JUnit. Pragmatyczne testy jednostkowe w Javie

Autorzy: Andy Hunt, Dave Thomas
Tłumaczenie: Jaromir Senczyk
ISBN: 83-246-0406-5
Tytuł oryginału: Pragmatic Unit Testing in Java with JUnit
Format: B5, stron: 192

Przetestuj swoje aplikacje podczas ich tworzenia

- Poznaj strukturę testów jednostkowych
- Stwórz poprawne testy jednostkowe
- Wykorzystaj moduły testowe w projekcie

Testy jednostkowe są niezwykle ważnym narzędziem programisty. Przeprowadzane podczas pisania aplikacji pozwalają na sprawdzenie poprawności kodu, wyłapanie błędów i szybkie usunięcie ich. W nowoczesnych metodykach wytwarzania oprogramowania testy jednostkowe są jednymi z najważniejszych elementów procesu. Tworzenie systemów bez korzystania z testów jednostkowych często porównywane jest do pisania programów na kartce. Nowoczesne narzędzia takie, jak JUnit bardzo ułatwiają przeprowadzanie testów jednostkowych, integrując się ze środowiskami programistycznymi.

Książka „JUnit. Pragmatyczne testy jednostkowe w Javie" to wprowadzenie do tematyki testów jednostkowych. Czytając ją poznasz ich znaczenie i nauczysz się stosować JUnit do projektowania i wykorzystywania testów. Dowiesz się, jak projektować testy jednostkowe w oparciu JUnit, co testować za ich pomocą i gdzie umieszczać kod testowy. W książce przeczytasz też o zasadach projektowania łatwego do testowania kodu oraz programowaniu sterowanym testami.

- Cele przeprowadzania testów jednostkowych
- Planowanie testów
- Implementacja testów z wykorzystaniem JUnit
- Zasady stosowania testów
- Automatyzowanie testów
- Określanie częstotliwości testowania
- Projektowanie kodu pod kątem testowania

Przekonaj się, jak bardzo testy jednostkowe ułatwią Ci pracę

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 4.
Co testować?

Trudno przewidzieć wszystkie możliwe błędy w działaniu klasy lub metody. Jeśli mamy sporo doświadczenia w tym zakresie, to intuicyjnie rozpoznajemy miejsca, w których mogą pojawić się kłopoty, i dzięki temu koncentrujemy się najpierw na przetestowaniu właśnie tych miejsc. Jednak dla mniej doświadczonych programistów wykrywanie kłopotliwych obszarów często jest frustrującym zadaniem. Tym bardziej że szczególnie łatwo przychodzi ono użytkownikom końcowym, co jest nie tylko irytujące, ale może stanowić także gwóźdź do trumny niejednej kariery programisty. Dlatego niezwykle cenne są wszelkie wskazówki mogące nam wskazać lub choćby przypomnieć obszary, na które szczególnie należy zwrócić uwagę podczas testowania.

Przyjrzyjmy się zatem sześciu różnym obszarom testowania, których wykorzystanie powinno zwiększyć naszą skuteczność w wykrywaniu błędów:

- Czy wyniki są **poprawne**?
- Czy **warunki brzegowe** zostały prawidłowo określone?
- Czy można sprawdzić relacje zachodzące w **odwrotnym kierunku**?
- Czy można sprawdzić wyniki, uzyskując je w **alternatywny sposób**?

- Czy można wymusić **warunki zajścia błędu**?
- Czy **charakterystyka efektywnościowa** jest poprawna?

Czy wyniki są poprawne?

Pierwszy, najbardziej oczywisty obszar testowania to sprawdzenie, czy wyniki działania testowanych metod są poprawne.

Kontrolę wyników przedstawiliśmy w rozdziale 2. na przykładzie prostej metody zwracającej największy element listy.

Testy te są zwykle najłatwiejsze do przeprowadzenia, a wiele spodziewanych wyników często określonych jest w specyfikacji wymagań. Jeśli nie, musisz odpowiedzieć sobie na pytanie:

Skąd mam wiedzieć, czy kod działa poprawnie?

