AKADEMIA TARNOWSKA
Wydział Politechniczny

Kierunek: ***Informatyka***

Specjalność: Informatyka Stosowana

Specjalizacja: Inżynieria Oprogramowania

2023/2024

Arkadiusz Nocek

PRACA INŻYNIERSKA

Testowanie kodu z wykorzystaniem frameworka Spock

Promotor pracy:

mgr inż Tomasz Gądek

Tarnów, 2024

Spis treści

1. Wstęp

W dzisiejszym zmieniającym się dynamicznie środowisku programistycznym, utrzymanie wysokiej jakości i niezawodności oprogramowania stało się niezwykle istotne. Testowanie kodu odgrywa kluczową rolę podczas tworzenia oprogramowania, ponieważ umożliwia weryfikację poprawności kodu oraz zachowanie systemu, który on oferuje. Wraz z postępem technologicznym, narzędzia do testowania ewoluowały, oferując nowe możliwości i funkcjonalności, umożliwiając skuteczne i efektywne testowanie kodu. Jednym z takich narzędzi testowych jest framework Spock, który zyskał popularność wśród programistów jako efektywne narzędzie do testowania w środowisku języka Groovy. Niniejsza praca ma na celu przybliżenie tego zaawansowanego frameworka testowego oraz zbadanie jego zastosowania i efektywności w procesie testowania oprogramowania.

1.1. Motywacja

Rozwój i złożoność współczesnych aplikacji, nie tylko dużych, ale i małych firm, wymaga zapewnienia im niezawodności. Jednakże nie zawsze się to udaje. Nie ładująca się strona, niemożność zalogowania się do serwisu, chaotyczny tekst na stronie bądź wyskakujący błąd w czasie używania aplikacji. Są to tylko przykłady, które na pewno, każdy doświadczył. Pomimo próby dostarczenia jak najlepszego oprogramowania, błędy te napotyka każdy kto korzysta z jakiejkolwiek technologii, która wymaga oprogramowania. Rozwój i rozbudowa oprogramowania to nie jedyne czynniki, które mogą obniżyć ich niezawodność, ale również rozwój urządzeń i technologii, z których te aplikacje korzystają. Przy każdej nowej wersji aplikacji i urządzenia, na które jest ono przeznaczone istnieje ryzyko powstania nowych błędów . Dlatego też wybór tematu tej pracy wynika z rozpoznania ogromnej roli, jaką odgrywa testowanie w dzisiejszym środowisku programistycznym. Motywacja do przeprowadzenia tej pracy opiera się na potrzebie zgłębienia wiedzy na temat zaawansowanych technik testowania, które mogą przyczynić się do zwiększenia niezawodności i jakości oprogramowania, co z kolei ma bezpośredni wpływ na satysfakcję użytkowników oraz powodzenie projektów biznesowych.

Framework Spock, będąc jednym z zaawansowanych narzędzi testowych, oferuje programistom nie tylko wygodne i elastyczne środowisko testowe, ale także możliwość skutecznego weryfikowania funkcjonalności i zachowania tworzonego oprogramowania. W kontekście rosnących oczekiwań klientów i konkurencji na rynku, umiejętne wykorzystanie frameworka Spock może przyczynić się do skrócenia cyklu wytwarzania oprogramowania, zmniejszenia ryzyka wystąpienia błędów oraz zwiększenia zaufania do tworzonego produktu.

Badanie efektywności tego narzędzia testowego jest kluczowym elementem tej pracy, mającym na celu dostarczenie praktycznych wskazówek i wniosków dla programistów oraz inżynierów oprogramowania, którzy dążą do poprawy jakości swojego kodu oraz procesu testowania.

1.2. Cel pracy

Celem niniejszej pracy jest przeprowadzenie kompleksowej analizy frameworka Spock oraz zbadanie jego skuteczności i efektywności w testowaniu kodu. Praca ma na celu zaprezentowanie zalet i możliwości tego narzędzia oraz zbadanie, w jaki sposób może on przyczynić się do zwiększenia jakości i niezawodności tworzonego oprogramowania, poprzez szczegółową analizę funkcji, składni i możliwości frameworka Spock. Praca dostarcza wiedzę na temat potencjalnych korzyści wynikających z jego implementacji w procesie tworzenia i testowania oprogramowania.

1.3. Zakres pracy

W ramach projektu inżynierskiego opisany został język Groovy, ze szczególnym uwzględnieniem możliwości w kontekście testowania. Ponadto, dokonano szczegółowego opisu frameworka Spock, jak i różnych rodzajów testowania w dziedzinie informatyki. Na część praktyczną zaimplementowane zostały dwie niezależnie od siebie działające aplikacje:

- System zarządzania studentami.
- Kalkulator matematyczny odwrotnej notacji polskiej.

Obydwie aplikacje zostaną szczegółowo przedstawione w dalszej części pracy.

2. Język Groovy

Język Groovy to zwinny i dynamiczny język programowania przeznaczony dla maszyny wirtualnej Javy opiera się na jej mocnych stronach, ale posiada dodatkowe funkcje o większych możliwościach zainspirowane językami takimi jak Python, Ruby i Smalltalk. Pozwala on na statyczną weryfikację typów i statyczną kompilację kodu celem zwiększenia jego niezawodności i wydajności. Ułatwia pisanie i budowanie skryptów dzięki podstawowym funkcjom przetwarzania, zdolnościom obiektowym i DSL Ant oraz zwiększa produktywność poprzez redukcję kodu podczas tworzenia aplikacji internetowych, aplikacji GUI, baz danych lub konsolowych. Upraszcza testowanie poprzez obsługę testów jednostkowych i tworzenie atrap "out-of-the-box" (W kontekście oprogramowania lub technologii, termin ten oznacza funkcję lub możliwość, która jest dostępna natychmiast po zainstalowaniu programu lub narzędzia, bez konieczności dokonywania dalszych ustawień lub modyfikacji[10]). Język Groovy płynnie integruje się ze wszystkimi istniejącymi klasami i bibliotekami Javy i kompiluje się bezpośrednio do kodu bajtowego Javy, co pozwala na jego wykorzystanie wszędzie tam, gdzie można go używać [6].

2.1. Deklarowanie zmiennych

Zmienne to konstrukcje programistyczne, dzięki którym można przechowywać dane. Wszystkie zmienne mają swoją nazwę i typ [11].

2.1.1. Typy zmiennych

Typ zmiennej definiuje rodzaje danych jakie zmienna może przechowywać. Dzięki nazwie każda zmienna ma jednoznaczny identyfikator [11]. Nie trzeba jawnie określać typów zmiennych podczas deklaracji, co pozwala na szybsze i bardziej zwięzłe pisanie kodu. Zmienne są dynamicznie typowane, co oznacza, że ich typy mogą być zmieniane w trakcie działania programu. Groovy jest językiem kompatybilnym z Javą, obsługuje wszystkie typy danych które są dostępne w Javie [4].

```
def dynamicVariable = "Groovy"
def dynamicNumber = 42
```

Listing 2.1 Przykład zmiennych dynamicznie typowanych w języku Groovy.

2.1.2. Sposoby deklaracji zmiennych

W większości języków programowania, wymaga się deklarowania typów, ponieważ w przeciwnym wypadku kompilator zgłosi błąd. Jednym z wyjątków jest język Groovy, w którym kompilator sam przypisuje typ zmiennej, jeśli nie ma przypisanego typu [11][4].

```
def dynamicVariable = "Groovy"
def dynamicNumber = 42
String text = "Groovy String"
int number = 42
List<String> stringList = ["one", "two", "three"]
```

Listing 2.2 Przykład deklaracji zmiennych dynamicznie i za pomocą deklarowania typów w języku Groovy.

