

# TestNG: testowanie jednostkowe nowej generacji

Bartosz Walter

Instytut Informatyki Politechniki Poznańskiej

## 1 Wprowadzenie

TestNG jest nowej generacji biblioteką do tworzenia testów jednostkowych w języku Java. Jej podstawowe koncepcje wywodzą się z wcześniejszej, doskonale znanej biblioteki JUnit, jednak w porównaniu do niej posiada ona kilka istotnych różnic, znacznie ułatwiających tworzenie testów i zarządzanie nimi.

Niewątpliwą zaletą JUnita jest jego prostota. Pozwala ona szybko zrozumieć zasady tworzenia i wykonywania testów jednostkowych oraz łatwo nimi zarządzać. Zaleta ta zaczyna być wadą w przypadku problemów z licznymi testami, jakie powstają w dużych projektach: wykonywane są zawsze wszystkie przypadki testowe, w arbitralnej kolejności, trudno też efektywnie nimi zarządzać. W takiej sytuacji możliwości tej biblioteki okazują się niewystarczające.

Te okoliczności, jak i wprowadzenie do języka Java mechanizmu adnotacji spowodowały zintensyfikowanie prac nad nową generacją bibliotek służących do tworzenia i wykonywania testów. Wśród nich znalazły się m.in. TestNG, opublikowany w 2004 roku, oraz nowa wersja JUnita 4.0, która znajduje się w stadium końcowych prac przed publikacją.

W tym artykule omówiono bibliotekę TestNG, przedstawiając jej zalety nad dotychczas opublikowanymi wersjami JUnita.

## 2 Biblioteka JUnit 3.x

### 2.1 Ogólne zasady testowania jednostkowego

Testowanie jednostkowe jest techniką służącą do weryfikacji poprawności oprogramowania i polega na sprawdzeniu, czy rozważana jednostka (zwykle jest nią pojedyncza metoda w klasie) dla zadanych argumentów daje poprawne, spodziewane wyniki. Jednostkowość polega na tym, że testom poddawana jest tylko wyizolowana klasa, z pominięciem wszelkich innych klas i zależności. Dzięki temu możliwe jest objęcie pojedynczym testem bardzo małego obszaru, co zwiększa możliwości jego powtórnego użycia w przyszłości.

Podstawowym pojęciem stosowanym w testowaniu jednostkowym jest przypadek testowy (nie należy tego mylić z klasą TestCase w JUnit, która oznacza klasę testującą;

każda instancja takiej klasy jej przypadkiem testowym). Jest to pojedyncze wykonanie testowanej metody dla określonych danych wejściowych oraz weryfikacja, czy dla tych danych otrzymany wynik jest zgodny z oczekiwanym. Ponieważ metoda może dawać różne wyniki w zależności od rodzaju przekazanych parametrów, dlatego prawidłowe jej przetestowanie może wymagać kilku przypadków testowych, z których każdy weryfikuje jedno z jej możliwych zachowań.

Na przykład, metoda ustawiająca wiek instancji klasy *Person* o sygnaturze *void setAge(int age)* może posiadać przypadki testowe dla następujących wartości wieku:

- 25, reprezentującą poprawną wartość
- -1, czyli wartość niedopuszczalną
- 10.2, która może być niedopuszczalna, jeżeli wiek jest wyrażany w pełnych latach
- 2000, którą także można uznać za wartość niepoprawną

## 2.2 Testowanie z wykorzystaniem JUnit 3.x

Twórcy biblioteki JUnit przyjęli i promują zasadę, że dla każdej klasy powinna istnieć odpowiadająca jej klasa testująca o takiej samej nazwie z przyrostkiem „Test”. Aby była ona poprawnie rozpoznawana, musi ona dziedziczyć z klasy  *junit.framework.TestCase* należącej do biblioteki – stanowi to swego rodzaju oznaczenie klasy testowej, jak też udostępnia grupę przydatnych metod.