Jeśli znalezienie satysfakcjonującej odpowiedzi na to pytanie nie jest możliwe, tworzenie kodu i jego testów może okazać się kompletnym marnotrawstwem czasu. A co w przypadku, gdy wymagania dopiero się krystalizują? Czy oznacza to, że nie możemy rozpocząć pisania kodu, dopóki wymagania nie zostaną dokładnie sprecyzowane?

Nie, wcale nie. Jeśli ostateczne wymaganie nie są jeszcze znane lub kompletne, zawsze możesz wymyślić własne wymagania jako punkt odniesienia. Może okazać się, że nie są one poprawne z punktu widzenia użytkownika, ale dzięki nim wiesz, jak powinien według Ciebie działać kod, i potrafisz odpowiedzieć na postawione wcześniej pytanie.

Oczywiście musisz potem, wspólnie z użytkownikami, zweryfikować swoje założenia. Definicja poprawności wyników może zmieniać się w trakcie powstawania kodu, ale na każdym etapie jego tworzenia powinieneś móc stwierdzić, czy kod działa tak, jak założyłeś.

Stosowanie plików zawierających dane testowe

W przypadku zestawów testów wymagających dużej ilości danych wejściowych można rozważyć ich umieszczenie (być może wraz z oczekiwanymi wynikami) w osobnych plikach wczytywanych przez testy jednostkowe. Implementacja takiego rozwiązania nie jest skomplikowana i niekoniecznie musi od razu wymagać zastosowania języka XML[1]. Poniżej przedstawiona została wersja testu `TestLargest`, która wczytuje wszystkie testy z pliku.

```
import junit.framework.*;
import java.io.*;
import java.util.ArrayList;
import java.util.StringTokenizer;

public class TestLargestDataFile extends TestCase {

  public TestLargestDataFile(String name) {
    super(name);
  }

  /* Wykonuje testy zapisane w pliku testdata.txt (nie testuje
   * przypadku wyjątku). Wyrzuca wyjątek w przypadku,
   * gdy któraś operacja wejścia i wyjścia zakończy się błędem.
   */

  public void testFromFile() throws Exception {
    String line;
    BufferedReader rdr = new BufferedReader(
                                    new FileReader(
                                    "testdata.txt"));

    while ((line = rdr.readLine()) != null) {

      if (line.startsWith("#")) { // Ignoruje komentarz
        continue;
      }

      StringTokenizer st = new StringTokenizer(line);
      if (!st.hasMoreTokens()) {
        continue; // Pusty wiersz
      }

      // Pobiera spodziewany wynik
      String val = st.nextToken();
      int expected = Integer.valueOf(val).intValue();
```

[1] To jest oczywiście żart, przecież zastosowanie języka XML jest obligatoryjne we wszystkich współczesnych projektach?

TestLargestDataFile.java

```
        // Oraz argumenty dla testowanej metody
        ArrayList argument_list = new ArrayList();

        while (st.hasMoreTokens()) {
          argument_list.add(Integer.valueOf(
                                    st.nextToken()));
        }

        // Zamienia listę obiektów w tablicę typu podstawowego
        int[] arguments = new int[argument_list.size()];
        for (int i=0; i < argument_list.size(); i++) {
          arguments[i] = ((Integer)argument_list.
                          get(i)).intValue();
        }

        // Wykonuje asercję
        assertEquals(expected,
                     Largest.largest(arguments));
      }
    }
  }
```

Plik danych testowych ma bardzo prosty format — każdy wiersz zawiera zbiór wartości. Pierwsza z nich jest oczekiwanym wynikiem, a pozostałe argumentami testowanej metody. Dodatkowo zastosowanie znaku # na początku wiersza umożliwia wprowadzanie komentarzy.