2.1.3. Zmienne statyczne i finalne

Elementy statyczne obejmują zarówno pola, jak i metody klasy, które mają zdolność istnienia niezależnie od egzystencji obiektu danej klasy. Takie metody bądź pola współdzielone są między wszystkimi instancjami klasy. Klasa bądź jej składowa może być finalna, przez co nie można jej dziedziczyć po deklaracji [11]. Deklaracja zmiennych finalnych zapewnia, że ich wartość nie może być zmieniona po zainicjalizowaniu [4].

```
class Example {
    final finalVarDeclaration = "Groovy"
    Example() {
        finalVarInConstructor = "final in Constructor"
    }
    final finalVarInConstructor
    void printVariables() {
        println("finalVarDeclaration: $finalVarDeclaration")
        println("finalVarInConstructor: $finalVarInConstructor")
    }
}
def example = new Example()
example.printVariables()
```

Listing 2.3 Przykład zmiennych statycznych i finalnych w języku Groovy.

2.2. Instrukcje warunkowe

Instrukcje warunkowe pełnią funkcję sprawdzania określonych warunków, zgodnie z ich nazwą. Dzięki nim, w zależności od prawdziwości danego warunku, możliwe jest realizowanie różnych operacji [11].

2.2.1. Składnia instrukcji warunkowych

Po słowie kluczowym “if” umieszcza się warunek do sprawdzenia w nawiasie okrągłym, a za nim blok instrukcji, który ma zostać wykonany w przypadku, kiedy warunek jest prawdziwy [11].

```
if (number > 0)
    println("Positive")
else if (number < 0)
    println("Negative")
else
    println("Zero")
```

Listing 2.4 Przykład instrukcji warunkowej w języku Groovy.

2.2.2. Obsługa wielokrotnych warunków

W przypadku instrukcji switch, po słowie kluczowym “switch”, umieszcza się nawias okrągły, a w nim dowolne wyrażenie, którego wynikiem będzie wartość arytmetyczna, lub ciąg znaków. Następnie umieszcza się blok “case”, gdzie po słowie kluczowym “case” umieszcza się oczekiwany wynik, a po dwukropku instrukcje do wykonania, jeśli wynik równa się wyrażeniu w nawiasie switch[11]. W Groovym istnieje możliwość obsługi wielu warunków za pomocą konstrukcji "switch", która może być bardziej czytelna w przypadku wielu opcji [4].

```
def color = "red"
switch (color) {
    case "red":
        println("color is red")
        break
    case "green":
        println("color is green")
        break
    case "blue":
        println("color is blue")
        break
    default:
        println("color is different than red,green or blue")
}
```

Listing 2.5 Przykład wielokrotnego warunku w języku Groovy.

2.2.3. Operatory porównania

Do operatorów porównań, należą operatory równości i operatory relacji, ich zadaniem jest porównywanie wartości zmiennych bądź obiektów. Operatory równości, występują w dwóch formach, jako „==”, dla którego wartość sprawdzanego wyrażenia jest prawdziwa, gdy porównywane zmienne są równe, bądź „!=”, dla którego wartość sprawdzanego wyrażenia jest prawdziwa, gdy porównywane zmienne są różne. Do operatorów relacji należą „<”, „>”, „<=”, „>=”. Kolejno, oznaczają (w stosunku do pierwszego): mniejsze, większe, mniejsze bądź równe, większe bądź równe.

```
def weather = "Sunny"
if (weather == "Sunny") {
    println("It’s a nice day!")
}
```

Listing 2.6 Przykład użycia operatora porównania w języku Groovy.

2.2.3. Operatory logiczne

Warunki logiczne wykorzystują operatory logiczne, są one symbolami, które działają na argumentach reprezentujących wartości logiczne. Do operatorów należą operatory arytmetyczne, logiczne, przypisania, relacyjne, bitowe [13]. Groovy umożliwia elastyczne stosowanie warunków logicznych za pomocą różnych operatorów logicznych, takich jak "&&", "||" i "!" [4].

```
def temperature = 25
def weather = "Sunny"
if (temperature > 20 && weather == "Sunny") {
    println("It’s a very nice day!")
}
def isWeekend = true
isWeekend ? println("Enjoy the weekend!") : null
```

Listing 2.7 Przykład użycia operatora logicznego w Groovy

2.3. Pętle

Pętle to struktury programistyczne, które umożliwiają realizację powtarzalnych operacji[11].

2.3.1. Składnia pętli

Standardowa pętla for składa się z zadeklarowania zmiennej i przypisania jej wartości, następnie tak długo, dopóki wyrażenie jest prawdziwe, wykonuje się instrukcję w bloku oraz zwiększa bądź zmniejsza wartość licznika pętli [11].

```
for (int i = 1; i <= 5; i++) {
    println("Iteration $i")
}
```

Listing 2.8 Przykład pętli for w języku Groovy.

2.3.2. Pętla for-each

Pętla for-each, zwana również rozszerzoną pętlą for, automatycznie iteruje po kolekcji obiektów lub też po tablicy [11]. Groovy pozwala na używanie pętli for-each na różnych typach danych co sprawia, że jest bardziej elastyczny [4].

```
def mixedData = [1, 'two', 3.0, true]
for (element in mixedData) {
    println(element)
}
```

Listing 2.9 Przykład pętli for-each w języku Groovy.

2.3.3. Obsługa pętli while

Podobnie jak pętla for, pętla while służy do wykonywania powtarzających się czynności. Możemy ją stosować, kiedy liczba powtarzanych operacji nie jest znana, ale jest to podział umowny, ponieważ oba typy pętli można zapisać w taki sposób, aby były swoimi funkcjonalnymi odpowiednikami [11].

```
def count = 0
while (count < 5) {
    println("Count: $count")
    count++
}
```

Listing 2.10 Przykład pętli while w języku Groovy.

2.4. Kolekcje

Kolekcje, to sposób grupowania obiektów. Ze względu, na użyteczność przy testowaniu, opisane zostaną tablice, listy i mapy, w dalszych podrozdziałach.

2.4.1. Obsługa tablic

Tablice to struktury danych pozwalające na przechowywanie w sposób uporządkowany danego zbioru elementów [11].

```
def colors=["red","green","blue"] as String[]
colors.putAt(0,"black")//  zmiana  pierwszej  pozycji  w  tablicy  na
"czarny"
println colors[0]// wypisanie pierwszej pozycji z tablicy
```

Listing 2.11 Przykład deklaracji tablicy i obsługi elementów w tablicy w języku Groovy.

2.4.2. Obsługa list

Listy można rozumieć jako bardziej elastyczną wersję tablicy. Oferuje ona określoną kolejność danych, łatwe dodawanie oraz usuwanie elementów, bez potrzeby deklarowania, bądź zmieniania rozmiaru listy [17]. Groovy zapewnia zaawansowane funkcje obsługi list, na całych listach za pomocą wbudowanych metod [4].

```
def groovyList = ["Element 1", "Element 2", "Element 3"]
groovyList << "New Element"// dodawanie nowego elementu na koniec
listy
def newElement = groovyList[1]// dostęp do elementu po indeksie
groovyList.add(1, "New Element on index 1")// dodawanie nowego
elementu na określonej pozycji
groovyList[1] = "Updated Element 2"// aktualizacja elementu na
indeksie 1
def drugiElementPoZmianie = groovyList[1]// dostęp do zmienionego
elementu
groovyList.remove("Element 3")// usunięcie z listy elementu
“Element 3”
groovyList.remove[0]// usunięcie elementu z indeksu 0
```

Listing 2.12 Przykład deklaracji listy i obsługi elementów w liście w języku Groovy.

2.4.3. Obsługa map

Mapy, pozwalają do danego parametru przypisać wartość klucza. Następnie, aby pobrać z mapy poszczególne dane, wykorzystujemy unikalny klucz [18]. Groovy umożliwia prostą i elastyczną obsługę map, włączając w to dynamiczne dodawanie i usuwanie kluczy oraz wartości. Możliwe jest stosowanie skróconej składni do manipulowania mapami, co przyczynia się do czytelności kodu[4].

```
def groovyMap = [:]
groovyMap["key1"] = "Value 1"
groovyMap["key2"] = "Value 2"
groovyMap.remove("key1")
groovyMap.each { key, value ->
    println("Key: $key, Value: $value")
}
```

Listing 2.13 Przykład deklaracji mapy i obsługa elementów w mapie w języku Groovy.