Klasa testująca zawiera metody stanowiące pojedyncze przypadki testowe. Przyjmuje się, że jednej testowanej metodzie odpowiada jeden lub więcej przypadków testowych. Ma to zwykle swoje odzwierciedlenie w nazwach przypadków testowych, które często zawierają nazwę metody testowanej oraz opis zachowania, jakie ten przypadek weryfikuje; należy jednak pamiętać, że JUnit narzuca pewne ograniczenia dotyczące sygnatur: metoda testująca musi zwracać typ *void*, nie przyjmować żadnych parametrów, a jej nazwa zaczynać się od przedrostka *test*. Metody nie stosujące się do tych zasad nie są rozpoznawane jako przypadki testowe.

Przypadki testowe wewnątrz klasy testującej są z założenia wykonywane w losowej kolejności, co jest zgodne z zasadą testowania jednostkowego. Istnieje wprawdzie możliwość ręcznego określenia porządku, wymaga ona jednak rekompilacji klas testujących.

Klasa testująca poza przypadkami testowymi może także posiadać dwie metody o sygnaturach *void setUp()* i *void tearDown()*, które są wykonywane odpowiednio przed i po wykonaniu każdego przypadku testowego. Służą one głównie do inicjalizacji, a następnie usunięcia obiektu testowanego i potrzebnego środowiska.

Klasy testujące napisane z wykorzystaniem JUnit mogą być zorganizowane w grupy (ang. *suites*), dzięki czemu łatwiej je wykonywać i zarządzać nimi.

## 2.3 Niedogodności JUnita

Prostota architektury oraz łatwość wykorzystania JUnita przysporzyła mu wielu użytkowników. Jednak z biegiem czasu i w miarę wzrostu złożoności testowanego opro-

gramowania wiele jego założeń okazało się zbyt restrukcyjnych, co uniemożliwiało jego wykorzystanie na większą skalę. Wśród nich należy wymienić następujące cechy:

- **Ścisłe przywiązanie klas testujących do hierarchii testów, szczególnie klasy *TestCase*.** Klasa testująca musi dziedziczyć określony interfejs, aby była rozpoznawana przez środowisko JUnit.
- **Konieczność stosowania konwencji w nazwach metod.**
- **Niemożliwość określania zależności pomiędzy przypadkami testowymi.** JUnit wykonuje wszystkie przypadki testowe, nawet w oczywisty sposób zależne od innych przypadków; ich niepowodzenie powoduje kaskadowe zgłaszanie błędów, które są łatwe do przewidzenia i nie niosą żadnej nowej informacji.
- **Utrudnione grupowanie klas testujących w grupy.** Powiązanie klas testujących w grupy wymaga rekompilacji kodu
- **Inicjalizacja obiektu testowanego jedynie na poziomie przypadku testowego.** Metody *setUp()* i *tearDown()* są wykonywane zawsze przed i po każdym przypadku testowym. Brakuje mechanizmu, który pozwalałby przenieść część inicjalizacji przed i po każdej klasie czy grupie.

Część tych niedogodności jest rekompensowana w części poprzez dodatki do JUnita (np. *Multiple Failures*) oraz podobne biblioteki (np. JUnitour, JTestCase, JFunc – wyczerpującą listę można znaleźć pod adresem <http://www.junit.org/>), które pozwalają uzupełnić jego funkcjonalność. Niestety, pozostałe wynikają z architektury i założeń przyjętych podczas projektowania tej biblioteki, a także dostępnych rozwiązań w samym języku Java.