A oto przykład zawartości takiego pliku:

testdata.txt

```
#
# Proste testy:
#
9 7 8 9
9 9 8 7
9 9 8 9
#
# Testy z użyciem wartości ujemnych:
#
-7 -7 -8 -9
-7 -8 -7 -8
-7 -9 -7 -8
#
# Testy mieszane:
#
7 -9 -7 -8 7 6 4
9 -1 0 9 -7 4
#
# Warunki brzegowe:
#
1 1
0 0
2147483647 2147483647
-2147483648 -2147483648
```

W przypadku kilku testów — jak choćby w powyższym przykładzie — wysiłek włożony w opracowanie odpowiedniego rozwiązania nie zwróci się. Jednak w przypadku zaawansowanej aplikacji wymagającej przeprowadzenia setek takich testów rozwiązanie wykorzystujące pliki danych na pewno warte jest zainteresowania.

Należy przy tym pamiętać, że dane testowe — obojętnie, czy umieszczone bezpośrednio w kodzie, czy w pliku danych testowych — same mogą być niepoprawne. Doświadczenie podpowiada nawet, że prawdopodobieństwo tego, iż dane testowe są nieprawidłowe, *jest większe* niż tego, że testowany kod jest niepoprawny. Zwłaszcza jeśli dane testowe zostały przygotowane drogą ręcznych obliczeń lub uzyskane z systemu, który zastępujemy naszym oprogramowaniem (ponieważ jego nowe możliwości mogą mieć wpływ na uzyskiwane wyniki). Jeśli wyniki testów są niepomyślne, warto najpierw sprawdzić kilka razy poprawność danych testowych, zanim zabierzemy się do poszukiwania błędu w kodzie.

Przedstawiony powyżej kod nie umożliwia testowania wyjątków. Zastanów się, w jaki sposób zaimplementować taką możliwość.

Wybierz taki sposób testowania, który pozwoli najłatwiej sprawdzić, czy metoda działa poprawnie.

Warunki brzegowe

W przykładzie z wyszukiwaniem największego elementu listy udało nam się zidentyfikować szereg warunków brzegowych: gdy największy element znajdował się na końcu listy, gdy lista zawierała wartość ujemną, gdy lista była pusta i tak dalej.

Identyfikacja warunków brzegowych jest jednym z najwartościowszych aspektów testowania, ponieważ pozwala wykryć i przetestować obszary, w których prawdopodobieństwo niepoprawnego działania kodu jest największe. Typowe warunki brzegowe powstają na skutek:

- wprowadzenia całkowicie błędnych lub niespójnych danych wejściowych, na przykład łańcucha `"!*W:X\&Gi/w~>g/h#WQ@"` jako nazwy pliku;
- nieprawidłowo sformatowanych danych, takich jak adres e-mail, który nie zawiera głównej domeny (`"fred@foobar."`);
- niewprowadzenia odpowiednich wartości (wtedy pojawiają się wartości `0`, `0.0`, `""` lub `null`);
- pojawienia się wartości znacznie przekraczających oczekiwania (na przykład wiek osoby równy 10 000 lat);
- pojawienia się duplikatów na listach, które nie powinny ich zawierać;
- wystąpienia list nieuporządkowanych zamiast uporządkowanych i na odwrót; spróbuj na przykład przekazać algorytmowi sortowania listę, która jest już posortowana, albo nawet posortuj ją w odwrotnym porządku;
- zakłócenia przewidywanego porządku zdarzeń, na przykład próby wydrukowania dokumentu przed zalogowaniem się.

Warunków brzegowych można poszukiwać w przedstawionych poniżej obszarach. W przypadku każdego z podanych niżej warunków należy zastanowić się, czy dotyczy on testowanej metody, a jeśli tak, to co się stanie, gdy zostanie naruszony:

- Zgodność — czy wartość jest zgodna z oczekiwanym formatem?
- Uporządkowanie — czy zbiór wartości jest odpowiednio uporządkowany?
- Zakres — czy wartość należy do przedziału oczekiwanych wartości?
- Odwołanie — czy kod odwołuje się do zewnętrznych obiektów, które są poza jego kontrolą?
- Istnienie — czy wartość istnieje (czyli jest różna od `null`, zera, obecna w zbiorze itd.)?