2.5. Range

Zakres określa sekwencję wartości zmiennej [19].

2.5.1. Składnia zakresów

W Groovym zakresy są zapisywane za pomocą operatora "..", co umożliwia definiowanie sekwencji wartości pomiędzy dwoma punktami. Możliwe jest tworzenie zakresów nie tylko z liczb, ale również daty, znaków i innych obiektów [4].

```
def integerRange = 1..5
integerRange.each { println it }
def charRange = 'a'..'z'
charRange.each { println it }
```

Listing 2.14 Przykład zakresu liczbowego i znakowego w języku Groovy.

2.6. Klasy

Klasy są opisami obiektów, mogą przechowywać dane i realizować zadania polecone przez programistę [11].

2.6.1. Składnia deklaracji klas

Składowymi elementami klasy są pola oraz metody. Pola przechowują dane, natomiast metody wykonują różne operacje [11]. W Groovym, składnia deklaracji klas jest bardziej zwięzła i elastyczna niż w Javie. Nie trzeba deklarować typów zmiennych, a konstruktor i metody mogą być zdefiniowane bezpośrednio w ciele klasy [4].

```
class Car {
    def brand
    def model
    def year
    Car(brand, model, year) {
        this.brand = brand
        this.model = model
        this.year = year
    }
    def displayInfo() {
        println "Car: $year $brand $model"
    }
}
```

Listing 2.15 Przykład deklaracji klasy w języku Groovy.

2.7. Interfejsy

Interfejsy, można traktować jako klasy czysto abstrakcyjne, niemające żadnej implementacji [11].

2.7.1. Składnia deklaracji interfejsów

Interfejsy zawierają metody, które muszą zostać zaimplementowane i zdefiniowane w danej klasie [20].

```
interface RunOrStop {
   String status
   void start()
   void stop()
}

class Car implements RunOrStop {
   def brand
   def model
   def year
   Car(brand, model, year) {
       this.brand = brand
       this.model = model
       this.year = year
   }
   def displayInfo() {
       println "Car: $year $brand $model"
   }
   String status=new String("stop")
   void start(){
       status="run"
   }
   void stop(){
       status="stop"
   }
}
```

Listing 2.16 Przykład deklaracji interfejsu i jego implementacja do klasy w języku Groovy.

2.8. Dziedziczenie

Dziedziczenie jest wyrażane za pomocą słowa extends, polega na przejęciu od klasy bazowej, dozwolonych przez klasę bazową, pól i metod oraz dodanie własnych do klasy potomnej [11].

2.8.1. Składnia dziedziczenia klas

Groovy umożliwia dziedziczenie po klasach w sposób zbliżony do Javy, co pozwala na elastyczne zarządzanie relacjami między klasami [4].

```
class Vehicle {
    def brand
    Vehicle(brand) { this.brand = brand }
    def start() { println "Starting the vehicle" }
}
class Car extends Vehicle {
    def model
    Car(brand, model) { super(brand); this.model = model }
    def drive() { println "Driving the car" }
}
```

Listing 2.17 Przykład dziedziczenia klasy w języku Groovy.

3. Framework Spock

Spock to framework, który umożliwia testowanie i generowanie specyfikacji, dla aplikacji Java i Groovy. Dzięki zależnościom JUnit, Spock jest kompatybilny z większością środowisk programistycznych (IDEs), narzędziami do budowy (build tools) oraz serwerami ciągłej integracji. Spock czerpie inspirację z JUnit, jMock, RSpec, Groovy, Scala, Vulcans [5].

3.1. Pierwsze kroki

Aby rozpocząć pracę z frameworkiem Spock, należy najpierw zainstalować go w środowisku programistycznym. Proces instalacji Spocka można wykonać przy użyciu menedżera zależności, narzędzi do zarządzania pakietami [5]:

```
import spock.lang.*
```

Listing 3.1 Import frameworka Spock w języku Groovy.

Po zainstalowaniu Spocka konieczne jest skonfigurowanie projektu testowego w środowisku programistycznym. W tej fazie należy zapewnić integrację Spocka z istniejącym kodem źródłowym aplikacji. W zależności od specyfiki projektu, mogą być wymagane dodatkowe ustawienia środowiska, które umożliwią prawidłowe działanie testów opartych na Spocku [5].

3.2. Specyfikacja

Aby rozpocząć testy należy, odziedziczyć klasę Specification, która jest dostępna w pakiecie spock.lang. Sposób umieszczenia jej w kodzie, wygląda następująco [5].

```
class MyFirstSpecification extends Specification {
  // pola instancji
  // metody przygotowujące środowisko testowe
  // metody funkcjonalności
  // metody pomocnicze
}
```

Listing 3.2 Przykład specyfikacji frameworka Spock[5].

3.3. Pola

Pożądaną praktyką jest inicjowanie pól instancji, na samym początku szkieletu specyfikacji. Obiekty przechowywane w polach instancji nie są współdzielone z metodami funkcjonalności. Oznacza to że, każda funkjonalność posiada swoją własną kopie obiektu, dzięki czemu izoluje się metody funkcjonalności od siebie nawzajem [5].

```
import spock.lang.*

class UserSpec extends Specification {
   String userName
   int number = 2
   def platform = new Platform()
   // metody do przygotowania środowiska testowego
   // metody funkcjonalności
   // metody pomocnicze
}
```

Listing 3.3 Przykład utworzenia pola instancji w języku Groovy, za pomocą frameworka Spock.

Jeśli jednak zajdzie potrzeba współdzielić obiekty, należy zastosować pole *@Shared*. Możliwe jest również zastosowanie pól statycznych dla stałych[5].

```
import spock.lang.*

class CalendarSpec extends Specification {
   @Shared def Date actualTime = new Date()
   static final timeInInt = 11.32
   // metody do przygotowania środowiska testowego
   // metody funkcjonalności
   // metody pomocnicze
}
```

Listing 3.4 Przykład wykorzystania pola @Shared, wraz z polami statycznymi w języku Groovy[5].

3.4. Metody przygotowujące środowisko testowe

Przygotowanie środowiska testowego można wykonać na wiele sposobów, framework Spock oferuję 4 możliwości ustawień, gdzie każda z nich jest opcjonalna, ale można ją zadeklarować tylko raz w każdym typie specyfikacji [5].

```
def setupSpec() {}// wykonuje się 1 raz przed wszystkimi testami w
ramach specyfikacji
def setup() {}// uruchamia się przed każdym testem
```

```
def cleanup() {}// uruchamia się po każdym teście
def cleanupSpec() {}// uruchamiana jest po zakończeniu wszystkich testów
```

Listing 3.5 Przykład utworzenia metod przygotowujących środowisko testowe w języku Groovy, frameworka Spock.

3.5. Kolejność wywoływania

Jeśli w podklasach specyfikacji nastąpi nadpisanie metod do przygotowania środowiska testowego, framework Spock sam znajdzie i ustawi metody do przygotowania środowiska testowego w odpowiedniej hierarchii obojętnie od ich kolejności w kodzie, utrzymując priorytet klasy nadrzędnej nad podrzędną klasą specyfikacji. Dzięki czemu mając wiele typów specyfikacji, ustawi je w następującej kolejności (Tabela 3.1) [5].

Kolejność wywoływania	Rodzaj_klasy.Metoda_do_przygotowania_środowiska_testowego
1	`nadrzedna_Klasa.setupSpec`
2	`podrzedna_Klasa.setupSpec`
3	`nadrzedna_Klasa.setup`
4	`podrzedna_Klasa.setup`
5	`metody funkcjonalności`
6	`podrzedna_Klasa.cleanup`
7	`nadrzedna_Klasa.cleanup`
8	`podrzedna_Klasa.cleanupSpec`
9	`nadrzedna_Klasa.cleanupSpec`

Tabela 3.1 Kolejność wykonywania metod do przygotowania środowiska testowego w frameworku Spock [5].

3.6. Metody Funkcjonalności

Metody funkcjonalności są rdzeniem specyfikacji. Określają właściwości i aspekty, które szuka się w systemie podanym w specyfikacji. Składają się z 4 faz, w określonej kolejności, gdzie 1 oraz 4 faza, nie jest obowiązkowa (Tabela 3.2).