### 3 Testy nowej generacji: TestNG

W celu rozwiązania wymienionych wyżej problemów powstała opublikowana we wrześniu 2004 roku wersja 1.0 biblioteki TestNG, która zachowując koncepcję testowania jednostkowego znaną z JUnit, w znacznym stopniu zmieniła sposób reprezentacji i konfiguracji testów. W dużej mierze korzysta ona z możliwości dostępnych w JDK 1.5, w szczególności z mechanizmu adnotacji, choć istnieje także jej wersja dla JDK 1.4 (wówczas zamiast adnotacji wykorzystywane znaczniki JavaDoc). Poniżej pokrótce omówiono najważniejsze cechy TestNG:

- **Separacja implementacji testów od ich wykonania.** Testy jednostkowe są wykonywane wielokrotnie, jednak nie zawsze konieczne jest wykonanie wszystkich przypadków testowych. TestNG pozwala specyfikować zakres testów niezależnie od kodu tych testów w postaci pliku XML. Dzięki temu można zdefiniować różne zestawy testów (za pomocą osobnych plików XML) do różnych zastosowań.
- **Rezygnacja z mechanizmu dziedziczenia z wyróżnionej klasy testującej.** Klasa testująca zapisana w TestNG nie musi być potomkiem konkretnej klasy należącej do biblioteki – może nią być dowolna zwykła klasa (tzw. POJO – *Plain Old Java Object*).

- **Koniec z konwencjami nazewniczymi.** Podobnie, nazwy metod-przypadków testowych nie muszą odpowiadać jakimkolwiek konwencjom. Wystarczy, że są oznaczone adnotacją `@Test`.
- **Parametryzacja przypadków testowych.** Przypadki testowe mogą przyjmować parametry typu `String`, których liczba i wartości są konfigurowane niezależnie od kodu testu.
- **Ziarnistość inicjalizacji i finalizacji.** Istnieje możliwość precyzyjnego ustalenia ziarnistości i zakresu inicjalizacji oraz finalizacji środowiska testowego: za pomocą adnotacji można oznaczyć metody wykonywane przed i po wszystkich przypadkach testowych w grupie (`@beforeSuite` i `@afterSuite`), w klasie testującej (`@beforeTestClass` i `@afterTestClass`) oraz – podobnie jak w JUnit'cie 3.x – pojedynczym przypadku testowym (`@beforeTestMethod` i `@afterTestMethod`)
- **Wykorzystanie asercji wbudowanych w język w miejsce asercji programowych.**
- **Specyfikowanie zależności między grupami przypadków testowych i określanie kolejności ich wykonywania.** Cecha ta, nieobecna w JUnit, pozwala m.in. na automatyczne pomijanie tych testów, których zależności już wcześniej nie zostały spełnione (tzn. zwróciły błędy). Pozwala także ręcznie oznaczać niektóre testy do pominięcia w aktualnym wykonaniu.
- **Proste wskazywanie oczekiwanych wyjątków.** Przypadek testowy sprawdzający pojawienie się oczekiwanego wyjątku zaimplementowany z wykorzystaniem JUnit 3.x musiał przechwycić ten wyjątek, a następnie go zignorować, natomiast brak wyjątku był sygnalizowany bezwarunkowym zgłoszeniem błędu. TestNG pozwala wskazać oczekiwany wyjątek za wyłączenie pomocą adnotacji

```
@Test
@ExpectedExceptions(NullPointerException.class)
public void metoda() {
    null.equals(null);
}
```

## 4 Przykład wykorzystania

W celu zilustrowania zasad, jakimi posługuje się TestNG, prześledźmy następujący przykład.

```
public class TestPrzykladowy {
    @Configuration(beforeTestClass = true)
    public void przygotowanie() {
    }

    @Configuration(afterTestClass = true)
```

```

public void sprzatanie() {
}

@Configuration(beforeTestMethod = true)
public void przedKazdymPrzypadkiemTestowym() {
}

@Test(groups = { "grupa-A" })
public void przypadekTestowy1() {
}

@Test(enabled = false)
public void przypadekTestowy2() {
}

@Test
@Parameters({ "parametr" })
@DependsOnMethods({ "przypadekTestowy2" })
public void przypadekTestowy3(String parametr) {
}
}

```

Klasa *TestPrzykladowy* zawiera przypadki testowe do wykonania. Metody *przygotowanie()* i *sprzatanie()* zostaną wykonane tylko raz, odpowiednio przed i po wszystkich przypadkach zawartych w tej klasie. Metoda *przedKazdymPrzypadkiemTestowym()* będzie wykonywana analogicznie do metody *setUp()* znanej z JUnit, czyli przed wykonaniem każdej metody testującej.

