Radosław Sokół

Testowanie aplikacji Java za pomocą JUnit

Helion

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

Printed in Poland.

Spis treści

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 TDD[1]. Wiąże ona bezpośrednio projekt architektury aplikacji z jej realizacją i pozwala uzyskać implementację ściśle spełniającą wszystkie wymogi. Z drugiej strony, charakteryzuje się wysoką pracochłonnością i wcale nie musi skutkować wysokim pokryciem testami krytycznych modułów programu. Dlatego w praktyce wprowadza się też **technikę programowania wspomaganego testami**, w skrócie TAD[2]. Choć nie wspomaga ona tak bardzo procesu projektowego, to jednak pozwala dokładniej przetestować tworzoną implementację.

W ostateczności zaś jakiekolwiek 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, jakiekolwiek 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 NPE[3] 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.checkArgument` — 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 **mniejszy** od określonej wartości granicznej; w przeciwnym przypadku zgłaszana jest sytuacja wyjątkowa `IndexOutOfBoundsException`,
- `Preconditions.checkPositionIndex` — sprawdza, czy podany indeks jest **mniejszy 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 >= 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

K

M

O

P

R

S

Ś

T

U

W

Z

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!

Helion

helion.pl

0 801 339900

0 601 339900

ISBN 978-83-283-3828-9

9 788328 338289

Cena: 39,00 zł