Kolejność fazy	Nazwa fazy
1	Ustawienie szkieletu funkcjonalności
2	Dostarczenie bodźca do systemu podanym w specyfikacji
3	Określenie odpowiedzi oczekiwanej z systemu podanego w specyfikacji
4	Wyczyszczenie szkieletu funkcjonalności

Tabela 3.2 Fazy metod funkcjonalności frameworka Spock [5].

Według konwencji nazywa się je za pomocą literałów tekstowych *String* [5].

```
def "konwencyjna nazwa metody funkcjonalnosci"() {
  // bloki funkcjonalności
}
```

Listing 3.6 Przykład konwencyjnej nazwy metody funkcjonalności frameworka Spock.

3.7. Bloki podstawowe

Każda metoda funkcjonalności musi posiadać co najmniej jeden oznakowany blok, aby osiągnąć którąkolwiek z faz podanych w tabeli 3.2. Oznakowany blok to taki, który zaczyna się od nazwy bloku plus dwukropek. Zasięg bloku to koniec metody bądź nowy blok w metodzie. Do bloków podstawowych zaliczamy *given, when, then.*

Ustawienie szkieletu funkcjonalności następuje w bloku *given*, przed którym nie może się znaleźć żaden inny blok oraz nie może się powtórzyć w tej samej metodzie funkcjonalności. Jego zadaniem jest określenie ustawień początkowych [5].

```
given:
def a = 2
def b = 3
def object= new Object()
```

Listing 3.7 Przykład utworzenia szkieletu funkcjonalności w bloku given w języku Groovy, za pomocą frameworka Spock.

Dostarczenie bodźca do systemu następuję w bloku *when*, po którym zawsze deklaruje się blok *then*, który dostarcza oczekiwaną odpowiedź z systemu. W bloku *when* wykonywana jest operacją, którą chcemy sprawdzić, natomiast blok *then* jest przeznaczony do weryfikacji, czy otrzymane rezultaty są zgodne z oczekiwaniami [5].

```
import spock.lang.*

class ExampleSpec extends Specification {
   // pola instancji
   // metody do przygotowania środowiska testowego
   def "przykładowy test"(){
       given:
       def a = 2
       def b = 3
       def object= new Object()
       when:
       def result = a + b

       then:
       result == 5
   }
   // metody pomocnicze
}
```

Listing 3.8 Przykład użycia bloków when i then w języku Groovy, za pomocą framewrka Spock.

Po dodaniu kodu bloku *when* i *then*, można już oczekiwać wyniku od frameworka spock. Po uruchomieniu programu framework Spock wygeneruje następujący wynik [5].

```
    Condition not satisfied:

result == 5
|      |
4      false
```

Listing 3.9 Odpowiedź zwrotna po przeprowadzeniu testu w języku Groovy, za pomocą frameworka Spock.

Framework nie tylko rejestruje błąd, ale również powiadamia użytkownika dlaczego wystąpił, co jest bardzo przydatne podczas testowania.

3.8. Bloki dodatkowe

Za bloki dodatkowe w frameworku Spock, ze względu na funkcje, można uznać bloki *expect, cleanup, where* i *and.*

Blok *expect* jest bardziej ograniczonym blokiem *then*, ponieważ może zawierać tylko warunki i definicje zmiennych. Może się przydać podczas opisywania bodźca i oczekiwanej reakcji w jednym wyrażeniu, nie stosując wcześniej bloku *when* [5].

```
import spock.lang.*

class ExampleSpec extends Specification {
   // pola instancji
   // metody do przygotowania środowiska testowego
   def "przykładowy test"(){
       given:
       def a = 2
       def b = 3
       def object= new Object()
       expect:
       a + b == 5
   }
   // metody pomocnicze
}
```

Listing 3.10 Przykład użycia bloku expect w języku Groovy, frameworka Spock.

Dzięki takiej operacji został uzyskany kod semantycznie równoważny, jak kod listingu 3.8.

Blok czyszczący *cleanup*, służy do zwalniania pamięci po skończeniu danej metody funkcjonalności. Nawet jeśli testowanie zostanie przerwane, na przykład na wskutek błędu, metoda *cleanup* zawsze się wykona. Żaden inny blok, oprócz bloku *where*, nie może się znaleźć poniżej bloku *cleanup*, w tej samej metodzie funkcjonalności [5].

```
import spock.lang.*

class ExampleSpec extends Specification {
   // pola instancji
   // metody do przygotowania środowiska testowego
   def "przykładowy test"(){
       given:
       def file = new File("/przykładowa/ścieżka")
       file.createNewFile()

       cleanup:
       file.delete()
   }
   // metody pomocnicze
}
```

Listing 3.11 Przykład użycia bloku cleanup w języku Groovy za pomocą frameworka Spock[5].

Blok *where* może się pojawić tylko na końcu metody. Służy do definiowania tabeli danych wejściowych, dzięki czemu można wykonać wielokrotne wykonanie tej samej metody funkcjonalności, z różnymi zestawami danych.

```
import spock.lang.*

class ExampleSpec extends Specification {
   // pola instancji
   // metody do przygotowania środowiska testowego
   def "przykładowy test wykorzystujący blok where"(){
       when:
       def result = add(a, b) // Przykładowa funkcja
       then:
       result == expectedResult // Warunek do spełnienia

       where:
       a | b | expectedResult // Przykładowa tabela danych
wejściowych
       2 | 3 | 5
       5 | 7 | 12
       8 | 2 | 10
   }
   // metody pomocnicze
}
```

Listing 3.12 Przykład wykorzystania bloku where w języku Groovy za pomocą frameworka Spock.

W podanym przykładzie metoda „add”, zostanie wywołana 3 razy, za pomocą różnych zestawów danych. Elementy kolumny pierwszej (a), zostaną podstawione do pierwszej danej wejściowej „add”, natomiast elementy drugiej kolumny (b), za drugą daną wejściową „add”. Następnie zostanie sprawdzone, czy wynik zwrócony z „add”, równa się elementom z 3 kolumny (expectedResult). Nazwy kolumn odpowiadają zmiennym w metodzie funkcjonalności, a każdy zestaw danych uruchamia się osobno.

3.9. Warunki

Warunki są bardzo ważnym składnikiem bloków *then* i *expect,* pomagają w automatycznym zweryfikowaniu założeń i warunków w kodzie, w danym czasie działania programu. Użycie warunku wiąże się z słowem *assert,* po którym pisze się warunek do spełnienia [5].

Jeśli framework Spock wychwyci błąd po słowie *assert*, wyświetli błąd i przerwie testowanie danego bloku, bądź jeśli błąd został wychwycony w konfiguracji lub bloku *given*, cała metoda funkcjonalności zostanie przerwana [5].

3.10. Wyjątki

Wyjątki służą do przechwytywania świadomych błędów w czasie testowania, bądź sprawdzenia ich. Definiuje się je słowem kluczowym *thrown()*, przekazując do nawiasu oczekiwany typ wyjątku [5].

```
import spock.lang.*

class ExampleSpec extends Specification {
   // pola instancji
   // metody do przygotowania środowiska testowego
   def "przykładowy test"(){
       when:
       int wynik = 1 / 0

       then:
       thrown(ArithmeticException)
   }
   // metody pomocnicze
}
```

Listing 3.13 Przykład wychwycenia błędu metodą thrown() w frameworku Spock.

W bloku *then* sprawdzane jest, czy błąd „AritheticException” (dzielenie przez 0) został zgłoszony w bloku *when* i kontynuuje dalsze testowanie. Jeśli błąd nie został wychwycony w metodzie *thrown*, test zakończy się niepowodzeniem [5].

Odwrotną metodą, metody *thrown()*, jest *notThrown(),* która przerwie testowanie, gdy zostanie wychwycony podany błąd. Stosuje się ją w podobny sposób co metodę *thrown* [5].

