Craig Walls

Spring w akcji

Wydanie III

Kompletne źródło informacji o Spring Framework!

Tytuł oryginału: Spring in Action, 3rd edition

Tłumaczenie: Jacek Kowolik (wstęp, rozdz. 1 – 4);
Krzysztof Wołowski (rozdz. 5 – 14)

Projekt okładki: Anna Mitka
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-246-4888-7

Original edition copyright © 2011 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2013 by HELION SA.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

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

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

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

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/sprwa3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/sprwa3.zip

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Przedmowa *11*
Podziękowania *13*
O książce *15*

CZĘŚĆ 1. PODSTAWY FRAMEWORKA SPRING

Rozdział 1. Zrywamy się do działania 21

1.1. Upraszczamy programowanie w Javie 22
1.1.1. Uwalniamy moc zawartą w POJO 23
1.1.2. Wstrzykujemy zależności 25
1.1.3. Stosujemy aspekty 29
1.1.4. Ograniczamy stosowanie kodu szablonowego dzięki wzorcom 33
1.2. Kontener dla naszych komponentów 36
1.2.1. Pracujemy z kontekstem aplikacji 37
1.2.2. Cykl życia komponentu 37
1.3. Podziwiamy krajobraz Springa 39
1.3.1. Moduły Springa 39
1.3.2. Rodzina projektów wokół Springa 42
1.4. Co nowego w Springu 46
1.4.1. Co nowego w Springu 2.5? 46
1.4.2. Co nowego w Springu 3.0? 47
1.4.3. Co nowego w rodzinie projektów otaczających Springa? 47
1.5. Podsumowanie 48

Rozdział 2. Tworzymy powiązania między komponentami 49

2.1. Deklarujemy komponenty 50
2.1.1. Tworzymy konfigurację Springa 51
2.1.2. Deklarujemy prosty komponent 51
2.1.3. Wstrzykujemy przez konstruktory 53
2.1.4. Ustalamy zakres komponentów 58
2.1.5. Inicjalizujemy i likwidujemy komponenty 59
2.2. Wstrzykujemy wartości do właściwości komponentu 61
2.2.1. Wstrzykujemy proste wartości 62
2.2.2. Wstrzykujemy referencje do innych beanów 63
2.2.3. Wiążemy właściwości w przestrzeni nazw p Springa 66
2.2.4. Wiążemy kolekcje 67
2.2.5. Wiążemy nic (null) 72
2.3. Wiążemy za pomocą wyrażeń 72
2.3.1. Podstawy wyrażeń w SpEL 73
2.3.2. Wykonujemy operacje na wartościach w języku SpEL 76
2.3.3. Przesiewamy kolekcje za pomocą SpEL 80
2.4. Podsumowanie 83

Rozdział 3. Ograniczamy użycie języka XML w konfiguracji Springa 85

3.1. Automatycznie wiążemy właściwości komponentów ... 86
3.1.1. Cztery rodzaje automatycznego wiązania ... 86
3.1.2. Domyślny styl automatycznego wiązania ... 90
3.1.3. Mieszamy wiązanie automatyczne i jawne ... 91
3.2. Wiążemy za pomocą adnotacji ... 92
3.2.1. Korzystamy z adnotacji @Autowired ... 92
3.2.2. Stosujemy automatyczne wiązanie zgodne ze standardami, korzystając z adnotacji @Inject ... 97
3.2.3. Posługujemy się wyrażeniami podczas wstrzykiwania przez adnotacje ... 99
3.3. Automatyczne wykrywanie komponentów ... 100
3.3.1. Oznaczamy komponenty adnotacjami dla automatycznego wykrywania ... 100
3.3.2. Dodajemy filtry do skanera komponentów ... 102
3.4. Używamy w Springu konfiguracji opartej na Javie ... 103
3.4.1. Przygotowujemy się do stosowania konfiguracji opartej na Javie ... 103
3.4.2. Definiujemy klasę konfiguracyjną ... 104
3.4.3. Deklarujemy prosty komponent ... 104
3.4.4. Wstrzykujemy w Springu za pomocą konfiguracji opartej na Javie ... 105
3.5. Podsumowanie ... 106

Rozdział 4. Aspektowy Spring 107

4.1. Czym jest programowanie aspektowe ... 108
4.1.1. Definiujemy terminologię dotyczącą AOP ... 109
4.1.2. Obsługa programowania aspektowego w Springu ... 112
4.2. Wybieramy punkty złączenia za pomocą punktów przecięcia ... 115
4.2.1. Piszemy definicje punktów przecięcia ... 116
4.2.2. Korzystamy z desygnatora bean() w Springu ... 117
4.3. Deklarujemy aspekty w języku XML ... 117
4.3.1. Deklarujemy porady before i after ... 119
4.3.2. Deklarujemy poradę around ... 121
4.3.3. Przekazujemy parametry do porady ... 123
4.3.4. Wprowadzamy nową funkcjonalność przez aspekty ... 125
4.4. Używamy adnotacji dla aspektów ... 127
4.4.1. Tworzymy poradę around za pomocą adnotacji ... 129
4.4.2. Przekazujemy argumenty do porad konfigurowanych przez adnotacje ... 130
4.4.3. Tworzymy wprowadzenie za pomocą adnotacji ... 131
4.5. Wstrzykujemy aspekty z AspectJ ... 132
4.6. Podsumowanie ... 135

CZĘŚĆ 2. PODSTAWY APLIKACJI SPRINGA

Rozdział 5. Korzystanie z bazy danych 139

5.1. Filozofia dostępu do danych Springa ... 140
5.1.1. Hierarchia wyjątków związanych z dostępem do danych w Springu ... 141
5.1.2. Szablony dostępu do danych ... 143
5.1.3. Klasy bazowe DAO ... 145
5.2. Konfiguracja źródła danych ... 147
5.2.1. Źródła danych JNDI ... 147
5.2.2. Źródła danych z pulą ... 147
5.2.3. Źródło danych oparte na sterowniku JDBC ... 149

5.3. **Używanie JDBC w Springu** ... **149**
5.3.1. Kod JDBC a obsługa wyjątków ... 150
5.3.2. Praca z szablonami JDBC ... 153
5.4. **Integracja Hibernate ze Springiem** ... **158**
5.4.1. Przegląd Hibernate ... 159
5.4.2. Deklarowanie fabryki sesji Hibernate ... 160
5.4.3. Hibernate bez Springa ... 162
5.5. **Spring i Java Persistence API** ... **163**
5.5.1. Konfiguracja fabryki menedżerów encji ... 164
5.5.2. Klasa DAO na bazie JPA ... 168
5.6. **Podsumowanie** ... **169**

Rozdział 6. Zarządzanie transakcjami **171**

6.1. **Zrozumienie transakcji** ... **172**
6.1.1. Cztery słowa kluczowe do zrozumienia transakcji ... 173
6.1.2. Zrozumienie obsługi zarządzania transakcjami w Springu ... 174
6.2. **Wybór menedżera transakcji** ... **175**
6.2.1. Transakcje JDBC ... 175
6.2.2. Transakcje Hibernate ... 176
6.2.3. Transakcje Java Persistence API ... 177
6.2.4. Transakcje Java Transaction API ... 178
6.3. **Programowanie transakcji w Springu** ... **178**
6.4. **Deklarowanie transakcji** ... **180**
6.4.1. Definiowanie atrybutów transakcji ... 180
6.4.2. Deklarowanie transakcji w XML ... 184
6.4.3. Definiowanie transakcji sterowanych adnotacjami ... 186
6.5. **Podsumowanie** ... **187**

Rozdział 7. Budowanie aplikacji sieciowych za pomocą Spring MVC **189**

7.1. **Wprowadzenie do Spring MVC** ... **190**
7.1.1. Cykl życia żądania w Spring MVC ... 190
7.1.2. Konfiguracja Spring MVC ... 192
7.2. **Budowa podstawowego kontrolera** ... **193**
7.2.1. Konfiguracja Spring MVC sterowanego adnotacjami ... 194
7.2.2. Definiowanie kontrolera strony głównej ... 195
7.2.3. Produkowanie widoków ... 198
7.2.4. Definiowanie widoku strony głównej ... 202
7.2.5. Dopracowanie kontekstu aplikacji Springa ... 203
7.3. **Kontroler obsługujący dane wprowadzane przez użytkownika** ... **205**
7.3.1. Budowa kontrolera przetwarzającego dane wprowadzane przez użytkownika ... 205
7.3.2. Wyświetlenie widoku ... 207
7.4. **Przetwarzanie formularzy** ... **208**
7.4.1. Wyświetlenie formularza rejestracyjnego ... 209
7.4.2. Przetwarzanie formularza ... 211
7.4.3. Walidacja przesyłanych danych ... 213
7.5. **Obsługa plików wysyłanych na serwer** ... **217**
7.5.1. Dodanie pola selektora plików do formularza ... 217
7.5.2. Odbieranie plików wysyłanych na serwer ... 218
7.5.3. Konfiguracja Springa do przesyłania plików na serwer ... 221
7.6. **Podsumowanie** ... **221**

Rozdział 8. Praca ze Spring Web Flow 223

8.1. Instalacja Spring Web Flow ... 224
8.1.1. Konfiguracja Spring Web Flow ... 224
8.2. Składowe przepływu ... 227
8.2.1. Stany ... 227
8.2.2. Przejścia ... 230
8.2.3. Dane przepływu ... 231
8.3. Łączymy wszystko w całość: zamówienie pizzy ... 233
8.3.1. Definiowanie bazowego przepływu ... 233
8.3.2. Zbieranie informacji o kliencie ... 237
8.3.3. Budowa zamówienia ... 242
8.3.4. Przyjmowanie płatności ... 245
8.4. Zabezpieczanie przepływu ... 247
8.5. Podsumowanie ... 247

Rozdział 9. Zabezpieczanie Springa 249

9.1. Wprowadzenie do Spring Security ... 250
9.1.1. Rozpoczynamy pracę ze Spring Security ... 250
9.1.2. Konfiguracyjna przestrzeń nazw Spring Security ... 251
9.2. Zabezpieczanie żądań sieciowych ... 252
9.2.1. Pośredniczenie w dostępie do filtrów serwletów ... 252
9.2.2. Minimalna konfiguracja bezpieczeństwa sieciowego ... 253
9.2.3. Przechwytywanie żądań ... 257
9.3. Zabezpieczanie elementów na poziomie widoku ... 259
9.3.1. Dostęp do informacji uwierzytelniających ... 260
9.3.2. Wyświetlanie z uprawnieniami ... 261
9.4. Uwierzytelnianie użytkowników ... 263
9.4.1. Konfiguracja repozytorium w pamięci operacyjnej ... 263
9.4.2. Uwierzytelnianie za pomocą bazy danych ... 264
9.4.3. Uwierzytelnianie za pomocą LDAP ... 266
9.4.4. Włączenie funkcji „pamiętaj mnie” ... 269
9.5. Zabezpieczanie metod ... 270
9.5.1. Zabezpieczanie metod z adnotacją @Secured ... 270
9.5.2. Adnotacja @RolesAllowed ze specyfikacji JSR-250 ... 271
9.5.3. Zabezpieczenia przed wykonaniem metody ze SpEL i po jej wykonaniu ... 271
9.5.4. Deklaracja przecięć bezpieczeństwa na poziomie metody ... 276
9.6. Podsumowanie ... 276

CZĘŚĆ 3. INTEGRACJA W SPRINGU

Rozdział 10. Praca ze zdalnymi usługami 279

10.1. Zdalny dostęp w Springu ... 280
10.2. Praca z RMI ... 282
10.2.1. Eksportowanie usługi RMI ... 283
10.2.2. Dowiązanie usługi RMI ... 285
10.3. Udostępnianie zdalnych usług za pomocą Hessian i Burlap ... 287
10.3.1. Udostępnianie funkcjonalności komponentu za pomocą Hessian/Burlap ... 288
10.3.2. Dostęp do usług Hessian/Burlap ... 290

10.4. **Obiekt HttpInvoker** ... 292
10.4.1. Udostępnianie komponentów jako usług HTTP ... 292
10.4.2. Dostęp do usług przez HTTP ... 293
10.5. **Publikacja i konsumpcja usług sieciowych** ... 294
10.5.1. Tworzenie punktów końcowych JAX-WS w Springu ... 295
10.5.2. Pośrednik usług JAX-WS po stronie klienta ... 299
10.6. **Podsumowanie** ... 300

Rozdział 11. Spring i model REST 301

11.1. **Zrozumienie REST** ... 302
11.1.1. Fundamenty REST ... 302
11.1.2. Obsługa REST w Springu ... 303
11.2. **Tworzenie kontrolerów korzystających z zasobów** ... 303
11.2.1. Kontroler niezgodny z konwencją REST pod lupą ... 304
11.2.2. Obsługa adresów URL typu RESTful ... 305
11.2.3. Czasowniki REST ... 308
11.3. **Reprezentacja zasobów** ... 311
11.3.1. Negocjowanie reprezentacji zasobu ... 311
11.3.2. Praca z konwerterami komunikatów HTTP ... 314
11.4. **Tworzenie klientów REST** ... 317
11.4.1. Operacje szablonu RestTemplate ... 319
11.4.2. Pobieranie zasobów za pomocą GET ... 320
11.4.3. Umieszczanie zasobów na serwerze za pomocą PUT ... 322
11.4.4. Usuwanie zasobów za pomocą DELETE ... 324
11.4.5. Wysyłanie danych zasobu za pomocą POST ... 325
11.4.6. Wymiana zasobów ... 327
11.5. **Wysyłanie formularzy typu RESTful** ... 329
11.5.1. Umieszczanie ukrytych pól metod w kodzie za pomocą JSP ... 330
11.5.2. Demaskowanie rzeczywistego żądania ... 330
11.6. **Podsumowanie** ... 332

Rozdział 12. Obsługa komunikatów w Springu 333

12.1. **Krótki wstęp do JMS** ... 334
12.1.1. Architektura JMS ... 335
12.1.2. Szacowanie korzyści związanych z użyciem JMS ... 337
12.2. **Konfiguracja brokera komunikatów w Springu** ... 338
12.2.1. Tworzenie fabryki połączeń ... 339
12.2.2. Deklaracja miejsca docelowego komunikatów ActiveMQ ... 340
12.3. **Szablon JMS Springa** ... 340
12.3.1. Kod JMS a obsługa wyjątków ... 341
12.3.2. Praca z szablonami JMS ... 342
12.4. **Tworzenie obiektów POJO sterowanych komunikatami** ... 347
12.4.1. Tworzenie odbiorcy komunikatów ... 348
12.4.2. Konfiguracja odbiorców komunikatów ... 349
12.5. **Używanie RPC opartego na komunikatach** ... 350
12.5.1. Praca z RPC opartym na komunikatach w Springu ... 350
12.5.2. Asynchroniczne RPC z Lingo ... 352
12.6. **Podsumowanie** ... 354

Rozdział 13. Zarządzanie komponentami Springa za pomocą JMX 357

13.1. Eksportowanie komponentów Springa w formie MBean 358
13.1.1. Udostępnianie metod na podstawie nazwy 361
13.1.2. Użycie interfejsów do definicji operacji i atrybutów komponentu zarządzanego 363
13.1.3. Praca z komponentami MBean sterowanymi adnotacjami 364
13.1.4. Postępowanie przy konfliktach nazw komponentów zarządzanych 365
13.2. Zdalny dostęp do komponentów zarządzanych 366
13.2.1. Udostępnianie zdalnych komponentów MBean 367
13.2.2. Dostęp do zdalnego komponentu MBean 367
13.2.3. Obiekty pośredniczące komponentów zarządzanych 369
13.3. Obsługa powiadomień 370
13.3.1. Odbieranie powiadomień 371
13.4. Podsumowanie 372

Rozdział 14. Pozostałe zagadnienia 373

14.1. Wyodrębnianie konfiguracji 374
14.1.1. Zastępowanie symboli zastępczych we właściwościach 374
14.1.2. Nadpisywanie właściwości 377
14.1.3. Szyfrowanie zewnętrznych właściwości 378
14.2. Wiązanie obiektów JNDI 380
14.2.1. Praca z tradycyjnym JNDI 380
14.2.2. Wstrzykiwanie obiektów JNDI 382
14.2.3. Wiązanie komponentów EJB w Springu 385
14.3. Wysyłanie wiadomości e-mail 386
14.3.1. Konfiguracja komponentu wysyłającego pocztę 386
14.3.2. Budowa wiadomości e-mail 388
14.4. Planowanie zadań wykonywanych w tle 392
14.4.1. Deklaracja zaplanowanych metod 393
14.4.2. Deklaracja metod asynchronicznych 395
14.5. Podsumowanie 396
14.6. Koniec…? 396

Skorowidz 399

Aspektowy Spring

W tym rozdziale omówimy:

- Podstawy programowania aspektowego
- Tworzenie aspektów z POJO
- Stosowanie adnotacji `@AspectJ`
- Wstrzykiwanie zależności do aspektów AspectJ

Gdy piszę ten rozdział, w Teksasie (gdzie mieszkam) mamy od kilku dni rekordowo wysokie temperatury. Jest strasznie gorąco. Podczas takiej pogody klimatyzacja jest koniecznością. Jednak wadą klimatyzacji jest zużycie energii elektrycznej, a za energię elektryczną trzeba zapłacić. Niewiele możemy zrobić w celu uniknięcia wydatków na to, by mieszkać w chłodnych i komfortowych warunkach. To dlatego w każdym domu jest zamontowany licznik energii elektrycznej, który rejestruje wszystkie zużyte kilowaty, i raz w miesiącu ktoś przychodzi odczytać ten licznik, aby zakład energetyczny dokładnie wiedział, jaki wystawić nam rachunek.