- Liczność — czy występuje dokładnie tyle wartości, ile jest oczekiwanych?
- Czas (absolutny i względny) — czy wszystkie zdarzenia zachodzą w oczekiwanej kolejności i we właściwym czasie?

Wymienione obszary warunków brzegowych omówimy w następnym rozdziale.

Sprawdzanie relacji zachodzących w odwrotnym kierunku

Działanie niektórych metod można przetestować, używając logiki ich działania w odwrotnym kierunku. Na przykład działanie metody wyznaczającej pierwiastek kwadratowy możemy sprawdzić, podnosząc wynik do kwadratu i porównując go z wartością wejściową:

```
public void testSquareRootUsingInverse() {
  double x = mySquareRoot(4.0);
  assertEquals(4.0, x * x, 0.0001);
}
```

Na podobnej zasadzie, wyszukując rekord w bazie danych, możesz sprawdzić działanie metody, która go tam wstawiła.

Jednak implementując odwrotną logikę działania metody, należy zachować szczególną ostrożność, ponieważ niepoprawne działanie metody może nie zostać wykryte z powodu tych samych błędów w implementacjach obu metod. Dlatego też tam, gdzie jest to tylko możliwe, najlepiej użyć innego kodu źródłowego do implementacji metody działającej w kierunku odwrotnym. W naszym przykładzie testu metody pierwiastkowania wykorzystaliśmy w tym celu zwykły operator mnożenia. W przypadku testowania metody wstawiającej rekord do bazy danych najlepiej użyć metody wyszukiwania dostarczonej przez producenta systemu zarządzania bazą danych.

Kontrola wyników na wiele sposobów

Wyniki działania testowanych metod możemy sprawdzać na wiele sposobów.

Zwykle bowiem istnieje więcej niż jeden sposób wyznaczenia interesującej nas wartości. Podczas implementacji kodu produkcyjnego wybieramy ten algorytm, który jest najefektywniejszy lub ma charakterystykę interesującą pod innym względem. Natomiast pozostałe możemy wykorzystać w celu sprawdzenia wyników podczas testowania. Technika ta jest szczególnie pomocna, gdy istnieje ogólnie znany, sprawdzony sposób uzyskania wyników, który jednak jest zbyt mało efektywny lub elastyczny, by mógł zostać użyty w kodzie produkcyjnym.

Mniej efektywnego algorytmu możemy użyć do sprawdzenia, czy algorytm zastosowany w wersji produkcyjnej daje takie same wyniki[2]:

```
public void testSquareRootUsingStd() {
  double number = 3880900.0;
  double root1 = mySquareRoot(number);
  double root2 = Math.sqrt(number);
  assertEquals(root2, root1, 0.0001);
}
```

Inny sposób zastosowania tej techniki polega na sprawdzeniu, czy różne dane sumują się we właściwy sposób. Załóżmy na przykład, że pracujemy nad systemem obsługi biblioteki. W takim systemie liczba egzemplarzy danej książki powinna zawsze zgadzać się z sumą egzemplarzy znajdujących się na półkach biblioteki i egzemplarzy wypożyczonych. Liczba poszczególnych egzemplarzy stanowi osobne dane, które mogą nawet być raportowane przez obiekty różnych klas, ale muszą być ze sobą zgodne i wobec tego mogą być używane podczas testowania.

[2] Niektóre arkusze kalkulacyjne (na przykład Microsoft Excel) stosują podobną technikę w celu sprawdzenia, czy do rozwiązania danego problemu użyte zostały odpowiednie modele i metody i czy wyniki działania różnych metod są ze sobą zgodne.

Wymuszanie warunków powstawania błędów

Gdy kod produkcyjny pracuje już w rzeczywistym systemie, narażony jest na sytuacje związane z różnego rodzaju błędami, takimi jak brak miejsca na dysku, awarie sieci czy niepoprawna praca innych aplikacji. Aby przetestować zachowanie kodu w takich sytuacjach, musimy umieć je wywołać.

