

Radosław Sokół



Testowanie aplikacji Java za pomocą

JUnit

Helion 

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Grafika na okładce została wykorzystana za zgodą Shutterstock.com

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/teapja>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:

<ftp://ftp.helion.pl/przyklady/teapja.zip>

ISBN: 978-83-283-3828-9

Copyright © Helion 2018

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wstęp	7
Rozdział 1. Testy jednostkowe	11
1.1. Pierwszy przykład	12
1.2. Ścieżki kodu	13
1.3. Testy jednostkowe	17
1.3.1. Wykorzystanie biblioteki JUnit	18
1.3.2. Klasy testów	18
1.3.3. Szkielet klasy testu	18
1.3.4. Pierwsze testy	20
1.3.5. Uruchomienie testów	22
1.3.6. Raport z testów	24
1.3.7. Spodziewanie się sytuacji wyjątkowej	25
1.3.8. Cykl życia klasy testów	29
1.3.9. Inne założenia testowe	34
1.3.10. Ignorowanie testów	35
1.3.11. Pokrycie testami	36
1.4. Refaktoryzacja	38
1.4.1. Klasa poddawana refaktoryzacji	39
1.4.2. Testy	40
1.4.3. Klasa po poprawkach	45
1.4.4. Refaktoryzacja	48
1.5. Podsumowanie	49
Rozdział 2. Imitacje	51
2.1. Klasy imitacji	52
2.1.1. Podstawy	53
2.1.2. Główny moduł aplikacji przed dostosowaniem do testów	57
2.1.3. Dostosowanie głównego modułu aplikacji do testów	58

2.1.4. Utworzenie imitacji klasy przechowywania danych	60
2.1.5. Testowanie głównego modułu aplikacji	61
2.1.6. Usuwanie usterek	64
2.2. Obiekty imitacji	65
2.2.1. Biblioteka Mockito	66
2.2.2. Najprostszy przykład wykorzystania	67
2.2.3. Imitacja modułu składowania danych ocen	72
2.2.4. Swobodne konfigurowanie obiektów imitacji	77
2.2.5. Zliczanie wywołań imitowanych metod	79
2.2.6. Tworzenie obiektów imitacji na bazie klas imitacji	81
2.3. Podsumowanie	84

Rozdział 3. Płynne definicje testów 87

3.1. Biblioteka AssertJ	88
3.2. Budowanie kryteriów	88
3.2.1. Klasa przykładowa	88
3.2.2. Testy jednostkowe	89
3.2.3. Kompletna treść przykładu	99
3.3. Inne kryteria	102
3.3.1. Typy skalarne	102
3.3.2. Typy tablicowe i kolekcje	105
3.4. Podsumowanie	107

Rozdział 4. Programowanie ukierunkowane na testy 109

4.1. Programowanie bazujące na kontraktach	110
4.1.1. Kontrakty wejściowe	110
4.1.2. Kontrakty wyjściowe	115
4.2. Programowanie ukierunkowane na testy	116
4.2.1. Projekt klasy	117
4.2.2. Testy jednostkowe	118
4.2.3. Uzupełnianie implementacji	121
4.2.4. Kompletna implementacja	124
4.3. Programowanie wspomagane testami	125
4.3.1. Interfejs klasy	126
4.3.2. Szkielet klasy	127
4.3.3. Implementacja i testy	128
4.4. Podsumowanie	145

Rozdział 5. Imitowanie baz danych	147
5.1. Dołączanie Derby do projektu	148
5.2. Definiowanie testowej jednostki utrwalania	149
5.3. Tworzenie zarządcy utrwalania	150
5.4. Szkielet klasy testowej	151
5.5. Przykład wykorzystania	154
5.5.1. Klasa encji zakładki strony	154
5.5.2. Klasa usługi zarządzania zakładkami	159
5.5.3. Całość tekstu źródłowego	165
5.6. Podsumowanie	169
Skorowidz	171

Rozdział 4.

