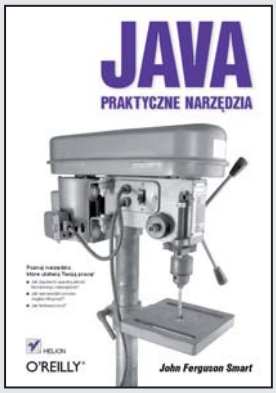
Java. Praktyczne narzędzia



Autor: John Ferguson Smart
Tłumaczenie: Mikołaj Szczepaniak
ISBN: 978-83-246-1932-0
Tytuł oryginału: Java Power Tools
Format: 168x237, stron: 888

Poznaj narzędzia, które okażą się niezbędne!

- Jak zapewnić wysoką jakość tworzonego rozwiązania?
- Jak wprowadzić proces ciągłej integracji?
- Jak testować kod?

Możliwości języka Java znają już chyba wszyscy. Dlatego warto jedynie wspomnieć o tym, że oprócz podstawowych narzędzi do tworzenia oprogramowania w tym języku, które zna każdy programista, istnieje wiele innych – przydatnych i użytecznych – aplikacji. Potrafią one w niezwykle skuteczny sposób przyśpieszyć oraz ułatwić programowanie w języku Java i sprawić, że będzie to zajęcie jeszcze przyjemniejsze. W żadnej innej książce nie znajdziesz tak szczegółowego omówienia tych narzędzi. Zatem jeśli wykorzystujesz język Java na co dzień, musisz ją mieć!

Dzięki tej książce poznasz 33 praktyczne narzędzia, które ułatwią Twoją pracę – narzędzia, które zwiększą niezawodność Twojego kodu, poprawią wydajność oraz zapewnią bezpieczeństwo Twoim plikom źródłowym. Autor książki omawia kilka grup narzędzi, a wśród nich aplikacje takie, jak Maven, Subversion, JUnit czy też Hudson. Dzięki książce „Java. Praktyczne narzędzia" dowiesz się, jak bardzo na jakość Twojego rozwiązania może wpłynąć proces ciągłej integracji oraz jak ważne są testy jednostkowe czy integracyjne. Ponadto autor książki omawia 29 innych narzędzi, które zwiększają komfort pracy. Otwórz spis treści i spójrz, jak cenne informacje są zawarte w tej książce!

- Wykorzystanie narzędzi kompilujących (Ant, Maven2)
- Zastosowanie systemów kontroli wersji (CVS, Subversion)
- Sposoby oceny jakości kodu (CheckStyle, PMD, FindBugs, Jupiter)
- Tworzenie wysokiej jakości dokumentacji
- Przygotowanie testów jednostkowych (JUnit, TestNG)
- Przeprowadzanie testów integracyjnych
- Systemy raportowania i śledzenia błędów (Bugzilla, Trac)
- Narzędzia pozwalające na wprowadzenie procesu ciągłej integracji (Continuum, Hudson)
- Sposoby przeprowadzania testów obciążeniowych
- Profilowanie i monitorowanie aplikacji za pomocą narzędzi dostępnych w pakiecie JDK oraz Eclipse

Zobacz, jak łatwo można wykonać skomplikowane zadania!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Słowo wstępne **17**

Przedmowa **19**

Wprowadzenie **33**

I Narzędzia kompilujące **37**

1. Przygotowywanie projektu z wykorzystaniem Anta **41**

1.1. Rola narzędzia Ant w procesie kompilacji 41
1.2. Instalacja Anta 41
1.3. Płynne wprowadzenie w świat Anta 44
1.4. Kompilowanie kodu Javy za pomocą Anta 51
1.5. Dostosowywanie skryptów kompilacji za pomocą właściwości 53
1.6. Przeprowadzanie testów jednostkowych za pomocą Anta 57
1.7. Generowanie dokumentacji za pomocą narzędzia Javadoc 75
1.8. Pakowanie gotowej aplikacji 77
1.9. Wdrażanie aplikacji 81
1.10. Automatyczne przygotowywanie środowiska dla uruchamianych skryptów kompilacji 83
1.11. Stosowanie zależności narzędzia Maven w Ancie wraz z zadaniami Mavena 85
1.12. Stosowanie Anta w środowisku Eclipse 89
1.13. Stosowanie Anta w środowisku NetBeans 89
1.14. Modyfikowanie kodu XML-a za pomocą zadania XMLTask 90
1.15. Konkluzja 95

2. Przygotowywanie projektu z wykorzystaniem Mavena 2 ..... **97**

2.1. Rola narzędzia Maven w procesie kompilacji 97
2.2. Maven i Ant 98
2.3. Instalacja Mavena 99
2.4. Kompilacje deklaratywne i model obiektu projektu Mavena 101

2.5. Zrozumieć cykl życia Mavena 2 112
2.6. Struktura katalogów Mavena 114
2.7. Konfigurowanie Mavena pod kątem naszego środowiska 115
2.8. Zarządzanie zależnościami w Mavenie 2 118
2.9. Poszukiwanie zależności za pośrednictwem witryny Maven Repository 126
2.10. Dziedziczenie i agregacja projektów 127
2.11. Tworzenie szablonu projektu za pomocą tzw. archetypów 131
2.12. Kompilacja kodu 135
2.13. Testowanie kodu 136
2.14. Pakowanie i wdrażanie naszej aplikacji 138
2.15. Wdrażanie aplikacji z wykorzystaniem narzędzia Cargo 140
2.16. Stosowanie Mavena w środowisku Eclipse 144
2.17. Stosowanie Mavena w środowisku NetBeans 147
2.18. Dostosowywanie procesu kompilacji do specyficznych potrzeb projektu za pomocą własnych modułów rozszerzeń 147
2.19. Konfigurowanie repozytorium korporacyjnego za pomocą narzędzia Archiva 154
2.20. Konfigurowanie repozytorium korporacyjnego z wykorzystaniem narzędzia Artifactory 166
2.21. Stosowanie narzędzia Ant w Mavenie 178
2.22. Archetypy zaawansowane 183
2.23. Stosowanie podzespołów 187

II Narzędzia kontroli wersji 193

3. Kontrola wersji z wykorzystaniem systemu CVS 195
3.1. Wprowadzenie do systemu CVS 195
3.2. Konfigurowanie repozytorium systemu CVS 196
3.3. Tworzenie nowego projektu w systemie CVS 196
3.4. Wypożyczanie projektu 198
3.5. Praca na plikach — aktualizowanie i zatwierdzanie plików z kodem źródłowym 200
3.6. Blokowanie repozytorium 204
3.7. Praca z mechanizmem zastępowania słów kluczowych 204
3.8. Praca z plikami binarnymi 205
3.9. Znaczniki systemu CVS 207
3.10. Tworzenie odgałęzień w systemie CVS 208
3.11. Scalanie zmian z odgałęzienia 210
3.12. Przeglądanie historii zmian 211
3.13. Wycofywanie zmian 213
3.14. Stosowanie CVS-a w systemie Windows 214

4. Kontrola wersji z wykorzystaniem systemu Subversion 217
4.1. Wprowadzenie do systemu Subversion 217
4.2. Instalacja systemu Subversion 221
4.3. Typy repozytoriów systemu Subversion 221
4.4. Konfigurowanie repozytorium systemu Subversion 223
4.5. Tworzenie nowego projektu w systemie Subversion 225
4.6. Wypożyczanie kopii roboczej 227
4.7. Importowanie istniejących plików do repozytorium systemu Subversion 228
4.8. Zrozumieć adresy URL repozytorium systemu Subversion 230
4.9. Praca z plikami 231
4.10. Sprawdzanie bieżącej sytuacji — polecenie status 235
4.11. Rozwiązywanie konfliktów 237
4.12. Stosowanie znaczników, odgałęzień i operacji scalania 239
4.13. Przywracanie poprzedniej rewizji 243
4.14. Blokowanie dostępu do plików binarnych 244
4.15. Zdejmowanie i przechwytywanie blokad 246
4.16. Udostępnianie zablokowanych plików tylko do odczytu za pomocą właściwości svn:needs-lock 248
4.17. Stosowanie właściwości 249
4.18. Historia zmian w systemie Subversion — rejestrowanie zdarzeń i określanie odpowiedzialności za zmiany 252
4.19. Konfigurowanie serwera systemu Subversion z wykorzystaniem serwera svnserve 253
4.20. Konfigurowanie bezpiecznego serwera svnserve 257
4.21. Konfigurowanie serwera Subversion z obsługą protokołu WebDAV/DeltaV 258
4.22. Konfigurowanie bezpiecznego serwera WebDAV/DeltaV 263
4.23. Dostosowywanie działania systemu Subversion za pomocą skryptów przechwytujących 264
4.24. Instalacja systemu Subversion w formie usługi systemu operacyjnego Windows 266
4.25. Sporządzanie kopii zapasowej i przywracanie repozytorium systemu Subversion 268
4.26. Stosowanie systemu Subversion w środowisku Eclipse 268
4.27. Stosowanie systemu Subversion w środowisku NetBeans 275
4.28. Stosowanie systemu Subversion w systemie operacyjnym Windows 281
4.29. Śledzenie usterek i kontrola zmian 287
4.30. Stosowanie systemu Subversion w Ancie 290
4.31. Konkluzja 292

III Ciągła integracja 293

5. Konfigurowanie serwera ciągłej integracji za pomocą narzędzia Continuum 297
5.1. Wprowadzenie do narzędzia Continuum 297
5.2. Instalacja serwera narzędzia Continuum 297

5.3. Ręczne uruchamianie i zatrzymywanie serwera 301
5.4. Sprawdzanie stanu serwera 302
5.5. Uruchamianie serwera narzędzia Continuum w trybie ze szczegółowymi komunikatami 302
5.6. Dodawanie grupy projektów 303
5.7. Dodawanie projektu Mavena 303
5.8. Dodawanie projektu Anta 306
5.9. Dodawanie projektu kompilowanego za pomocą skryptu powłoki 307
5.10. Zarządzanie kompilacjami projektu 307
5.11. Zarządzanie użytkownikami 309
5.12. Konfigurowanie mechanizmów powiadomień 311
5.13. Konfigurowanie planowanych kompilacji 311
5.14. Diagnozowanie procesu kompilacji 314
5.15. Konfigurowanie serwera poczty elektronicznej narzędzia Continuum 314
5.16. Konfigurowanie portów witryny internetowej serwera Continuum 315
5.17. Automatyczne generowanie witryny Mavena za pomocą narzędzia Continuum 316
5.18. Konfigurowanie zadania ręcznej kompilacji 317
5.19. Konkluzja 319

6. Konfigurowanie serwera ciągłej integracji za pomocą narzędzia CruiseControl 321
6.1. Wprowadzenie do narzędzia CruiseControl 321
6.2. Instalacja narzędzia CruiseControl 322
6.3. Konfigurowanie projektu Anta 323
6.4. Powiadamianie członków zespołu za pomocą mechanizmów publikujących 329
6.5. Konfigurowanie projektu Mavena 2 w narzędziu CruiseControl 336
6.6. Panel administracyjny narzędzia CruiseControl 338
6.7. Dodatkowe narzędzia 339
6.8. Konkluzja 340

7. LuntBuild — serwer ciągłej integracji z interfejsem WWW 341
7.1. Wprowadzenie do narzędzia LuntBuild 341
7.2. Instalowanie narzędzia LuntBuild 341
7.3. Konfigurowanie serwera LuntBuild 343
7.4. Dodawanie projektu 345
7.5. Wykorzystywanie zmiennych projektowych do numerowania wersji 352
7.6. Diagnostyka wyników kompilacji 353
7.7. Stosowanie narzędzia LuntBuild w środowisku Eclipse 355
7.8. Raportowanie w systemie LuntBuild o pokryciu testami z wykorzystaniem narzędzia Cobertura 359
7.9. Integrowanie narzędzia LuntBuild z Mavenem 365
7.10. Konkluzja 370