3.11. Interakcje

Testowanie oparte na interakcji opisuje sposób, w jaki obiekty komunikują się między sobą. Jego celem, jest sprawdzanie stanu obiektów, po interakcji.

```
import spock.lang.*

class ExampleSpec extends Specification {
   // pola instancji
   // metody do przygotowania środowiska testowego
   def "przykładowy test"(){
       given:
       def mockObject = Mock(Calculator)  // Przykładowy obiekt
"mockObject" klasy "Class"

       when:
       def wynik = mockObject.add(2, 3) // Przykładowa funkcja add

       then:
       1 * mockObject.add(2, 3) >> 5  // Sprawdzane jest czy w
bloku when dana funkcja została wywołana dokładne raz, z dokładnie
podanymi danymi i czy wynik się zgadza
   }
   // metody pomocnicze
}
```

Listing 3.14 Przykład interakcji w języku Groovy za pomocą frameworka Spock.

W podanym listingu testowanie zostanie zakończone pomyślnie, ponieważ metoda „add”, została wywołana dokładnie raz, z dokładnie elementami 2 i 3, a jej wynik będzie się równał w założeniu 5.

3.12. Rozszerzenia

Framework Spock posiada 4 wbudowane rozszerzenia (Tabela 3.3), które są oparte na przechwytywaniu. Aktywowane są za pomocą adnotacji, zwanymi **dyrektywami**. Oprócz dyrektywy *@Timeout*, mogą być stosowane tylko przed ciałem metody funkcjonalności [5].

Nazwa dyrektywy	Opis działania dyrektywy
`@Timeout`	Ustawia limit czasu dla wykonania metody funkcjonalności. Może być użyte przed całą specyfikacją.
`@Ignore`	Ignoruje każdą metodę funkcjonalności, z tą dyrektywą.
`@IgnoreRest`	Ignoruje wszystkie inne metody funkcjonalności, oprócz tej, która jest oznaczoną tą dyrektywą.
`@FailsWith()`	Oczekuje wystąpienia podanego błędu w nawiasie w danej metodzie funkcjonalności.

Tabela 3.3 Wbudowane rozszerzenia frameworka Spock[5].

3.13. Specyfikacja jako dokumentacja

Framework Spock oferuje dynamiczne pisanie dokumentacji, pozwala on na pisanie tekstu słownego, po każdym bloku, na przykład *when* [5].

```
when: "udokumentowany blok when, wraz z opisem działania"
```

Listing 3.15 Przykład dynamicznego pisania dokumentacji w bloku, za pomocą frameworka Spock.

Spock umożliwia również dodawanie etykiet za pomocą specjalnego bloku *and*. Nie wpływa on na działanie innych bloków, nie przerywa ich działania [5].

```
and: "udokumentowany blok and, wraz z opisem sytuacji zaistniałej w
kodzie"
```

Listing 3.16 Przykład dynamicznego pisania dokumentacji w bloku and, za pomocą frameworka Spock.

Za pomocą biblioteki „spock-reports”, możliwe jest wygenerowanie raportu HTML, wraz z wynikami testów Spock i komentarzami zapisanymi w specyfikacjach [5].

4. Testowanie

Testowanie oprogramowania należy do dziedziny inżynierii systemów, jest to projektowanie i urzeczywistnianie specjalnego rodzaju oprogramowania systemowego, który wypróbuje inny system, w taki sposób, aby znaleźć błędy [2].

4.1. Piramida testów

Piramida testów to koncepcja optymalizacji czasu wykonywania się testów jako całości, określając ilość potrzebnych testów oraz ich koszt w zależności od ich poziomu. Koszt testu oznacza czas wykonywania testu oraz ilość kodu potrzebnego na implementacje testu. Wyróżnia się 3 poziomy w piramidzie testów, są to [1] :

- testy akceptacyjne,
- testy integracyjne,
- testy jednostkowe.

Testy jednostkowe, są rodzajem testów nastawione na sprawdzanie poprawności poszczególnych klas, metod, obiektów. Ukierunkowane są na szybkie działanie, a ich koszt powinien być najmniejszy w porównaniu do innych rodzajów testów. Zajmują najniższe miejsce w piramidzie testów, oznacza to, że ich ilość powinna być znacznie większa od innych rodzajów testów. Związane jest to z ich małym kosztem i niskim czasem wykonywania [1].

Testy integracyjne, służą do sprawdzania stanu obiektów po interakcji z innymi obiektami bądź systemami. Badają poprawność przetwarzania danych w systemie. Koszt testów integracyjnych jest większy od testów jednostkowych. Zajmują środkowe miejsce w piramidzie testów, więc ich ilość powinna być znacząco mniejsza od testów jednostkowych [1].

Testy akceptacyjne, są skierowane na sprawdzanie poprawności systemu w czasie rzeczywistego działania programu, naśladując przy tym przyszłego użytkownika. Znajdują się na samym szczycie piramidy testów, ponieważ ich koszt jest największy spośród innych rodzajów testów. Wiąże się z tym również ich ilość, powinna być dobrze zaplanowana i ograniczona do minimum, ponieważ czas ich wykonywania jest największy [1].

4.2. F.I.R.S.T.

Fast, Independent, Repeatable, Self-validating, Timely czyli szybkie, odizolowane, powtarzalne, klarowne i punktualne. Są to główne określenia, zbioru zasad techniki pisania testów F.I.R.S.T [1].

Pierwsza zasada, szybkie, ma skłonić programistę do napisania testów, które wykonają się w jak najkrótszym czasie, tak aby zminimalizować czas testowania do minimum [1].

Druga zasada, odizolowane, ma spowodować, że testy będą niezależne od siebie. Pomimo iż uruchamiane w całych zestawach, nie powinny w maksymalnym stopniu współdziałać ze sobą w czasie testowania. Oznacza to, że każdy test powinien otrzymać własną kopię danych, odizolowanych od siebie nawzajem [1].

Trzecia zasada, powtarzalne, jest określeniem na testy, które są powtarzalne w każdych warunkach. Oznacza to iż, powinny być zależne tylko od kodu, który testują i być niezależnymi od możliwych przerw w dostępności do danych [1].

Czwarta zasada, klarowne, testy muszą dać jasną odpowiedź swojego powodzenia, bądź niepowodzenia. W przypadku niepowodzenia, mają dokładnie określić, co nie zadziałało i dlaczego[1].

Piąta zasada, punktualne, ma skłonić programistę do pisania testów równolegle z kodem programu i zminimalizowanie do minimum, pisania testów, po wdrożeniu funkcjonalności [1].

4.3. TDD

TDD (Test Driven Development), jest metodą programowania, w której testy piszę się przed kodem, który mają testować. Metoda ta ma zmusić programistę, jeszcze przed napisaniem kodu, do zastanowienia się, co ma dokładnie zrobić. Dodatkowo zapewnia koncentrację na wymaganiach i na implementowaniu tylko jednej rzeczy naraz. Sama procedura TDD, jest podzielona na 3 wielokrotnie powtarzane kroki, zwanymi potocznie *Czerwony - Zielony - Refaktoryzacja* [3]:

1. Napisz test dla kolejnej funkcjonalności, którą chcesz dodać (Czerwony).
2. Napisz kod funkcjonalny, aż test zakończy się pomyślnie (Zielony).
3. Popraw zarówno nowy, jak i stary kod, aby uzyskać dobrą strukturę (Refaktoryzacja).

5. Praktyczne zastosowanie Spock Framework

Część praktyczna projektu inżynierskiego składa się z dwóch, niezależnych od siebie aplikacji, napisanych w języku Groovy. Ich celem jest zaprezentowanie możliwości frameworka Spock w kontekście testowania.

5.1. Wykorzystane technologie

Przy tworzeniu aplikacji zostały wykorzystane następujące technologie :

- IntelliJ IDEA,
- Gradle,
- Maven Central,
- Groovy,
- Spock,
- Spring Boot.