Programowanie ukierunkowane na testy

Pokazywane w poprzednich rozdziałach przykłady nie prezentowały jednolitej metodologii. W niektórych przypadkach praca rozpoczynała się od zapisania testu, w innych zaś najpierw powstawał kod, a potem dopiero metoda testująca, czy ten kod działa zgodnie z oczekiwaniami.

W dużych projektach informatycznych konieczne jest usystematyzowanie podejścia do kwestii testów. Najbardziej rozpowszechniona jest **technika programowania ukierunkowanego na testy**, w skrócie TDD¹. Wiąże ona bezpośrednio projekt architektury aplikacji z jej realizacją i pozwala uzyskać implementację ściśle spełniającą wszystkie wymogi. Z drugiej strony, charakteryzuje się wysoką prędkością i wcale nie musi skutkować wysokim pokryciem testami krytycznych modułów programu. Dlatego w praktyce wprowadza się też **technikę programowania wspomaganego testami**, w skrócie TAD². Choć nie wspomaga ona tak bardzo procesu projektowego, to jednak pozwala dokładniej przetestować tworzoną implementację.

W ostateczności zaś jakiegokolwiek testowanie jest zawsze lepsze od braku testów. Jeśli nawet zespół nie jest w stanie z dnia na dzień wdrożyć technik TDD lub TAD, sukcesem będzie, jeżeli z poziomu braku jakichkolwiek testów przejdzie na uzupełnianie testami tworzonego oprogramowania. Oczywiście, nie oznacza to końca drogi: należy dalej przekonywać o zaletach TAD i TDD, świecić dobrym przykładem i starać się, by cały zespół rozwijał się w kierunku programowania ukierunkowanego na testy.

Zanim jednak przejdziemy do omawiania poszczególnych metodologii pisania testów, warto zapoznać się z techniką dużo łatwiejszą do wdrożenia, czyli programowaniem bazującym na kontraktach.

¹ Ang. *Test Driven Development*.

² Ang. *Test Assisted Development*.

4.1. Programowanie bazujące na kontraktach

Jednym ze sposobów zapewnienia wysokiej jakości oprogramowania jest stosowanie **techniki kontraktów**. Zakłada się w niej, że każdy podprogram opisany jest **kontraktem wejściowym** (ang. *preconditions*) oraz **kontraktem wyjściowym** (ang. *postconditions*). Kontrakt wejściowy gwarantuje, że wszystkie dane potrzebne do realizacji operacji są poprawne (należą do dziedziny funkcji), a wyjściowy — że wynik nie wychodzi poza zakres dopuszczalnych wartości (należy do przeciwdziedziny).

Największą zaletą stosowania kontraktów jest szybkie i dokładne lokalizowanie defektów oprogramowania. W tradycyjnie pisanych programach błędy pojawiają się gdzieś w głębi kodu i propagują do warstw komunikacji z użytkownikiem, gdzie mogą — ale nie muszą — zostać dostrzeżone. Jeśli nawet zostaną wykryte przez program lub zauważone przez użytkownika, często ich pierwotna przyczyna jest tak dobrze ukryta, że jej znalezienie i usunięcie może być albo bardzo czasochłonne, albo wręcz niemożliwe.

Jeżeli jednak każdy podprogram jest wyposażony w zestaw kontraktów definiujących dziedzinę i przeciwdziedzinę, jakiegokolwiek naruszenie reguł skutkuje zgłoszeniem błędu od razu. Początkowo takie podejście może sprawiać więcej problemów niż przynosić korzyści, gdyż program zgłasza dużo więcej błędów i często całkowicie przerywa swoje działanie lub przynajmniej realizację jednej z funkcji. Docelowo jednak, gdy błędy zostaną usunięte, a sytuacje wyjątkowe przechwycone i zamienione na czytelne dla człowieka komunikaty błędów, aplikacja staje się mniej wrażliwa na błędne dane oraz błędy w implementacji algorytmów.

