



Java[®] EE

Zaawansowane wzorce projektowe

Tytuł oryginału: Professional Java® EE Design Patterns

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-1315-6

Copyright © 2015 by John Wiley & Sons, Inc., Indianapolis, Indiana.

All Rights Reserved.

This translation published under license with the original publisher John Wiley & Sons, Inc.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise without either the prior written permission of the Publisher.

The Wrox Brand trade dress is a trademark of John Wiley & Sons, Inc. in the United States and/or other countries. Used by permission.

Wiley, the Wiley logo, Wrox, the Wrox logo, Wrox Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Java is registered trademark of Oracle America, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

Translation copyright © 2015 by Helion S.A.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek 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/javeez>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

O autorach	11
O korektorze merytorycznym	13
Podziękowania	15
Przedmowa	17
Wprowadzenie	19

CZĘŚĆ I WPROWADZENIE DO WZORCÓW PROJEKTOWYCH W JAVIE EE

Rozdział 1. Krótki przegląd wzorców projektowych	27
Czym jest wzorzec projektowy	28
Jak odkryto wzorce i do czego są potrzebne	29
Wzorce w realnym świecie	30
Podstawy wzorców projektowych	31
Wzorce w rozwiązaniach dla biznesu	31
Od Javy do Javy dla biznesu	31
Pojawienie się wzorców Javy dla przedsiębiorstw	32
Wzorce projektowe a wzorce biznesowe	33
Stare dobre wzorce projektowe spotykają Javę EE	33
Kiedy wzorce stają się antywzorcami	35
Podsumowanie	35
Rozdział 2. Podstawy Javy EE	37
Architektura wielowarstwowa	38
Warstwa kliencka	39
Warstwa logiki biznesowej	40
Komponent sieciowy	40
Warstwa EIS	40
Serwery Java EE	41
Profil sieciowy Javy EE	42
Podstawowe zasady Javy EE	42

Konwencja przed konfiguracją	43
CDI	43
Interceptory	44
Podsumowanie	45
Ćwiczenia	45

CZĘŚĆ II IMPLEMENTACJA WZORCÓW PROJEKTOWYCH W JAVIE EE

Rozdział 3. Wzorzec Fasada	49
Czym jest Fasada	50
Diagram klas wzorca Fasady	51
Implementacja wzorca Fasada w zwykłym kodzie źródłowym	52
Implementacja wzorca Fasada w Javie EE	53
Fasada z ziarnami bezstanowymi	53
Fasada ze stanowym ziarnem	55
Gdzie i kiedy używać wzorca Fasada	55
Podsumowanie	56
Ćwiczenia	56
Rozdział 4. Wzorzec Singleton	57
Czym jest Singleton	58
Diagram klas wzorca Singleton	59
Implementacja wzorca Singleton w zwykłym kodzie	59
Implementacja wzorca Singleton w Javie EE	63
Ziarna singletonowe	63
Wykorzystywanie singletonów przy uruchamianiu	64
Określanie kolejności uruchamiania	65
Współbieżność	67
Gdzie i kiedy używać wzorca Singleton	69
Podsumowanie	69
Ćwiczenia	70
Rozdział 5. Wstrzykiwanie zależności i CDI	71
Na czym polega wstrzykiwanie zależności	72
Implementacja wstrzykiwania zależności w zwykłym kodzie	72
Implementacja wstrzykiwania zależności w Javie EE	75
Adnotacja @Named	76
Wstrzykiwanie zależności i kontekst (CDI)	77
CDI a EJB	78
Ziarna CDI	78
Adnotacja @Inject	79
Konteksty i zakres	79
Nazewnictwo i EL	80
Ziarna CDI jako wsparcie dla JSF	80
Kwalifikatory	81
Alternatywy	81

Stereotypy	82
Inne wzorce związane z CDI	82
Podsumowanie	83
Ćwiczenia	83
Rozdział 6. Wzorzec Fabryka	85
Czym jest Fabryka	86
Metoda Fabryczna	86
Implementacja Metody Fabrycznej w zwykłym kodzie	88
Fabryka Abstrakcyjna	89
Implementacja Fabryki Abstrakcyjnej w zwykłym kodzie	90
Implementacja wzorca Fabryka w Javie EE	91
Okiełznać moc kontenera CDI	97
Gdzie i kiedy używać wzorców fabrycznych	100
Podsumowanie	101
Ćwiczenia	101
Rozdział 7. Wzorzec Dekorator	103
Czym jest wzorzec Dekorator	104
Diagram klas wzorca Dekorator	105
Implementacja wzorca Dekorator w zwykłym kodzie	106
Implementacja wzorca Dekorator w Javie EE	109
Dekoratory bez konfiguracji XML	113
Gdzie i kiedy używać wzorca Dekorator	114
Podsumowanie	115
Ćwiczenia	115
Rozdział 8. Programowanie aspektowe (interceptory)	117
Co to jest programowanie aspektowe	118
Implementacja AOP w zwykłym kodzie	120
Aspekty w Javie EE i interceptory	122
Cykl życia interceptora	125
Interceptory na poziomie domyślnym	125
Kolejność interceptorów	126
Interceptory CDI	128
Gdzie i kiedy używać interceptorów	130
Podsumowanie	131
Rozdział 9. Asynchroniczność	133
Co to jest programowanie asynchroniczne	134
Wzorzec Asynchroniczność	134
Implementacja asynchroniczności w zwykłym kodzie	136
Programowanie asynchroniczne w Javie EE	138
Asynchroniczne ziarna	138
Asynchroniczne serwlety	140
Gdzie i kiedy stosować programowanie asynchroniczne	143
Podsumowanie	144
Ćwiczenia	145