Teraz wyobraźmy sobie, co by się stało, gdyby taki licznik zniknął i nikt nie przychodził, by sprawdzić nasze zużycie energii elektrycznej. Załóżmy, że do każdego właściciela domu należałoby skontaktowanie się z zakładem energetycznym i zgłoszenie swojego zużycia energii. Choć jest możliwe, że jakiś obsesyjny właściciel domu uważnie rejestrowałby każde użycie światła, telewizji czy klimatyzacji, większość by się tym nie przejmowała. Większość podałaby swoje przybliżone zużycie, a niektórzy nie zgłosiliby zużycia w ogóle. Sprawdzanie zużycia energii byłoby kłopotliwe, a pokusa, by nie zapłacić, mogłaby okazać się zbyt silna.

Ta forma systemu płatności za energię elektryczną mógłaby być świetna dla klientów, lecz byłaby daleka od ideału z punktu widzenia zakładów energetycznych. To dlatego wszyscy mamy w domu liczniki energii i dlatego raz w miesiącu ktoś zagląda, by odczytać licznik i poinformować zakład energetyczny o zużyciu.

Niektóre funkcje systemów oprogramowania są podobne do liczników energii elektrycznej w naszych domach. Funkcje te muszą być stosowane w wielu miejscach w naszej aplikacji, lecz byłoby niewygodne, gdybyśmy musieli wywoływać je w każdym miejscu.

Kontrola zużycia energii elektrycznej jest ważną funkcją, lecz nie zajmuje najwyższej pozycji w świadomości większości właścicieli domów. Koszenie trawników, odkurzanie dywanów czy sprzątanie łazienki to czynności, w które posiadacz domu jest aktywnie zaangażowany. Natomiast kontrola zużycia elektryczności w domu jest z punktu widzenia właściciela domu czynnością pasywną (choć byłoby cudownie, gdyby koszenie trawników też było czynnością pasywną — zwłaszcza w takie gorące dni).

W oprogramowaniu niektóre czynności są powszechne w większości aplikacji. Rejestrowanie transakcji, ich bezpieczeństwo i zarządzanie nimi są ważne, lecz czy powinny być czynnościami, w których aktywnie uczestniczą obiekty naszej aplikacji? Czy może lepiej by było, aby obiekty naszej aplikacji pozostały skoncentrowane na problemach z dziedziny biznesowej, do jakich zostały zaprojektowane, i pozostawiły pewne aspekty do obsługi przez kogoś innego?

Funkcje, które przenikają wiele miejsc w aplikacji, noszą w branży programistycznej nazwę **zagadnień przecinających**. W typowej sytuacji takie zagadnienia przecinające są koncepcyjnie rozdzielone od biznesowej logiki aplikacji (choć często bezpośrednio w nią wbudowane). Oddzielenie zagadnień przecinających od logiki biznesowej jest miejscem, gdzie do pracy włącza się **programowanie aspektowe** (AOP).

W rozdziale 2. nauczyliśmy się, jak używać wstrzykiwania zależności do zarządzania obiektami naszej aplikacji i ich konfigurowania. Podobnie jak DI pozwala rozdzielić od siebie poszczególne obiekty aplikacji, AOP pozwala oddzielić zagadnienia przecinające od obiektów, których one dotyczą.

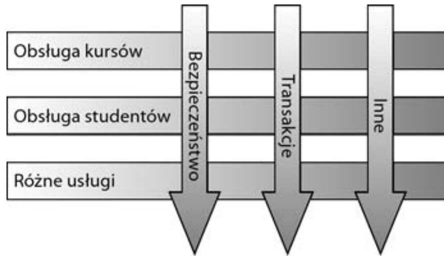
Logowanie jest popularnym przykładem zastosowania aspektów. Jednak nie jest to jedyne zastosowanie, w którym korzystne będzie użycie aspektów. Podczas lektury tej książki zobaczymy kilka praktycznych zastosowań aspektów, w tym transakcje deklaracyjne, bezpieczeństwo oraz pamięć podręczną.

W tym rozdziale zostanie omówiona obsługa aspektów w Springu, w tym sposób deklarowania zwykłych klas, aby stały się aspektami, oraz możliwość zastosowania adnotacji podczas tworzenia aspektów. Dodatkowo dowiemy się, w jaki sposób przy użyciu AspectJ — innej popularnej implementacji AOP — możemy uzupełnić działanie środowiska AOP Springa. Lecz najpierw, zanim się zajmiemy transakcjami, bezpieczeństwem i pamięcią podręczną, dowiedzmy się, w jaki sposób aspekty są implementowane w Springu, zaczynając od paru zupełnych podstaw AOP.

4.1. Czym jest programowanie aspektowe

Jak wcześniej wspomnieliśmy, aspekty pomagają zamykać w modułach zagadnienia przecinające. W skrócie, zagadnienie przecinające można opisać jako dowolny mechanizm, którego wpływ jest używany na wielu miejscach w aplikacji. Bezpieczeństwo na

przykład jest zagadnieniem przecinającym, w tym sensie, że wiele metod w aplikacji może podlegać stosowaniu reguł bezpieczeństwa. Na rysunku 4.1 pokazano obrazową ilustrację zagadnień przecinających.



Rysunek 4.1. Aspekty zamykają w modułach zagadnienia przecinające, które dotyczą logiki zawartej w wielu spośród obiektów aplikacji

Na rysunku 4.1 pokazano typową aplikację, która jest podzielona na moduły. Głównym zadaniem każdego z modułów jest realizowanie usług ze swojej szczególnej dziedziny. Lecz każdy moduł potrzebuje także podobnych mechanizmów pomocniczych, takich jak bezpieczeństwo czy zarządzanie transakcjami.

Popularną obiektową techniką ponownego użycia tej samej funkcjonalności jest stosowanie dziedziczenia albo delegacji. Ale dziedziczenie może prowadzić do zburzenia hierarchii obiektów, jeśli ta sama klasa bazowa jest stosowana w całej aplikacji. Z kolei stosowanie delegacji bywa uciążliwe, ponieważ może być potrzebne skomplikowane wywołanie delegacji.

Aspekty stanowią alternatywę dla dziedziczenia i delegacji, która w wielu sytuacjach może być bardziej eleganckim rozwiązaniem. Korzystając z techniki AOP, nadal definiujemy popularne mechanizmy w jednym miejscu, lecz za pomocą deklaracji definiujemy, gdzie i jak mechanizmy te zostaną zastosowane, bez konieczności modyfikowania klasy, do której mają zastosowanie nowe mechanizmy. Odtąd zagadnienia przekrojowe możemy zamknąć w modułach w postaci specjalnych klas zwanych **aspektami**. Wynikają z tego dwie korzyści. Po pierwsze, logika dla każdego zagadnienia znajduje się teraz w jednym miejscu, zamiast być rozrzuconą po całym kodzie. Po drugie, nasze moduły usługowe będą teraz bardziej uporządkowane, ponieważ zawierają jedynie kod dotyczący ich głównego zadania (albo podstawowych funkcji), zaś zagadnienia drugorzędne zostały przeniesione do aspektów.

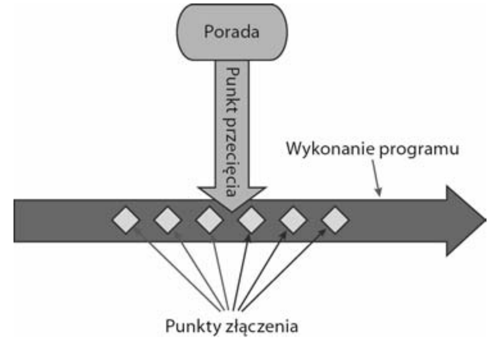
4.1.1. Definiujemy terminologię dotyczącą AOP

Podobnie jak w przypadku innych technologii, wraz z rozwojem AOP powstał specyficzny żargon opisujący tę technikę programistyczną. Aspekty są często opisywane za pomocą pojęć takich jak „porada", „punkt przecięcia" czy „punkt złączenia". Na rysunku 4.2 pokazano, jak są ze sobą powiązane te zagadnienia.

Niestety, wiele spośród pojęć używanych do opisywania AOP jest dalekie od intuicyjności. Są one jednak obecnie częścią pojęcia „programowanie aspektowe" i musimy się z nimi zapoznać w celu zrozumienia tej nowej techniki programistycznej. Zanim przejdziemy do praktyki, nauczmy się języka, w którym będziemy rozmawiać.

PORADA

Gdy osoba odczytująca licznik pokaże się w naszym domu, jej zadaniem jest poinformowanie zakładu energetycznego o wskazywanej liczbie kilowatogodzin. Z pewnością osoba ta ma listę domów, które musi odwiedzić, zaś informacje, które dostarcza zakładowi energetycznemu, są ważne. Lecz głównym zadaniem osoby odczytującej liczniki jest sama czynność notowania zużycia energii.



Rysunek 4.2. **Funkcjonalność aspektu (porada) jest wplatana do wykonania programu w jednym lub więcej punktach złączenia**

Podobnie, aspekty mają główne zadanie — czynność, której wykonanie jest celem ich istnienia. W terminologii programowania aspektowego zadanie aspektu jest nazywane **poradą**.

Porada jest dla aspektu definicją zarówno czynności, która ma zostać wykonana, jak i właściwego momentu jej wykonania. Czy aspekt powinien być wykonany przed wywołaniem metody? Po jej wykonaniu? Zarówno przed wywołaniem metody, jak i po niej? A może tylko wtedy, gdy metoda zgłosi wyjątek? Aspekty w Springu mogą działać z pięcioma rodzajami porad:

- *Before* — Funkcjonalność porady jest wykonywana przed wywołaniem metody z poradą.
- *After* — Funkcjonalność porady jest wykonywana po zakończeniu działania metody z poradą, niezależnie od wyniku jej działania.
- *After-returning* — Funkcjonalność porady jest wykonywana po prawidłowym zakończeniu metody z poradą.
- *After-throwing* — Funkcjonalność porady jest wykonywana po zgłoszeniu wyjątku przez metodę z poradą.
- *Around* — Porada realizuje tę samą funkcjonalność zarówno przed wywołaniem, jak i po zakończeniu metody z poradą.

PUNKTY ZŁĄCZENIA

Zakład energetyczny obsługuje wiele domów, prawdopodobnie wręcz całe miasto. W każdym domu zamontowany jest licznik energii elektrycznej, zatem każdy dom jest potencjalnym celem dla osoby odczytującej liczniki. Potencjalnie osoba taka mogłaby odczytywać wiele różnych przyrządów pomiarowych, lecz w ramach swoich obowiązków służbowych powinna przyjąć za cel właśnie liczniki energii zamontowane w domach.

W podobny sposób w naszej aplikacji mogą być tysiące miejsc, w których moglibyśmy zastosować poradę. Miejsca te są nazywane **punktami złączenia**. Punkt złączenia jest taką pozycją w przebiegu wykonania programu, w której może zostać wpięty aspekt. Takim punktem może być wywołanie metody, zgłoszenie wyjątku czy nawet modyfikacja zawartości pola. Są to pozycje, w których kod aspektu może zostać włączony w normalny przebieg działania naszej aplikacji, wzbogacając tym jej działanie.

Punkty przecięcia

Nie jest możliwe, by jedna osoba odczytująca liczniki zdołała odwiedzić wszystkie domy, którym elektryczność dostarcza ten sam zakład energetyczny. Pojedynczej osobie jest przypisany jedynie pewien podzbiór wszystkich domów. Podobnie porada nie musi koniecznie dołączać aspektu do każdego punktu złączenia w aplikacji. **Punkty przecięcia** pozwalają zawęzić listę punktów złączenia, do których zastosowany będzie aspekt.

Jeśli porada definiowała *czynność* i *moment jej wykonania* przez aspekt, to punkty przecięcia definiują *miejsce*, w którym czynność ta zostanie wykonana. Definicja punktu przecięcia dopasowuje jeden lub więcej punktów złączenia, w których należy wpleść poradę. Często określamy takie punkty przecięcia za pomocą jawnego wskazania nazw metod i klas albo za pomocą wyrażeń regularnych, które dopasowują do wzorców nazwy klas i metod. Niektóre frameworki aspektowe pozwalają na tworzenie dynamicznych punktów przecięcia, w których decyzja, czy zastosować poradę, jest podejmowana w trakcie działania, na przykład na podstawie wartości parametrów metod.

Aspekty

Gdy osoba odczytująca liczniki rozpoczyna dzień pracy, wie zarówno, co należy do jej obowiązków (kontrola zużycia energii elektrycznej), jak również z których domów powinna zbierać tę informację. Zatem wie ona wszystko, co potrzeba, by wywiązać się ze swoich obowiązków.

Aspekt jest sumą porady i punktów przecięcia. Wzięte razem porada i punkty przecięcia definiują wszystko, co można powiedzieć o aspekcie — jaką czynność wykonuje, gdzie i kiedy.

Wprowadzenia

Wprowadzenie pozwala nam dodawać nowe metody lub atrybuty do istniejących klas. Na przykład możemy skonstruować klasę z poradą, o nazwie `Auditable`. Będzie ona przechowywała informację o momencie ostatniej modyfikacji obiektu. Może to być klasa tak prosta, że będzie posiadała tylko jedną metodę, nazwaną `setLastModified(Date)` i zmienną o zasięgu instancji, która będzie przechowywała stan tej klasy. Taką nową metodę i zmienną możemy wprowadzić do istniejących klas bez konieczności ich modyfikowania, wzbogacając je o nowe działanie i zmienną stanu.

Wplatanie

Wplatanie jest procesem zastosowania aspektu do obiektu docelowego w celu utworzenia nowego obiektu z pośrednikiem. Aspekty są wplatane do docelowych obiektów we wskazanych punktach złączenia. Wplatanie może mieć miejsce w różnych momentach cyklu życia docelowego obiektu:

- *W czasie kompilacji* — Aspekty zostają wplecione, gdy klasa docelowa jest kompilowana. W tym celu potrzebny jest specjalny kompilator. Wplatający kompilator w AspectJ wplata aspekty w ten sposób.
- *W czasie ładowania klasy* — Aspekty zostają wplecione, gdy klasa docelowa jest ładowana do maszyny wirtualnej Javy (JVM). W tym celu potrzebny jest specjalny `ClassLoader`, który rozszerza kod bajtowy docelowej klasy, zanim

zostanie ona wprowadzona do aplikacji. Opcja *wplatania w czasie ładowania* (ang. *load-time weaving*, LTW) wprowadzona w wersji 5 AspectJ realizuje wplatanie aspektów w ten sposób.

- *W czasie działania* — Aspekty zostają wplecione w jakimś momencie podczas działania aplikacji. W typowej sytuacji kontener aspektowy będzie dynamicznie generował obiekt pośredniczący, który zostanie połączony z obiektem docelowym za pomocą delegacji podczas wplatania aspektów. Z tego sposobu korzysta Spring AOP podczas wplatania aspektów.

To wiele nowych pojęć, z którymi trzeba się zapoznać. Wracając do rysunku 4.2, zobaczymy, że porada zawiera zachowanie zagadnienia przecinającego, które ma być włączone do obiektów aplikacji. Punktami złączenia są wszystkie punkty w przebiegu działania aplikacji, które są potencjalnymi miejscami zastosowania porady. Najważniejszym, co należy tu sobie uświadomić, jest fakt, że punkty przecięcia definiują, które punkty złączenia będą brane pod uwagę.

Gdy już się troszkę oswoiliśmy z najbardziej podstawową częścią terminologii dotyczącej programowania aspektowego, przekonajmy się, jak te podstawowe koncepcje AOP zostały zaimplementowane w Springu.