8. Ciągła integracja z wykorzystaniem narzędzia Hudson 371
8.1. Wprowadzenie do narzędzia Hudson 371
8.2. Instalacja narzędzia Hudson 371
8.3. Zarządzanie katalogiem domowym Hudsona 372
8.4. Instalacja aktualizacji 373
8.5. Konfigurowanie Hudsona 374
8.6. Dodawanie nowego zadania kompilacji 376
8.7. Organizowanie zadań 381
8.8. Monitorowanie kompilacji 382
8.9. Przeglądanie i awansowanie wybranych kompilacji 383
8.10. Zarządzanie użytkownikami 385
8.11. Uwierzytelnianie i bezpieczeństwo 386
8.12. Przeglądanie zmian 386
8.13. Moduły rozszerzeń Hudsona 387
8.14. Śledzenie wyników testów 388
8.15. Śledzenie mierników kodu źródłowego 388
8.16. Raportowanie o pokryciu kodu 390

9. Konfigurowanie platformy natychmiastowej komunikacji za pomocą serwera Openfire 393
9.1. Natychmiastowa komunikacja w projekcie informatycznym 393
9.2. Instalacja serwera Openfire 394
9.3. Konfigurowanie użytkowników i kont użytkowników serwera Openfire 394
9.4. Uwierzytelnianie użytkowników z wykorzystaniem zewnętrznej bazy danych 396
9.5. Uwierzytelnianie użytkowników na serwerze POP3 397
9.6. Organizowanie wirtualnych spotkań zespołu z wykorzystaniem czatu grupowego 398
9.7. Rozszerzanie funkcjonalności serwera Openfire za pomocą modułów rozszerzeń 400
9.8. Stosowanie serwera Openfire z systemem Continuum 400
9.9. Stosowanie serwera Openfire z systemem CruiseControl 401
9.10. Stosowanie serwera Openfire z narzędziem LuntBuild 402
9.11. Wysyłanie komunikatów Jabbera z poziomu aplikacji Javy za pośrednictwem interfejsu API Smack 402
9.12. Wykrywanie obecności interfejsu API Smack 405
9.13. Otrzymywanie wiadomości z wykorzystaniem interfejsu API Smack 405

IV Testy jednostkowe 407

10. Testowanie kodu z wykorzystaniem frameworku JUnit 409
10.1. Frameworki JUnit 3.8 i JUnit 4 409
10.2. Testowanie jednostkowe z wykorzystaniem frameworku JUnit 4 410
10.3. Konfigurowanie i optymalizacja przypadków testów jednostkowych 412

10.4. Proste testy wydajności z wykorzystaniem limitów czasowych 414
10.5. Prosta weryfikacja występowania wyjątków 415
10.6. Stosowanie testów sparametryzowanych 415
10.7. Stosowanie metody assertThat() i biblioteki Hamcrest 418
10.8. Teorie we frameworku JUnit 4 421
10.9. Stosowanie frameworku JUnit 4 w projektach Mavena 2 423
10.10. Stosowanie frameworku JUnit 4 w projektach Anta 423
10.11. Selektywne wykonywanie testów frameworku JUnit 4 w Ancie 426
10.12. Testy integracyjne 428
10.13. Korzystanie z frameworku JUnit 4 w środowisku Eclipse 429

11. Testowanie nowej generacji z wykorzystaniem frameworku TestNG 433
11.1. Wprowadzenie do frameworku TestNG 433
11.2. Tworzenie prostych testów jednostkowych za pomocą frameworku TestNG 433
11.3. Definiowanie pakietów testów frameworku TestNG 435
11.4. Moduł rozszerzenia frameworku TestNG dla środowiska Eclipse 437
11.5. Stosowanie frameworku TestNG w Ancie 440
11.6. Korzystanie z frameworku TestNG w Mavenie 2 443
11.7. Zarządzanie cyklem życia testów 444
11.8. Stosowanie grup testów 449
11.9. Zarządzanie zależnościami 451
11.10. Testowanie równoległe 454
11.11. Parametry testów i testowanie sterowane danymi 455
11.12. Weryfikacja wyjątków 456
11.13. Obsługa błędów częściowych 456
11.14. Ponowne wykonywanie testów zakończonych niepowodzeniem 457

12. Maksymalizacja pokrycia testami za pomocą narzędzia Cobertura 459
12.1. Pokrycie testami 459
12.2. Uruchamianie narzędzia Cobertura za pośrednictwem Anta 460
12.3. Weryfikacja pokrycia kodu testami frameworku TestNG 463
12.4. Interpretacja raportu narzędzia Cobertura 465
12.5. Wymuszanie dużego pokrycia kodu 467
12.6. Generowanie raportów narzędzia Cobertura w Mavenie 469
12.7. Integracja testów pokrycia kodu z procesem kompilacji Mavena 471
12.8. Badanie pokrycia kodu w środowisku Eclipse 473
12.9. Konkluzja 475

V Testy integracyjne, funkcjonalne, obciążeniowe i wydajnościowe ...477

13. Testowanie aplikacji frameworku Struts z wykorzystaniem frameworku StrutsTestCase ... 481
13.1. Wprowadzenie 481
13.2. Testowanie aplikacji frameworku Struts 482
13.3. Wprowadzenie do frameworku StrutsTestCase 483
13.4. Testy obiektów zastępczych z wykorzystaniem frameworku StrutsTestCase 483
13.5. Testowanie mechanizmów obsługi błędów w aplikacji frameworku Struts 488
13.6. Dostosowywanie środowiska testowego 489
13.7. Testy wydajnościowe pierwszego stopnia 489
13.8. Konkluzja 490

14. Testy integracyjne baz danych z wykorzystaniem frameworku DbUnit ... 491
14.1. Wprowadzenie 491
14.2. Przegląd 491
14.3. Struktura frameworku DbUnit 493
14.4. Przykładowa aplikacja 497
14.5. Wypełnianie bazy danych 498
14.6. Weryfikacja bazy danych 506
14.7. Zastępowanie wartości 510
14.8. Alternatywne formaty zbiorów danych 516
14.9. Obsługa niestandardowych testów danych 520
14.10. Pozostałe zastosowania 524

15. Testy wydajnościowe z wykorzystaniem frameworku JUnitPerf ... 533
15.1. Wprowadzenie do frameworku JUnitPerf 533
15.2. Badanie wydajności za pomocą klasy TimedTest 534
15.3. Symulowanie obciążenia za pomocą klasy LoadTest 536
15.4. Przeprowadzanie testów wydajnościowych, które nie gwarantują bezpieczeństwa przetwarzania wielowątkowego 539
15.5. Oddzielanie testów wydajnościowych od testów jednostkowych w Ancie 540
15.6. Oddzielanie testów wydajnościowych od testów jednostkowych w Mavenie 541

16. Wykonywanie testów obciążeniowych i wydajnościowych za pomocą narzędzia JMeter ... 543
16.1. Wprowadzenie 543
16.2. Instalacja narzędzia JMeter 544
16.3. Testowanie prostej aplikacji internetowej 544
16.4. Projektowanie struktury naszego przypadku testowego 550
16.5. Rejestrowanie i wyświetlanie wyników testu 553

16.6. Rejestrowanie przypadku testowego za pomocą serwera proxy narzędzia JMeter 556
16.7. Testowanie z wykorzystaniem zmiennych 558
16.8. Testowanie na wielu komputerach 560

17. Testowanie usług sieciowych za pomocą narzędzia SoapUI 563
17.1. Wprowadzenie 563
17.2. Wprowadzenie do narzędzia SoapUI 563
17.3. Instalacja narzędzia SoapUI 565
17.4. Instalacja lokalnej usługi sieciowej 565
17.5. Testowanie usług sieciowych za pomocą narzędzia SoapUI 567
17.6. Przeprowadzanie testów obciążeniowych za pomocą narzędzia SoapUI 573
17.7. Uruchamianie narzędzia SoapUI z poziomu wiersza poleceń 576
17.8. Uruchamianie narzędzia SoapUI za pośrednictwem Anta 578
17.9. Uruchamianie narzędzia SoapUI za pośrednictwem Mavena 579
17.10. Testy ciągłe 580
17.11. Konkluzja 581

18. Profilowanie i monitorowanie aplikacji Javy za pomocą narzędzi pakietu Sun JDK 583
18.1. Narzędzia profilujące i monitorujące pakietu Sun JDK 583
18.2. Nawiązywanie połączenia z aplikacją Javy i monitorowanie jej działania za pomocą narzędzia JConsole 583
18.3. Monitorowanie zdalnej aplikacji na serwerze Tomcat za pomocą narzędzia JConsole 587
18.4. Wykrywanie i identyfikacja wycieków pamięci za pomocą narzędzi pakietu JDK 588
18.5. Diagnozowanie wycieków pamięci z wykorzystaniem zrzutów sterty oraz narzędzi jmap i jhat 593
18.6. Wykrywanie zakleszczeń 595

19. Profilowanie aplikacji Javy w środowisku Eclipse 599
19.1. Profilowanie aplikacji z poziomu środowiska IDE 599
19.2. Platforma TPTP środowiska Eclipse 599
19.3. Instalacja platformy TPTP 601
19.4. Platformy TPTP i Java 6 601
19.5. Podstawowe techniki profilowania z wykorzystaniem platformy TPTP 602
19.6. Ocena użycia pamięci na podstawie wyników podstawowej analizy pamięci 607
19.7. Analiza czasu wykonywania 609
19.8. Wyświetlanie statystyk pokrycia 610
19.9. Stosowanie filtrów zawężających uzyskiwane wyniki 611
19.10. Profilowanie aplikacji internetowej 613
19.11. Konkluzja 613

20. Testowanie interfejsów użytkownika 615
20.1. Wprowadzenie 615
20.2. Testowanie aplikacji internetowej za pomocą narzędzia Selenium 615
20.3. Testowanie graficznych interfejsów Swinga za pomocą narzędzia FEST 642
20.4. Konkluzja 651

VI Narzędzia pomiaru jakości..... 653

21. Wykrywanie i wymuszanie standardów kodowania za pomocą narzędzia Checkstyle 657
21.1. Wymuszanie standardów kodowania za pomocą narzędzia Checkstyle 657
21.2. Stosowanie narzędzia Checkstyle w środowisku Eclipse 659
21.3. Modyfikowanie reguł narzędzia Checkstyle w środowisku Eclipse 663
21.4. Dostosowywanie reguł narzędzia Checkstyle z wykorzystaniem plików konfiguracyjnych w formacie XML 665
21.5. Dostosowywanie pracy narzędzia Checkstyle — reguły, bez których możemy sobie poradzić, i kilka reguł, z których warto korzystać 667
21.6. Stosowanie narzędzia Checkstyle do definiowania reguł dla nagłówków w kodzie źródłowym 671
21.7. Wstrzymywanie testów narzędzia Checkstyle 672
21.8. Korzystanie z narzędzia Checkstyle w Ancie 673
21.9. Korzystanie z narzędzia Checkstyle w Mavenie 674