4.1.1. Kontrakty wejściowe

Kontrakt wejściowy ma za zadanie zweryfikowanie, czy dane przekazane do podprogramu (argumenty podprogramu) są poprawne. Instrukcje wymuszające zgodność z kontraktem dają pewność, że kod podprogramu nie będzie wykonywany w przypadku, gdy dane nie należą do dziedziny. Zmniejsza to prawdopodobieństwo wystąpienia trzech bardzo istotnych problemów:

- niewykrycia istnienia błędów w pozostałych modułach programu, korzystających z danego podprogramu, które to błędy powodują przekazywanie błędnych danych do wymienionego podprogramu,
- uzyskania błędnych wyników obliczeń, błędów działania programu lub ujawnienia wrażliwych informacji w efekcie nieprzewidzianego zachowania podprogramu w obliczu niepoprawnych (przypadkowo uszkodzonych lub celowo spreparowanych) danych,
- dalszego propagowania błędu w efekcie uzyskania błędnych (a może wręcz nienależących do przeciwdziedziny) wyników, przekazywanych następnie do kolejnych podprogramów lub modułów funkcjonalnych programu.

Aby zdefiniować kontrakt wejściowy podprogramu, należy określić:

- dziedzinę, czyli możliwe wartości bezpośrednich parametrów podprogramu; wszystkie wartości wykraczające poza dziedzinę muszą powodować zgłoszenie sytuacji wyjątkowej `IllegalArgumentException`,
- dozwolony obszar stanu obiektu, czyli wartości pól obiektu, do którego klasy należy metoda; wszystkie inne wartości uniemożliwiają poprawne wykonanie metody i muszą powodować zgłoszenie sytuacji wyjątkowej `IllegalStateException`.

Kwestią dyskusyjną jest sposób reagowania na przekazywanie jako argumentów podprogramów wartości `null`. Zgodnie z jedną szkołą, predestynowana do zgłaszania pustych referencji przekazywanych jako argumenty jest klasa `NullPointerException`. Zgodnie z drugą, błędy NPE³ powinny być zarezerwowane dla nieoczekiwanych odwołań do pustych referencji, zaś efekt negatywnej weryfikacji kontraktu powinien — niezależnie od pierwotnej przyczyny niezgodności — skutkować zgłoszeniem `IllegalArgumentException`. W ramach książki będę trzymał się tej drugiej zasady, jednak czytelnicy mogą wybrać bardziej odpowiadające im rozwiązanie.

Najprostszym sposobem weryfikowania kontraktów wejściowych jest wykorzystanie instrukcji warunkowej `if` i sprawdzenie, czy kontrakt został spełniony. Listing 4.1 przedstawia klasę `Circle` opisującą okrąg (koło), w której zastosowano technikę kontraktów wejściowych do zapewnienia, by przekazywany promień nigdy nie był ujemny. Warto zauważyć, że weryfikacja kontraktu znajduje się bezpośrednio w miejscu, gdzie ustalany lub zmieniany jest stan obiektu, gdyż zgodnie z zasadami hermetyzacji obiekt nigdy nie może zostać wprowadzony w błędny stan. Sprawdzanie stanu obiektu dopiero w metodach `getRadius()`, `getArea()` i `getCircumference()` byłoby zatem błędem z dwóch powodów:

- pogorszeniu uległaby wydajność programu, gdyż te same weryfikacje musiałyby być realizowane przy każdym wywoływaniu (teoretycznie trywialnego) akcesora,
- obiekt mógłby zostać wprowadzony w niepoprawny stan, który mógłby „wyciekać” z nieodpowiednio zabezpieczonych innych metod klasy lub zostać pobrany choćby za pomocą mechanizmu refleksji.