4.1.2. Obsługa programowania aspektowego w Springu

Nie wszystkie aspektowe frameworki są skonstruowane tak samo. Mogą się różnić tym, jak bogaty model punktów złączeń posiadają. Niektóre pozwalają stosować porady na poziomie modyfikacji pól, podczas gdy inne udostępniają jedynie punkty złączeń związane z wywołaniem metod. Mogą się także różnić sposobem i momentem wplatania aspektów. Niezależnie od konkretnego przypadku, możliwość tworzenia punktów przecięcia, definiujących punkty złączenia, w których powinny zostać wplecione aspekty, jest tym, co stanowi aspektowy framework.

Wiele się zmieniło w branży frameworków aspektowych przez ostatnie kilka lat. Zostały poczynione porządki, w wyniku których niektóre spośród frameworków zostały połączone ze sobą, z kolei inne zaczynają zanikać. W roku 2005 projekt AspectWerkz został połączony z AspectJ, co było ostatnią istotną zmianą w świecie AOP, pozostawiając nam do wyboru trzy dominujące frameworki aspektowe:

- AspectJ (*http://eclipse.org/aspectj*),
- JBoss AOP (*http://www.jboss.org/jbossaop*),
- Spring AOP (*http://www.springframework.org*).

Ponieważ książka ta jest poświęcona frameworkowi Spring, skupimy się na Spring AOP. Mimo to występuje wiele podobieństw między projektami Spring i AspectJ, zaś obsługa AOP w Springu zawiera wiele zapożyczeń z projektu AspectJ.

Obsługa AOP w Springu ma cztery odmiany:

- Klasyczna obsługa AOP w Springu na bazie obiektów pośredniczących.
- Aspekty sterowane adnotacją @AspectJ.
- Aspekty zbudowane z „czystych” POJO.
- Wstrzykiwane aspekty z AspektJ (dostępne we wszystkich wersjach Springa).

Pierwsze trzy pozycje są właściwie odmianami obsługi AOP przez Springa na bazie obiektów pośredniczących. W efekcie obsługa AOP w Springu jest ograniczona do przechwytywania metod. Jeśli potrzebujemy w Springu rozszerzonego przechwytywania prostych metod (na przykład przechwytywania konstruktora albo właściwości), będziemy musieli rozważyć implementację aspektów w AspectJ, być może korzystając ze wstrzykiwania zależności w Springu, by wstrzyknąć komponenty Springa do aspektów w AspectJ.

Co takiego? Nie będzie omówienia klasycznego AOP w Springu?

Słowo *klasyczne* zwykle budzi pozytywne skojarzenia. Klasyczne samochody, klasyczny turniej golfowy czy też klasyczna Coca-Cola — wszystkie są czymś dobrym.

Lecz klasyczny model programowania aspektowego w Springu wcale nie jest zbyt dobry. Oczywiście, był czymś świetnym jak na swoje czasy. Lecz obecnie istnieją w Springu znacznie bardziej uporządkowane i łatwiejsze sposoby pracy z aspektami. W porównaniu do prostego deklaracyjnego modelu AOP i AOP opartego na adnotacjach klasyczny model AOP w Springu robi wrażenie ociężałego i nadmiernie skomplikowanego. Bezpośrednie wykorzystanie `ProxyFactoryBean` może być męczące.

Zatem zdecydowałem, że do bieżącego wydania tej książki nie włączę w ogóle omówienia klasycznego AOP w Springu. Jeśli naprawdę ciekawi cię, Czytelniku, jak działa ten model AOP, możesz sprawdzić w pierwszych dwóch wydaniach. Lecz sądzę, że przekonasz się, iż praca z nowymi modelami AOP w Springu jest znacznie łatwiejsza.

W tym rozdziale dowiemy się więcej o powyższych technikach aspektowych w Springu. Jednak zanim rozpoczniemy, ważne, byśmy zrozumieli kilka kluczowych zagadnień w aspektowym frameworku Spring.

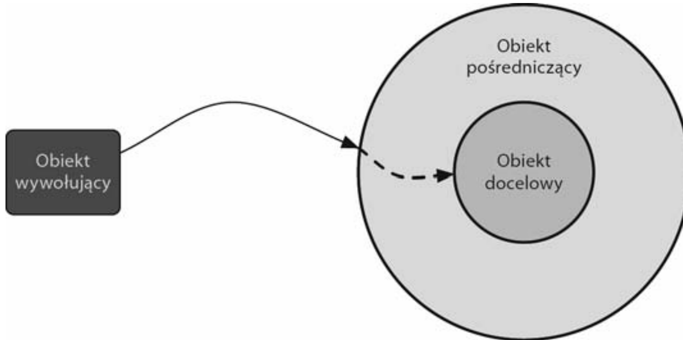
Porada Springa jest zapisana w Javie

Wszystkie porady, jakie utworzymy w Springu, są zapisane w standardowych klasach Javy. W ten sposób czerpiemy korzyść z możliwości konstruowania naszych aspektów w tym samym zintegrowanym środowisku programistycznym (IDE), którego na co dzień używamy do pisania programów w Javie. Co więcej, punkty przecięcia, które definiują miejsca, gdzie porady powinny zostać zastosowane, zwykle zapisujemy w języku XML, jako część naszego pliku konfiguracji Springa. To oznacza, że zarówno kod aspektów, jak i składnia konfiguracji będą naturalne dla programistów Javy.

Inaczej jest w AspectJ. Choć obecnie AspectJ obsługuje już aspekty oparte na adnotacjach, AspectJ pojawił się także jako rozszerzenie języka Java. Takie podejście ma zarówno zalety, jak i wady. Posiadanie specjalizowanego języka do obsługi aspektów zwiększa możliwości takiego języka i pozwala na precyzyjniejszą kontrolę jego zachowania, a także bogatszy zestaw narzędzi aspektowych. Lecz osiągamy to kosztem konieczności nauki nowego narzędzia i nowej składni.

Spring dołącza porady do obiektów w trakcie pracy

Spring wplata aspekty do zarządzanych przez niego komponentów w trakcie pracy, opakowując je w klasy pośredniczące. Jak pokazano na rysunku 4.3, klasa pośrednicząca zawiera docelowy komponent, przechwytuje wywołania metod z poradą i przekierowuje wywołania tych metod do komponentu docelowego.



Rysunek 4.3. Aspekty w Springu zostały zaimplementowane jako obiekty pośredniczące, które opakowują obiekt docelowy. Obiekt pośredniczący przechwytuje wywołania metod, wykonuje dodatkową logikę aspektu, a następnie wywołuje metodę docelową

W czasie pomiędzy przechwyceniem przez obiekt pośredniczący wywołania metody a momentem wywołania metody komponentu docelowego obiekt pośredniczący realizuje logikę aspektu.

Spring nie tworzy obiektu z pośrednikiem, dopóki aplikacja nie będzie potrzebowała określonego komponentu. Jeśli korzystamy z `ApplicationContext`, obiekt z pośrednikiem zostanie utworzony, gdy kontekst będzie ładował wszystkie należące do niego komponenty z `BeanFactory`. Ponieważ Spring tworzy obiekty pośredniczące w czasie działania, korzystanie z AOP w Springu nie zmusza nas do stosowania specjalnego kompilatora, który umożliwiałby wplatanie aspektów.

SPRING OBSŁUGUJE JEDYNIE PUNKTY ZŁĄCZENIA ZWIĄZANE Z METODAMI

Jak wspomnieliśmy wcześniej, w poszczególnych implementacjach AOP stosowane są różne modele punktów złączenia. Ponieważ Spring bazuje na dynamicznych obiektach pośredniczących, obsługuje tylko punkty złączenia związane z metodami. W tym różni się od niektórych innych frameworków aspektowych, takich jak AspectJ czy JBoss, które poza punktami złączenia związanymi z metodami oferują także obsługę punktów złączenia związanych z polami oraz z konstruktorami. Nieobecność w Springu punktów przecięcia związanych z polami uniemożliwia nam tworzenie porad o bardzo dużej precyzji, takich jak przejęcie aktualizacji pola w obiekcie. Zaś bez punktów przecięcia związanych z konstruktorami nie dysponujemy sposobem, by zastosować poradę podczas tworzenia instancji beana.

Przechwytywanie wywołań metod powinno zaspokoić większość, jeśli nie wszystkie, z naszych potrzeb. Jeśli okaże się, że potrzebujemy czegoś więcej niż tylko przechwytywanie wywołań metod, będziemy mogli uzupełnić funkcjonalność AOP w Springu za pomocą AspectJ.

Teraz mamy już ogólne pojęcie, do czego służy programowanie aspektowe i w jaki sposób jest obsługiwane przez Springa. Nadszedł czas, by zakasać rękawy i wziąć się za tworzenie aspektów w Springu. Zacznijmy od deklaracyjnego modelu AOP w Springu.

4.2. Wybieramy punkty złączenia za pomocą punktów przecięcia

Jak wcześniej wspominaliśmy, punktów przecięcia używamy do wskazania miejsca, w którym powinna zostać zastosowana porada aspektu. Obok porady, punkty przecięcia stanowią najbardziej podstawowe składniki aspektu. Jest zatem ważne, byśmy wiedzieli, w jaki sposób wiązać punkty przecięcia.

Programując aspektowo w Springu, definiujemy punkty przecięcia za pomocą pochodzącego z frameworka AspectJ języka wyrażeń punktów przecięcia. Jeśli jesteśmy już zaznajomieni ze środowiskiem AspectJ, wówczas definiowanie punktów przecięcia w Springu wyda się nam naturalne. Lecz w razie gdyby AspectJ był dla nas frameworkiem nowym, ten podrozdział może służyć jako szybki samouczek pisania punktów przecięcia w stylu AspectJ. Poszukującym bardziej szczegółowego omówienia frameworka AspectJ oraz pochodzącego z tego frameworka języka wyrażeń punktów przecięcia z całego serca polecam drugie wydanie książki *AspectJ in Action*, którą napisał Ramnivas Laddad.

Najważniejszym, co powinniśmy wiedzieć na temat punktów przecięcia w stylu AspectJ w zastosowaniu do programowania aspektowego w Springu, jest fakt obsługiwania przez Springa jedynie podzbioru desygnatorów punktów przecięcia dostępnych w AspectJ. Przypomnijmy sobie, że Spring AOP bazuje na obiektach pośredniczących i niektóre wyrażenia opisujące punkty przecięcia niczego nie wnoszą dla programowania aspektowego bazującego na obiektach pośredniczących. W tabeli 4.1 zawarto zestawienie desygnatorów punktów przecięcia pochodzących z AspectJ, które są obsługiwane przez Spring AOP.

Tabela 4.1. Spring do definiowania aspektów wykorzystuje język wyrażeń punktów przecięcia pochodzący z AspectJ

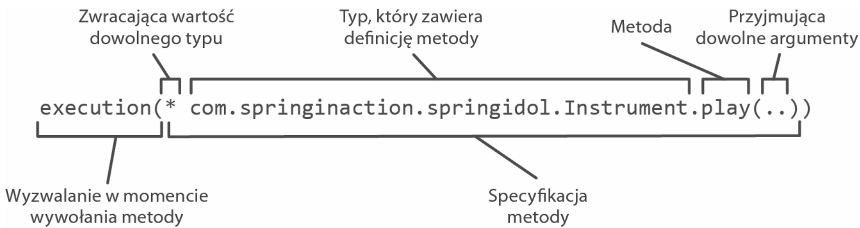
Desygnator w stylu AspectJ	Opis
`args()`	Ogranicza dopasowanie punktów złączenia do wywołań metod, których argumenty są instancjami określonych typów.
`@args()`	Ogranicza dopasowanie punktów złączenia do wywołań metod, których argumenty są opatrzone adnotacjami określonych typów.
`execution()`	Dopasowuje punkty złączenia, które są wywołaniami metod.
`this()`	Ogranicza dopasowanie punktów złączenia do takich, które posiadają w obiekcie pośredniczącym AOP referencję do beana określonego typu.
`target()`	Ogranicza dopasowanie punktów złączenia do takich, w których obiekt docelowy jest instancją określonego typu.
`@target()`	Ogranicza dopasowanie do punktów złączenia, w których klasa wywoływanego obiektu jest opatrzona adnotacją określonego typu.
`within()`	Ogranicza dopasowanie do punktów złączenia będących instancjami określonych typów.
`@within()`	Ogranicza dopasowanie do punktów złączenia będących instancjami typów, które są opatrzone określoną adnotacją (w zastosowaniu dla Spring AOP wywołania metod zadeklarowanych w typach opatrzonych określoną adnotacją).
`@annotation`	Ogranicza dopasowanie punktów złączenia do takich, w których obiekt będący przedmiotem działania punktów złączenia jest opatrzony określoną adnotacją.

Próba użycia któregoś z desygnatorów pochodzących z AspectJ, niewymienionego w powyższej tabeli, będzie skutkowała zgłoszeniem wyjątku `IllegalArgumentException`.

Przeglądając listę obsługiwanych desygnatorów, zwróćmy uwagę, że `execution` jest jedynym desygnatorem, który faktycznie realizuje dopasowanie. Wszystkich pozostałych używamy do ograniczania takich dopasowań. To znaczy, że podstawowy jest desygnator `execution`, którego użyjemy w każdej definicji punktu przecięcia, jaką napiszemy. Pozostałych będziemy używać do ograniczania zasięgu punktu przecięcia.

4.2.1. Piszemy definicje punktów przecięcia

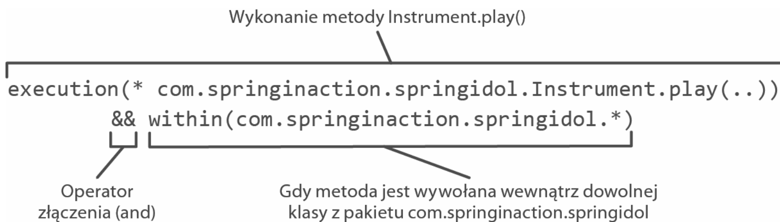
Przykładowo możemy użyć wyrażenia pokazanego na rysunku 4.4, by zastosować poradę za każdym wywołaniem metody `play()`, należącej do klasy `Instrument`.



Rysunek 4.4. Wybieramy metodę play(), zdefiniowaną w klasie Instrument, za pomocą wyrażenia punktu przecięcia w stylu AspectJ

Użyliśmy desygnatora `execution()`, by wybrać metodę `play()`, należącą do klasy `Instru`↪`ment`. Specyfikacja metody rozpoczyna się od gwiazdki, która oznacza, że nie ma dla nas znaczenia, jaki będzie typ wartości zwróconej przez metodę. Następnie podajemy pełną, kwalifikowaną nazwę klasy oraz nazwę metody, którą chcemy wybrać. Dla listy parametrów metody użyliśmy podwójnej kropki (`..`), co oznacza, że punkt przecięcia może wybrać dowolną spośród metod `play()`, niezależnie od tego, jakie parametry przyjmują poszczególne z nich.

Teraz załóżmy, że chcemy zawęzić zasięg tego punktu przecięcia jedynie do pakietu `com.springinaction.springidol`. W takiej sytuacji możemy ograniczyć dopasowanie, dodając do wyrażenia desygnator `within()`, jak pokazano na rysunku 4.5.



Rysunek 4.5. Ograniczamy zasięg punktu przecięcia za pomocą desygnatora within()

Zauważmy, że użyliśmy operatora `&&`, by połączyć desygnatory `execution()` oraz `within()` za pomocą relacji *koniunkcji* (w której warunkiem dopasowania punktu przecięcia jest dopasowanie obydwu desygnatorów). Podobnie, mogliśmy użyć operatora `||`, by utworzyć relację *alternatywy*. Z kolei operator `!` pozwala zanegować wynik działania desygnatora.

Ponieważ znak & jest symbolem specjalnym w języku XML, możemy swobodnie zastąpić notację && operatorem and, gdy zapisujemy specyfikację punktów przecięcia w pliku konfiguracyjnym XML Springa. Podobnie, możemy użyć operatorów or oraz not, zastępując nimi, odpowiednio, notację || i !.

4.2.2. Korzystamy z desygnatora bean() w Springu

W wersji 2.5 Springa wprowadzono nowy desygnator bean(), rozszerzający listę zawartą w tabeli 4.1. Pozwala on wskazywać komponenty za pomocą ich nazw w wyrażeniu punktu przecięcia. Desygnator bean() przyjmuje jako argument nazwę komponentu i ogranicza działanie punktu przecięcia do tego konkretnego komponentu.