Nie jest to trudne, gdy dotyczy na przykład przekazania kodowi niepoprawnych parametrów, ale już symulacja błędów sieci wymaga zastosowania odpowiednich technik. Jedną z nich — polegającą na użyciu obiektów imitujących — omówimy w rozdziale 6. Zanim tam dotrzesz, spróbuj zastanowić się, jakie rodzaje błędów lub ograniczeń wprowadzanych przez środowisko wykonywania kodu należy przetestować? Sporządź krótką ich listę, zanim przejdziesz do dalszej lektury.

Zastanów się nad tym, zanim przejdziesz do dalszej lektury...

A oto kilka typowych problemów, które udało nam się wymyślić.

- Brak wolnej pamięci.
- Brak wolnego miejsca na dysku.
- Nieprawidłowy czas systemu.
- Brak dostępu do sieci i błędy transmisji.
- Przeciążenie systemu.
- Ograniczona paleta kolorów.
- Zbyt duża lub zbyt mała rozdzielczość obrazu.

Charakterystyka efektywnościowa

Kolejnym obszarem wartym przetestowania jest charakterystyka efektywnościowa. Nie chodzi tutaj o samą efektywność działania kodu, lecz o sposób jej zmiany w odpowiedzi na zwiększającą się ilość danych wejściowych, rosnący poziom komplikacji rozwiązywanego problemu i tym podobne.

W tym przypadku zależy nam przede wszystkim na przeprowadzeniu testu regresji charakterystyki efektywnościowej. Często bowiem zdarza się, że pewna wersja systemu pracuje poprawnie, ale już następna okazuje się zaskakująco wolna. Zwykle nie wiemy wtedy, jaka jest tego przyczyna, kto, kiedy i dlaczego wprowadził modyfikacje, które pogorszyły efektywność kodu. A użytkownicy zniecierpliwieni czekają na rozwiązanie problemu.

Aby uniknąć takiego scenariusza, możemy wykonać kilka podstawowych testów, które upewnią nas, że krzywa efektywności jest stabilna w nowej wersji kodu. Załóżmy na przykład, że napisaliśmy filtr pozwalający identyfikować witryny internetowe, do których dostęp chcemy zablokować.

Stworzony kod działa poprawnie w przypadku kilkudziesięciu witryn, ale czy równie efektywnie będzie się zachowywać w przypadku 10 000 lub nawet 100 000 witryn? Aby się o tym przekonać, musimy napisać odpowiedni test jednostkowy.

```
public void testURLFilter() {
  Timer timer = new Timer();
  String naughty_url = "http://www.xxxxxxxxxx.com";

  // Najpierw wyszukaj niedozwolony URL na małej liście
  URLFilter filter = new URLFilter(small_list);
  timer.start();
  filter.check(naughty_url);
  timer.end();

  assertTrue(timer.elapsedTime() < 1.0);
  // Następnie wyszukaj niedozwolony URL na większej liście
  filter = new URLFilter(big_list);

  timer.start();
  filter.check(naughty_url);
  timer.end();

  assertTrue(timer.elapsedTime() < 2.0);

  // Na koniec wyszukaj niedozwolony URL na liście o jak największym rozmiarze
  filter = new URLFilter(huge_list);

  timer.start();
  filter.check(naughty_url);
  timer.end();

  assertTrue(timer.elapsedTime() < 3.0);
}
```

Przeprowadzenie takiego testu daje nam gwarancję, że kod spełnia wymagania odnośnie do jego efektywności. Ponieważ jego wykonanie zajmuje kilka sekund, nie musimy uruchamiać go za każdym razem. Wystarczy, że będziemy go wykonywać wraz z innymi testami w nocy lub co kilka dni, a zostaniemy zaalarmowani o pojawiających się problemach z efektywnością kodu wystarczająco wcześnie, aby poprawić je na czas.

Istnieje wiele dekoratorów umożliwiających dokładniejsze pomiary efektywności poszczególnych metod, symulację warunków silnego obciążenia i tym podobne. Jednym z takich produktów jest dostępny bezpłatnie JUnitPerf[3].

[3] *http://www.clarkware.com*