LISTING 4.1. Klasa `Circle`

```
import java.io.Serializable;
public final class Circle implements Serializable {
    public static Circle withRadius(final double r) {
        return new Circle(r);
    }
    public double getRadius() {
        return r;
    }
}
```

³ NPE: `NullPointerException`.

```
    }  
    public double getCircumference() {  
        return 2.0 * Math.PI * r;  
    }  
    public double getArea() {  
        return Math.PI * r * r;  
    }  
    private Circle(final double r) {  
        if (r < 0.0) {  
            throw new IllegalArgumentException("Invalid radius: " + r);  
        }  
        this.r = r;  
    }  
    private final double r;  
}
```

Dodatkowo listing 4.2 przedstawia zestaw testów związanych z klasą `Circle`. Testy te nie są jednostkowe w czystej formie, gdyż każdy z nich sprawdza zarówno poprawność tworzenia obiektu, jak i wyniki zwracane przez poszczególne akcesory. Jest to jednak uzasadnione specyfiką klasy, która głównie reprezentuje podzbiór liczb zmiennoprzecinkowych z określoną funkcją (reprezentacja nieumiejscowionej w przestrzeni figury geometrycznej o określonym rozmiarze). Trzy zdefiniowane testy sprawdzają działanie całej klasy dla ujemnego, zerowego oraz dodatniego promienia, pokrywając w ten sposób obydwa zakresy parametru (niepoprawny i poprawny) oraz punkt graniczny.

LISTING 4.2. Testy jednostkowe dla klasy `Circle`

```
import org.junit.Test;  
import static org.assertj.core.api.Assertions.assertThat;  
public class CircleTest {  
    @Test(expected = IllegalArgumentException.class)  
    public void negativeRadius() {  
        Circle.withRadius(-0.1);  
    }  
    @Test  
    public void zeroRadius() {  
        final Circle circle = Circle.withRadius(0.0);  
        assertThat(circle.getRadius()).isZero();  
        assertThat(circle.getCircumference()).isZero();  
        assertThat(circle.getArea()).isZero();  
    }  
    @Test  
    public void positiveRadius() {  
        final Circle circle = Circle.withRadius(1.0);  
        assertThat(circle.getRadius()).isEqualTo(1.0);  
        assertThat(circle.getCircumference()).isEqualTo(2.0 * Math.PI);  
        assertThat(circle.getArea()).isEqualTo(Math.PI);  
    }  
}
```

Pisanie za każdym razem litanii instrukcji warunkowych `if` byłoby jednak pracochłonne i męczące, a także niepotrzebnie zwiększałoby objętość tekstu źródłowego klasy i podatność na błędy. Z tego powodu wiele bibliotek programistycznych oferuje rozwiązania systematyzujące definiowanie kontraktów wejściowych. Przykładowo biblioteka *Google Guava* zawiera klasę `Preconditions`, której dwie najważniejsze metody statyczne, w wielu przeciążonych wariantach, pozwalają sprawdzić kontrakt wejściowy zarówno na poziomie dziedziny, jak i stanu obiektu. Oto one:

- `Preconditions.checkNotNull` — sprawdza, czy spełniony jest podany warunek logiczny; w przeciwnym przypadku zgłaszana jest sytuacja wyjątkowa `IllegalArgumentException`,
- `Preconditions.checkState` — sprawdza, czy spełniony jest podany warunek logiczny; w przeciwnym przypadku zgłaszana jest sytuacja wyjątkowa `IllegalStateException`.

Dodatkowo dla specyficznych zastosowań, w których należy zgłaszać innego typu sytuacje wyjątkowe, dostępne są trzy inne metody statyczne:

- `Preconditions.checkElementIndex` — sprawdza, czy podany indeks jest **mniej** od określonej wartości granicznej; w przeciwnym przypadku zgłaszana jest sytuacja wyjątkowa `IndexOutOfBoundsException`,
- `Preconditions.checkPositionIndex` — sprawdza, czy podany indeks jest **mniej lub równy** określonej wartości granicznej; w przeciwnym przypadku zgłaszana jest sytuacja wyjątkowa `IndexOutOfBoundsException`,
- `Preconditions.checkNotNull` — sprawdza, czy podana referencja jest różna od `null`; w przeciwnym przypadku zgłaszana jest sytuacja wyjątkowa `NullPointerException`.