Przykładowo, rozważmy poniższy punkt przecięcia:

```
execution(* com.springinaction.springidol.Instrument.play()) and bean(eddie)
```

Informujemy tu Springa, że chcemy zastosować poradę aspektu do wykonania metody play() klasy Instrument, lecz ograniczając się do wywołań z komponentu o nazwie eddie.

Możliwość zawężenia punktu przecięcia do określonego komponentu może być cenna w niektórych sytuacjach, lecz możemy także użyć negacji, by zastosować aspekt do wszystkich komponentów, z wyjątkiem posiadającego określoną nazwę:

```
execution(* com.springinaction.springidol.Instrument.play()) and !bean(eddie)
```

W tym wypadku porada aspektu zostanie wpleciona do wszystkich komponentów, które mają nazwę różną od eddie.

Teraz, gdy omówiliśmy podstawy pisania punktów przecięcia, zmierzmy się z pisaniem porad i deklarowaniem aspektów, które będą z tych punktów przecięcia korzystały.

4.3. Deklarujemy aspekty w języku XML

Jeśli jesteś obeznany z klasycznym modelem programowania aspektowego w Springu, wiesz, że praca z ProxyFactoryBean jest bardzo niewygodna. Twórcy Springa zauważyli ten problem i postanowili udostępnić lepszy sposób na deklarowanie aspektów w tym frameworku. Rezultat tych starań znalazł się w przestrzeni nazw aop Springa. Zestawienie elementów konfiguracyjnych AOP umieszczono w tabeli 4.2.

W rozdziale 2., pokazując przykład wstrzykiwania zależności, zorganizowaliśmy turniej talentów o nazwie *Idol Springa*. W przykładzie tym utworzyliśmy powiązania dla kilku wykonawców, jako elementów <bean>, umożliwiając im zademonstrowanie ich uzdolnień. To wszystko było wspaniałą rozrywką. Lecz tego rodzaju przedsięwzięcie potrzebuje widowni albo będzie bezcelowe.

Zatem, aby zilustrować działanie Spring AOP, utwórzmy klasę Audience dla naszego turnieju talentów. Funkcje widowni definiuje klasa z listingu 4.1.

Jak widzimy, klasa Audience niczym szczególnym się nie wyróżnia. Jest to podstawowa klasa Javy, zawierająca kilka metod. Możemy także zarejestrować tę klasę jako beana w kontekście aplikacji Springa:

```
<bean id="audience"
      class="com.springinaction.springidol.Audience" />
```

Tabela 4.2. Elementy konfiguracyjne Spring AOP upraszczają deklarację aspektów bazujących na POJO

Element konfiguracji AOP	Zastosowanie
`<aop:advisor>`	Definiuje doradcę aspektowego.
`<aop:after>`	Definiuje aspektową poradę *after* (niezależną od wyniku działania metody zaopatrzonej w poradę).
`<aop:after-returning>`	Definiuje aspektową poradę *after-returning* (po pomyślnym zakończeniu działania metody).
`<aop:after-throwing>`	Definiuje aspektową poradę *after-throwing* (po zgłoszeniu wyjątku przez metodę).
`<aop:around>`	Definiuje aspektową poradę *around* (zarówno przed wykonaniem metody, jak i po zakończeniu jej działania).
`<aop:aspect>`	Definiuje aspekt.
`<aop:aspect-autoproxy>`	Przełącza w tryb aspektów sterowanych adnotacjami z użyciem @AspectJ.
`<aop:before>`	Definiuje aspektową poradę *before* (przed wykonaniem metody).
`<aop:config>`	Element nadrzędnego poziomu w konfiguracji aspektowej. Większość elementów `<aop:*>` powinna znajdować się wewnątrz elementu `<aop:config>`.
`<aop:declare-parents>`	Wprowadza do obiektów z poradą dodatkowe interfejsy, implementowane w przezroczysty sposób.
`<aop:pointcut>`	Definiuje punkt przecięcia.

Listing 4.1. Klasa Audience dla naszego turnieju talentów

```
package com.springinaction.springidol;
public class Audience {
  public void takeSeats() {                    ← Przed występem
    System.out.println("Widzowie zajmują miejsca.");
  }

  public void turnOffCellPhones() {            ← Przed występem
    System.out.println("Widzowie wyłączają telefony komórkowe.");
  }

  public void applaud() {                      ← Po występie
    System.out.println("Brawooo! Oklaski!");
  }

  public void demandRefund() {                 ← Po nieudanym występie
    System.out.println("Buu! Oddajcie pieniądze za bilety!");
  }
}
```

Mimo skromnego wyglądu klasy `Audience` niezwykłe w niej jest to, że ma ona wszystko, czego potrzeba do utworzenia aspektu. Potrzebuje ona jedynie odrobiny specjalnych aspektowych czarów Springa.

4.3.1. Deklarujemy porady before i after

Korzystając z elementów konfiguracyjnych Spring AOP, jak pokazano na listingu 4.2, możemy z komponentu audience utworzyć aspekt.

Listing 4.2. Definiujemy aspekt audience, korzystając z elementów konfiguracyjnych Spring AOP

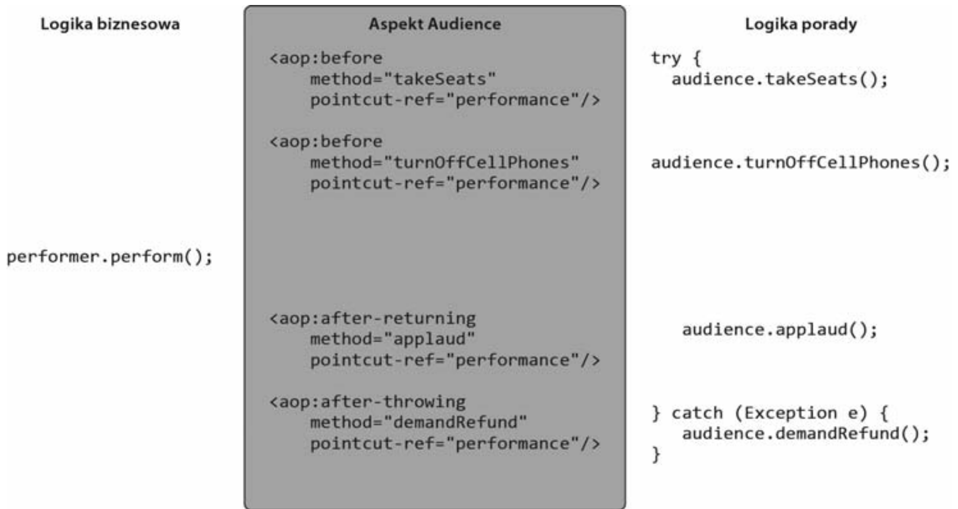
```
<aop:config>
  <aop:aspect ref="audience">                       ← Referencja do komponentu audience
    <aop:before pointcut=
        "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="takeSeats" />                                  ← Przed występem

    <aop:before pointcut=
        "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="turnOffCellPhones" />                          ← Przed występem

    <aop:after-returning pointcut=
        "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="applaud" />                                    ← Po występie

    <aop:after-throwing pointcut=
        "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="demandRefund" />                         ← Po nieudanym występie
  </aop:aspect>
</aop:config>
```

Pierwszym, co zauważamy w temacie elementów konfiguracyjnych Spring AOP, jest fakt, że prawie wszystkie z nich powinny być użyte wewnątrz kontekstu elementu <aop:config>. Jest od tej reguły kilka wyjątków, lecz podczas deklarowania komponentów jako aspektów zawsze zaczynamy od elementu <aop:config>.

Wewnątrz elementu <aop:config> możemy zadeklarować jeden albo więcej aspektów, doradców lub punktów przecięcia. Na listingu 4.2 za pomocą elementu <aop:aspect> zadeklarowaliśmy pojedynczy punkt przecięcia. Atrybut ref zawiera referencję do komponentu w postaci POJO, który będzie użyty jako dostawca funkcjonalności aspektu — w tym wypadku jest to komponent audience. Komponent, do którego referencję zawiera atrybut ref, będzie dostarczał metody wywoływane przez każdą z porad w aspekcie.

Aspekt posiada cztery różne porady. Dwa elementy <aop:before> definiują porady *realizowane przed wykonaniem metody*, które wywołają metody takeSeats() oraz turnOffCellPhones() (zadeklarowane z użyciem atrybutu method) komponentu Audience, przed wykonaniem jakiejkolwiek metody dopasowanej do punktu przecięcia. Element <aop:after-returning> definiuje *poradę realizowaną po powrocie z metody*, która wywoła metodę applaud() po punkcie przecięcia. Jednocześnie element <aop:after-throwing> definiuje *poradę realizowaną po zgłoszeniu wyjątku*, która wywoła metodę demandRefund(), jeśli zostanie zgłoszony jakiś wyjątek. Na rysunku 4.6 pokazano, jak logika porady jest wplatana między logikę biznesową.

We wszystkich elementach porad atrybut pointcut definiuje punkt przecięcia, w którym zostanie zastosowana porada. Wartość, jaką podamy w atrybucie pointcut, jest punktem przecięcia zdefiniowanym w składni wyrażeń punktów przecięcia w stylu AspectJ.



Rysunek 4.6. Aspekt Audience zawiera cztery porady, wplatające swoją logikę wokół metod, które zostaną dopasowane do punktu przecięcia

Zapewne zwróci naszą uwagę fakt, że wszystkie elementy porad posiadają taką samą wartość atrybutu `pointcut`. To dlatego, że wszystkie porady mają zostać zastosowane w tym samym punkcie przecięcia. Widzimy tu złamanie reguły „nie powtarzaj się" (ang. *don't repeat yourself*, DRY). Jeśli później postanowilibyśmy zmienić punkt przecięcia, musielibyśmy dokonać modyfikacji w czterech różnych miejscach.

Aby uniknąć duplikatów w definicji punktów przecięcia, możemy się zdecydować na zdefiniowanie nazwanego punktu przecięcia za pomocą elementu `<aop:pointcut>`. Kod z listingu 4.3 demonstruje sposób użycia elementu `<aop:pointcut>` wraz z elementem `<aop:aspect>` do zdefiniowania nazwanego punktu przecięcia, którego będziemy mogli użyć we wszystkich elementach porad.

Listing 4.3. Definiujemy nazwany punkt przecięcia, by wyeliminować nadmiarowe definicje punktów przecięcia

```
<aop:config>
  <aop:aspect ref="audience">
    <aop:pointcut id="performance" expression=
        "execution(* com.springinaction.springidol.Performer.perform(..))"
        />                          ← Definicja punktu przecięcia
    <aop:before
        pointcut-ref="performance"
        method="takeSeats" />                ←
    <aop:before
        pointcut-ref="performance"
        method="turnOffCellPhones" />        ←
    <aop:after-returning                       Referencje
        pointcut-ref="performance"             do punktu przecięcia
        method="applaud" />                  ←
    <aop:after-throwing
        pointcut-ref="performance"
        method="demandRefund" />             ←
```

```
  </aop:aspect>
</aop:config>
```

Teraz punkt przecięcia jest zdefiniowany w jednym miejscu, do którego odwołuje się wiele elementów porad. Element `<aop:pointcut>` definiuje punkt przecięcia o nazwie `performance`. Jednocześnie wszystkie elementy porad zostały zmienione na odwołania do nazwanego punktu przecięcia za pomocą atrybutu `pointcut-ref`.

Jak pokazano na listingu 4.3, element `<aop:pointcut>` definiuje punkt przecięcia, do którego mogą się odwoływać wszystkie porady wewnątrz tego samego elementu `<aop:aspect>`. Lecz możemy także zdefiniować punkty przecięcia, które będą widoczne z wielu aspektów. W tym celu musielibyśmy umieścić element `<aop:pointcut>` bezpośrednio wewnątrz elementu `<aop:config>`.

4.3.2. *Deklarujemy poradę around*

Obecna implementacja aspektu `Audience` działa rewelacyjnie, lecz podstawowe porady `before` i `after` mają pewne ograniczenia. W szczególności dużym wyzwaniem jest wymiana informacji między parą porad `before` i `after` bez uciekania się do przechowywania tej informacji w zmiennych składowych.

Przykładowo, wyobraźmy sobie, że poza wyłączeniem telefonów komórkowych i oklaskiwaniem wykonawców po zakończeniu, chcielibyśmy, by widzowie zerknęli na zegarki i poinformowali nas o czasie trwania danego występu. Jedynym sposobem osiągnięcia tego za pomocą porad `before` i `after` jest zanotowanie momentu rozpoczęcia w poradzie `before` i raportowanie czasu trwania w jakiejś poradzie `after`. Lecz musielibyśmy przechować moment rozpoczęcia w zmiennej składowej. Ponieważ zaś komponent `Audience` jest instancją klasy singletonowej, nie byłoby to rozwiązaniem bezpiecznym ze względu na pracę wielowątkową, gdybyśmy w taki sposób przechowywali informację o stanie.

Pod tym względem porada `around` ma przewagę nad parą porad `before` i `after`. Za pomocą porady `around` możemy osiągnąć ten sam efekt, co za pomocą osobnych porad `before` i `after`, lecz wykonując wszystkie czynności za pomocą jednej metody. Dzięki umieszczeniu w jednej metodzie całego zbioru porad nie ma potrzeby przechowywania informacji o stanie w zmiennej składowej.

Przykładowo rozważmy nową metodę `watchPerformance()`, zdefiniowaną na listingu 4.4.

Listing 4.4. Metoda watchPerformance() realizuje funkcjonalność aspektowej porady around

```
public void watchPerformance(ProceedingJoinPoint joinpoint) {
    try {
      System.out.println("Widzowie zajmują miejsca.");
      System.out.println("Widzowie wyłączają telefony komórkowe.");
      long start = System.currentTimeMillis();              <— Przed występem

      joinpoint.proceed();         <— Przejście do metody opatrzonej poradą

      long end = System.currentTimeMillis();      <— Po występie
      System.out.println("Brawooo! Oklaski!");
      System.out.println("Występ trwał " + (end - start)
          + " miliskund.");
```

```
    } catch (Throwable t) {
      System.out.println("Buu! Oddajcie pieniądze za bilety!");   ← Po nieudanym
    }                                                                występie
}
```

Pierwszym, co zauważamy w tej nowej metodzie porady, jest fakt, że otrzymuje ona parametr `ProceedingJoinPoint`. Obiekt ten jest konieczny, abyśmy byli w stanie wywołać docelową metodę z wnętrza naszej porady. Metoda porady wykonuje wszystkie swoje zadania, po czym, gdy jest gotowa do przekazania sterowania do metody docelowej, wywoła metodę `proceed()` obiektu `ProceedingJoinPoint`.

Zwróćmy uwagę, że sprawą krytyczną jest, abyśmy pamiętali o umieszczeniu w poradzie wywołania metody `proceed()`. Gdybyśmy o tym szczególe zapomnieli, w efekcie nasza porada blokowałaby dostęp do metody docelowej. Może byłby to efekt zgodny z naszymi zamierzeniami, lecz jest znacznie bardziej prawdopodobne, że naprawdę chcielibyśmy, aby metoda docelowa została w jakimś momencie wywołana.

Jeszcze jedną interesującą informacją jest fakt, że podobnie jak istnieje możliwość zablokowania dostępu do metody docelowej przez pominięcie wywołania metody `pro↳ceed()`, możemy także umieścić w poradzie wielokrotne wywołanie tej metody. Jednym z powodów, by tak postąpić, może być implementowanie logiki ponawiania próby wykonania pewnej czynności w sytuacji, gdy realizująca ją metoda docelowa zakończyła się niepowodzeniem.

W przypadku aspektu `audience` metoda `watchPerformance()` zawiera całą funkcjonalność wcześniejszych czterech metod realizujących porady, lecz tym razem zawartych w jednej, włącznie z tym, że jest odpowiedzialna za obsługę zgłoszonych przez nią samą wyjątków. Zauważmy też, że tuż przed wywołaniem należącej do punktu złączenia metody `proceed()` w lokalnej zmiennej zostaje zapisany bieżący czas. Tuż po odzyskaniu sterowania od wywołanej metody wyświetlany jest komunikat o czasie jej trwania.

Deklaracja porady `around` nie różni się znacząco od deklaracji innych rodzajów porad. Musimy jedynie posłużyć się elementem `<aop:around>`, jak na listingu 4.5.

Listing 4.5. Definiujemy aspekt audience z użyciem porady around

```
<aop:config>
  <aop:aspect ref="audience">
    <aop:pointcut id="performance2" expression=
        "execution(* com.springinaction.springidol.Performer.perform(..))"
        />

    <aop:around
        pointcut-ref="performance2"
        method="watchPerformance()" />        ← Deklaracja porady around
  </aop:aspect>
</aop:config>
```

Podobnie jak w przypadku elementów XML dotyczących innych porad, element `<aop:↳around>` otrzymuje punkt przecięcia oraz nazwę metody realizującej poradę. Użyliśmy tu tego samego punktu przecięcia co wcześniej, lecz atrybutowi `method` nadaliśmy wartość wskazującą na nową metodę `watchPerformance()`.