5.2. System zarządzania studentami

Celem Systemu zarządzania studentami, jest utworzenie i obsługa bazy danych na serwerze API. W bazie danych znajdują się 2 tabele, “Students” oraz “Grades”. Celem aplikacji jest dodawanie, edytowanie, bądź usuwanie studentów oraz ocen do bazy danych. Na aplikację System zarządzania studentami, składa się 11 klas :

- “Application” - klasa wykorzystująca SpringBootAplication, której zadaniem jest uruchomienie i wyłączenie serwera.
- “DatabaseStatusController” - klasa mapująca odpowiedź serwera na podanym adresie.
- “HealthCheckController” - klasa mapująca odpowiedź serwera na podanym adresie.
- “Grades” - klasa tworząca tabele “Grades” w bazie danych.
- “GradesRepository” - interfejs, którego celem jest bezpośrednia komunikacja z bazą danych z tabelą “Grades”.
- “GradesService” - klasa zawierająca metody, których celem jest określone działanie na danych w tabeli “Grades” w bazie danych.
- “GradesController” - klasa mapująca wywoływanie metod z klasy “GradesService”.
- “Students” - klasa tworząca tabele “Students” w bazie danych.
- “StudentsRepository” - interfejs, którego celem jest bezpośrednia komunikacja z bazą danych z tabelą “Students”.
- “StudentsService” - klasa zawierająca metody, których celem jest określone działanie na danych w tabeli “Students” w bazie danych.

●“StudentsController” - klasa mapująca wywoływanie metod z klasy “StudentsService”.

Oraz 2 klasy testowe :

●“ApplicationSpec” - klasa testowa, której celem jest zbadanie poprawności uruchomienia się serwera Spring Boot oraz poprawność odpowiedzi na podanych portach.

●“StudentsGradesSpec” - klasa testowa, której celem jest zbadanie poprawności obsługi bazy danych na serwerze Spring Boot.

5.2.1. Klasa StudentsService

Do klasy określającej tabelę “Students”, została dodana adnotacja @OneToMany, której celem jest połączenie z tabelą “Grades” z adnotacją @ManyToOne. Oznacza to, że do każdego studenta, można dodać wiele ocen, ale do każdej oceny tylko jednego studenta.

W klasie “StudentsService”, znajduje się 5 metod podstawowych, dodawanie, znajdowanie, zapisywanie, usuwanie studenta z tabeli Students.

Pierwsza z nich, “saveStudent”, zapisuje studenta w tabeli Students:

```
def saveStudent(Students student) {
    studentsRepository.save(student)
}
```

Listing 5.1 Metoda “saveStudents” w klasie “StudentsService”.

Druga z nich, “addNewStudent”, dodaje studenta do tabeli, za pomocą metody “saveStudents”:

```
def addNewStudent(String name) {
    def student = new Students(name: name)
    saveStudent(student)
}
```

Listing 5.2 Metoda “addNewStudent” w klasie “StudentsService”.

Trzecia, “getStudentById”, wyszukuje i zwraca danego studenta pod wskazanym id:

```
def getStudentById(Long id) {
    studentsRepository.findById(id).orElse(null)
}
```

Listing 5.3 Metoda “getStudentById” w klasie “StudentsService”.

Czwarta i piąta, “deleteStudentByIds” oraz “deleteAllStudent”, usuwa studenta pod wskazanym id lub wszystkich studentów:

```
def deleteStudentById(Long StudentId){
   studentsRepository.findById(StudentId)ifPresent {
       studentsRepository.delete(it)
   }
}
@Transactional
def deleteAllStudents() {
   studentsRepository.deleteAll();
}
```

Listing 5.4 Metoda “eleteStudentById” i “deleteAllStudentsd” w klasie “StudentsService”.

5.2.2. Klasa GradesService

Na klasę “GradesService”, składa się z 6 metod podstawowych o tym samym działaniu jak w klasie “StudentsService” tzn. dodawanie, znajdowanie, zapisywanie, usuwanie oceny w tabeli Grades . Ze względu na funkcje, to właśnie w tej klasie, dodatkowo przeprowadzane są operacje na ocenach. Osiem dodatkowych metod znajduje się w klasie “GradesService”.

Pierwsza z nich, “getGradesByStudentAndSubject”, metoda zwraca obiekty tabeli Grades, które są przypisane do danego studenta do danego przedmiotu:

```
def getGradesByStudentAndSubject(Students student, String subject)
{
   gradesRepository.findAllByStudentAndNameOfSubject(student,
subject)
}
```

Listing 5.5 Metoda “getGradesByStudentAndSubject” w klasie “GradesService”.

Druga metoda, “getGradesByStudent”, zwraca obiekty z tabeli “Grades”, które są przypisane do danego studenta.

```
def getGradesByStudent(Students student) {
   gradesRepository.findAllByStudent(student)
}
```

Listing 5.6 Metoda “getGradesByStudent” w klasie “GradesService”.

Trzecia, “calculateAverageGradeForSubject”, oblicza średnią arytmetyczną, dla danego studenta, dla danego przedmiotu i zwraca wynik :

```
def calculateAverageGradeForSubject(Long studentId, String subject)
{
    def student =studentsRepository.findById(studentId).orElse(null)
    if (student) {
        def grades = getGradesByStudentAndSubject(student, subject)
        if (grades) {
            def sum = grades.collect { it.grade }.sum()
            return sum / grades.size()
        }
    }
    return null
}
```

Listing 5.7 Metoda “calculateAverageGradeForSubject” w klasie “GradesService”.

Czwarta z nich, “calculateWeightedAverageGradeForSubject”, oblicza średnią ważoną, dla danego studenta oraz przedmiotu:

```
def calculateWeightedAverageGradeForSubject(Long studentId, String
subject) {
    def student =
studentsRepository.findById(studentId).orElse(null)
    if (student) {
        def grades = getGradesByStudentAndSubject(student, subject)
        if (grades) {
            def weightedSum = grades.collect { it.grade * it.weight
}.sum()
            def totalWeight = grades.collect { it.weight }.sum()
            return weightedSum / totalWeight
        }
    }
    return null
}
```

Listing 5.8 Metoda “calculateWeightedAverageGradeForSubject” w klasie “GradesService”.

W piątej metodzie, “alculateMedianGradeForSubject”, obliczana jest mediana dla danego studenta oraz przedmiotu:

```
def calculateMedianGradeForSubject(Long studentId, String subject)
{
   def student =studentsRepository.findById(studentId).orElse(null)
   if (student) {
       def grades = getGradesByStudentAndSubject(student, subject)
       if (grades) {
           def sortedGrades = grades.sort { it.grade }
           def size = sortedGrades.size()

           if (size % 2 == 0) {
               def middle1 = sortedGrades[size / 2 - 1].grade
               def middle2 = sortedGrades[size / 2].grade
               return (middle1 + middle2) / 2
           } else {
               return sortedGrades[size / 2].grade
           }
       }
   }
   return null
}
```

Listing 5.9 Metoda “calculateWeightedAverageGradeForSubject” w klasie “GradesService”.

Trzy ostatnie metody, obliczaja średnią arytmetyczną, ważona i medianę, dla wszystkich przedmiotów, dla danego studenta.

5.2.3. Klasy testowe

Na metody przygotowujące środowisko testowe w obydwu klasach testowych, składa się metoda setupSpec() oraz cleanupSpec(). W przypadku klas testowych “ApplicationSpec” oraz “StudentsGradesSpec”, ich zadaniem jest uruchomienie serwera Spring Boot przed pierwszym testem w specyfikacji oraz wyłączenie serwera po wykonaniu się ostatniego testu w specyfikacji.

```
def setupSpec() {
   and:"Inicjalizacja aplikacji Spring Boot przed 1 testem"
   App = new Application()
   App.main()
}

def cleanupSpec() {
   and:"Zamykanie serwera po zakończeniu ostatniego testu"
   App.mainStop()
}
```

Listing 5.10 Metody przygotowujące środowisko testowe w specyfikacjach aplikacji “System zarządzania studentami”.