Każda z tych metod przyjmuje dodatkowo opcjonalny parametr tekstowy lub obiektowy, którego treść jest wykorzystywana przy konstruowaniu opisu sytuacji wyjątkowej w przypadku wykrycia naruszenia kontraktu. Pozwala to znacząco zwiększyć czytelność pojawiających się w czasie działania aplikacji błędów oraz skrócić czas poszukiwania przyczyny błędów.

Tekst konstruktora klasy `Circle`, zapisany z wykorzystaniem klasy `Preconditions`, wyglądałby następująco:

```
private Circle(final double r) {
    Preconditions.checkArgument(r >= 0.0, "Invalid radius");
    this.r = r;
}
```

Najbardziej rzuca się w oczy krótszy i prostszy zapis. Zamiast uniwersalnej instrukcji `if`, którą łatwo pomylić z główną logiką programu, wykorzystujemy specyficzny zapis, łatwo przekładający się na naturalne zdania angielskie. Większy nacisk jest też położony na

zapis **dziedziny** podprogramu, a nie niepoprawnego obszaru parametrów: zamiast „negatywnego” wyrażenia $r < 0.0$ stosujemy „pozytywny” zapis $r \geq 0.0$, jasno wskazujący na to, jakie argumenty będą uznane za poprawne.

Istotną zmianą jest rezygnacja z podawania wartości, która doprowadziła do pojawienia się błędu. Jest to spowodowane odmiennym prowadzeniem sterowania w obydwu przypadkach. W przypadku stosowania instrukcji `if` ścieżka błędu powodująca zgłoszenie sytuacji wyjątkowej jest wykonywana jedynie wtedy, gdy zachodzi odpowiedni warunek. Możemy sobie zatem pozwolić na łączenie tekstu z błędną wartością. W drugim przypadku argument metody `Preconditions.checkArgument()` stanowiący komunikat błędu jest wyliczany **zawsze**, również wtedy, kiedy nie zachodzą odpowiednie ku temu warunki. Ponieważ konwersja liczby zmiennoprzecinkowej na tekst oraz łączenie dwóch fragmentów tekstu to operacje o wyraźnym wpływie na wydajność, nie możemy sobie pozwolić na taką rozrzutność.

Jest jednak pewne rozwiązanie. Należy zastosować wariant metody, który pozwala podać szablon tekstu komunikatu oraz parametry, wstawiane w określone miejsca szablonu:

```
private Circle(final double r) {
    Preconditions.checkArgument(r >= 0.0, "Invalid radius: %s", r);
    this.r = r;
}
```

To rozwiązanie na pierwszy rzut oka pozwala zachować zalety wynikające z dokładności komunikatu błędu bez płacenia ceny utraty wydajności. Jednak nie jest to do końca prawdą. W języku Java metody pobierające zmienną liczbę argumentów otrzymują do przetworzenia tablicę obiektów. W tym przypadku wartość typu `double` zostanie poddana automatycznej konwersji do typu obiektowego `Double`, co wiąże się z utworzeniem nowego obiektu na stercie oraz wywołaniem konstruktora. Powstanie też w pamięci nowa tablica jednoelementowa. Jest to oczywiście dużo mniejszy koszt niż w przypadku każdorazowej konwersji i łączenia napisów, jednak w oprogramowaniu pisanym z myślą o maksymalnej wydajności może być nie do pominięcia.