4.3.3. *Przekazujemy parametry do porady*

Jak dotąd nasze aspekty miały prostą budowę i nie przyjmowały żadnych parametrów. Jedynym wyjątkiem była metoda `watchPerformance()`, którą napisaliśmy w celu realizowania przykładowej porady `around`. Otrzymywała ona parametr `ProceedingJoinPoint`. Poza tym jednym przypadkiem nie zajmowaliśmy naszym poradom uwagi kontrolą wartości parametrów przekazywanych do metody docelowej. Było to jednak jak najbardziej poprawne, ponieważ metoda `perform()`, dla której pisaliśmy porady, nie przyjmowała parametrów.

Zdarzają się jednak sytuacje, gdy może się okazać pożyteczne, by porada nie tylko opakowywała metodę, lecz także kontrolowała wartości parametrów przekazywanych do tej metody.

By się przekonać, jak to działa, wyobraźmy sobie nowy typ zawodnika w turnieju talentów *Idol Springa*. Ten nowy zawodnik będzie występował z pokazami czytania w myślach i zostanie zdefiniowany za pomocą interfejsu `MindReader`:

```
package com.springinaction.springidol;

public interface MindReader{
  void interceptThoughts(String thoughts);

  String getThoughts();
}
```

Interfejs `MindReader` zawiera definicje dwu podstawowych czynności: przechwytywania myśli ochotnika i informowania o ich treści. Prostą implementacją interfejsu `MindReader` jest klasa `Magician`:

```
package com.springinaction.springidol;

public class Magician implements MindReader{
  private String thoughts;

  public void interceptThoughts(String thoughts){
    System.out.println("Przechwytuję myśli ochotnika: ");
    this.thoughts=thoughts;
  }

  public StringgetThoughts(){
    return thoughts;
  }
}
```

Musimy dostarczyć czytającemu w myślach kogoś, kogo myśli mógłby odczytać. W tym celu zdefiniujemy interfejs `Thinker`:

```
package com.springinaction.springidol;

public interface Thinker{
  void thinkOfSomething(String thoughts);
}
```

Klasa `Volunteer` stanowi podstawową implementację interfejsu `Thinker`:

```
package com.springinaction.springidol;

public class Volunteer implements Thinker{
  private String thoughts;

  public void thinkOfSomething(String thoughts){
    this.thoughts = thoughts;
  }

  public String getThoughts(){
    return thoughts;
  }
}
```

Szczegóły klasy `Volunteer` nie są jakoś strasznie interesujące ani ważne. Interesujący jest sposób, w jaki klasa `Magician` będzie przechwytywać myśli klasy `Volunteer` za pomocą Spring AOP.

Aby osiągnąć taki stopień telepatii, będziemy musieli skorzystać z tych samych elementów `<aop:aspect>` oraz `<aop:before>`, których używaliśmy już wcześniej. Lecz tym razem skonfigurujemy je w taki sposób, by przekazywały do porady parametry metody docelowej.

```
<aop:config>
  <aop:aspectref="magician">
    <aop:pointcutid="thinking"
      expression="execution(*
      com.springinaction.springidol.Thinker.thinkOfSomething(String))
           and args(thoughts)"/>

    <aop:before
        pointcut-ref="thinking"
        method="interceptThoughts"
        arg-names="thoughts" />
  </aop:aspect>
</aop:config>
```

Klucz do umiejętności pozazmysłowych klasy `Magician` znajduje się w definicji punktu przecięcia oraz atrybucie `arg-names` elementu `<aop:before>`. Punkt przecięcia identyfikuje metodę `thinkOfSomething()` klasy `Thinker`, oczekującą argumentu typu `String`. Następnie zaś parametr `args` pozwala zarejestrować argument jako wartość `thoughts`.

Jednocześnie deklaracja porady `<aop:before>` odwołuje się do argumentu `thoughts`, wskazując, że powinien on zostać przekazany do metody `interceptThoughts()`.

Od tego momentu, kiedykolwiek zostanie wywołana metoda `thinkOfSomething()` w beanie `volunteer`, klasa `Magician` przechwyci te myśli. By to udowodnić, utworzymy prostą klasę testową z poniższą metodą:

```
@Test
public void magicianShouldReadVolunteersMind(){
  volunteer.thinkOfSomething("Dama Kier");

  assertEquals("Dama Kier", magician.getThoughts());
}
```

Bardziej szczegółowo zagadnienie pisania w Springu testów jednostkowych i testów integracyjnych omówimy w następnym rozdziale. Na razie zapamiętajmy, że rezultat testu okaże się pozytywny, ponieważ `Magician` zawsze będzie wiedział wszystko, o czymkolwiek `Volunteer` pomyśli.

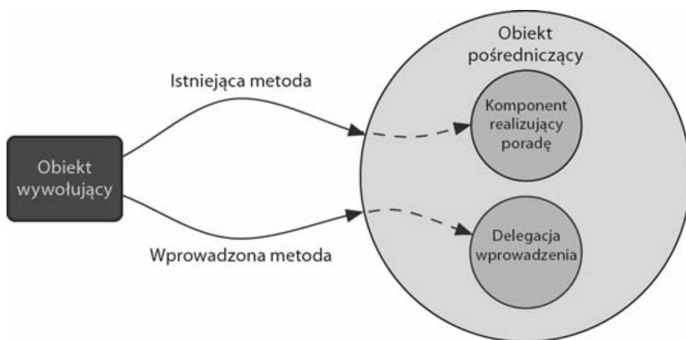
Teraz dowiedzmy się, w jaki sposób użyć techniki Spring AOP, by dodać nową funkcjonalność do istniejących obiektów dzięki mocy wprowadzenia.

4.3.4. Wprowadzamy nową funkcjonalność przez aspekty

W niektórych językach, jak Ruby czy Groovy, istnieje pojęcie klas otwartych. Umożliwiają one dodawanie nowych metod do obiektu albo klasy bez konieczności bezpośredniego modyfikowania definicji takiego obiektu czy też klasy. Niestety, Java nie jest językiem aż tak dynamicznym. Gdy klasa została skompilowana, niewiele można zrobić w celu dodania do niej nowej funkcjonalności.

Lecz jeśli się dobrze zastanowić, czy nie to właśnie staraliśmy się robić w tym rozdziale za pomocą aspektów? Oczywiście, nie dodaliśmy do obiektów żadnej nowej metody, lecz dodawaliśmy nowe mechanizmy obok metod wcześniej definiowanych przez odpowiednie obiekty. Jeśli aspekt może opakowywać istniejące metody w dodatkowe mechanizmy, czemu nie dodać nowych metod do obiektu? Właściwie, korzystając z aspektowej koncepcji zwanej **wprowadzeniem**, aspekty mogą dodawać zupełnie nowe metody do beanów w Springu.

Przypomnijmy sobie, że w Springu aspekty są po prostu obiektami pośredniczącymi, które implementują te same interfejsy, co komponent docelowy. Jaki byłby skutek, gdyby, poza tymi wspólnymi interfejsami, obiekt pośredniczący implementował jeszcze jakiś inny interfejs? Wówczas każdy komponent realizujący poradę aspektu sprawiałby wrażenie, jakby także implementował ten nowy interfejs. Dzieje się tak, mimo że klasa, której instancją jest nasz komponent, nie zawiera implementacji tego dodatkowego interfejsu. Na rysunku 4.7 pokazano, jak to działa.



Rysunek 4.7. Spring AOP pozwala na wprowadzanie nowych metod do komponentu. Obiekt pośredniczący przechwytuje wywołania i deleguje do innego obiektu, który implementuje daną metodę

Na rysunku 4.7 możemy zauważyć, że w momencie wywołania wprowadzonego interfejsu obiekt pośredniczący deleguje wywołanie do pewnego innego obiektu, który zawiera implementację tego nowego interfejsu. W efekcie uzyskujemy pojedynczy komponent, którego implementacja jest rozdzielona pomiędzy więcej niż jedną klasę.

By wcielić ten pomysł w życie, powiedzmy, że chcemy wprowadzić do klas wszystkich wykonawców z naszych przykładów poniższy interfejs Contestant:

```
package com.springinaction.springidol;

public interfaceContestant{
  void receiveAward();
}
```

Zakładamy, że możemy zajrzeć do każdej implementacji klasy Performer i zmodyfikować je wszystkie tak, aby implementowały także interfejs Contestant. Z projektowego punktu widzenia może to jednak być niezbyt rozważny ruch (ponieważ implementacje interfejsów Contestant i Performer niekoniecznie muszą w sensie logicznym zawierać się nawzajem w sobie). Co więcej, może nawet okazać się niemożliwe, by zmienić wszystkie implementacje interfejsu Performer. Szczególnie jeśli pracujemy z implementacjami tworzonymi przez osoby trzecie, możemy nie mieć dostępu do kodu źródłowego.

Szczęśliwie wprowadzenia przez aspekty mogą nam pomóc sobie z tym poradzić, nie wymagając poświęcania decyzji projektowych lub inwazyjnych działań względem istniejących implementacji. Aby skorzystać z tego mechanizmu, musimy posłużyć się elementem <aop:declare-parents>:

```
<aop:aspect>
  <aop:declare-parents
    types-matching="com.springinaction.springidol.Performer+"
    implement-interface="com.springinaction.springidol.Contestant"
    default-impl="com.springinaction.springidol.GraciousContestant"
  />
</aop:aspect>
```

Jak sugeruje znaczenie nazwy elementu <aop:declare-parents>, pozwala on na zadeklarowanie, że komponent, którego dotyczy porada, otrzyma nowe obiekty nadrzędne w hierarchii obiektów. W tym konkretnym wypadku deklarujemy za pomocą atrybutu types-matching, że typy pasujące do interfejsu Performer powinny posiadać także, jako klasę nadrzędną, interfejs Contestant (wskazany przez atrybut implement-interface). Ostatnią sprawą do rozstrzygnięcia jest położenie implementacji metod interfejsu Contestant.

Istnieją dwa sposoby, by wskazać implementację wprowadzonego interfejsu. W tym wypadku korzystamy z atrybutu default-impl, by wprost wskazać implementację za pomocą jej w pełni kwalifikowanej nazwy klasy. Alternatywnie, moglibyśmy wskazać implementację za pomocą atrybutu delegate-ref:

```
<aop:declare-parents
  types-matching="com.springinaction.springidol.Performer+"
  implement-interface="com.springinaction.springidol.Contestant"
  delegate-ref="contestantDelegate"
/>
```

Atrybut delegate-ref odwołuje się do komponentu w Springu jako delegacji wprowadzenia. Zakładamy tu, że komponent o nazwie contestantDelegate istnieje w kontekście Springa:

```
<bean id="contestantDelegate"
      class="com.springinaction.springidol.GraciousContestant"/>
```

Różnica między bezpośrednim wskazaniem delegacji za pomocą atrybutu `default-impl` oraz pośrednim przez atrybut `delegate-ref` polega na tym, że w tym drugim rozwiązaniu mamy komponent Springa, który sam może podlegać wstrzykiwaniu, otrzymać poradę albo w jakiś inny sposób zostać skonfigurowany przez Springa.

4.4. Używamy adnotacji dla aspektów

Podstawową funkcją wprowadzoną w wersji 5 AspectJ była możliwość użycia adnotacji do tworzenia aspektów. We wcześniejszych wersjach pisanie aspektów w AspectJ wymagało nauki rozszerzenia do języka Java. Lecz adnotacyjny model wprowadzony w AspectJ uprościł tworzenie aspektu z dowolnej klasy dzięki wprowadzeniu kilku nowych adnotacji. Ten nowy mechanizm jest znany pod nazwą **@AspectJ**.

Wracając do klasy `Audience`, widzieliśmy, że zawierała ona wszystkie funkcje, jakie powinni realizować widzowie, lecz żadnego ze szczegółów, które czyniłyby z tej klasy aspekt. To zmuszało nas do deklarowania porady i punktów przecięcia w pliku XML.

Lecz dysponując adnotacjami w stylu @AspectJ, możemy wrócić do naszej klasy `Audience` i uczynić z niej aspekt, nie potrzebując do tego celu żadnej dodatkowej klasy ani deklaracji komponentu. Na listingu 4.6 pokazano nową wersję klasy `Audience`, dzięki adnotacjom przemienioną w aspekt.

Listing 4.6. Opatrujemy klasę Audience adnotacją, by utworzyć aspekt

```
package com.springinaction.springidol;

import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Audience{
  @Pointcut(
       "execution(* com.springinaction.springidol.Performer.perform(..))")
  public void performance() {              <- Definicja punktu przecięcia
  }

  @Before("performance()")
  public void takeSeats() {                <- Przed występem
    System.out.println("Widzowie zajmują miejsca.");
  }

  @Before("performance()")
  public void turnOffCellPhones() {        <- Przed występem
    System.out.println("Widzowie wyłączają telefony komórkowe.");
  }

  @AfterReturning("performance()")
  public void applaud() {                  <- Po występie
    System.out.println("Brawooo! Oklaski!");
  }

  @AfterThrowing("performance()")
```

```
  public void demandRefund() {                          ← Po nieudanym występie
    System.out.println("Buu! Oddajcie pieniądze za bilety!");
  }
}
```

Nowa wersja klasy `Audience` jest opatrzona adnotacją `@Aspect`. Adnotacja ta wskazuje, że klasa nie jest już tylko jednym z wielu POJO, lecz jest aspektem.

Adnotacji `@Pointcut` używamy do zdefiniowania punktu przecięcia możliwego do wielokrotnego użycia wewnątrz aspektu w stylu @AspectJ. Wartość przekazana adnotacji `@Pointcut` jest wyrażeniem punktu przecięcia — tu wskazującym, że punkt przecięcia powinien zostać dopasowany do metody `perform()` klasy `Performer`. Nazwa punktu przecięcia pochodzi od nazwy metody, do której zastosowana jest adnotacja. Zatem ten punkt przecięcia będzie się nazywał `performance()`. To, co właściwie znajduje się w ciele metody `performance()`, nie ma specjalnie znaczenia, właściwie mogłaby to być metoda pusta. Sama metoda jest tylko znacznikiem, zapewniającym adnotacji `@Pointcut` punkt, w którym będzie mogła się zaczepić.

Każda spośród metod naszej widowni została opatrzona adnotacją porady. Adnotacja `@Before` została zastosowana zarówno do metody `takeSeats()`, jak i do `turnOffCellPho` ↪`nes()`, wskazując tym samym, że te dwie metody realizują porady *before*. Adnotacja `@AfterReturning` wskazuje, że metoda `applaud()` realizuje poradę `after-returning`. Zaś metoda `demandRefound()` otrzymała adnotację `@AfterThrowing`, zatem zostanie wywołana w razie zgłoszenia jakichś wyjątków podczas występu.

Nazwa punktu przecięcia `performance()` jest przekazana jako wartość parametru do wszystkich adnotacji porad. W ten sposób informujemy metody porad, gdzie powinny zostać zastosowane.

Zauważmy, że poza adnotacjami oraz metodą `performance()` — niewykonującą żadnych operacji — klasa `Audience` pozostała funkcjonalnie bez zmian. To oznacza, że nadal jest ona prostym obiektem Javy i możemy jej używać jako takiego obiektu. Nadal możemy także wiązać tę klasę w Springu w następujący sposób:

```
<bean id="audience"
      class="com.springinaction.springidol.Audience" />
```

Ponieważ klasa `Audience` zawiera wszystko, co potrzebne, by zdefiniować jej własne punkty przecięcia i porady, już więcej nie potrzebujemy deklaracji punktów przecięcia i porad w pliku konfiguracyjnym XML. Pozostała tylko jedna czynność, jaką musimy wykonać, aby Spring zaczął stosować klasę `Audience` jako aspekt. Należy zadeklarować w kontekście Springa komponent automatycznego pośredniczenia, który przekształca komponenty opatrzone adnotacjami w stylu @AspectJ w porady obiektów pośredniczących.