Pierwsza specyfikacja, “ApplicationSpec”, składa się z dwóch metod funkcjonalności (testów), których celem jest sprawdzenie poprawności odpowiedzi z serwera.

```
def "Sprawdzenie aktywności serwera"() {
   given :"Tworzenie klienta HTTP"
   HttpClient httpClient = HttpClient.newHttpClient();
   HttpRequest httpRequest = HttpRequest.newBuilder()
           .uri(URI.create("http://localhost:8080/"))
           .GET()
           .build();

   when:"wysłanie żądania GET"
   HttpResponse<String> httpResponse = httpClient.send(httpRequest,
HttpResponse.BodyHandlers.ofString());

   then:"sprawdzenie kodu statusu"
   httpResponse.statusCode()==200
   httpResponse.body()=="Spring Boot API jest aktywne"
}
def "Sprawdzenie aktywności bazy danych na serwerze"() {
   given :"Tworzenie klienta HTTP"
   HttpClient httpClient = HttpClient.newHttpClient();
```

```
HttpRequest httpRequest = HttpRequest.newBuilder()

.uri(URI.create("http://localhost:8080/database-status"))
           .GET()
           .build();

   when:"wysłanie żądania GET"
   HttpResponse<String> httpResponse = httpClient.send(httpRequest,
HttpResponse.BodyHandlers.ofString());

   then:"sprawdzenie kodu statusu"
   httpResponse.statusCode()==200
   httpResponse.body()=="Baza danych jest aktywna"
}
```

Listing 5.11 Metody funkcjonalności w klasie “ApplicationSpec”, aplikacji “System zarządzania studentami”.

Druga specyfikacja, “StudentsGradesSpec”, sprawdza poprawność dodania, edytowania bądź usuwania danych w tabelach “Students” i “Grades”. Dodatkowo sprawdzana jest poprawność wyliczania średnich arytmetycznych, ważonych lub median studentów na podstawie ocen z tabeli “Grades”. Ze względu na dużą ilość kodu, zostaną pokazane w listingach tylko wybrane metody specyfikacji “StudentsGradesSpec”.

```
def "Sprawdzanie poprawności usuwania oceny"(){
   given :"Dodawanie studenta i oceny"
   def student = studentsService.addNewStudent("Jacek Kowalski")
   def add = gradesService.addNewGrade("Fizyka",5.0,student,2.0)

   when :"Usuwanie oceny"
   gradesService.deleteGradeById(add.id)

   then :"Sprawdzanie czy ocena została usunięta"
   gradesService.getGradeById(add.id)==null
}
```

Listing 5.12 Metoda funkcjonalności sprawdzająca poprawność usuwania oceny studenta w aplikacji “System zarządzania studentami”.

```
def "Sprawdzanie poprawności wyliczania średniej ważonej dla danego
studenta dla danego przedmiotu przy wielu ocenach"(){
   given :"Dodawanie studenta i oceny"
   def student = studentsService.addNewStudent("Jacek Kowalski")
   gradesService.addNewGrade("Fizyka",5.0,student,2.0)
   gradesService.addNewGrade("Matematyka",3.0,student,1.0)
   gradesService.addNewGrade("Matematyka",2.0,student,1.0)
   gradesService.addNewGrade("Matematyka",2.0,student,1.0)
   gradesService.addNewGrade("Fizyka",4.0,student,1.0)
   gradesService.addNewGrade("Fizyka",4.5,student,2.0)

   when :"Tworzenie zmiennej average do której zostanie zwrócony
wynik funkcji"
   def average =
gradesService.calculateWeightedAverageGradeForSubject(student.id,"F
izyka")

   then :"Sprawdzanie poprawności wyniku"
   average==4.6
}
```

Listing 5.13 Metoda funkcjonalności sprawdzająca poprawność wyliczania średniej ważonej w aplikacji "System zarządzania studentami".

Dzięki frameworkowi Spock, przeprowadzone testy zakończyły się pomyślnie:

```
StudentsGradesSpec > Sprawdzanie poprawnosci dodawania uzytkownikow PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci dodawania ocen PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci skladni w dodawanych ocenach PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci aktualizacji oceny PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci usuwania oceny PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci wyliczania sredniej arytmetycznej dla danego studenta dla danego przedmiotu gdy nie ma zadnej oceny PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci wyliczania sredniej arytmetycznej dla danego studenta dla danego przedmiotu przy jednej ocenie PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci wyliczania sredniej arytmetycznej dla danego studenta dla danego przedmiotu przy dwoch ocenach PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci wyliczania sredniej arytmetycznej dla danego studenta dla danego przedmiotu przy wielu ocenach PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci wyliczania sredniej wazonej dla danego studenta dla danego przedmiotu gdy nie ma zadnej oceny PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci wyliczania sredniej wazonej dla danego studenta dla danego przedmiotu przy jednej ocenie PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci wyliczania sredniej wazonej dla danego studenta dla danego przedmiotu przy dwoch ocenach PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci wyliczania sredniej wazonej dla danego studenta dla danego przedmiotu przy wielu ocenach PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci wyliczania mediany dla danego studenta dla danego przedmiotu gdy nie ma zandej oceny PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci wyliczania mediany dla danego studenta dla danego przedmiotu przy jednej ocenie PASSED
StudentsGradesSpec > Sprawdzanie poprawno◆ci wyliczania mediany dla danego studenta dla danego przedmiotu przy 3 ocenach PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci wyliczania mediany dla danego studenta dla danego przedmiotu przy wielu ocenach PASSED
StudentsGradesSpec > Sprawdzanie poprawnosci wyliczania wszystkich srednich dla danego studenta dla wszystkich przedmiotow PASSED
```

Listing 5.14 Rezultat testów, klasy "StudentsGrades", aplikacji "System zarządzania studentami".

5.3. Kalkulator matematyczny odwrotnej notacji polskiej

Aplikacja “Kalkulator ONP” (odwrotna notacja polska), składa się z jednej klasy “RPN”, w której znajduje się algorytm do obliczania wyniku wyrażenia zapisanego w ONP, do której dołączona jest jedna klasa testowa, “CalcSpec”.

5.2.1. Klasa RPN

Klasa “RPN” składa się z 3 metod, “parseExpression”, “compute” oraz “main”.

Do metody “main”, wprowadzany jest ciąg znaków, jako równanie ONP, następnie za pomocą metody “parseExpression” algorytm oblicza wynik równania ONP i zwraca wynik:

```
private static String parseExpression(String expr) {
   def parsedExpr = expr.replaceAll("[^\\^\\*\\+\\-\\d/\\s]", "")
   def trimmedExpr = parsedExpr.replaceAll("\\s+", " ")
   return trimmedExpr
}
static Double compute(String expr) {
   def validExpr = parseExpression(expr)
   def stack = []

   validExpr.split("\\s").each { token ->
       def secondOperand = 0.0
       def firstOperand = 0.0

       switch (token) {
           case "+":
               secondOperand = stack.pop()
               firstOperand = stack.pop()
               stack.push(firstOperand + secondOperand)
               break
           case "-":
               secondOperand = stack.pop()
               firstOperand = stack.pop()
               stack.push(firstOperand - secondOperand)
               break
           case "*":
               secondOperand = stack.pop()
               firstOperand = stack.pop()
               stack.push(firstOperand * secondOperand)
               break
           case "/":
               secondOperand = stack.pop()
               firstOperand = stack.pop()
```

```
                if (secondOperand == 0.0) {
                    throw new ArithmeticException("Nie można dzielić przez
0!")
                }

                stack.push(firstOperand / secondOperand)
                break
            case "^":
                secondOperand = stack.pop()
                firstOperand = stack.pop()
                stack.push(Math.pow(firstOperand, secondOperand))
                break
            default:
                stack.push(token.toDouble())
                break
        }
    }
    def result = Double.parseDouble(stack.pop().toString())
    return result
}
```

Listing 5.15 Metoda “parseExpression” i “compute” w klasie “RPN”, aplikacji “Kalkulator matematyczny odwrotnej notacji polskiej”.