Technika zwiększania szczegółowości komunikatów błędów sprawdza się dużo lepiej w przypadku, gdy danymi szczegółowymi są obiekty, do których referencje można przekazać bez konieczności tworzenia nowych obiektów. Należy jej unikać, jeżeli danymi są wartości typów prostych lub należy zachować najwyższą wydajność. W takim przypadku trzeba albo używać wariantów metod klasy `Preconditions`, które zwracają stały komunikat błędu bez informacji szczegółowych, albo zastosować tradycyjne podejście wykorzystujące instrukcję warunkową `if`.

Skorowidz

A

adnotacja, 17
 @Before, 69
 @Rule, 151
 @Test, 18
 @VisibleForTesting, 92
Apache Derby, 148
AssertJ, 9, 87, 88
 kryteria testów, 88
 testy jednostkowe, 89

B

bazy danych, 147
biblioteka
 AssertJ, 87, 88
 EqualsVerifier, 97
 Google Guava, 113
 Guava, 115
 JUnit, 17
 Mockito, 66, 83
błąd dzielenia przez zero, 94

C

cykl życia klasy testów, 29

D

definiowanie
 testów, 87
 testowej jednostki utrwalania, 149
dostosowywanie do testów, 57
dziedziczenie, 19
dzielenie przez zero, 94

E

encje, 154
EqualsVerifier, 9

G

Google Guava, 9

I

ignorowanie testów, 35
imitacja, 51
 modułu składowania danych, 72
 NotesStorageMock, 61
imitowane podsystemy, 52
imitowanie
 baz danych, 147
 dołączanie Derby, 148
 szkielet klasy testowej, 151
 testowa jednostka utrwalania, 149
 zarządca utrwalania, 150
klas finalnych, 83
interfejs, 58
 Bookmark, 154
 klasy, 126
 NotesService, 58
 NotesStorage, 75, 82

J

jednostka utrwalania, 149
JUnit, 9, 17, 18

K

klasa

BookmarkEntity, 157, 165
 BookmarkEntityTest, 166
 BookmarkServiceImpl, 160
 BookmarkServiceImplTest, 168
 Circle, 111
 Clock, 46
 ClockTest, 41, 46
 DatabaseTestSkeleton, 153
 encji, 154
 Fraction, 89, 100
 FractionTest, 101
 InventoryImpl, 127, 141
 InventoryImplTest, 142
 Note, 53
 NotesService, 57
 NotesServiceImplTest, 61
 NotesStorageMock, 82
 Preconditions, 113
 ProductRating, 19
 ReversePhrase, 67
 ReverseWords, 67, 69

klasy

imitacji, 52
 testów, 18

kolekcje, 105

konfigurowanie obiektów imitacji, 77

kontrakty

wejściowe, 110
 wyjściowe, 115

kryteria

testów, 88
 weryfikacji, 81

M

metoda

add(), 43
 after(ms), 81
 andUnequalExample(), 98
 apply(), 80
 are(c), 105
 areAtLeast(n, c), 105
 areAtLeastOne(c), 105
 areAtMost(n, c), 105

areExactly(n, c), 105
 areNot(c), 105
 atLeast(n), 81
 atLeastOnce(), 81
 atMost(n), 81
 calls(n), 81
 contains(s), 104
 contains(x, i), 105
 contains(x, y, z...), 105
 containsAll(c), 105
 containsExactly(x, y, z...), 105
 containsExactlyElementsOf(c), 105
 containsExactlyInAnyOrder, 106
 containsIgnoringCase(s), 104
 containsNull(), 106
 containsOnly(x, y, z...), 106
 containsOnlyDigits(), 104
 containsOnlyElementsOf(c), 106
 containsOnlyOnce(s), 104
 containsOnlyOnce(x, y, z...), 106
 containsPattern(s), 104
 doesNotContain(s), 104
 doesNotContain(x, i), 106
 doesNotContain(x, y, z...), 106
 doesNotContainNull(), 106
 doesNotContainPattern(s), 104
 doesNotEndWith(s), 104
 doesNotHaveDuplicates(), 106
 doesNotMatch(s), 104
 doesNotStartWith(s), 104
 endsWith(s), 104
 endWith(x, y, z...), 106
 equals(), 95
 filteredOn(c), 106
 forRelaxedEqualExamples(), 98
 Fraction.of(), 93, 95
 getAll(), 164
 hasAtLeastOneElementOfType(c), 106
 hashCode(), 97
 hasLineCount(n), 104
 hasOnlyElementOfType(c), 106
 hasOnlyElementOfTypes(c1, c2...), 106
 hasSameSizeAs(s), 104
 hasSize(n), 104, 106
 hhasSameSizeAs(c), 106
 isBetween(a, b), 102
 isBlank(), 104