Do tego celu Spring posiada klasę tworzącą automatycznych pośredników, nazwaną `AnnotationAwareAspectJAutoProxyCreator`. Możemy zarejestrować obiekt klasy `Annota` ↪`tionAwareAspectJAutoProxyCreator` jako element `<bean>` w kontekście Springa, lecz wymagałoby to bardzo dużo pisania (uwierz mi, Czytelniku... już kilka razy zdarzyło mi się pisać taką deklarację). Zamiast tego, w celu uproszczenia tej dość długiej nazwy, Spring posiada specjalny element konfiguracyjny w przestrzeni nazw `aop`, który jest znacznie łatwiejszy do zapamiętania:

```
<aop:aspectj-autoproxy />
```

Element `<aop:aspectj-autoproxy/>` utworzy w kontekście Springa komponent klasy `AnnotationAwareAspectJAutoProxyCreator` i będzie automatycznie pośredniczył w wywołaniach z komponentów, których metody zostają dopasowane do punktów przecięcia zdefiniowanych za pomocą adnotacji `@Pointcut` w beanach opatrzonych adnotacją `@Aspect`.

By móc użyć elementu konfiguracyjnego `<aop:aspectj-autoproxy>`, musimy pamiętać o umieszczeniu przestrzeni nazw `aop` w naszym pliku konfiguracyjnym Springa:

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:aop="http://www.springframework.org/schema/aop"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
     http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

Powinniśmy być świadomi, że element `<aop:aspectj-autoproxy>` korzysta jedynie z adnotacji w stylu @AspectJ jako przewodnika podczas tworzenia aspektów opartych na obiektach pośredniczących. Gdybyśmy zajrzeli „pod maskę", przekonalibyśmy się, że nadal są to aspekty w stylu Springa. To jest istotne, ponieważ oznacza, że choć używamy adnotacji w stylu @AspectJ, nadal zachowuje ważność ograniczenie do pośredniczenia jedynie w wywołaniach metod. Jeśli potrzebowalibyśmy wykorzystać pełne możliwości frameworka AspectJ, będziemy musieli posłużyć się bibliotekami uruchomieniowymi AspectJ, zamiast polegać na Springu, pozwalającym jedynie na tworzenie aspektów na bazie obiektów pośredniczących.

Warto także wspomnieć w tym miejscu, że zarówno element `<aop:aspect>`, jak i adnotacje w stylu @AspectJ są efektywnymi sposobami na przekształcenie POJO w aspekt. Element `<aop:aspect>` posiada jednak jedną szczególną zaletę w porównaniu do @AspectJ. Polega ona na tym, że nie potrzebujemy kodu źródłowego klasy, która realizuje funkcjonalność aspektu. Korzystając z @AspectJ, musimy opatrzyć adnotacjami klasę i metody, a to wymaga posiadania dostępu do kodu źródłowego. Element `<aop:aspect>` może zaś odwoływać się do dowolnego komponentu.

Teraz dowiemy się, jak za pomocą adnotacji w stylu AspectJ utworzyć poradę `around`.

4.4.1. Tworzymy poradę around za pomocą adnotacji

Podobnie jak w przypadku konfigurowania Spring AOP z użyciem plików XML, korzystając z adnotacji w stylu @AspectJ, nie jesteśmy ograniczeni do porad `before` i `after`. Możemy także zdecydować się na zastosowanie porady `around`. W tym celu powinniśmy użyć adnotacji `@Around`, jak w poniższym przykładzie:

```
@Around("performance()")
  public voidwatchPerformance(ProceedingJoinPointjoinpoint){
    try {
      System.out.println("Widzowie zajmują miejsca.");
      System.out.println("Widzowie wyłączają telefony komórkowe.");

      longstart=System.currentTimeMillis();
      joinpoint.proceed();
      longend=System.currentTimeMillis();

      System.out.println("Brawooo! Oklaski!");
```

```
      System.out.println("Występ trwał " + (end - start)
          + " miliskund.");
    } catch(Throwable t){
       System.out.println("Buu! Oddajcie pieniądze za bilety!");
    }
  }
```

Adnotacja `@Around` wskazuje, że metoda `watchPerformance()` ma być zastosowana jako porada `around` w punkcie przecięcia `performance`. Powinno to wyglądać w dziwnie znajomy sposób, ponieważ jest to dokładnie ta sama metoda `watchPerformance()`, którą widzieliśmy wcześniej. Jedyna różnica polega na tym, że teraz została ona opatrzona adnotacją `@Around`.

Jak być może pamiętamy z wcześniejszych przykładów, nie wolno zapomnieć o jawnym wywołaniu w poradzie `around` metody `proceed()`, co zapewnia wywołanie metody docelowej. Lecz samo opatrzenie metody adnotacją `@Around` nie wystarczy do umożliwienia wywołania metody `proceed()`. Wiemy, że metoda, która będzie realizowała poradę `around`, musi otrzymać jako argument obiekt `ProceedingJoinPoint`, a następnie wywołać metodę `proceed()` tego obiektu.

4.4.2. Przekazujemy argumenty do porad konfigurowanych przez adnotacje

Dostarczanie parametrów do porad za pośrednictwem adnotacji w stylu @AspectJ nie różni się istotnie od sposobu, w jaki osiągaliśmy to za pomocą deklaracji aspektów w plikach konfiguracyjnych XML Springa. Właściwie, w sporej większości elementy języka XML, których użyliśmy wcześniej, przekładają się prawie bezpośrednio na odpowiedniki w postaci adnotacji w stylu @AspectJ, jak możemy się przekonać na przykładzie nowej wersji klasy `Magician` z listingu 4.7.

Listing 4.7. Tworzymy aspekt z klasy Magician za pomocą adnotacji w stylu @AspectJ

```
package com.springinaction.springidol;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect public class Magician implements MindReader {
  private String thoughts;

  @Pointcut("execution(* com.springinaction.springidol."
      + "Thinker.thinkOfSomething(String)) && args(thoughts)")
  public void thinking(String thoughts) {
  }

  @Before("thinking(thoughts)")
  public void interceptThoughts(String thoughts) {
    System.out.println("Przechwytuję myśli ochotnika: " + thoughts);
    this.thoughts = thoughts;
  }

  public String getThoughts() {
```

Deklaracja parametryzowanego punktu przecięcia (adnotacja do `@Pointcut`)

Przekazujemy parametry do porady (adnotacja do `@Before`)

```
    return thoughts;
  }
}
```

Element <aop:pointcut> został zastąpiony adnotacją `@Pointcut`, zaś element <aop:before> zastąpiła adnotacja `@Before`. Jedyna znacząca zmiana polega na tym, że adnotacje @AspectJ mogą, bazując na składni Javy, wyznaczać szczegółowo parametry przekazywane do porady. Zatem w konfiguracji opartej na adnotacjach nie potrzebujemy odpowiednika atrybutu `arg-names` z elementu <aop:before>.

4.4.3. *Tworzymy wprowadzenie za pomocą adnotacji*

Wcześniej zademonstrowaliśmy sposób na wprowadzenie nowego interfejsu do istniejącego komponentu bez zmian w kodzie źródłowym tego komponentu, za pomocą elementu <aop:declare-parents>. Wróćmy teraz raz jeszcze do tamtego przykładu, lecz tym razem korzystając z konfiguracji aspektów za pomocą adnotacji.

Odpowiednikiem elementu <aop:declare-parents> w @AspectJ jest adnotacja `@Decla` ↪`re-Parents`. Adnotacja ta, użyta wewnątrz klasy opatrzonej adnotacją `@Aspect`, działa niemal identycznie jak element w XML, który zastąpiła. Użycie adnotacji `@DeclareParents` demonstruje listing 4.8.

Listing 4.8. Wprowadzamy interfejs Contestant za pomocą adnotacji w stylu @AspectJ

```
package com.springinaction.springidol;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect public class ContestantIntroducer {

  @DeclareParents(                                  ← Dodajemy interfejs Contestant
      value = "com.springinaction.springidol.Performer+",
      defaultImpl = GraciousContestant.class)
  public static Contestant contestant;
}
```

Jak widzimy, klasa `ContestantIntroducer` jest aspektem. Jednak w odróżnieniu od aspektów, jakie do tej pory utworzyliśmy, nie realizuje ona porad *before*, *after* czy też *around*. Zamiast tego wprowadza do beanów klasy `Performer` interfejs `Contestant`. Podobnie jak element <aop:declare-parents>, adnotacja `@DeclareParents` składa się z trzech części:

- Atrybut `value` jest odpowiednikiem atrybutu `types-matching` w elemencie <aop:declare-parents>. Identyfikuje on rodzaje komponentów, do których wprowadzony będzie interfejs.
- Atrybut `defaultImpl` jest odpowiednikiem atrybutu `default-impl` w elemencie <aop:declare-parents>. Identyfikuje on klasę, która będzie zawierała implementację wprowadzonego interfejsu.
- Statyczna właściwość, która jest opatrzona adnotacją `@DeclareParents`, identyfikuje wprowadzany interfejs.

Podobnie jak w przypadku każdego aspektu, musimy zadeklarować klasę `Contestant➥Introducer` jako komponent w kontekście aplikacji Springa:

```
<bean class="com.springinaction.springidol.ContestantIntroducer" />
```

Element `<aop:aspect-autoproxy>` będzie tam sięgał po ten komponent. Gdy odkryje komponent opatrzony adnotacją `@Aspect`, automatycznie utworzy obiekt pośredniczący, który będzie delegował wywołania albo do docelowego komponentu, albo do obiektu implementującego wprowadzony interfejs, zależnie od tego, do którego z nich wywoływana metoda należy.

Sprawą, która zwróci naszą uwagę, jest fakt, że adnotacja `@DeclareParents` nie posiada odpowiednika atrybutu `delegate-ref` z elementu `<aop:declare-parents>`. Dzieje się tak, ponieważ `@DeclareParents` jest adnotacją w stylu @AspectJ. @AspectJ jest projektem niezależnym od Springa i z tego powodu jego adnotacje nie są świadome istnienia komponentów Springa. Wskutek tego, jeśli potrzebujemy delegata do komponentu, który został skonfigurowany w Springu, wówczas adnotacja `@DeclareParents` może okazać się nieodpowiednia i będziemy musieli uciec się do zastosowania w konfiguracji elementu `<aop:declare-parents>`.

Spring AOP umożliwia oddzielenie zagadnień przecinających od biznesowej logiki aplikacji. Lecz, jak widzieliśmy, aspekty w Springu ciągle jeszcze są oparte na obiektach pośredniczących i obowiązuje je ograniczenie wyłącznie do porad związanych z wywołaniami metod. Jeśli potrzebujemy czegoś więcej niż tylko obsługi pośredniczenia w wywołaniach metod, może warto, byśmy rozważyli użycie aspektów AspectJ. W następnym punkcie zobaczymy, jak można użyć tradycyjnych aspektów z AspectJ w aplikacji Springa.

4.5. Wstrzykujemy aspekty z AspectJ

Choć Spring AOP jest wystarczającym rozwiązaniem dla wielu zastosowań aspektów, wypada słabo w porównaniu z rozwiązaniem AOP, jakim jest AspectJ. AspectJ obsługuje wiele typów punktów przecięcia, które nie są możliwe w Spring AOP.

Punkty przecięcia w konstruktorach, na przykład, są przydatne, gdy potrzebujemy zastosować poradę podczas tworzenia obiektów. W odróżnieniu od konstruktorów w niektórych innych językach obiektowych, konstruktory w Javie różnią się od zwykłych metod. Z tego powodu bazująca na obiektach pośredniczących obsługa programowania aspektowego w Springu okazuje się zdecydowanie niewystarczająca, gdy chcemy zastosować poradę podczas tworzenia obiektu.

W znacznej większości aspekty w AspectJ są niezależne od Springa. Choć mogą być wplatane do aplikacji opartych na Javie, w tym aplikacji w Springu, zastosowanie aspektów z AspectJ wprowadza odrobinę zamieszania po stronie Springa.

Jednak każdy dobrze zaprojektowany i znaczący aspekt prawdopodobnie będzie zależał od innych klas, które będą go wspomagały podczas działania. Jeśli aspekt jest zależny od jednej lub więcej klas w trakcie realizacji porady, możemy z poziomu tego aspektu tworzyć instancje takich współpracujących obiektów. Albo, jeszcze lepiej, możemy posłużyć się wstrzykiwaniem zależności w Springu, by wstrzykiwać komponenty do aspektów w AspectJ.

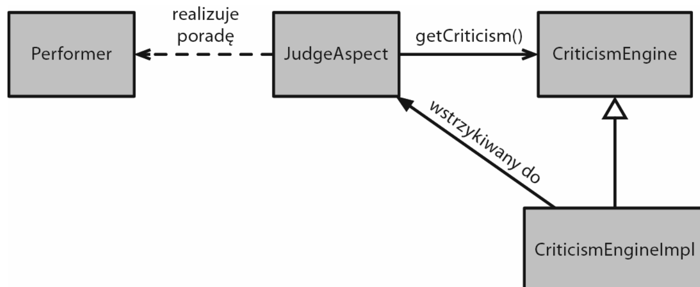
By to zobrazować, utwórzmy kolejny aspekt dla turnieju talentów *Idol Springa*. Turniej talentów potrzebuje jurora. Zatem utworzymy aspekt jurora w środowisku AspectJ. Aspekt ten, zdefiniowany na listingu 4.9, nazwiemy `JudgeAspect`.

Listing 4.9. Implementacja w AspectJ jurora dla turnieju talentów

```
package com.springinaction.springidol;

public aspect JudgeAspect {
  public JudgeAspect() {}

  pointcut performance() : execution(* perform(..));

  after() returning() : performance() {
    System.out.println(criticismEngine.getCriticism());
  }

  // wstrzyknięty obiekt
  private CriticismEngine criticismEngine;
  public void setCriticismEngine(CriticismEngine criticismEngine) {
    this.criticismEngine = criticismEngine;
  }
}
```

Głównym zadaniem aspektu `JudgeAspect` jest komentowanie występu po jego zakończeniu. Punkt przecięcia `performance()` z listingu 4.9 zostanie dopasowany do metody `perform()`. W połączeniu z poradą `after()returning()` otrzymujemy aspekt, który reaguje na zakończenie występu.

Tym, co okazuje się interesujące w listingu 4.9, jest fakt, że juror nie komentuje występu sam we własnym zakresie. Zamiast tego aspekt `JudgeAspect` współpracuje z obiektem `CriticismEngine`, wywołując jego metodę `getCriticism()`, aby uzyskać krytyczny komentarz po występie. By uniknąć niepotrzebnego wiązania między aspektem `JudgeAspect` i obiektem `CriticismEngine`, aspekt `JudgeAspect` otrzymuje referencję do obiektu `CriticismEngine` za pomocą wstrzykiwania przez metodę dostępową. Relacja ta została zobrazowana na rysunku 4.8.



Rysunek 4.8. Aspekty też potrzebują wstrzykiwania. Spring może wstrzykiwać zależności do aspektów w AspectJ, zupełnie jakby to były zwykłe komponenty

`CriticismEngine` sam jest tylko interfejsem, który deklaruje prostą metodę `getCriticism()`. Oto implementacja tego interfejsu (listing 4.10).

Listing 4.10. Implementacja interfejsu CriticismEngine, z którego korzysta aspekt JudgeAspect

```
package com.springinaction.springidol;

public class CriticismEngineImpl implements CriticismEngine {
  public CriticismEngineImpl() {}

  public String getCriticism() {
    int i = (int) (Math.random() * criticismPool.length);

    return criticismPool[i];
  }

  // wstrzyknięty obiekt
  private String[] criticismPool;
  public void setCriticismPool(String[] criticismPool) {
    this.criticismPool = criticismPool;
  }
}
```

Klasa `CriticismEngineImpl` implementuje interfejs `CriticismEngine`, losowo wybierając krytyczny komentarz z listy wstrzykniętej krytyki. Klasa ta może zostać zadeklarowana jako bean w Springu za pomocą poniższego kodu w języku XML:

```
<bean id="criticismEngine"
    class="com.springinaction.springidol.CriticismEngineImpl">
  <property name="criticisms">
    <list>
      <value>I'm not being rude, but that was appalling.</value>
      <value>You may be the least talented person in this show.</value>
      <value>Do everyone a favor and keep your day job.</value>
    </list>
  </property>
</bean>
```

Jak dotąd nieźle. Mamy już implementację interfejsu `CriticismEngine`, z którego będzie korzystał aspekt `JudgeAspect`. Wszystko, co pozostało, to powiązanie klasy `Criticism` ↪`EngineImpl` z aspektem `JudgeAspect`.

