unit Testing with JUnit 5*, Apress, 2017

[5] Radosław Sokół, *Testowanie aplikacji Java za pomocą JUnit*, Helion, 2018

[6] Catalin Tudose, *JUnit in Action, Third Edition*, 2020

[7] *A Guide to JUnit 5* [ONLINE] https://www.baeldung.com/junit-5

[8] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman, *The Java® Languag Specification, Java SE 21 Edition* [ONLINE] https://docs.oracle.com/javase/specs/

[9] Przemysław Bykowski, *Piramida, Diament i Trofeum – Jak Rozplanować Testy Automatyczne w Aplikacji* [ONLINE] https://bykowski.pl/piramida-diament-i-trofeum-jak-rozplanowac-testy-automatyczne-w-aplikacji/

[10] *Testy automatyczne oprogramowania. Co warto wiedzieć?* [ONLINE] https://chcenawczoraj.pl/software/testy-automatyczne-oprogramowania-co-warto-wiedziec

[11] Lokesh Gupta, *JUnit 5 Expected Exception – assertThrows() Example* [ONLINE] https://howtodoinjava.com/junit5/expected-exception-example/

[12] *Mockito* [ONLINE] https://javadoc.io/static/org.mockito/mockito-core/5.8.0/org/mockito/Mockito.html

[13] *Refaktoryzacja kodu* [ONLINE] https://javastart.pl/baza-wiedzy/wprowadzenie/refaktoryzacja

[14] *Testing* [ONLINE] https://www.jetbrains.com/help/idea/testing.html

[15] *JUnit 5* [ONLINE] https://junit.org/junit5/

[16] Piotr Pelczar, *Mockito w pigułce* [ONLINE] https://softwareskill.pl/mockito-w-pigulce

[17] Piotr Pelczar, *Zasady FIRST* [ONLINE] https://softwareskill.pl/zasady-first

[18] Mike Pitt, *What is the "happy path"?* [ONLINE] https://blog.zeplin.io/what-is-the-happy-path

[19] *JUnit 5 Parameterized Tests* [ONLINE] https://reflectoring.io/tutorial-junit5-parameterized-tests/

[20] *Adnotacje w języku Java* [ONLINE] https://www.samouczekprogramisty.pl/adnotacje-w-jezyku-java/

[21] Aleksandra Skalska, *Testowanie niezbędnym elementem tworzenia oprogramowania. Jakie są jego typy, rodzaje i poziomy?* [ONLINE] https://udigroup.pl/blog/testowanie-oprogramowania-typy-rodzaje-poziomy/

[22] *Co to jest asercja?* [ONLINE] https://testelka.pl/asercja/

[23] Sebastian Zawadzki, *Testy manualne vs. automatyczne – poznaj kluczowe różnice* [ONLINE] https://smartbees.pl/blog/testy-manualne-vs-automatyczne

[24] Marek Żukowicz, *Piramida testów i ciągła integracja* [ONLINE] https://testerzy.pl/baza-wiedzy/artykuly/piramida-testow-i-ciagla-integracja

[25] Hüdai Semih Çavdar, *Test Driven Development — 101* [ONLINE] https://medium.com/trendyol-tech/test-driven-development-101-a0038f257bd2

Spis rysunków

Spis listingów

Listing 5.7. Klasa GridSamples (fragment) [opracowanie własne]