isCloseTo(a, offset), 102
 isCloseTo(a, procent), 102
 isEmpty(), 104, 106
 isEqualTo(a), 102
 isEqualTo(a, offset), 103
 isEqualToIgnoringCase(s), 104
 isEqualToIgnoringWhitespace(s), 104
 isGreaterThan(a), 102
 isGreaterThanOrEqualTo(a), 103
 isLessThan(a), 103
 isLessThanOrEqualTo(a), 103
 isNaN(), 103
 isNegative(), 103
 isNotCloseTo(a, offset), 103
 isNotCloseTo(a, procent), 103
 isNotEmpty(), 104, 106
 isNotEqualTo(a), 103
 isNotEqualToIgnoringCase(s), 104
 isNotEqualToIgnoringWhitespace(s), 104
 isNotNegative(), 103
 isNotPositive(), 103
 isNotZero(), 103
 isNullOrEmpty(), 104
 isOne(), 103
 isPositive(), 103
 isSorted(), 107
 isStrictlyBetween(a, b), 103
 isSubstringOf(s), 104
 matches(s), 105
 Mockito.doAnswer(), 74
 never(), 81
 of(), 93
 only(), 81
 Preconditions.checkArgument(), 113
 Preconditions.checkElementIndex(), 113
 Preconditions.checkNotNull(), 113
 Preconditions.checkPositionIndex(), 113
 Preconditions.checkState(), 113, 115
 remove(), 164
 spy(), 83
 startsWith(s), 105
 startsWith(x, y, z...), 107
 suppress(), 98
 timeout(ms), 81
 times(n), 81
 toString(), 99
 usingComparator(c), 105

usingElementComparator(c), 107
 usingGetClass(), 98
 verify(), 79
 withIgnoredFields(), 98
 withNonNullFields(), 98
 withOnlyTheseFields(), 98
 Mockito, 9, 66, 83
 moduł
 logiki biznesowej, 60
 składowania danych, 60, 72

O

obiekt Clock, 44
 obiekty imitacji, 52, 65
 konfigurowanie, 77
 tworzenie, 81
 odczyt bieżącego czasu, 41
 operacje
 biznesowe, 58
 na bazie danych, 57

P

płynne definicje testów, 87
 płynny interfejs programowania, 87
 porównywanie obiektów, 95
 poziom pokrycia kodu testami, 36
 programowanie
 bazujące na kontraktach, 55, 110
 ukierunkowane na testy, 109, 116
 kompletna implementacja, 124
 projekt klasy, 117
 testy jednostkowe, 118
 uzupełnianie implementacji, 121
 wspomagane testami, 125
 implementacja, 128
 interfejs klasy, 126
 szkielet klasy, 127
 testy, 128

R

refaktoryzacja, 38, 48
 refleksja, 17

S

skrót dla obiektów, 97
 skutki uboczne, 74
 słowo kluczowe
 interface, 58
 sprawdzenie działania metody, 99
 systemy CI, 147
 szkielet klasy testu, 18