Zanim zademonstrujemy, jak zrealizować wstrzykiwanie, powinniśmy wiedzieć, że aspekty w AspectJ mogą być wplatane do naszej aplikacji zupełnie bez angażowania Springa. Jednak jeśli chcemy użyć wstrzykiwania zależności w Springu, by wstrzykiwać klasy współpracujące do aspektu w AspectJ, musimy zadeklarować aspekt jako element `<bean>` w konfiguracji Springa. Poniższa deklaracja elementu `<bean>` realizuje wstrzykiwanie beana `criticismEngine` do aspektu `JudgeAspect`:

```
<bean class="com.springinaction.springidol.JudgeAspect"
    factory-method="aspectOf">
  <propertyname="criticismEngine" ref="criticismEngine"/>
</bean>
```

W dużej części ta deklaracja elementu `<bean>` nie różni się w istotny sposób od wszystkich innych deklaracji komponentów, jakie występują w Springu. Jedyna poważna różnica polega na użyciu atrybutu `factory-method`. Normalnie instancje komponentów w Springu tworzy kontener Springa, lecz aspekty w AspectJ są tworzone przez bibliotekę

uruchomieniową AspectJ. Do momentu, gdy Spring uzyska możliwość wstrzyknięcia komponentu typu `CriticismEngine` do aspektu `JudgeAspect`, istnieje już instancja klasy `JudgeAspect`.

Ponieważ Spring nie odpowiada za tworzenie instancji aspektu `JudgeAspect`, nie możemy po prostu zadeklarować klasy `JudgeAspect` jako komponentu w Springu. Zamiast tego potrzebujemy sposobu, by Spring uzyskał uchwyt do instancji klasy `JudgeAspect`, która została właśnie utworzona przez AspectJ, tak abyśmy mogli wstrzyknąć do niej obiekt `CriticismEngine`. Zgodnie z konwencją, wszystkie aspekty w AspectJ posiadają statyczną metodę `aspectOf()`, która zwraca singleton będący instancją aspektu. Zatem, by uzyskać instancję aspektu, musimy użyć atrybutu `factory-method`, by wywołać metodę `aspectOf()`, zamiast próbować wywoływać konstruktor klasy `JudgeAspect`.

W skrócie, Spring nie korzysta z deklaracji `<bean>`, jakiej używaliśmy wcześniej, do tworzenia instancji klasy `JudgeAspect` — instancja ta została już utworzona przez bibliotekę uruchomieniową AspectJ. Zamiast tego Spring otrzymuje referencję do aspektu przez metodę `aspectOf()` fabryki, a następnie realizuje wstrzykiwanie do niego zależności zgodnie z przepisem w elemencie `<bean>`.

4.6. Podsumowanie

Programowanie aspektowe jest potężnym uzupełnieniem programowania obiektowego. Dzięki aspektom możemy rozpocząć grupowanie zachowań aplikacji, dotychczas rozproszonych po całej aplikacji, w modułach wielokrotnego użytku. Możemy wówczas zadeklarować, gdzie i w jaki sposób dane zachowanie będzie zastosowane. Pozwala to na ograniczenie niepotrzebnego powielania kodu i pozwala, aby podczas konstruowania klas skupić się na ich głównych funkcjach.

Spring zapewnia aspektowe środowisko uruchomieniowe, które pozwala nam na dodawanie aspektów wokół wywołań metod. Nauczyliśmy się, jak możemy wplatać porady przed wywołaniem metody, po jej wywołaniu oraz wokół niego, a także dodać dostosowane zachowanie do obsługi wyjątków.

Mamy możliwość podjęcia kilku decyzji co do sposobu użycia aspektów przez naszą aplikację w Springu. Wiązanie porad i punktów przecięcia w Springu jest znacznie prostsze dzięki dodaniu obsługi adnotacji @AspectJ i uproszczonemu schematowi konfiguracji.

Na koniec, zdarzają się sytuacje, gdy Spring AOP jest mechanizmem niewystarczającym i musimy przejść na AspectJ, aby korzystać z aspektów o większych możliwościach. Na wypadek takich sytuacji zerknęliśmy na sposób użycia Springa, by wstrzykiwać zależności do aspektów w AspectJ.

Do tego momentu omówiliśmy podstawy frameworka Spring. Dowiedzieliśmy się, jak skonfigurować kontener Springa i jak zastosować aspekty do obiektów zarządzanych przez Springa. Jak widzieliśmy, te podstawowe techniki dają świetną możliwość tworzenia aplikacji złożonych z luźno powiązanych obiektów. W następnym rozdziale dowiemy się, w jaki sposób luźne wiązanie przez techniki DI i AOP pozwala na testowanie przez programistę i jak zapewnić pokrycie testami kodu w Springu.

Część 2

Podstawy aplikacji Springa

W części 1. omówiliśmy podstawowy kontener Springa i oferowane przez niego możliwości w zakresie wstrzykiwania zależności (DI) i programowania aspektowego (AOP). Mając tę podstawową wiedzę, w części 2. dowiemy się, w jaki sposób tworzyć aplikacje biznesowe w środowisku Springa.

Większość aplikacji utrwala informacje biznesowe w relacyjnej bazie danych. Rozdział 5., „Korzystanie z bazy danych", dostarczy informacji na temat wsparcia Springa dla utrwalania danych. Jednym z omawianych w tym rozdziale zagadnień będzie wsparcie Springa dla JDBC, dzięki któremu można wydatnie zredukować kod potrzebny przy pracy z JDBC. Omówiona zostanie też integracja Springa z rozwiązaniami trwałości opartymi na mechanizmach odwzorowań obiektowo-relacyjnych, takich jak Hibernate czy JPA.

Utrwalone dane muszą pozostać spójne. W rozdziale 6., „Zarządzanie transakcjami", nauczysz się deklaratywnego stosowania strategii transakcyjnych w obiektach aplikacji przy pomocy programowania aspektowego.

W rozdziale 7., „Budowanie aplikacji sieciowych za pomocą Spring MVC", poznasz podstawy Spring MVC, frameworka sieciowego zbudowanego na bazie Springa. Odkryjesz szeroki wybór kontrolerów Spring MVC do obsługi żądań sieciowych i zobaczysz, jak w przezroczysty sposób podpinać parametry żądania pod obiekty biznesowe, dbając jednocześnie o poprawność danych i obsługę błędów.

Z rozdziału 8., „Praca ze Spring Web Flow", dowiesz się, jak budować oparte na przepływach, konwersacyjne aplikacje sieciowe, używając frameworka Spring Web Flow.

Jako że bezpieczeństwo jest istotnym aspektem każdej aplikacji, w rozdziale 9., „Zabezpieczanie Springa", omówimy wreszcie użycie frameworka Spring Security w celu ochrony informacji aplikacji.

Skorowidz

A

Acegi Security, 250
ACID, 173
ActiveMQ, 338
adnotacja
  @Around, 129
  @Aspect, 129
  @AspectJ, 127
  @Async, 395
  @Autowired, 92–96, 391
  @Bean, 104
  @Configuration, 104
  @Controller, 206
  @DeclareParents, 131
  @Entity, 161
  @Inject, 97
  @ManagedResource, 364
  @PathVariable, 307
  @Pointcut, 128
  @PreAuthorize, 272
  @Qualifier, 95, 98
  @Repository, 163
  @RequestMapping, 307, 331
  @RequestParam, 207
  @ResponseBody, 314
  @RolesAllowed, 271
  @Scheduled, 393
  @Secured, 270
  @SkipIt, 102
  @StringedInstrument, 96
  @Transactional, 180
  @Value, 99
adnotacje standardowe, 92, 100
adres URL, 262
  przepływu, 236
  typu RESTful, 305
  typu RESTless, 305
  zdalnej usługi, 294
agent MBean, 359
algorytm
  MD5, 269
  szyfrowania, 379
AOP, aspect-oriented programming, 22, 30, 108
aplikacja Spitter, 255
aplikacje
  klienckie JMS, 338
  aplikacje sieciowe, 189
architektura JMS, 335
asembler
  InterfaceBasedMBeanInfoAssembler, 363
  MetadataMBeanInfoAssembler, 364
  MethodExclusionMBeanInfo, 362
  MethodNameBasedMBeanInfoAssembler, 361
asemblery informacji MBean, 361
AspectJ, 113, 115
aspekt, 29, 109, 114
  Audience, 120
  w XML, 117
asynchroniczna obsługa komunikatów, 354
asynchroniczne
  RPC, 352
  wywołania, 353
atrybut
  default-init-method, 60
  delegate-ref, 126
  destroy-method, 59
  factory-method, 57
  hash, 267
  init-method, 59
  p:song, 66
  scope, 58
  system-properties-mode, 376
  value, 62
atrybuty
  transakcji, 180
  żądania, 203
automatyczne
  rejestrowanie, 103
  wiązanie, 85–90, 99, *Patrz także* wiązanie
  wykrywanie, 85, 100
autoryzacja @PostAuthorize, 273
AWS, Amazon Web Service, 220

B

bezpieczeństwo
aplikacji sieciowych, 259
na poziomie widoku, 260
sieciowe, 253
biblioteka
JAXB, 324
znaczników, 330
blok catch, 142
błędy walidacji, 214, 216
brokery komunikatów, message brokers, 335, 338
brudne odczyty, dirty reads, 182
Burlap, 287

C

Caucho Burlap, 280
Caucho Hessian, 280
chciwe pobieranie, eager fetching, 158
ciasteczko, 269
CMT, container managed transactions, 174
CRUD, create read update delete, 309
cykl życia komponentu, 37
czas wykonania, runtime, 155

D

dane przepływu, 231
DAO, Data Access Objects, 41, 140
definiowanie
aspektu audience, 122
atrybutów transakcji, 180
bazowego przepływu, 233
kafelków, 201
kontrolera, 195
punktów przecięcia, 120
transakcji, 186
widoku, 202
deklarowanie
aspektów, 117
dostawcy uwierzytelnienia LDAP, 266
fabryki, 160
komponentu, 104
porad, 119
transakcji, 180, 184
desygnator, 116
bean(), 117
execution(), 116
within(), 116
DI, dependency injection, 19, 29, 50, 64, 132, 250
dodawanie funkcjonalności, 125
domyślne właściwości, 376
dostawca uwierzytelnienia, 264, 266
dostęp do
adresu URL, 262
atrybutów komponentu, 368
danych, 41, 139, 154
filtrów serwletów, 252
informacji uwierzytelniających, 260
komponentów, 367
komponentów EJB, 385
metody, 272
S3, 220
składników kolekcji, 81
szablonu JdbcTemplate, 146
usług, 290, 293
usług JMS, 352
usług RMI, 282
zdalnego komponentu, 367
zdalnych usług, 281
dostęp tylko do odczytu, 183
dowiązanie szablonu JMS, 343

E

egzekutor przepływu, 225
EJB, Enterprise JavaBeans, 22
EJB 2, 24
eksport
asynchronicznej usługi, 353
komponentu, 359
eksporter
HttpInvokerServiceExporter, 292
JmsInvokerServiceExporter, 350
RmiServiceExporter, 284, 350
eksportowanie
komponentów, 358
usługi Burlap, 290
usługi Hessian, 288
element
<amq:connectionFactory>, 339
<aop:around>, 122
<aop:aspect-autoproxy>, 132
<aop:aspectj-autoproxy/>, 129
<aop:config>, 119
<aop:declare-parents>, 126
<aop:pointcut>, 120
<authentication-manager>, 264
<beans>, 51
<constructor-arg>, 54, 340
<context:annotation-config>, 92
<context:component-scan>, 100

<context:mbean-export>, 364
<decision-state>, 239
<div>, 202
<end-state>, 229
<entry>, 70, 359
<evaluate>, 232
<filter>, 252
<flow:flow-executor>, 225
<flow:flow-location>, 226
<flow:flow-location-pattern>, 225
<form>, 330
<global-method-security>, 270, 276
<http>, 253
<http-basic>, 256
<if>, 241
<input>, 244
<intercept-url>, 257
<jdbc-user-service>, 264
<jee:jndi-lookup>, 382
<jee:local-slsb>, 385
<jms:listener>, 350
<ldap-server>, 268
<list>, 67
<map>, 67, 70
<mvc:resources>, 193
<null/>, 72
<on-entry>, 246
<property>, 62
<props>, 67
<secured>, 247
<security:authentication>, 260
<security:authorize>, 261
<set>, 67, 69, 232
<sf:errors>, 214
<sf:form>, 330
<subflow-state>, 229
<task:annotation-driven/>, 393
<tx:advice>, 185
<tx:annotation-driven>, 186
<tx:method>, 185
<util:list>, 81
elementy konfiguracyjne, 67, 118
e-mail, 386

F

fabryka
komponentów, 36
menedżerów encji, 165
połączeń JMS, 339
sesji Hibernate, 160
fantomy, phantom read, 182
filtr, 102
annotation, 102
aspectj, 102
assignable, 102
custom, 102
HiddenHttpMethodFilter, 331
regex, 102
filtry
serwletów, 250, 253
Spring Security, 253
format JSON, 329
formularz, 208
logowania, 254
rejestracyjny, 209
formularze typu RESTful, 329
funkcja hasRole(), 272
funkcjonalność porady, 110

H

hasło, 267
hasło szyfrowania, 379
Hessian, 287
Hibernate, 158, 162
hierarchia wyjątków, 142

I

iBATIS, 139
IDE, 113
identyfikatory, 105
implementacja
SpitterDao, 163
SplitterController, 205
informacja o lokalizacji, 326
inicjalizacja na żądanie posiadacza, 57
instalacja
ActiveMQ, 339
Spring Web Flow, 224
interfejs
AlertService, 351
Contestant, 126
CriticismEngine, 134
DisposableBean, 60
InitializingBean, 60
Instrument, 61
javax.management.NotificationListener, 371
MailSender, 386
MindReader, 123
Performer, 50
Session, 160

interfejs
SpitterService, 283
Thinker, 123
izolacja, isolation, 173, 183

J

JAX-RPC, 295
JAX-WS, 295
JDBC, 35, 139, 149–152, 158
JDO, Java Data Objects, 159, 164
jednostka
organizacyjna, organization unit, 267
utrwalania, persistence unit, 165
język
AspectJ, 113, 115
SpEL, 72, 80, 257
JMS, Java Message Service, 41, 334
JMX, Java Management Extensions, 357, 372
JNDI, Java Naming and Directory Interface, 147, 380–384, 396
JPA, Java Persistence API, 149, 164
zarządzane przez aplikację, 165
zarządzane przez kontener, 166
JSP, 330
JSR-330, 97
JTA, Java Transaction API, 174

K

kaskadowość, cascading, 159
katalog
LDAP, 263
META-INF, 165
klasa
Audience, 118, 127
Auditorium, 59
BraveKnight, 27, 31
CriticismEngineImpl, 134
DamselRescuingKnight, 25
HomeController, 196
HttpInvokerServiceExporter, 292
Instrumentalist, 61, 62
java.util.Properties, 81
JdbcDaoSupport, 146
JmsTemplate102, 343
Juggler, 51, 53
Magician, 130
MimeMessageHelper, 389
Minstrel, 33
Piano, 64
PoeticJuggler, 54, 56
Properties, 71
Saxophone, 63
SimpleJdbcDaoSupport, 157
Sonnet29, 55
SpitterEmailServiceImpl, 387
SpitterServiceEndpoint, 296
SpringBeanAutowiringSupport, 296
Stage, 57
StandardPBEStringEncryptor, 379
Volunteer, 123
klasy
bazowe DAO, 145, 156
otwarte, 125
singletonowe, 57
statyczne, 57
szablonowe, 144
klient
REST, 317
SpitterService, 285
usługi RMI Spitter, 285
klucz prywatny, 269
kod szablonowy, 34
kolejka spittle.alert.queue, 353
kolejki, queues, 335
kolekcje typu map, 69
komponent, 50, 59
MBeanServerConnectionFactoryBean, 368
BasicDataSource, 148
dataSource, 377
HessianServiceExporter, 288
JaxWsPortProxyFactoryBean, 299
Minstrel, 32
RmiServiceExporter, 288
SpitterEmailServiceImpl, 387
spitterService, 284
TransactionProxyFactoryBean, 180
komponenty
EJB, 385
encyjne, entity beans, 163
encyjne BMP, 163
encyjne CMP, 163
jako usługi, 292
MBean, 364
MDB, 348
sterowane komunikatami, 347
w kontenerze, 38
wysyłające pocztę, 387
zagnieżdżone, 65
zarządzane, managed beans, 357, 361, 367
dynamic MBeans, 358
model MBeans, 358

open MBeans, 358
standard MBeans, 357
komunikacja
asynchroniczna, 335
RPC, 338
synchroniczna, 334, 337
konfiguracja
Acegi, 251
beanów, 51
bezpieczeństwa sieciowego, 253
brokera komunikatów, 338
danych szyfrowania, 379
fabryki menedżerów encji, 164
JPA, 165
konteneru, 52
kontrolera Hessian, 289
odbiorców komunikatów, 349
puli, 148
rejestru przepływów, 225
repozytorium, 263
serwera LDAP, 268
Spring MVC, 192, 194
Spring Web Flow, 224
Springa, 103, 105
usługi RMI, 283
źródła danych, 147
konflikt nazw komponentów, 366
koniec przepływu, 242
konsumowanie komunikatów, 346
konteksty aplikacji, 28, 36, 203
kontener, 36, 50, 60
odbiorcy komunikatów, 349
podstawowy, 39
kontrola nad atrybutami, 360
kontroler, 193
DisplaySpittleController, 304
HomeController, 195
spittle, 205
konwertery komunikatów HTTP, 314