22. Wstępne wykrywanie błędów za pomocą narzędzia PMD 677
22.1. Narzędzie PMD i statyczna analiza kodu 677
22.2. Korzystanie z narzędzia PMD w środowisku Eclipse 677
22.3.Konfiguracja reguł narzędzia PMD w środowisku Eclipse 680
22.4. Więcej o zbiorach reguł narzędzia PMD 681
22.5. Pisanie własnych zbiorów reguł narzędzia 684
22.6. Generowanie raportu narzędzia PMD w środowisku Eclipse 685
22.7. Wstrzymywanie reguł narzędzia PMD 686
22.8. Wykrywanie praktyki „wytnij i wklej" za pomocą narzędzia CPD 687
22.9. Stosowanie narzędzia PMD w Ancie 688
22.10. Stosowanie narzędzia PMD w Mavenie 691

23. Wstępne wykrywanie błędów za pomocą narzędzia FindBugs 693
23.1. FindBugs jako wyspecjalizowany zabójca błędów 693
23.2.Stosowanie narzędzia FindBugs w środowisku Eclipse 695
23.3. Wybiórcze zawieszanie stosowania reguł za pomocą filtrów narzędzia FindBugs 697
23.4. Stosowanie adnotacji narzędzia FindBugs 698

23.5. Korzystanie z narzędzia FindBugs w Ancie 700
23.6. Korzystanie z narzędzia FindBugs w Mavenie 702
23.7. Konkluzja 704

24. Analiza wyników — półautomatyczne przeglądy kodu za pomocą narzędzia Jupiter 705
24.1. Wprowadzenie do Jupitera — narzędzia do przeglądania kodu w środowisku Eclipse 705
24.2. Instalacja narzędzia Jupiter w środowisku Eclipse 706
24.3. Zrozumieć proces przeglądów kodu narzędzia Jupiter 706
24.4. Prowadzenie przeglądów własnego kodu 708
24.5. Konfiguracja 709
24.6. Ustawianie domyślnych wartości konfiguracyjnych 713
24.7. Przeglądy indywidualne 714
24.8. Przeglądy zespołowe 716
24.9. Faza wprowadzania poprawek 719
24.10. Wewnętrzne działania Jupitera 719
24.11. Konkluzja 721

25. Koncentrujmy się na tym, co naprawdę ważne — narzędzie Mylyn 723
25.1. Wprowadzenie do narzędzia Mylyn 723
25.2. Instalacja rozszerzenia Mylyn 724
25.3. Śledzenie zadań i problemów 725
25.4. Korzystanie z repozytoriów zadań 727
25.5. Koncentrowanie się na wybranych zadaniach z wykorzystaniem mechanizmów zarządzania kontekstami 731
25.6. Korzystanie ze zbiorów zmian środowiska Eclipse 734
25.7. Współdzielenie kontekstu z pozostałymi programistami 736
25.8. Konkluzja 737

26. Monitorowanie statystyk kompilacji 739
26.1. Wprowadzenie 739
26.2. Narzędzie QALab 739
26.3. Mierzenie ilości kodu źródłowego za pomocą modułu rozszerzenia StatSCM 747
26.4. Statystyki narzędzia StatSVN w Ancie 748

VII Narzędzia do zarządzania problemami 751

27. Bugzilla 753
27.1. Wprowadzenie do narzędzia Bugzilla 753
27.2. Instalacja narzędzia Bugzilla 753
27.3. Konfigurowanie środowiska narzędzia Bugzilla 757

27.4. Zarządzanie kontami użytkowników 758
27.5. Ograniczanie dostępu do bazy danych z wykorzystaniem grup użytkowników 760
27.6. Konfigurowanie produktu 762
27.7. Śledzenie postępu z wykorzystaniem tzw. kamieni milowych 764
27.8. Zarządzanie grupami produktów z wykorzystaniem klasyfikacji 764
27.9. Przeszukiwanie błędów 765
27.10. Tworzenie nowego błędu 767
27.11. Cykl życia błędu reprezentowanego w systemie Bugzilla 768
27.12. Tworzenie harmonogramu rozsyłania powiadomień (pojękiwania) 770
27.13. Dostosowywanie pól systemu Bugzilla do potrzeb konkretnego projektu 771
27.14. Konkluzja 772

28. Trac — lekkie zarządzanie projektami 773
28.1. Wprowadzenie do narzędzia Trac 773
28.2. Instalacja narzędzia Trac 774
28.3. Definiowanie projektu narzędzia Trac 776
28.4. Uruchamianie narzędzia Trac w formie autonomicznego serwera 778
28.5 Konfiguracja polecenia tracd jako usługi systemu Windows 779
28.6. Instalacja narzędzia Trac na serwerze Apache 780
28.7. Administrowanie witryną internetową Traca 781
28.8. Zarządzanie kontami użytkowników 783
28.9. Dostosowywanie witryny internetowej narzędzia Trac — korzystanie z funkcji witryn typu wiki 786
28.10. Stosowanie systemu zarządzania biletami Traca 790
28.11. Aktualizowanie błędów reprezentowanych w narzędziu Trac na podstawie zawartości repozytorium systemu Subversion 794
28.12. Modyfikowanie pól biletów Traca 795
28.13. Konfigurowanie powiadomień wysyłanych pocztą elektroniczną 797
28.14. Raportowanie z wykorzystaniem zapytań i raportów Traca 797
28.15. Zarządzanie postępami prac za pomocą map drogowych i diagramów linii czasu 800
28.16. Przeglądanie repozytorium z kodem źródłowym 802
28.17. Stosowanie kanałów RSS i formatu iCalendar 802
28.18. Dostosowywanie stron witryny wiki za pomocą skryptów Pythona 805
28.19. Konkluzja 806

VIII Narzędzia do dokumentacji technicznej 807

29. Komunikacja w ramach zespołu projektowego za pośrednictwem witryny Mavena 2 809
29.1. Witryna internetowa Mavena 2 jako narzędzie komunikacyjne 809
29.2. Konfigurowanie mechanizmu generowania witryny o projekcie Mavena 810

29.3. Włączanie do witryny Mavena raportów generowanych przez inne narzędzia 815
29.4. Tworzenie dedykowanego projektu witryny Mavena 819
29.5. Definiowanie szkicu witryny 821
29.6. Architektura mechanizmu generującego witryny Mavena 822
29.7. Stosowanie fragmentów kodu 826
29.8. Modyfikowanie wyglądu i sposobu obsługi witryny Mavena 827
29.9. Udostępnianie witryny 830

30. Automatyczne generowanie dokumentacji technicznej 833
30.1. Wprowadzenie 833
30.2. Wizualizacja struktury bazy danych za pomocą narzędzia SchemaSpy 833
30.3. Generowanie dokumentacji kodu źródłowego za pomocą Doxygena 841
30.4. Umieszczanie diagramów notacji UML w dokumentacji narzędzia Javadoc z wykorzystaniem narzędzia UmlGraph 850
30.5. Konkluzja 854

Bibliografia 855

Skorowidz.... 857

ROZDZIAŁ 10.

Testowanie kodu z wykorzystaniem frameworku JUnit

10.1. Frameworki JUnit 3.8 i JUnit 4

JUnit w chwili wprowadzenia na rynek był naprawdę rewolucyjnym oprogramowaniem — od tego czasu powstało mnóstwo przydatnych rozszerzeń tego frameworku ułatwiających nam wykonywanie testów jednostkowych w najbardziej wyspecjalizowanych obszarach. Wiele z tych rozszerzeń do tej pory bazuje na frameworku JUnit 3.*x*. Kilka takich rozszerzeń omówimy w dalszej części tej książki. W niniejszym podrozdziale spróbujemy sobie przypomnieć framework 3.8, aby lepiej rozumieć dalszy materiał poświęcony zmianom wprowadzonym w nowszych frameworkach, jak JUnit 4 czy TestNG (patrz rozdział 20.).

We frameworku JUnit 3 pisane przez nas testy jednostkowe mają postać klas Javy określanych mianem przypadków testowych. Wszystkie przypadki testowe tego frameworku muszą rozszerzać klasę `TestCase`. Testy jednostkowe implementujemy w formie metod tych klas — definiując te metody, musimy przestrzegać specjalnych konwencji nazewniczych: metody testowe muszą zwracać `void`, nie mogą pobierać żadnych parametrów, a ich nazwy muszą się rozpoczynać od słowa `test`. Także nazwy klas testowych muszą być zgodne z prostą konwencją — nazwa każdej takiej klasy musi się kończyć słowem `Test`.

Poniżej przedstawiono prostą klasę testową frameworku JUnit 3.8 testującą inną klasę, która z kolei odpowiada za obliczanie podatku od wartości dodanej (ang. *Value Added Tax — VAT*), nazywanego też podatkiem od towarów i usług. Przyjmijmy, że podstawowa stawka podatku VAT wynosi 22 procent. Nasz klasa testu jednostkowego może mieć następującą postać:

```
public class PriceCalculatorTest extends TestCase {

    public void testCalculateVAT() {
       calculator = new PriceCalculator();
       double amountWithVat = calculator.calculatePriceWithVAT(100.00);
       assertEquals("Podstawowa stawka VAT wynosi 22%", 122.00, amountWithVat, 0.0);
    }
}
```

Klasa bazowa `TestCase` oferuje mnóstwo metod z rodziny `assert`: `assertEquals()`, `assert➥True()`, `assertNotNull()` i wiele innych. Właśnie wymienione metody składają się na jądro testów jednostkowych, ponieważ za ich pośrednictwem wykonujemy nasze testy. Metody `assert` służą do sprawdzania, czy uzyskiwane wyniki są zgodne z wartościami oczekiwanymi.

Za pośrednictwem pierwszego parametru metody `assert` możemy przekazać opcjonalny komunikat, który w przyszłości powinien nam ułatwić identyfikację błędu (szczególnie jeśli korzystamy z dużej liczby testów jednostkowych).

Metody `setUp()` i `tearDown()` (zwróćmy uwagę na wielkie litery!) można przykryć wersjami odpowiednio inicjalizującymi i przywracającymi (przed i po każdym teście) stan środowiska testowego, w którym wykonujemy nasz kod. Jeśli na przykład korzystamy z wielu przypadków testowych operujących na obiekcie `calculator`, możemy zdecydować o jego jednorazowym utworzeniu w kodzie metody `setUp()`:

```
public class PriceCalculatorTest extends TestCase {

    PriceCalculator calculator;

    protected void setUp() throws Exception {
        calculator = new PriceCalculator();
    }

    public void testCalculateVAT() {
        double amountWithVat = calculator.calculatePriceWithVAT(100.00);
        assertEquals("Podstawowa stawka VAT wynosi 22%", 122.00, amountWithVat, 0.0);
    }
    // Pozostałe testy obiektu calculator...
}
```

Możliwości frameworku JUnit 3 oczywiście nie ograniczają się do zaprezentowanych mechanizmów, jednak uzyskana wiedza o architekturze tego frameworku powinna w zupełności wystarczyć do zrozumienia innowacji wprowadzonych w nowszych frameworkach i rozszerzeń frameworku JUnit 3 omawianych w pozostałych rozdziałach. Framework JUnit 4 pod wieloma względami przewyższa framework JUnit 3, jednak wersja 3.8 wciąż cieszy się dużą popularnością, a wiele atrakcyjnych modułów rozszerzeń nadal nie doczekało się aktualizacji do wersji 4. W kolejnych podrozdziałach tego rozdziału skoncentrujemy się wyłącznie na frameworku JUnit 4.