Ś

ścieżki kodu, 13

T

TAD, 126
 TDD, Test-Driven Development, 116
 testowa jednostka utrwalania, 149
 testowanie, 7
 błędu dzielenia przez zero, 94
 głównego modułu, 61
 klasy Circle, 112
 klasy Fraction, 91
 klasy FractionTest, 101
 klasy Note, 54
 klasy ReverseWordsTest, 69
 testy, 40
 cykl życia, 29
 jednostkowe, 11, 17, 89
 ignorowanie, 35
 poziom pokrycia, 36
 raporty, 24
 rozmyte, 37
 sytuacje wyjątkowe, 25
 ścieżki kodu, 13
 uruchamianie, 22
 wartość rzeczywista, 21
 wartość spodziewana, 21
 zakres dziedzinowy, 15
 zakres implementacyjny, 15
 założenia, 34

testReversePhrase(), 80
 TFD, Tests First Development, 117
 tworzenie
 imitacji klasy, 60
 obiektów imitacji, 81
 zarządcy utrwalania, 150
 typy
 skalarne, 102
 tablicowe, 105

U

ułamek wymierny, 89
 usuwanie usterek, 64

W

wartość
 rzeczywista, 21
 spodziewana, 21
 weryfikacja, 81
 wydajność, 60
 wyjątek
 DuplicateBookmarkException, 160
 IllegalArgumentException, 41, 55
 IllegalStateException, 113
 UnsupportedOperationException, 79, 127
 wzorzec projektowy Singleton, 57

Z

zakładki strony, 154
 zależności, 51
 założenia testowe, 34
 zarządca utrwalania, 150
 zarządzanie zakładkami, 159
 zasada pojedynczej odpowiedzialności, 58
 zliczanie wywołań imitowanych metod, 79
 złożenie, 19

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

Testuj swój kod profesjonalnie — pisz testy jednostkowe!

- **Poznaj niezbędne narzędzia**
- **Twórz testy jednostkowe**
- **Korzystaj z technik TDD i TAD**

Jednym z największych wyzwań współczesnej inżynierii oprogramowania jest zapewnienie właściwej jakości tworzonych produktów, co można osiągnąć w dużej mierze dzięki odpowiedniej weryfikacji kodu. Odpowiedzią na potrzeby branży są testy automatyczne, a w szczególności — testy jednostkowe. Nie tylko umożliwiają one ciągłe, bezproblemowe i szybkie sprawdzanie poprawności pisanego kodu, lecz również pozwalają wyeliminować błędy wprowadzane w czasie rozwoju programu. Stanowią także dokumentację zastosowanej architektury oraz sposobu jej implementacji.

Jeśli nie chcesz zostać w tyle, dołącz do tych, którzy piszą testy jednostkowe do swoich programów tworzonych w języku Java. Pomoże Ci w tym książka prezentująca koncepcję UT, bibliotekę JUnit, rozszerzenie AssertJ oraz biblioteki EqualsVerifier, Mockito i Google Guava. Poznasz dzięki niej sposoby pisania testów, refaktoryzowania kodu oraz imitowania zarówno pojedynczych modułów funkcjonalnych, jak i całych baz danych. Dowiesz się też, jak w praktyce stosować metodyki TDD i TAD. A wszystkiego nauczysz się dzięki wziętym z życia przykładom oraz zadaniom do samodzielnego wykonania.

- **Tworzenie testów jednostkowych za pomocą biblioteki JUnit**
- **Ścieżki kodu, pokrycie testami i refaktoryzacja kodu**
- **Unikanie zależności za pomocą imitacji i zastosowanie Mockito**
- **Płynne definiowanie testów za pomocą biblioteki AssertJ**
- **Programowanie ukierunkowane na testy i wspomagane testami**
- **Imitowanie baz danych przy użyciu rozwiązania Apache Derby**

Popraw jakość swoich programów — skorzystaj z testów jednostkowych!

	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI <i>Sięgnij po więcej! ▶</i>	
 helion.pl		ISBN 978-83-283-3828-9	
 0 801 339900	AKADEMIA IT & BUSINESS		
 0 601 339900	WWW.SZKOLENIA.HELION.PL	9 788328 338289	
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 39,00 zł	