L

LDAP, 266, 268
LDIF, LDAP Data Interchange Format, 268
leniwe ładowanie, lazy loading, 158
leniwe ładowanie obiektów JNDI, 383
limit czasowy, timeout, 184
Lingo, 352, 354
localhost, 268
logowanie, 254
logowanie przez formularz, 254
luźne wiązanie, 27, 64

M

MBean, 357
MDB, message-driven bean, 334, 347
MDP, message-driven POJO, 348
mechanizm utrwalania danych, 169
menedżer
encji, 164
transakcji, 175
metadane, 321
metoda
addAttribute(), 207
addCustomer(), 242
addInline(), 390
addSpitter(), 154, 211
addSpitterFromForm(), 218
aspectOf(), 135
createMessage(), 345
createMimeMessage(), 388
createSpittle(), 310
delete(), 324
displaySpittle(), 305
embark(), 25, 27
exchange(), 327
getBean(), 58
getCriticism(), 133
getEmployeeById(), 35
getFirst(), 322
getForEntity(), 320
getForObject(), 320
getHeaders(), 321
getLastModified(), 322
getObject(), 346
getRecentSpittles(), 196
getSimpleJdbcTemplate(), 157
getSpitter(), 315
getSpittle(), 307
getStatusCode(), 322
hasPermission(), 274
hasProfanity(), 275
invoke(), 368
listSpittlesForSpitter(), 207
main(), 28
Math.random(), 76
perform(), 91
postForEntity(), 326
postForLocation(), 327
postForObject(), 325
postSpitterForObject(), 326
proceed(), 130
put(), 323

metoda
putSpittle(), 309
queryNames(), 368
receive(), 346
retrieveSpittlesForSpitter(), 318, 321
saveImage(), 220
saveSpittle(), 179, 395
selectSong(), 75
send(), 345
sendSimpleSpittleEmail(), 388
sendSpittleEmailWithAttachment(), 389
setNotificationPublisher(), 371
setSong(), 62
setSpittlesPerPage(), 368
showHomePage(), 196, 360, 362
singBeforeQuest(), 33
toList(), 246
toUpperCase, 75
toUpperCase(), 75
updateSpitter(), 317
updateSpittle(), 323
watchPerformance(), 121–123
metody
asynchroniczne, 395
do przetwarzania zasobów, 308
dostępowe, 361
eksportowane, 363
fabryki, 56
HTTP, 308
idempotentne, 308
RestTemplate, 319
zaplanowane, 393
miejsca docelowe, destinations, 335
domyślne, 345
komunikatów, 340, 345
MIME, Multipurpose Internet Mail Extensions, 388
model, 191
model-widok-kontroler, 42
moduł AOP, 41
moduły, 39
moduły Spring Security, 251
MVC, Model-View-Controller, 42, 189

N

nadpisywanie właściwości, 377
nagłówek Accept, 315
negocjacja zawartości, 311, 314
niepodzielność, atomicity, 173
niepowtarzalne odczyty, nonrepeatable reads, 182

O

obiekt
EntityManagerFactory, 167
HttpInvoker, 292
MDP, 349
ResponseEntity, 322
obiekty
dostępu do danych, 41
nadpisujące właściwości, 374, 377
pośredniczące, 299, 369
pośrednika zdalnego, 369
pośredników, 291
wywołujące HTTP, 294
zastępujące właściwości, 374, 378
obraz wewnętrzny, inline image, 390
obsługa
AOP, 113
filtrów, 253
komunikatów, 335
publikacja-subskrypcja, 336
punkt-punkt, 336
komunikatów asynchroniczna, 354
komunikatów JMS, 351
plików, 218
powiadomień, 370
REST, 303
RPC, 281
wyjątków, 150, 341
żądań, 212
żądań przepływu, 226
odbieranie
komunikatów, 341, 346, 349
powiadomień, 371
odwzorowania obiektowo-relacyjne, 41, 159
odwzorowanie SimpleUrlHandlerMapping, 290
ograniczenia bezpieczeństwa, 276
okresowe uruchamianie zadań, 394
operacje matematyczne, 76
operator
!, 116
&&, 116
?, 75
[], 82
and, 117
Elvis, 79
not, 117
or, 117
T(), 75, 246
wyboru, 82
operatory w SpEL, 76
ORM, object-relational mapping, 41, 139, 159

osadzanie parametrów w adresach, 306
OSGi Blueprint Container, 44

P

pakiet org.springframework.beans.factory.
annotation, 98
parametry nazwane, 156
plik
    coupon.png, 389
    db.properties, 375
    deliveryWarning.jspx, 241
    emailTemplate.vm, 392
    foo.xml, 37
    home.jsp, 203
    knights.xml, 28
    konfiguracyjny, 66
    list.jsp, 208
    persistence.xml, 165
    spitter-security.xml, 252
    spring-idol.xml, 52
    users.ldif, 268
pliki
    JAR, 40
    LDIF, 268
pobieranie
    danych, 152
    zasobów, 320
podprzepływ
    order, 243
    płatności, 245
    zamówienia, 245
POJO, Plain Old Java Obiects, 22
pole
    _hidden, 331
    _method, 330
połączenie z bazą danych, 374
porada, 110, 119
    after, 33, 119
    around, 121, 129
    before, 33, 119
porównywanie haseł, 267
postautoryzacja metod, 273
postfiltrowanie metod, 273
pośrednik RMI, 286
powiadomienia JMX, 370
powiązania między obiektami, 50
poziomy izolacji, 182
preautoryzacja metod, 272
producent widoków, 198, 200, 312
program Cron, 395
programowanie
    aspektowe, 22, 30, 108
    funkcyjne, 302
    transakcji, 178
propagacja, 181
protokół JMXMP, 367
przechwytywanie żądań, 257
przeciążanie operacji, 319
przejścia, transitions, 227, 230
przejścia globalne, 231
przejście cancel, 231
przekazywanie argumentów do porad, 130
przekształcanie
    w komponent, 364
    w aspekt, 33
przepływ, 231
    _flowExecutionKey, 239
    bazowy, 233
    customer, 242
    identyfikujący klienta, 237
    nadrzędny, 234
przestrzeń nazw, 52
    amq, 339
    Spring Security, 251
przesyłanie plików, 221
przetwarzanie
    formularza, 208, 211
    warunkowe, 78
publikacje komponentu zarządzanego, 365
publikator NotifcationPublisher, 370
pula komponentu BasicDataSource, 148
punkty
    końcowe, 295–298
    przecięcia, 111, 132
    złączenia, 110, 114

R

reguła DRY, 120
reguły walidacji, 213
rejestr przepływów, 225
relacja alternatywy, 116
repozytoria, 140
resolver danych wieloczęściowych, 221
REST, Representational State Transfer, 190, 221,
301, 318
RESTful, 304
RESTless, 303
RMI, Remote Method Invocation, 42, 280, 282
rodzaje
    adnotacji, 92
    menedżerów encji, 164
    stanów, 227

rozszerzenia Springa, 46
RPC, remote procedure call, 280, 350
rzutowanie kolekcji, 82

S

schematy działania, workflow, 189
serializacja obiektów, 291
serwer
JMXConnectorServer, 367
LDAP, 268
MBean, 359
pocztowy, 387
serwlet
dyspozytora, 191
nasłuchujący, 204
sesja pocztowa JNDI, 387
sesje kontekstowe, 162
skanowanie komponentów, 102
składowe przepływu, 227
SOA, service-oriented architecture, 294
specyfikacja LDIF, 268
SpEL, Spring Expression Language, 72, 80, 257
spójność, consistency, 173
Spring
3.0, 47
AOP, 117
Batch, 43
Dynamic Modules, 44
Faces, 48
Integration, 43
JavaScript, 48
LDAP, 44
Mobile, 44
MVC, 190, 255
Rich Client, 45
Roo, 45
Security, 43, 250
Security 2.0, 48
Security 3.0, 271
Social, 44
Web Flow, 43, 48, 224, 247
Web Services, 43
.NET, 45
-Flex, 45
stałe propagacji, 181
stan
addCustomer, 242
buildOrder, 236
checkDeliveryArea, 238
createPizza, 242, 244
identifyCustomer, 235, 236
registrationForm, 240
showOrder, 242
takePayment, 236
thankCustomer, 236
welcome, 239, 241
stany, states, 227
akcji, 228
decyzyjne, 228
końcowe, 229, 236
obiektów, 317
początkowe, 235
podprzepływów, 229
widoków, 227, 243
zasobów, 315
sterownik JDBC, 149
stopnie izolacji, 183
szablon, template, 144
JdbcTemplate, 153, 155
JMS Springa, 340
JmsTemplate, 342
NamedParameterJdbcTemplate, 153
RestTemplate, 318
SimpleJdbcTemplate, 153, 155
TransactionTemplate, 179
Velocity, 391, 393
szablony
dostępu do danych, 143
JDBC, 153
wiadomości, 391
szyfrator łańcuchów, string encryptor, 379
szyfrowanie, 267
szyfrowanie właściwości, 378

T

technologie zdalnego dostępu, 281
tematy, topics, 335
testowanie, 42
testowanie kontrolera, 197
transakcje, 171
Hibernate, 176
Java Persistence API, 177
Java Transaction API, 178
JDBC, 175
programowe, 180
transformacja żądań, 331
trwałość, durability, 173
tworzenie
aspektów, 127
beanów, 59
fabryki połączeń, 339
klientów REST, 317

tworzenie
komponentów, 56
kwalifikatorów, 95
obiektów POJO, 347
odbiorcy komunikatów, 348
porady around, 129
punktów końcowych, 295
spittle'ów, 310
szablonów wiadomości, 391
usługi RMI, 283
wprowadzenia, 131
typy MIME, 312

U

uaktualnianie na gorąco, 383
udostępnianie
metod, 361
zdalnych komponentów, 367
uprawnienie
ROLE_ADMIN, 247
ROLE_SPITTER, 266
URI, uniform resource identifier, 305
URL, uniform resource locator, 305
usługa
Burlap, 290
Simple Storage Service, 219
Spitter, 187, 286, 290
usługi
asynchroniczne, 353
Hessian, 288
RMI, 282, 283
sieciowe, 42, 294
systemowe, 30
użytkownika, 264
zdalne, 280, 282, 300
usuwanie
komponentu, 60
spittle, 274
zasobów, 324
uwierzytelnianie
HTTP, 256
użytkowników, 263–269
za pomocą bazy danych, 264
za pomocą LDAP, 266

W

walidacja, 189, 213
warstwa sieciowa aplikacji, 190, 194
wiadomość MIME, 388
wiązanie, 27
automatyczne, 85–90, 99
autodetect, 87, 90
byName, 86
byType, 86, 88
constructor, 87, 89
jawne, 91
kolekcji, 67
kolekcji typu map, 69
kolekcji właściwości, 71
list, 68
komponentów EJB, 385
między obiektami, 25
null, 72
obiektów JNDI, 380
opcjonalne, 94
punktów końcowych, 296
za pomocą adnotacji, 92
za pomocą wyrażeń, 72
widok, 191
formularza, 210
strony głównej, 202
widoki
Tiles, 200
wewnętrzne, 198
właściwości
komponentów, 62
systemowe, 376
właściwość
connectionFactory, 354
database, 167
destination, 354
managedInterfaces, 363
notificationListenerMappings, 372
registrationBehaviorName, 366
serviceInterface, 354
spittlesPerPage, 358
wplatanie, 111
wprowadzenie, 111, 125
WSDL, 300
współbieżność, 182
wstrzykiwanie
obiektów JNDI, 382
przez adnotacje, 99
przez konstruktor, 26, 56
przez wywołanie metod, 62
referencji, 54, 63
zależności, 29, 50, 64, 132, 250
wycofanie transakcji, 184
wydajność
postrzegana, 395
rzeczywista, 395

wyjątek
CustomerNotFoundException, 240
DataAccessExeption, 143
java.rmi.RemoteException, 281
NullPointerException, 75
SQLException, 141, 151
wyjątki
dostępu do danych, 143
JDBC, 143
kontrolowane, checked, 184
wykrywanie komponentów, 100
wyliczenie PaymentType, 246
wylogowywanie, 256
wymiana zasobów, 327
wymuszanie HTTPS, 259
wyodrębnianie konfiguracji, 374
wyrażenia
bezpieczeństwa, 258
logiczne, 78
programu Cron, 395
regularne, 79
w SpEL, 73–76
wyrażenie hasRole(), 258
wysyłanie
danych zasobu, 325
formularzy, 329
komunikatów, 341, 344
plików, 217
spittle'a, 344
wiadomości, 386–390
zasobów, 323
wyszukiwanie obiektów, 380
wyświetlenie widoku, 207
wywołania zwrotne, callbacks, 144
wzorzec, 33, 192

Z

zabezpieczanie
metod, 270, 272
przepływu, 247
żądań sieciowych, 252
zabezpieczenia na poziomie kanału, 258
zagadnienia
przecinające, 108
przekrojowe, 29
zakres
komponentów, 58
propagacji, 181
załączniki wiadomości, 388
zapis pliku, 219
zapis w pamięci podręcznej, 383
zapytania SQL, 152
zarządzanie
stanem, 189
transakcjami, 174
zarządzany atrybut, managed attribute, 359
zasięg
danych przepływu, 232
kontekstu, 59
zasoby REST, 303
zastępowanie
właściwości, 375
zmiennych, 376
zdalne
usługi, 280, 282, 300
wywołanie procedury, 280
wywoływanie metod, 42, 280
zdalny dostęp, remoting, 280
zdalny dostęp JMX, 367
zintegrowane środowisko programistyczne, 113
zmienne środowiskowe, 379
znacznik, *Patrz także* element
<security:authentication>, 260
<security:authorize>, 261
<sf:errors>, 214
<sf:form>, 330
znak &, 117
znak kontynuacji wiersza, 17
znaki kropek (..), 116

Ź

źródło danych
JDBC, 149
JNDI, 147
z pulą, 147

Ż

żądania przepływu, 226
żądanie
DELETE, 310, 324
GET, 309, 320
HttpServletRequest, 252
POST, 310, 325
PUT, 322

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA

Sprawdź, jak Spring Framework ułatwia życie!

Spring Framework odmienił sposób konstruowania aplikacji w języku Java. Prawdopodobnie jest on najczęściej wybieranym narzędziem do tworzenia aplikacji, niezależnie od tego, czy są one internetowe, czy biurkowe. Czym zasłużył sobie na taką popularność? Niezwykle wygodna konfiguracja, ogromna liczba różnego rodzaju bibliotek, przemyślana architektura to tylko niektóre z jego atutów. Jeżeli do tego dodać ogromną społeczność, chętną do udzielania wszelkich porad, otrzymujemy wyjątkowe narzędzie do zadań specjalnych.

Kolejne wydanie książki uwzględnia wszystkie zmiany wprowadzone w trzeciej wersji Spring Framework. A jest ich sporo. Dzięki nim praca z tym narzędziem stała się jeszcze prostsza. W trakcie lektury dowiesz się, jak zminimalizować użycie XML do konfiguracji, wstrzykiwać zależności oraz korzystać z potencjału programowania aspektowego.

Ponadto znajdziesz tu komplet informacji na temat współpracy z bazami danych, a takie terminy jak transakcje, JPA, JDBC przestaną być Ci obce. Twoją ciekawość powinien wzbudzić rozdział poświęcony bezpieczeństwu — dzięki Spring Security implementacja tego kluczowego elementu aplikacji staje się o niebo przyjaźniejsza. Książka ta jest skarbnicą informacji o Spring Framework, którą powinien zainteresować się każdy programista języka Java!

Dzięki Spring Framework:

- zbudujesz lepszą architekturę Twojej aplikacji
- wykorzystasz potencjał aspektów i wstrzykiwania zależności
- zagwarantujesz bezpieczeństwo Twoim rozwiązaniom
- stworzysz zaawansowaną aplikację!

Nr katalogowy: 11955

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne: 0 801 339900
0 601 339900

helion.pl księgarnia internetowa

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Cena: 69,00 zł

ISBN 978-83-246-4888-7

Informatyka w najlepszym wydaniu