Metoda *przypadekTestowy1()* jest przypadkiem testowym i należy do grupy o nazwie „grupa-A”, dzięki czemu jest logicznie powiązany z innymi testami z tej grupy i może być wykonany razem z nimi. Metoda *przypadekTestowy2()* również jest metodą testującą, ale nie należy do żadnej nazwanej grupy, a przy tym jest wyłączona, tzn. nie zostanie wykonana podczas uruchamiania testów.

Metoda *przypadekTestowy3()* jest bardziej skomplikowana. Jej adnotacje wskazują, że może ona przyjąć parametr o nazwie „parametr”, który zostanie zdefiniowany w pliku konfiguracyjnym. Ponadto przypadek ten zależy od metody *przypadekTestowy2()*: jeżeli on się nie powiedzie, to wówczas *przypadekTestowy3()* zostanie pominięty w wykonywaniu.

Klasa ta może zostać uruchomiona przy wykorzystaniu następującego pliku konfiguracyjnego *testng.xml*:

```

<!DOCTYPE suite SYSTEM
    "http://testng.org/testng-1.0.dtd">
<suite name="Zestaw1" verbose="1">
  <test name="Test1">
    <classes>
      <parameter name="parametr" value="abc"/>
      <class name="TestPrzykladowy"/>
    </classes>
  </test>
</suite>

```

```
    </classes>
  </test>
</suite>
```

Plik *testng.xml* daje dużo większe możliwości selekcji i konfiguracji testów – w celu zapoznania się ze szczegółami należy zajrzeć do dokumentacji biblioteki.

Testy uruchomić można na kilka sposobów, m. In. Za pomocą systemu Ant oraz z linii poleceń. Poniżej podano ten drugi sposób:

```
java -classpath testng.jar;%CLASSPATH% -ea
    org.testng.TestNG testng.xml
```

Przełącznik *-ea* powoduje włączenie mechanizmu asercji w maszynie wirtualnej (właśnie te asercje wykorzystuje TestNG), natomiast *org.testng.TestNG* jest klasą startową, która czytuje podany plik konfiguracyjny i uruchamia przypadki testowe).

## 5 Podsumowanie

TestNG otwiera nowe możliwości tworzenia i uruchamiania testów jednostkowych. Stanowi on rozwiązanie nie tylko dla osób rozpoczynających pracę z testowaniem, ale także dla tych, którzy muszą uruchamiać setki już istniejących przypadków testowych. Oferuje on dwa rozwiązania:

- Testy JUnit 3.x można bez zmian uruchamiać w środowisku TestNG
- Posiada translator, który automatycznie uzupełnia treść klas testujących o adnotacje TestNG.

TestNG posiada także swój plugin dla popularnego środowiska Eclipse, który znacznie ułatwia pracę z nową biblioteką.

W niedługim czasie zostanie opublikowana wersja 4.0 JUnita, która rozwiązuje wiele problemów w podobny sposób jak TestNG. Wygląda więc na to, że nic tak dobrze nie robi produktowi, jak oddech konkurencji na plecach.

## Literatura

1. C. Beust, A. Popescu, *TestNG: Testing. The next generation*. <http://www.testng.org/>
2. F. Diotalevi, *TestNG makes Java unit testing a breeze*.  
<http://www-128.ibm.com/developerworks/java/library/j-testng/>
3. T. Janaudy, *TestNG: The next generation of unit-testing*.  
[http://www.javaworld.com/javaworld/jw-04-2005/jw-0404-testng\\_p.html](http://www.javaworld.com/javaworld/jw-04-2005/jw-0404-testng_p.html)
4. N. Ehreke: *Annotations to the rescue*. <http://www.javaworld.com/javaworld/jw-08-2005/jw-0801-annotations.html>
5. TestNG Eclipse plugin. <http://testng.org/doc/eclipse.html>