10.2. Testowanie jednostkowe z wykorzystaniem frameworku JUnit 4

W świecie frameworków testów jednostkowych JUnit jest de facto standardem. Jest powszechnie stosowany i doskonale znany niemal każdemu programiście. JUnit oferuje też wiele przydatnych rozszerzeń stworzonych z myślą o bardziej wyspecjalizowanych procesach testowych. Framework JUnit (w oryginalnej wersji autorstwa Kenta Becka i Ericha Gammy) jest uważany za rozwiązanie, które (przynajmniej teoretycznie) spopularyzowało praktyki testów jednostkowych wśród programistów Javy. Okazuje się jednak, że wskutek spadku dynamiki zmian wprowadzanych w podstawowym interfejsie API w ostatnich latach powstało i zyskało popularność kilka innych, jeszcze bardziej innowacyjnych frameworków, na przykład TestNG (patrz rozdział 20.).

JUnit 3 nakłada na programistów wiele ograniczeń, które nie znajdują żadnego uzasadnienia w dobie Javy 5, adnotacji i paradygmatu odwrócenia sterowania (ang. *Inversion of Control — IoC*). We frameworku JUnit 3 klasy testów muszą rozszerzać klasę bazową samego frameworku JUnit, a testy muszą być definiowane zgodnie ze specjalnymi konwencjami nazewnictwa — nie możemy użyć w roli klasy testu dowolnej klasy Javy. Klasy testów frameworku JUnit 3

są inicjalizowane za każdym razem, gdy wykonujemy jakiś test, co znacznie utrudnia refaktoryzację i optymalizację kodu testowego. JUnit 3 w żaden sposób nie wspiera na przykład testowania sterowanego danymi (czyli wykonywania testów na danych pochodzących z zewnątrz). We frameworku JUnit 3 brakuje też takich mechanizmów jak funkcje zarządzania zależnościami pomiędzy testami czy grupami testów.

JUnit 4 jest niemal całkowicie przebudowanym interfejsem API JUnit, który ma na celu wykorzystanie postępu obserwowanego w świecie technologii Javy w ciągu ostatnich kilku lat. Framework JUnit 4 jest prostszy, łatwiejszy w użyciu i bardziej elastyczny od swojego poprzednika; oferuje też kilka nowych funkcji! JUnit 4 wprowadza mnóstwo nowych mechanizmów, które mogą nam znacznie ułatwić pisanie testów jednostkowych, w tym obsługę adnotacji i bardziej elastyczny model inicjalizacji klas testów. We frameworku JUnit test może mieć postać dowolnej klasy Javy, a metody testów nie muszą być zgodne z żadnymi konwencjami nazewniczymi.

Sprawdźmy więc, jak nasze testy kalkulatora podatkowego (patrz podrozdział 10.1) wyglądałyby we frameworku JUnit 4:

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class PriceCalculatorTest {

   @Test
   public void calculateStandardVAT() {
     PriceCalculator calculator = new PriceCalculator();
     double vat = calculator.calculatePriceWithVAT(100.00);
     assertEquals(vat, 122.00 , 0.0);
   }

   @Test
   public void calculateReducedVAT() {
     PriceCalculator calculator = new PriceCalculator();
     double vat = calculator.calculatePriceWithReducedVAT(100.00);
     assertEquals(vat, 105.00 , 0.0);
   }

}
```

Warto w pierwszej kolejności zwrócić uwagę na brak konieczności rozszerzania konkretnej klasy przez przypadki testowe frameworku JUnit 4 (takie wymaganie obowiązywało we frameworku JUnit 3). Podobnie jak TestNG, framework JUnit 4 wykorzystuje adnotacje do oznaczania metod, które powinny być traktowane jako testy jednostkowe. Za testy jednostkowe uważa się wszystkie metody oznaczone adnotacją `@Test`. JUnit 4 co prawda nie narzuca nam żadnej konwencji nazewniczej (metody testów nie muszą się rozpoczynać od słowa `test`, jak `testThis()` czy `testThat()`), ale wymaga, by metody testów jednostkowych zwracały `void` i nie pobierały żadnych parametrów. Teoretycznie można by nawet umieszczać testy jednostkowe w tej samej klasie, w której znajduje się testowany kod, jednak w praktyce lepszym rozwiązaniem jest definiowanie kodu testowego w odrębnych klasach.

Klasa `org.junit.Assert` zawiera tradycyjne metody `assert` frameworku JUnit 3.*x*, do których zdążyliśmy się przyzwyczaić i które tak lubimy. We frameworku JUnit 3 metody `assert` były definiowane w klasie `TestCase`, czyli klasie bazowej dla wszystkich klas testów tego frameworku — dzięki temu można było z nich korzystać w dowolnych testach. Z zupełnie inną sytuacją mamy do czynienia w przypadku frameworku JUnit 4, gdzie klasy testów nie muszą dziedziczyć po klasie `TestCase`. Nie ma jednak powodów do zmartwień — możemy dla tej klasy

użyć operacji statycznego importowania, aby korzystać z niezbędnych klas `assert` (w tym `assertEquals`, `assertNotNull` itp.; patrz przykłady w dalszej części tego rozdziału) w dokładnie taki sam sposób jak w testach jednostkowych frameworku JUnit 3.*x*.

Alternatywnym rozwiązaniem jest stosowanie wyrażeń `assert` dostępnych w Javie 5:

```
assert (vat == 100*PriceCalculator.DEFAULT_VAT_RATE);
```

Wyrażenie w tej formie sprawia wrażenie bardziej eleganckiego, jednak musimy pamiętać o pewnej pułapce — Java ignoruje nasze wyrażenia `assert`, chyba że w wierszu poleceń użyjemy opcji `-ea` (od ang. *enable assertions*).

10.3. Konfigurowanie i optymalizacja przypadków testów jednostkowych

Jak każdy kod źródłowy, testy jednostkowe wymagają efektywnego kodowania i — w razie konieczności — refaktoryzacji. Framework JUnit 4 oferuje kilka adnotacji, które mogą nam to zadanie bardzo ułatwić. Adnotacja `@Before` wskazuje metodę, która musi być wywołana przed każdym testem, czyli w praktyce zastępuje znaną z frameworku JUnit 3.*x* metodę `setup()`. Możemy też użyć adnotacji `@After` do wskazania metod przywracających stan środowiska testowego po każdym wykonanym teście. W tym przypadku metoda `initialize()` będzie wywoływana przed, a metoda `tidyup()` po każdym teście jednostkowym:

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class PriceCalculatorTest {

   private PriceCalculator calculator;

   @Before
   public void initialize() {
     calculator = new PriceCalculator();
   }

   @Test
   public void calculateStandardVAT() {
     PriceCalculator calculator = new PriceCalculator();
     double vat = calculator.calculatePriceWithVAT(100.00);
     assertEquals(vat, 122.00 , 0.0);
   }

   @Test
   public void calculateReducedVAT() {
     PriceCalculator calculator = new PriceCalculator();
     double vat = calculator.calculatePriceWithReducedVAT(100.00);
     assertEquals(vat, 105 , 0.0);
   }

   @After
   public void tidyup() {
     calculator.close();
     calculator = null;
   }

}
```

Takie rozwiązanie wciąż nie jest optymalne. JUnit oferuje kilka innych adnotacji, których można z powodzeniem używać do dodatkowego doskonalenia kodu naszych testów jednostkowych. W pewnych sytuacjach warto poprawić efektywność testów przez skonfigurowanie niektórych zasobów przed wykonaniem któregokolwiek z testów jednostkowych zdefiniowanych w danej klasie i ich zwolnienie po zakończeniu wykonywania testów tej klasy. Cel ten można osiągnąć odpowiednio za pomocą adnotacji `@BeforeClass` i `@AfterClass`. Metody oznaczone adnotacją `@BeforeClass` zostaną wywołane tylko raz, przed wykonaniem któregokolwiek z testów jednostkowych definiowanych przez daną klasę. Jak łatwo się domyślić, metody oznaczone adnotacją `@AfterClass` zostaną wywołane dopiero po zakończeniu wszystkich testów. W powyższym przykładzie obiekt `calculator` zostałby utworzony tylko raz (na początku testów jednostkowych) i zniszczony dopiero po wykonaniu wszystkich testów. Klasę tę można uzupełnić o metodę `reset()` wywoływaną przed każdym testem jednostkowym i odpowiedzialną za każdorazowe ponowne inicjalizowanie testowanego obiektu `calculator`. Możliwy sposób implementacji tak zoptymalizowanej klasy testów jednostkowych przedstawiono poniżej:

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class PriceCalculatorTest {

   private PriceCalculator calculator;

   @BeforeClass
   public void initialize() {
     calculator = new PriceCalculator();
   }

   @Before
   public void resetCalculator() {
     calculator.reset();
   }

   @Test
   public void calculateStandardVAT() {
     PriceCalculator calculator = new PriceCalculator();
     double vat = calculator.calculatePriceWithVAT(100.00);
     assertEquals(vat, 122.00 , 0.0);
   }

   @Test
   public void calculateReducedVAT() {
     PriceCalculator calculator = new PriceCalculator();
     double vat = calculator.calculatePriceWithReducedVAT(100.00);
     assertEquals(vat, 105 , 0.0);
   }

   @AfterClass
   public void tidyup() {
     calculator.close();
   }

}
```

10.4. Proste testy wydajności z wykorzystaniem limitów czasowych

Jednym z najprostszych sposobów przeprowadzania testów wydajności jest sprawdzanie, czy określony test zawsze jest wykonywany w określonych ramach czasowych. Takie rozwiązanie bywa szczególnie przydatne w przypadku zapytań wykonywanych na bazie danych z użyciem takich narzędzi odwzorowań obiektowo-relacyjnych jak Hibernate. Nawet proste błędy w plikach odwzorowań tego narzędzia mogą skutkować znacznie wydłużonymi czasami odpowiedzi (także w przypadku stosunkowo prostych zapytań). W przeciwieństwie do tradycyjnego testu jednostkowego, test z określonym limitem czasowym umożliwia wykrywanie tego rodzaju błędów.

Tego rodzaju testy sprawdzają się także w roli mechanizmów wykrywających pętle nieskończone, chociaż wskazanie fragmentów kodu, które mogą zawierać tego rodzaju konstrukcje, jest oczywiście nieporównanie trudniejsze.

Opisaną technikę zintegrowano bezpośrednio z adnotacją `@Test`, która umożliwia ustawianie górnego limitu czasu, w którym dany test musi się zakończyć — w przeciwnym razie po upłynięciu tego czasu test kończy się błędem. W tym celu należy zdefiniować parametr `timeout` (reprezentujący limit czasowy wyrażony w milisekundach) adnotacji `@Test`:

```
@Test(timeout=100)
public void lookupVAT() {
   double vat = calculator.lookupRateForYear(2006);
   assertEquals(vat, VAT_RATE_IN_2006 , 0.0);
}
```

Jeśli użyte zapytanie zajmuje testowanej funkcji więcej niż 100 milisekund, nasz test kończy się niepowodzeniem:

```
Testsuite: com.wakaleo.jpt.alexandria.services.PriceCalculatorTest
Tests run: 3, Failures: 0, Errors: 1, Time elapsed: 0.136 sec

Testcase: calculateStandardVAT took 0.009 sec
Testcase: lookupVAT took 0.128 sec
        Caused an ERROR
test timed out after 100 milliseconds
java.lang.Exception: test timed out after 100 milliseconds
```

W przypadku niektórych metod, od których oczekujemy wysokiej wydajności i których efektywność ma kluczowe znaczenie dla funkcjonowania naszej aplikacji, warto dodatkowo sprawdzić, czy oferowana przepustowość spełnia nasze oczekiwania. Oczywiście im mniejsza będzie wartość limitu czasowego, tym większe będzie ryzyko wystąpienia sytuacji, w której jakiś czynnik zewnętrzny spowalniający nasze testy doprowadzi do nieuzasadnionego przekroczenia tego limitu. Na przykład w poniższym przypadku testowym sprawdzamy, czy średni czas wykonywania metody `calculateInterest()` nie przekracza milisekundy:

```
@Test(timeout=50)
public void perfTestCalculateInterest() {
    InterestCalculator calc = new InterestCalculatorImpl();
    for(int i = 0 ; i < 50; i++) {
        calc.calculateInterest(principal, interestRate, startDate, periodInDays);
    }
}
```

Tego rodzaju testy gwarantują nam, że uzyskiwane wyniki będą zbliżone do rzeczywistości i że badane metody nie są szczególnie powolne — nie powinniśmy być zbyt wymagający.

10.5. Prosta weryfikacja występowania wyjątków

W niektórych przypadkach warto sprawdzać, czy w określonych okolicznościach następuje prawidłowe generowanie wyjątków. We frameworku JUnit 3.*x* to dość pracochłonne zadanie wiąże się z koniecznością przechwytywania wyjątku — jeśli wyjątek uda się przechwycić, przyjmujemy, że test zakończył się pomyślnie; w przeciwnym razie test kończy się niepowodzeniem. We frameworku JUnit 4 mamy do dyspozycji parametr `expected` adnotacji `@Test`, któremu należy przypisać klasę oczekiwanego wyjątku (właśnie ten wyjątek powinien zostać wygenerowany zgodnie z naszym planem). W poniższym (dość mało realistycznym) przykładzie oczekujemy od aplikacji wygenerowania wyjątku `IllegalArgumentException`, jeśli dany rok jest mniejszy od przyjętego progu. We frameworku JUnit 4 odpowiedni test jest bardzo prosty:

```
@Test(expected = IllegalArgumentException.class)
public void lookupIllegalVATYear() {
   double vat = calculator.lookupRateForYear(1066);
}
```

Jeśli badana metoda nie wygeneruje wyjątku `IllegalArgumentException`, nasz test zakończy się niepowodzeniem:

```
Testsuite: com.wakaleo.jpt.alexandria.services.PriceCalculatorTest
Tests run: 3, Failures: 1, Errors: 0, Time elapsed: 0.114 sec

Testcase: calculateStandardVAT took 0.009 sec
Testcase: lookupVAT took 0.01 sec
Testcase: lookupIllegalVATYear took 0.003 sec
        FAILED
Expected exception: java.lang.IllegalArgumentException
junit.framework.AssertionFailedError: Expected exception:
java.lang.IllegalArgumentException
```

10.6. Stosowanie testów sparametryzowanych

Pisanie testów jednostkowych jest dość nużące, zatem wielu programistów próbuje iść na skróty. Okazuje się jednak, że od pewnych czynności nie uciekniemy — dobre testy jednostkowe muszą weryfikować działanie funkcji biznesowych dla rozmaitych danych, jak przypadki skrajne, klasy danych itp. Ten sam test może się zakończyć pomyślnie dla jednego zbioru danych, by chwilę później wykazać poważne błędy dla innego zbioru. Jeśli jednak programista musi napisać odrębny przypadek testowy dla każdej wartości (zgodnie z najlepszymi praktykami testowania), najprawdopodobniej jego kod będzie weryfikował stosunkowo niewielki zbiór wartości. Czyż nie byłoby wspaniale, gdybyśmy mogli wielokrotnie wykonywać ten sam test jednostkowy z wykorzystaniem różnych danych?

Okazuje się, że JUnit 4 oferuje dopracowany mechanizm ułatwiający nam testowanie kodu na dowolnych zbiorach danych. Za pomocą tego mechanizmu możemy zdefiniować kolekcję danych testowych i wymusić jej automatyczne wypełnianie w ramach naszych metod testów jednostkowych. Przeanalizujmy teraz prosty przykład. Przypuśćmy, że musimy napisać klasę wyznaczającą wysokość podatku dochodowego dla określonych dochodów we wskazanym roku. Interfejs naszej klasy biznesowej może mieć następującą postać:

```
public interface TaxCalculator {
    public double calculateIncomeTax(int year, double taxableIncome);
}
```

Wyznaczanie podatku dochodowego z reguły wymaga wykonywania kilku niełatwych obliczeń. W większości krajów stosuje się system podatków progresywnych, gdzie stawki podatkowe rosną wraz ze wzrostem opodatkowanych dochodów. Stawki definiuje się dla odrębnych przedziałów dochodów. Co więcej, same progi podatkowe (a więc także przedziały dochodów) nierzadko są zmieniane w kolejnych latach. W przypadku aplikacji odpowiedzialnej za tego rodzaju obliczenia niezwykle ważne jest przetestowanie wartości z każdego przedziału, a także przypadków skrajnych. W tej sytuacji powinniśmy opracować kolekcję danych testowych obejmujących możliwie wiele dochodów, lat i oczekiwanych obciążeń podatkowych. Sprawdźmy, jak można to zrobić.

JUnit 4 umożliwia nam definiowanie zbiorów danych testowych, które można następnie przekazywać do naszych testów jednostkowych. W tym przypadku musimy przetestować różne dochody podlegające opodatkowaniu w różnych przedziałach podatkowych. W prezentowanym przykładzie skoncentrujemy się tylko na roku 2006, jednak w rzeczywistej aplikacji powinniśmy poddać testom wiele lat podatkowych. Nasze zbiory testowe będą więc zawierać po trzy wartości: opodatkowane dochody, rok podatkowy oraz prawidłową wysokość podatku dochodowego.

Korzystanie z tych danych testowych wymaga skonfigurowania sparametryzowanej klasy testowej. Może to być zwykła klasa testowa z konstruktorem otrzymującym na wejściu kilka parametrów, a konkretnie po jednym parametrze dla każdej wartości naszego zbioru danych. Oznacza to, że w analizowanym przypadku wspomniany konstruktor będzie pobierał trzy parametry: opodatkowane dochody, rok podatkowy i oczekiwaną wysokość podatku dochodowego. Sparametryzowana klasa testowa z reguły obejmuje zmienne składowe reprezentujące każde z tych pól. Za inicjalizację tych pól odpowiada konstruktor, a właściwe metody testów jednostkowych wykorzystują je w czasie testowania.

JUnit tworzy odrębny obiekt naszej klasy testów dla każdego wiersza danych testowych, po czym wykonuje na tych danych testy jednostkowe (metody) tej klasy. Oznacza to, że jeśli nasze dane testowe obejmują 20 wierszy, JUnit utworzy obiekt naszej klasy 20 razy i każdorazowo wykona testy jednostkowe na innym wierszu tego zbioru danych.

Sprawdźmy teraz, jak można ten mechanizm zaimplementować. Kompletny kod naszej klasy testowej (dla fikcyjnych progów podatkowych) przedstawiono poniżej:

```
@RunWith(Parameterized.class)
public class TaxCalculatorTest {

    @Parameters
    public static Collection data() {
        return Arrays.asList(new Object[][]{
                /*  Dochód   Rok   Podatek */
                {     0.00, 2006,     0.00},
                { 10000.00, 2006,  1950.00},
                { 20000.00, 2006,  3900.00},
                { 38000.00, 2006,  7410.00},
                { 38001.00, 2006,  7410.33},
                { 40000.00, 2006,  8070.00},
                { 60000.00, 2006, 14670.00},
                {100000.00, 2006, 30270.00},
        });
    }
```

```
    private double revenue;
    private int    year;
    private double expectedTax;

    public TaxCalculatorTest(double input, int year, double expectedTax) {
        this.revenue = revenue;
        this.year = year;
        this.expectedTax = expectedTax;
    }

    @Test public void calculateTax() {
        TaxCalculator calculator = getTaxCalculator();
        double calculatedTax = calculator.calculateIncomeTax(year, revenue);
        assertEquals(expectedTax, calculatedTax);
    }

    private TaxCalculator getTaxCalculator() {
        TaxCalculator calculator = new TaxCalculatorImpl();
        return calculator;
    }
}
```

Przeanalizujmy teraz poszczególne fragmenty tej klasy. Po pierwsze, musimy użyć adnotacji `@RunWith` wskazującej na klasę `Parameterized`, aby zasygnalizować frameworkowi JUnit, że nasza klasa testowa zawiera sparametryzowane przypadki testowe:

```
@RunWith(Parameterized.class)
public class TaxCalculatorTest {...
```

Musimy teraz sporządzić kolekcję naszych danych testowych. W tym celu definiujemy funkcję oznaczoną adnotacją `@Parameters` i zwracającą dane testowe w formie kolekcji. Dane testowe wewnętrznie często mają postać listy tablic. W naszym przypadku dane testowe przyjmują formę listy tablic wartości, gdzie każda tablica obejmuje trzy elementy: dochód, rok i oczekiwaną wysokość podatku dochodowego (od danego dochodu osiągniętego we wskazanym roku podatkowym):

```
    @Parameters
    public static Collection data() {
        return Arrays.asList(new Object[][]{
                /*  Dochód   Rok   Podatek */
                {     0.00, 2006,     0.00},
                { 10000.00, 2006,  1950.00},
                { 20000.00, 2006,  3900.00},
                { 38000.00, 2006,  7410.00},
                { 38001.00, 2006,  7410.33},
                { 40000.00, 2006,  8070.00},
                { 60000.00, 2006, 14670.00},
                {100000.00, 2006, 30270.00},
        });
    }
```

Jak już wspomniano, kiedy framework JUnit 4 wykonuje naszą klasę testową, w rzeczywistości tworzy po jednym obiekcie tej klasy dla każdego wiersza kolekcji danych testowych. W tej sytuacji musimy zdefiniować zmienne składowe reprezentujące te wartości, a także konstruktor publiczny odpowiedzialny za ich inicjalizację, aby framework JUnit mógł tworzyć kolejne obiekty z właściwymi danymi testowymi:

```
    private double revenue;
    private int    year;
    private double expectedTax;
```

```
    public TaxCalculatorTest(double revenue, int year, double expectedTax) {
        this.revenue = revenue;
        this.year = year;
        this.expectedTax = expectedTax;
    }
```

Możemy teraz przetestować nasz kod z wykorzystaniem tych wartości:

```
@Test
public void calculateTax() {
    TaxCalculator calculator = getTaxCalculator();
    double calculatedTax = calculator.calculateIncomeTax(year, revenue);
    assertEquals(expectedTax, calculatedTax);
}
```

Kiedy uruchomimy te testy jednostkowe, okaże się, że nasze testy zostaną wykonane wielokrotnie — osobno dla każdego wiersza użytych danych testowych:

```
Testsuite: com.wakaleo.jpt.alexandria.services.TaxCalculatorTest
Tests run: 8, Failures: 0, Errors: 0, Time elapsed: 0.119 sec

Testcase: calculateTax[0] took 0.012 sec
Testcase: calculateTax[1] took 0.001 sec
Testcase: calculateTax[2] took 0.002 sec
Testcase: calculateTax[3] took 0.001 sec
Testcase: calculateTax[4] took 0.001 sec
Testcase: calculateTax[5] took 0.001 sec
Testcase: calculateTax[6] took 0.002 sec
Testcase: calculateTax[7] took 0.003 sec
```

Warto pamiętać o możliwości umieszczania wielu testów jednostkowych w jednej sparametryzowanej klasie testów (podobnie jak w przypadku tradycyjnych klas testów jednostkowych). Każda metoda testu jednostkowego będzie wywoływana osobno dla każdego wiersza danych testowych.

10.7. Stosowanie metody assertThat() i biblioteki Hamcrest

We frameworku JUnit 4.4 wprowadzono nowe pojęcie dla wyrażeń asercji, aby intencje programistów były bardziej zrozumiałe i łatwiejsze w interpretacji. Opisywana koncepcja, której oryginalnym pomysłodawcą był Joe Walnes[1], sprowadza się do stosowania metody `assertThat` łącznie ze zbiorem wyrażeń dopasowujących (określanych też mianem ograniczeń lub predykatów), co w wielu przypadkach znacznie poprawia czytelność testów. Na przykład poniższa klasa sprawdza, czy w danej sytuacji testowana funkcja wyznacza zerowy podatek dochodowy:

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;

public class TaxCalculatorTest {

   @Test
   public void calculateTax() {
       TaxCalculator calculator = getTaxCalculator();
```

[1] Patrz *http://joe.truemesh.com/blog/000511.html*.

```
        double calculatedTax = calculator.calculateIncomeTax(2007, 0);
        assertThat(calculatedTax, is(0.0));
    }
}
```

Wywołanie `assertThat(calculatedTax, is(0.0))` jest dużo bardziej czytelne niż wywołanie `assertEquals(calculatedTax, 0.0, 0.0)`, choć oczywiście wszystko zależy od osobistych preferencji programisty. Sam uważam wywołanie w tej formie za bardziej naturalne. Jest krótsze i nie zmusza nas do podświadomego tłumaczenia samego wyrażenia `assertsEquals` na zdanie „no dobrze, zatem wyznaczany podatek musi być równy zero". W przypadku pierwszego wyrażenia nasz mózg od razu dochodzi do interpretacji: „świetnie, zakładamy, że podatek będzie zerowy", co zajmuje nieporównanie mniej czasu.