5.2.2. Klasa testowa

Specyfikacja “CalcSpec”, składa się z czterech metod funkcjonalności. Pierwsza z nich sprawdza poprawność matematyczną otrzymanego wyniku, po wprowadzeniu równania, a trzy następne metody funkcjonalności, sprawdzają przypadki błędnego wprowadzenia wyrażenia ONP :

```
package com.example
import spock.lang.*
/**
* Klasa testów Spock, sprawdzająca popranowść działania kalkulatora ONP
*/
class CalcSpec extends Specification {
    def "Test poprawności wyników operacji arytmetycznych"() {
        expect: "Deklaracja równania i sprawdzanie oczekiwanego wyniku"
        RPN.compute(input) == expectedResult

        where: "Implementacja tabeli danych z wykorzystaniem bloku where"
        input       | expectedResult
        "2 3 +"     | 5.0
        "5 3 -"     | 2.0
```

```
        "4 5 *"      | 20.0
        "10 2 /"     | 5.0
        "2 3 ^"      | 8.0
        "3 4 2 * +" | 11.0
        "2 3 + "     | 5.0
    }
    def "Test dzielenia przez zero"() {
        when: "Deklaracja równania"
        def result = RPN.compute("5 0 /")

        then: "Sprawdzanie czy podany wyjątek został wychwycony"
        thrown(ArithmeticException)
    }
    def "Test niepoprawnego wyrażenia - za mało operandów"() {
        when: "Deklaracja równania"
        def result = RPN.compute("2 3 + +")

        then: "Sprawdzanie czy podany wyjątek został wychwycony"
        thrown(NoSuchElementException)
    }
    def "Test niepoprawnego wyrażenia - puste wyrażenie"() {
        when: "Deklaracja równania"
        def result = RPN.compute("")

        then: "Sprawdzanie czy podany wyjątek został wychwycony"
        thrown(NumberFormatException)
    }
}
```

Listing 5.16 Klasa “CalcSpec”, aplikacji “Kalkulator matematyczny odwrotnej notacji polskiej”.

Za pomocą frameworka Spock, przeprowadzenie testów zakończyło się sukcesem.

```
CalcSpec > Test poprawnosci wynikow operacji arytmetycznych > Test poprawnosci wynikow operacji arytmetycznych [input: 2 3 +, expectedResult: 5.0, #0] PASSED
CalcSpec > Test poprawnosci wynikow operacji arytmetycznych > Test poprawnosci wynikow operacji arytmetycznych [input: 5 3 -, expectedResult: 2.0, #1] PASSED
CalcSpec > Test poprawnosci wynikow operacji arytmetycznych > Test poprawnosci wynikow operacji arytmetycznych [input: 4 5 *, expectedResult: 20.0, #2] PASSED
CalcSpec > Test poprawnosci wynikow operacji arytmetycznych > Test poprawnosci wynikow operacji arytmetycznych [input: 10 2 /, expectedResult: 5.0, #3] PASSED
CalcSpec > Test poprawnosci wynikow operacji arytmetycznych > Test poprawnosci wynikow operacji arytmetycznych [input: 2 3 ^, expectedResult: 8.0, #4] PASSED
CalcSpec > Test poprawnosci wynikow operacji arytmetycznych > Test poprawnosci wynikow operacji arytmetycznych [input: 3 4 2 * +, expectedResult: 11.0, #5] PASSED
CalcSpec > Test poprawnosci wynikow operacji arytmetycznych > Test poprawnosci wynikow operacji arytmetycznych [input: 2 3 + , expectedResult: 5.0, #6] PASSED
CalcSpec > Test dzielenia przez zero PASSED
CalcSpec > Test niepoprawnego wyrazenia - za malo operandow PASSED
CalcSpec > Test niepoprawnego wyrazenia - puste wyrazenie PASSED
CalcSpec > Test pojedynczego rownania PASSED
```

Listing 5.17 Rezultat testów klasy “CalcSpec”, aplikacji “Kalkulator matematyczny odwrotnej notacji polskiej”.

6. Podsumowanie i wnioski

Celem pracy dyplomowej było przeprowadzenie kompleksowej analizy frameworka Spock oraz zbadanie jego skuteczności i efektywności w testowaniu kodu. Postawione cele i założenia zostały w pełni zrealizowane. Język Groovy, który obsługuje framework Spock, został dokładnie opisany w kontekście testowania, natomiast sam framework skrupulatnie przeanalizowany według funkcjonalności. Dzięki opisaniu różnych rodzajów testowania w dziedzinie informatyki osiągnięto dokładny obraz możliwości frameworka Spock w zależności od podejścia do testowania. Zaplanowane aplikacje zostały w pełni zrealizowane w języku Groovy i przetestowane zgodnie z zasadami testowania frameworka Spock i jego możliwości.

Zrealizowane aplikacje sprostały następującym opisanym funkcjonalnościom frameworka spock. Każda aplikacja wykorzystywała framework Spock przy przeprowadzeniu testów oraz zasady tworzenia specyfikacji Spock. Obie aplikacje wykorzystywały pola do deklaracji obiektów oraz korzystały z metod przygotowujących środowisko testowe, jak i metody funkcjonalności. Aplikacje zawierały bloki podstawowe, a aplikacja Kalkulator, również bloki dodatkowe. Każda aplikacja korzystała z warunków, natomiast w aplikacji Kalkulator, także z wyjątków. We wszystkich aplikacjach została zastosowana specyfikacja jako dokumentacja.

Omówiona praca pozwoliła na ugruntowanie wiedzy w trakcie studiów z wielu zakresów, w tym przede wszystkim z zakresu tworzenia testów w języku Groovy za pomocą Spock oraz rozszerzenie wiedzy o testowaniu w informatyce, co jest bardzo kluczowe na rynku pracy informatyka.

Bibliografia

[1] Ron Patton. *Testowanie oprogramowania.* 2002

[2] Robert V. Binder. *Testowanie systemów obiektowych : modele, wzorce i narzędzia.* 2003[3] Viktor Farcic. *TDD : programowanie w Javie sterowane testami : naucz się podstaw metodyki TDD* . 2015

[4] Dierk König, Paul King, Hamlet D'Arcy, Cédric Champeau. *Groovy in Action Second Edition.* 2015

[5] *Spock Framework Reference Documentation* [online] https://spockframework.org/spock/docs/2.3/index.html (15.10.2023)

[6] *Apache Groovy Documentation* [online] https://groovy-lang.org/documentation.html (15.10.2023)

[7] I*ntelij IDEA Documentation* [online] https://www.jetbrains.com/help/idea/getting-started.html (15.10.2023)

[8] *Introduction to Testing with Spock and Groovy* [online] https://www.baeldung.com/groovy-spock (15.10.2023)

[9] *Ant from Gradle* [online] https://docs.gradle.org/current/userguide/ant.html(15.10.2023)

[10] *Custom vs Configured vs Out-of-the-Box Solutions*[online] https://getnimblex.com/custom-vs-configured-vs-out-of-the-box-solutions/(15.10.2023)

[11] Marcin Lis. *Java : praktyczny kurs.* 2015

[12] *Zmienne - JavaStrat*[online] https://javastart.pl/baza-wiedzy/java-podstawy-jezyka/zmienne#zmienne-finalne(15.10.2023)

[13] *Operatory logiczne*[online] https://stormit.pl/operatory-logiczne/(15.10.2023)

[14] *Java Switch Case:The Improved Version*[online] https://dev.to/abh1navv/java-switch-case-the-improved-version-1012(15.10.2023)

[15] *The For-Each Loop*[online] https://docs.oracle.com/javase/8/docs/technotes/guides/language/foreach.html(15.10.2023)

[16] *Kolekcje w języku java*[online] https://www.samouczekprogramisty.pl/kolekcje-w-jezyku-java/(15.10.2023)

[17] *Kolekcje(Listy)*[online] https://javappa.com/kurs-java/kolekcje-listy(15.10.2023)

[18] *Struktury danych*[online]
https://javastart.pl/baza-wiedzy/algorytmy/struktury-danych-mapa(15.10.2023)

[19] *Groovy - Ranges*[online]
https://www.tutorialspoint.com/groovy/groovy_ranges.html(15.10.2023)

[20] *Interfejsy - JavaStart*[online]
https://javastart.pl/baza-wiedzy/programowanie-obiektowe/interfejsy(15.10.2023)

Spis tabel

Spis listingów