Bardziej czytelne testy oznaczają też większą niezawodność i łatwość w utrzymaniu. Jeśli interpretacja naszych testów jest prostsza, dużo łatwiej i szybciej możemy stwierdzić, czy są prawidłowe.

Wyrażenie dopasowujące `equalTo` (lub `is`, czyli jego skrócona forma) może być z powodzeniem wykorzystywane w roli bardziej czytelnej wersji metody `assertEquals`:

```
String result = "czerwony";

assertThat(result, equalTo("czerwony"));
```

Opisywane wyrażenia można też łączyć w bardziej złożone zadania. Możemy na przykład wykorzystać wyrażenie dopasowujące `anyOf` do sprawdzenia, czy zmienna `color` zawiera łańcuch `"czerwony"`, `"zielony"` lub `"niebieski"`:

```
assertThat(color, anyOf(is("czerwony"),is("zielony"),is("niebieski")));
```

W razie konieczności możemy skojarzyć z naszym testem opis, który dodatkowo ułatwi jego interpretację:

```
String color = "hebanowy";
assertThat("czarny to czarny", color, is("czarny"));
```

Powyższe wyrażenie spowoduje wygenerowanie komunikatu o błędzie uzupełnionego o nasz opis:

```
<<< FAILURE!
java.lang.AssertionError: czarny to czarny
Expected: "czarny"
     got: "hebanowy"
...
```

Możemy też użyć intuicyjnego wyrażenia dopasowującego `not`, które neguje wszystkie pozostałe wyrażenia dopasowujące:

```
String color = "czarny";
assertThat(color, is(not(("biały"))));
```

Te nowe metody w rzeczywistości pochodzą z zewnętrznej biblioteki nazwanej Hamcrest. Wachlarz wyrażeń dopasowujących oferowanych w ramach frameworku JUnit 4.4 jest dość ograniczony. Można jednak ten zbiór uzupełnić, dołączając do realizowanego projektu bibliotekę *hamcrest-all.jar*. Wspomniany interfejs API można pobrać z witryny internetowej biblioteki Hamcrest[2]. Jeśli korzystamy z Mavena, możemy po prostu dodać odpowiednią referencję do pliku POM:

[2] Patrz *http://code.google.com/p/hamcrest/downloads/list*.

```
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-all</artifactId>
    <version>1.1</version>
    <scope>test</scope>
</dependency>
```

W ten sposób zastępujemy statyczne wyrażenie importujące bibliotekę `org.hamcrest.Core ↪Matchers` wyrażeniem importującym bardziej rozbudowaną bibliotekę `org.hamcrest. ↪Matchers`. Prezentowane rozwiązanie daje nam dostęp do znacznie bogatszego zbioru wyrażeń dopasowujących. Niektóre z tych dodatkowych wyrażeń zostaną omówione w dalszej części tego podrozdziału.

Do najbardziej interesujących wyrażeń dopasowujących należą mechanizmy upraszczające operacje na kolekcjach. Na przykład wyrażenie `hasItem` można z powodzeniem wykorzystywać do przeszukiwania zawartości struktury typu `List` (w przypadku struktur tablicowych ten sam efekt można uzyskać, stosując wyrażenie `hasItemInArray`):

```
List<String> colors = new ArrayList<String>();
colors.add("czerwony");
colors.add("zielony");
colors.add("niebieski");
...
assertThat(colors, hasItem("czerwony"));
```

Wyrażeń dopasowujących `hasItem` i `hasItemInArray` można używać do konstruowania skomplikowanych testów operujących na listach wartości. Poniżej przedstawiono przykład sprawdzania, czy dana lista nie zawiera żadnych elementów:

```
List<Integer> ages = new ArrayList<Integer>();
ages.add(20);
ages.add(30);
ages.add(40);
...
assertThat(ages, not(hasItem(lessThan(18))));
```

I odwrotnie, wyrażenie dopasowujące `isIn` umożliwia nam sprawdzanie, czy interesująca nas lista zawiera konkretny obiekt:

```
assertThat(20, isIn(ages));
```

Obsługa kolekcji nie ogranicza się tylko do list. Wyrażeń dopasowujących `hasKey` i `hasValue` można używać do sprawdzania, czy dana mapa (struktura typu `Map`) zawiera odpowiednio interesujący nas klucz lub wartość:

```
Map map = new HashMap();
map.put("color", "czerwony");
...
assertThat(map, hasValue("czerwony"));
```

Istnieje nawet wyrażenie dopasowujące `hasProperty`, które umożliwia nam testowanie właściwości obiektów:

```
Client client = new Client();
client.setClientName("Janina");
...
assertThat(client, hasProperty("clientName", is("Janina")));
```

W tym podrozdziale dokonaliśmy przeglądu zaledwie kilku możliwych zastosowań tego rodzaju wyrażeń. Inne dostępne rozwiązania można znaleźć w dokumentacji najnowszej wersji tego API. Wyrażenia dopasowujące w tej formie umożliwiają nam tworzenie bardziej czytelnych

i łatwiejszych w utrzymaniu testów, co z kolei stwarza szansę lepszego, szybszego i prostszego kodowania naszych testów.

10.8. Teorie we frameworku JUnit 4

Inną nową i niezwykle przydatną (choć wciąż uważaną za element eksperymentalny) funkcją wprowadzoną we frameworku 4.4 jest pojęcie *teorii* (przypuszczenia). Teoria wyraża ogólne przekonanie, które pozostaje prawdziwe dla wielu (być może nieskończenie wielu) zbiorów danych. Wszelkie ograniczenia zbiorów danych, dla których stosuje się daną teorię, określa się mianem założeń.

Programista w pierwszej kolejności definiuje zbiór punktów danych na potrzeby testów swojej teorii. Punkt danych jest (z reguły stałym) elementem danych testowych identyfikowanym przez adnotację `@DataPoint`. Alternatywnym rozwiązaniem jest użycie zautomatyzowanych narzędzi analizujących nasz kod i automatycznie tworzących zbiory danych wzmacniających lub obalających teorię. Na przykład poniżej definiujemy prawidłowe wartości dla lat 2007 i 2008:

```
@DataPoint public static int YEAR_2007 = 2007;
@DataPoint public static int YEAR_2008 = 2008;
```

Możemy teraz użyć innego zbioru danych do zdefiniowania danych testowych wykorzystywanych w roli potencjalnych dochodów podatników:

```
@DataPoint public static double INCOME_1 = 0.0;
@DataPoint public static double INCOME_2 = 0.01;
@DataPoint public static double INCOME_3 = 100.0;
@DataPoint public static double INCOME_4 = 13999.99;
@DataPoint public static double INCOME_5 = 14000.0;
```

Aby zdefiniować test wykorzystujący teorię, należy w miejsce standardowej adnotacji `@Test` użyć adnotacji `@Theory`. Teoria jest zwykłą metodą otrzymującą na wejściu pewną liczbę parametrów. Framework sam określa, których punktów danych należy użyć dla poszczególnych parametrów naszych metod testowych, na podstawie ich typów. Każdy punkt danych jest przekazywany za pośrednictwem każdego parametru tego samego typu. Takie rozwiązanie stwarza pewne problemy, jeśli stosujemy wiele parametrów tego samego typu. Jak się za chwilę przekonamy, do ograniczania możliwych wartości przypisywanych poszczególnym parametrom służą tzw. założenia.

Kolejnym krokiem jest zdefiniowanie wspomnianych założeń za pomocą adnotacji `@assumeThat`. Stosując założenia w ramach przypadku testowego wykorzystującego teorię, możemy łatwo ograniczyć dane testowe, które będą używane podczas wykonywania tego przypadku testowego. W poniższym przykładzie ograniczamy zakres testów naszego przypadku do roku 2007 i dochodów z przedziału od 0 do 14 tys. złotych:

```
assumeThat(year, is(2007));
```

oraz

```
assumeThat(income, both(greaterThan(0.00)).and(lessThan(14000.00)));
```

Moduł rozszerzenia JUnitRunner wykonuje dany test dla wszystkich możliwych kombinacji punktów danych zgodnych z założeniami, czyli w tym przypadku dla kombinacji stałej `YEAR_2007` i stałych `INCOME_2`, `INCOME_3` oraz `INCOME_4`:

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;
import static org.junit.Assume.assumeThat;

import java.math.BigDecimal;

import org.junit.experimental.theories.DataPoint;
import org.junit.experimental.theories.Theories;
import org.junit.experimental.theories.Theory;
import org.junit.runner.RunWith;

@RunWith(Theories.class)
public class TaxCalculatorTheoryTest {

    @DataPoint public static int YEAR_2007 = 2007;
    @DataPoint public static int YEAR_2008 = 2008;
    @DataPoint public static BigDecimal INCOME_1 = new BigDecimal(0.0);
    @DataPoint public static double INCOME_2 = 0.01;
    @DataPoint public static double INCOME_3 = 100.0;
    @DataPoint public static double INCOME_4 = 13999.99;
    @DataPoint public static double INCOME_5 = 14000.0;

    @SuppressWarnings("unchecked")
    @Theory
    public void lowTaxRateIsNineteenPercent(int year, double income) {
        assumeThat(year, is(2007));
        assumeThat(income, allOf(greaterThan(0.00),lessThan(14000.00)));

        TaxCalculator calculator = getTaxCalculator();
        double calculatedTax = calculator.calculateIncomeTax(year, income);
        double expectedIncome = calculatedTax * 1000/195;
        assertThat(expectedIncome, closeTo(income,0.001));
        System.out.println("Rok: " + year + ", Dochód: " + income + ", Podatek: "
        + calculatedTax);
    }

    private TaxCalculator getTaxCalculator() {
        return new TaxCalculatorImpl();
    }
}
```

W wyniku wykonania tego kodu otrzymamy następujące dane:

```
Rok: 2007, Dochód: 0.01, Podatek: 0.0019500000000000001
Rok: 2007, Dochód: 100.0, Podatek: 19.5
Rok: 2007, Dochód: 13999.99, Podatek: 2729.99805
```

Dla uproszczenia wykorzystujemy wartości typu `double`. W prawdziwej aplikacji biznesowej prawdopodobnie należałoby użyć typu gwarantującego większą precyzję operacji na danych pieniężnych, czyli typu `BigDecimal` lub dedykowanej klasy `Money`.

W razie niepowodzenia tego testu zostanie wyświetlony opisowy komunikat obejmujący szczegóły punktów danych, które doprowadziły do błędu:

```
org.junit.experimental.theories.internal.ParameterizedAssertionError:
lowTaxRateIsNineteenPercent(2007, 0.01)
Caused by: java.lang.AssertionError:
    Expected: is <0.01>
    Got: is <0.0>
```

Możemy teraz dodać do tego testu inne teorie, aby zweryfikować inne podzbiory naszych danych testowych. Możliwy zbiór punktów danych (po zastosowaniu założeń) jest stosowany osobno dla każdej takiej teorii.

10.9. Stosowanie frameworku JUnit 4 w projektach Mavena 2

Maven 2 do wykonywania testów jednostkowych wykorzystuje moduł rozszerzenia Surefire (patrz podrozdział 2.13). Moduł rozszerzenia obsługuje testy jednostkowe zarówno frameworku JUnit 3, jak i frameworku JUnit 4 — klasy testów muszą się znajdować w katalogu *test*, a Maven automatycznie je wykrywa i uruchamia. Można nawet łączyć testy frameworków JUnit 3 i JUnit 4 w ramach tej samej aplikacji. Testy jednostkowe wykonujemy dokładnie tak samo jak pozostałe testy Mavena, czyli za pomocą polecenia `mvn test`:

```
$ mvn test
[INFO] Scanning for projects...
...
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
...
Results :

Tests run: 68, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4 seconds
[INFO] Finished at: Tue Aug 14 22:28:51 GMT+12:00 2007
[INFO] Final Memory: 7M/67M
[INFO] ------------------------------------------------------------------------
```

Polecenie `mvn test` wykonuje zarówno testy frameworku JUnit 3, jak i frameworku JUnit 4, po czym generuje standardowy zbiór raportów modułu rozszerzenia Surefire, obejmujący zebrane wyniki wszystkich naszych testów. Takie rozwiązanie jest bardzo korzystne w sytuacji, gdy chcemy korzystać z unikatowych funkcji frameworku JUnit 4 w normalnych testach jednostkowych i jednocześnie zachować możliwość stosowania kilku doskonałych bibliotek testujących napisanych dla frameworku JUnit 3, na przykład ze StrutsTestCase'a (patrz rozdział 19.), frameworku testowego Spring MVC lub rozszerzenia DBUnit.

10.10. Stosowanie frameworku JUnit 4 w projektach Anta

Obsługa frameworku JUnit 4 w wersjach Anta sprzed wydania 1.7.0 pozostawiała wiele do życzenia. Okazuje się jednak, że począwszy od wspomnianej wersji, testy frameworku JUnit 4 są w pełni obsługiwane i łatwe w konfiguracji. W tym podrozdziale przeanalizujemy kroki składające się na procesy konfiguracji, kompilacji i wykonywania testów JUnit 4 za pośrednictwem Anta.

Aby nasze rozważania były kompletne, przeanalizujemy cały skrypt kompilacji Anta. Większa część tego pliku powinna być zrozumiała dla programistów obeznanych z Antem (patrz rozdział 1.). W pierwszej części tego pliku definiujemy katalogi projektu i typowe zadania:

```
<project name="JUnit-Tests-Sample" default="runtests" basedir=".">
    <property name="junit.home"   value="/home/john/tools/junit4.1" />
    <property name="java.src"     value="src/main/java" />
    <property name="test.src"     value="src/test/java" />
    <property name="build.dir"    value="target" />
    <property name="java.classes" value="${build.dir}/classes" />
    <property name="test.classes" value="${build.dir}/test-classes" />
    <property name="test.reports" value="${build.dir}/test-reports" />

    <target name="init">
        <mkdir dir="${java.classes}"/>
        <mkdir dir="${test.classes}"/>
        <mkdir dir="${test.reports}"/>
    </target>

    <target name="clean">
        <delete dir="${build.dir}"/>
    </target>
```

Następnie musimy zdefiniować zadanie odpowiedzialne za kompilację naszego kodu Javy:

```
    <target name="compile" depends="init" >
        <javac srcdir="${java.src}" destdir="${java.classes}" >
            <include name="**/*.java"/>
        </javac>
    </target>
```

Także w tym przypadku mamy do czynienia ze standardowymi konstrukcjami skryptu kompilacji — ograniczamy się do kompilowania kodu Javy za pomocą standardowego zadania `<javac>`. Bardziej interesujące elementy można znaleźć w kolejnym fragmencie tego pliku, gdzie ustawiamy ścieżkę do klas wskazującą na plik JAR frameworku JUnit 4.1 i skompilowane klasy naszej aplikacji. Obie ścieżki wykorzystujemy następnie do skompilowania testów jednostkowych frameworku JUnit 4. Ponieważ framework JUnit 4 zapewnia zgodność wstecz z frameworkiem JUnit 3, testy jednostkowe napisane w obu tych interfejsach API można z powodzeniem stosować łącznie (bez ryzyka występowania konfliktów) w ramach tego samego projektu:

```
    <path id="test.classpath">
        <pathelement location="${junit.home}/junit-4.1.jar" />
        <pathelement location="${java.classes}" />
    </path>

    <target name="compiletests" depends="compile">
        <javac srcdir="${test.src}" destdir="${test.classes}">
            <classpath refid="test.classpath" />
            <include name="**/*.java"/>
        </javac>
    </target>
```

Jesteśmy wreszcie gotowi do właściwego uruchomienia naszych testów jednostkowych. Ant 1.7.0 oferuje nowe, udoskonalone zadanie stworzone z myślą o obsłudze zarówno testów frameworku JUnit 3, jak i testów frameworku JUnit 4. Typowe zadanie testu frameworku JUnit użyte w skrypcie kompilacji Anta 1.7.0 ma następującą postać:

```
    <target name="runtests" depends="compiletests">
        <junit printsummary="yes" haltonfailure="yes">
          <classpath>
              <path refid="test.classpath" />
              <pathelement location="${test.classes}"/>
          </classpath>
```

```
        <formatter type="plain"/>
        <formatter type="xml"/>

        <batchtest fork="yes" todir="${test.reports}">
          <fileset dir="${test.src}">
               <include name="**/*Test*.java"/>
          </fileset>
        </batchtest>
      </junit>
    </target>
```

Pierwszym interesującym elementem (przynajmniej z perspektywy użytkowników frameworku JUnit korzystających ze starszych wersji Anta) jest ścieżka do klas. Właśnie za pośrednictwem tego elementu sygnalizujemy Antowi, gdzie należy szukać pliku JAR frameworku JUnit 4. Warto o tym wspomnieć choćby dlatego, że aż do wydania Anta 1.6.5 użytkownicy zainteresowani korzystaniem z zadania frameworku JUnit musieli umieszczać kopię pliku *junit.jar* w katalogu *lib* Anta. Mimo że przytoczone wymaganie było udokumentowane w podręczniku użytkownika Anta i oficjalnie zadeklarowane jako zgodne z zamierzeniami twórców tego narzędzia, w najlepszym razie można je uznać za niefortunne. Począwszy od Anta 1.7.0, wystarczy zadeklarować plik JAR frameworku JUnit 4 w zagnieżdżonym elemencie `<class ➥path>`.

W kolejnej części należy zdefiniować listę obiektów formatujących. Wyniki testów można generować w wielu różnych formatach — opcja `plain` oznacza zwykłe pliki testowe, natomiast opcja `xml` oznacza bardziej szczegółowe raporty w popularnym formacie XML.

Za właściwe wykonywanie testów odpowiada element `<batchtest>`, który uruchamia wszystkie testy frameworku JUnit odnalezione we wskazanym zbiorze plików. W tym kontekście testy jednostkowe frameworku JUnit 3 są traktowane tak samo jak testy frameworku JUnit 4.

Po wywołaniu tego celu powinniśmy otrzymać dane wynikowe podobne do poniższych:

```
$ ant runtests
Buildfile: build.xml

init:

compile:
    [javac] Compiling 11 source files to
    /home/john/Documents/book/java-power-tools/src/sample-
code/alexandria/target/classes

compiletests:
    [javac] Compiling 4 source files to
    /home/john/Documents/book/java-power-tools/src/sample-code/alexandria/target/
    test-classes

runtests:
    [junit] Running com.wakaleo.jpt.alexandria.domain.CatalogTest
    [junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 4.493 sec
    [junit] Running com.wakaleo.jpt.alexandria.domain.LegacyJUnit3CatalogTest
    [junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0.041 sec
    [junit] Running com.wakaleo.jpt.alexandria.services.PriceCalculatorTest
    [junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 0.048 sec
    [junit] Running com.wakaleo.jpt.alexandria.services.TaxCalculatorTest
    [junit] Tests run: 8, Failures: 0, Errors: 0, Time elapsed: 0.054 sec

BUILD SUCCESSFUL
```

10.11. Selektywne wykonywanie testów frameworku JUnit 4 w Ancie

Zadanie `<junit>` Anta jest narzędziem wyjątkowo elastycznym — za jego pośrednictwem możemy między innymi wskazywać testy jednostkowe, które mają być wykonywane. W tym podrozdziale zostaną omówione rozmaite techniki wyboru takich testów.

Wykonywanie pojedynczych testów

Testy jednostkowe najczęściej wykonuje się całymi pakietami za pomocą elementu `<batchtest>`. Okazuje się jednak, że można te testy wykonywać także pojedynczo z użyciem elementu `<test>`:

```
<target name="runtest" depends="compiletests">
    <junit printsummary="yes" haltonfailure="yes">
        ...
        <test name="com.wakaleo.jpt.alexandria.domain.CatalogTest"/>
    </junit>
</target>
```

Wywołanie tego celu spowoduje wykonanie tylko testów jednostkowych zawartych we wskazanej klasie:

```
$ ant runtest
Buildfile: build.xml

init:

compile:

compiletests:

runtest:
    [junit] Running com.wakaleo.jpt.alexandria.domain.CatalogTest
    [junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 9.02 sec

BUILD SUCCESSFUL
```

Możemy też zdecydować o wyłączeniu jakiegoś testu ze zbioru głównych testów jednostkowych za pomocą elementu `<exclude>`:

```
<target name="runtests" depends="compiletests">
    <junit printsummary="yes" haltonfailure="yes">
      ...
      <batchtest fork="yes" todir="${test.reports}">
        <fileset dir="${test.src}">
            <include name="**/*Test*.java"/>
            <exclude name="**/CatalogTest.java"/>
        </fileset>
      </batchtest>
    </junit>
</target>
```

Warunkowe wykonywanie testów

W pewnych sytuacjach warto zrezygnować z wykonywania wszystkich klas testów przy okazji każdej procedury przeprowadzania testów jednostkowych. Niektóre rodzaje testów — w tym testy obciążeniowe, integracyjne i wydajnościowe — mogą być dość czasochłonne, zatem ich

wykonywanie po każdej kompilacji aplikacji bywa kłopotliwe. Testy jednostkowe powinny być krótkie i treściwe. Testy wymagające więcej czasu i mocy obliczeniowej procesora należy stosować tylko wtedy, gdy jest to naprawdę konieczne. Na komputerze programistów tego rodzaju testy są wykonywane tylko na żądanie; za ich systematyczne przeprowadzanie z reguły odpowiada serwer integracji.

Jednym ze sposobów realizacji tego celu jest użycie atrybutu `if` elementu `<batchtest>`. Atrybut `if` określa właściwość, której ustawienie jest warunkiem wykonania wskazanych testów jednostkowych; w przeciwnym przypadku testy zostaną po prostu pominięte.

Poniższy cel zostanie przetworzony, pod warunkiem że będzie ustawiona właściwość `perf` ↪`tests`:

```
<target name="runperftests" depends="compiletests">
    <junit printsummary="yes" haltonfailure="yes">
      ...
      <batchtest fork="yes" todir="${test.reports}" if="perftests">
        <fileset dir="${test.src}">
            <include name="**/*PerfTest*.java"/>
        </fileset>
      </batchtest>
    </junit>
</target>
```

Jeśli właściwość `perftests` nie zostanie ustawiona, testy wydajnościowe nigdy nie zostaną wykonane, nawet jeśli nasz cel zostanie wywołany wprost (według nazwy):

```
$ ant runperftests
Buildfile: build.xml

init:

compile:

compiletests:

runperftests:

BUILD SUCCESSFUL
Total time: 1 second
```

Jeśli jednak ustawimy właściwość `perftests` (przypisując jej dowolną wartość), testy wydajnościowe zostaną prawidłowo wykonane. Właściwości można ustawiać na wiele różnych sposobów: bezpośrednio w pliku kompilacji, w pliku właściwości ładowanym przez skrypt kompilacji (za pomocą zadania `<property>`) lub z poziomu wiersza poleceń:

```
$ ant runperftests -Dperftests=true
Buildfile: build.xml

init:

compile:

compiletests:

runperftests:
    [junit] Running com.wakaleo.jpt.alexandria.domain.CatalogPerfTest
    [junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 7.227 sec
    [junit] Running com.wakaleo.jpt.alexandria.services.PriceCalculatorPerfTest
    [junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 0.192 sec

BUILD SUCCESSFUL
```

Typowym zastosowaniem właściwości w tej formie (ustawianej z poziomu wiersza poleceń) jest stosowanie naszego zadania na serwerze integracyjnym — testy wydajnościowe i integracyjne z reguły wykonywane są tylko na tym serwerze, nie na komputerach programistów.

Stosując tę technikę, nie możemy zapominać o konieczności wyłączenia tych testów z głównego testu jednostkowego za pomocą omówionego wcześniej elementu `<exclude>`.

10.12. Testy integracyjne

Testy jednostkowe nie tylko powinny być krótkie i treściwe, ale też powinny generować interesujące nas wyniki możliwie szybko. Testy jednostkowe nie powinny korzystać z takich zasobów zewnętrznych jak bazy danych czy frameworki aplikacji internetowych. Właśnie dlatego często stosuje się interfejsy, obiekty zastępcze i rozmaite inne techniki gwarantujące, że każdy komponent będzie testowany niezależnie od pozostałych.

Prędzej czy później będziemy jednak chcieli sprawdzić, jak nasze komponenty ze sobą współpracują. Weryfikację tego aspektu projektu określa się mianem testów integracyjnych. Na tym etapie możemy sprawdzić obiekty DAO frameworku Hibernate, korzystając z prawdziwej bazy danych (zamiast z bazy wbudowanej), wykonać zapytania pokonujące całą drogę od warstwy usług do bazy danych i z powrotem lub zasymulować działanie przeglądarki użytkownika z użyciem specjalnego narzędzia, na przykład Selenium. Można też sprawdzić, jak nasza aplikacja radzi sobie z dużym obciążeniem i czy jest właściwie przygotowana do obsługi wielu jednoczesnych żądań. Tego rodzaju testy są oczywiście bardzo ważne, jednak z reguły okazują się zdecydowanie zbyt czasochłonne, aby programiści mogli je każdorazowo wykonywać wraz ze zwykłymi testami jednostkowymi. Zbyt wolne testy jednostkowe zniechęcają programistów do testowania, zatem powinniśmy znaleźć skuteczny sposób wyodrębnienia szybkich testów jednostkowych z grupy wolnych testów integracyjnych.

W Mavenie można tak skonfigurować moduł rozszerzenia Surefire, aby sam określał, które testy należy wykonywać w fazie testów jednostkowych (w odpowiedzi na polecenie *`mvn test`*), aktóre powinny być wykonywane na etapie testów integracyjnych (w odpowiedzi na polecenie *`mvn integration-test`*). W poniższym przykładzie nazwy testów integracyjnych kończą się wyrażeniem `IntegrationTest`. Mamy więc do czynienia z wyjątkowo prostą konwencją — w razie potrzeby można oczywiście zdefiniować alternatywną, własną konwencję. Poniższy plik konfiguracyjny wyłącza testy integracyjne ze zbioru zwykłych testów jednostkowych (skojarzonych z fazą `test`) i kojarzy je z odrębną fazą `integration-test`:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <executions>
          <execution>
            <id>unit-tests</id>
            <phase>test</phase>
```

```
            <goals>
              <goal>test</goal>
            </goals>
            <configuration>
              <excludes>
                  <exclude>**/*IntegrationTest.java</exclude>
              </excludes>
            </configuration>
          </execution>
          <execution>
            <id>integration-tests</id>
            <phase>integration-test</phase>
            <goals>
              <goal>test</goal>
            </goals>
            <configuration>
              <includes>
                  <include>**/*IntegrationTest.java</include>
              </includes>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Aby wykonać testy wydajnościowe, wystarczy wywołać fazę testów integracyjnych:

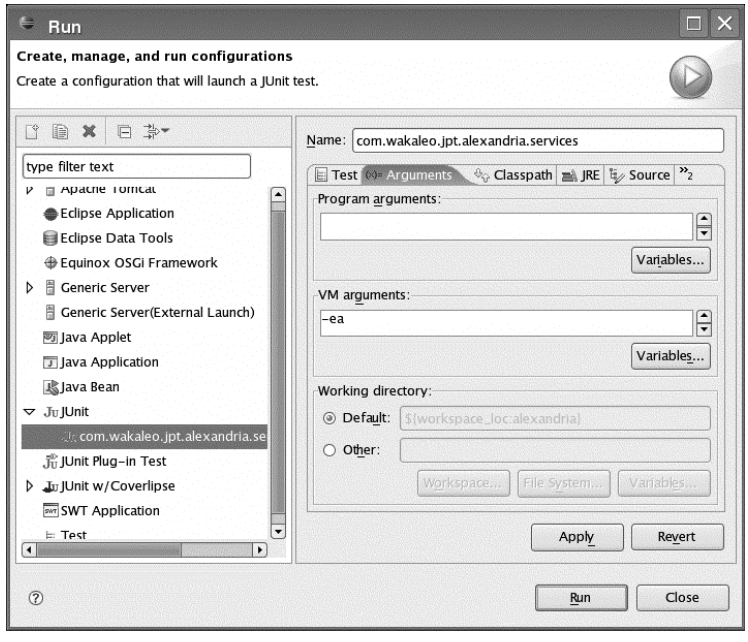
```
$ mvn integration-test
```

10.13. Korzystanie z frameworku JUnit 4 w środowisku Eclipse

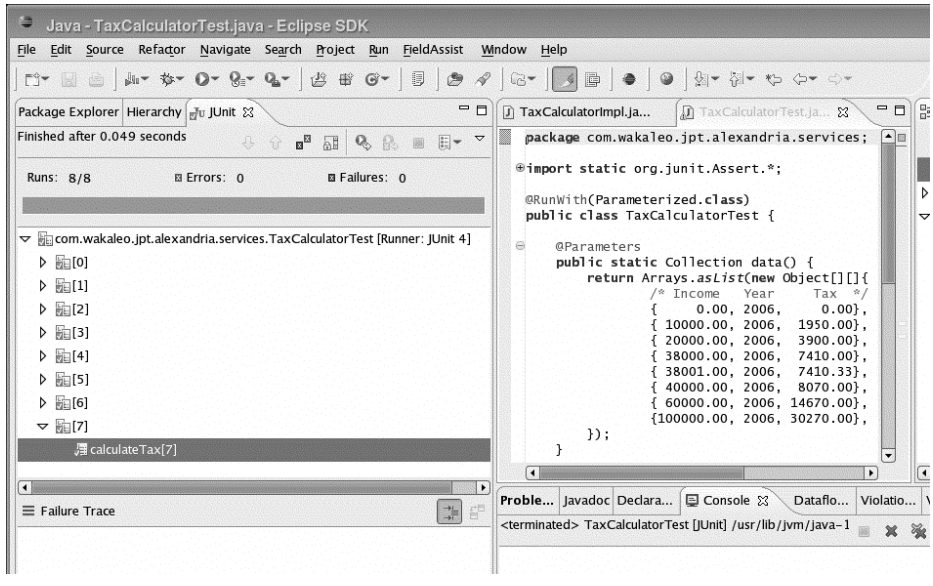
Korzystanie z pośrednictwa zintegrowanego środowiska wytwarzania (IDE) jest bodaj najprostszym i najbardziej efektywnym sposobem wykonywania testów jednostkowych. Framework JUnit 4 wprost doskonale integruje się ze środowiskiem Eclipse — testy tego frameworku można wywoływać w dokładnie taki sam sposób jak testy frameworku JUnit 3, czyli za pomocą opcji *Run As…* i *JUnit Test*. Jeśli korzystamy z asercji Javy 5, powinniśmy dodatkowo użyć opcji `-ea` (od ang. *enable assertions*) w oknie konfiguracyjnym *Run* (patrz rysunek 10.1). W przeciwnym razie nasze asercje zostaną po prostu zignorowane.

Co więcej, Eclipse prawidłowo obsługuje też takie mechanizmy frameworku JUnit 4 jak testy sparametryzowane (patrz rysunek 10.1).

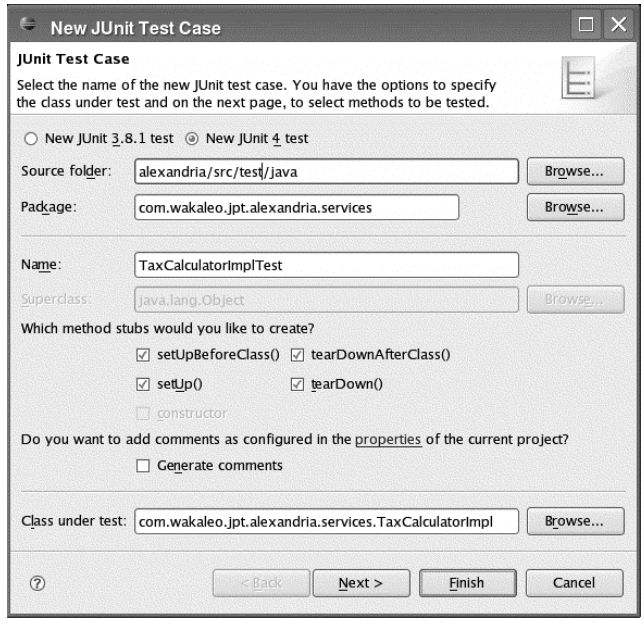
Środowisko Eclipse dodatkowo oferuje możliwość tworzenia nowych przypadków testowych frameworku JUnit 4 za pomocą opcji *New…* i *JUnit Unit Test* (patrz rysunek 10.3). Za pośrednictwem tego okna dialogowego możemy tworzyć klasy testów frameworków JUnit 3.8 lub JUnit 4 (oba typy testów jednostkowych można stosować jednocześnie w ramach tego samego projektu).



Rysunek 10.1. Konfigurowanie testów frameworku JUnit 4, aby korzystały z operacji asercji



Rysunek 10.2. Uruchamianie testów frameworku JUnit 4 z poziomu środowiska Eclipse



Rysunek 10.3. Tworzenie w środowisku Eclipse nowego testu frameworku JUnit 4