Rozdział 10. Usługa odmierzenia czasu	147
Czym jest usługa odmierzenia czasu	148
Implementacja czasomierza w Javie EE	150
Czasomierze automatyczne	150
Czasomierze programowe	151
Wyrażenia czasowe	153
Transakcje	156
Podsumowanie	156
Ćwiczenia	157
Rozdział 11. Wzorzec Obserwator	159
Czym jest Obserwator	160
Opis	160
Diagram klas wzorca Obserwator	162
Implementacja wzorca Obserwator w zwykłym kodzie	162
Implementacja wzorca Obserwator w Javie EE	164
Gdzie i kiedy używać wzorca Obserwator	169
Podsumowanie	170
Ćwiczenia	170
Rozdział 12. Wzorzec Dostęp do Danych	171
Czym jest wzorzec Dostęp do Danych	172
Diagram klas Dostępu do Danych	172
Ogólne informacje o wzorcu Dostęp do Danych	173
Wzorzec Obiekt Transferu Danych	173
API JPA i ORM	174
Implementacja wzorca Dostęp do Danych w Javie EE	174
Implementacja DAO bezpieczna pod względem typów	179
Gdzie i kiedy używać wzorca Dostęp do Danych	180
Podsumowanie	180
Ćwiczenia	180
Rozdział 13. REST-owe usługi sieciowe	181
Co to jest REST	182
Sześć warunków REST	183
Klient-serwer	183
Jednolity interfejs	184
Bezstanowość	184
Możliwość zapisywania danych w buforze	184
System warstwowy	184
Kod na żądanie	184
Model dojrzałości Richardsona	185
Poziom 0. — zwykły XML	185
Poziom 1. — zasoby	185
Poziom 2. — czasowniki HTTP	185
Poziom 3. — kontrolki hipermedialne	185

Projektowanie REST-owego interfejsu API	185
Nazewnictwo zasobów	186
Rzeczowniki, a nie czasowniki	186
Znaczenie nazw	187
Liczba mnoga	187
Metody HTTP	187
REST w akcji	188
Rzeczownik users	188
Rzeczowniki topics i posts	189
Implementacja REST w Javie EE	191
HATEOAS	194
Gdzie i kiedy używać REST	196
Podsumowanie	197
Ćwiczenia	197
Rozdział 14. Wzorzec Model – Widok – Kontroler	199
Czym jest wzorzec MVC	200
Typy wzorca MVC	201
Implementacja wzorca MVC w zwykłym kodzie	202
Implementacja wzorca MVC w Javie EE	206
Servlet FacesServlet	206
Implementacja wzorca MVC przy użyciu servletu FacesServlet	206
Gdzie i kiedy używać wzorca MVC	208
Podsumowanie	209
Ćwiczenia	209
Rozdział 15. Inne wzorce projektowe w Javie EE	211
Co to są gniazda sieciowe	212
Co to jest warstwa pośrednia do przekazywania wiadomości	214
Co to jest architektura mikrousługowa	215
Architektura monolityczna	215
Skalowalność	217
Dekompozycja na usługi	218
Zalety architektury mikrousługowej	218
Nie ma nic za darmo	219
Wnioski	220
Kilka antywzorców na zakończenie	220
Uberklasa	220
Architektura Lazani	221
Pan Kolumb	221
Korzyści z przyjaźni	221
Supernowoczesność	222
Szwajcarski scyzoryk	222

CZĘŚĆ III PODSUMOWANIE

Rozdział 16. Wzorce projektowe — dobre, złe i brzydkie 225

 Dobre — wzorce sukcesu225

 Złe — nadużywanie i błędne stosowanie wzorców227

 Brzydkie228

 Podsumowanie230

Skorowidz 231

Wstrzykiwanie zależności i CDI

ZAWARTOŚĆ ROZDZIAŁU:

- wprowadzenie do technik wstrzykiwania zależności;
- znaczenie wstrzykiwania zależności w Javie EE;
- implementacja wstrzykiwania zależności w zwykłym kodzie;
- implementacja wstrzykiwania zależności w Javie EE;
- wprowadzenie do kontekstowego wstrzykiwania zależności;
- najważniejsze różnice dzielące kontenery CDI i EJB.

PRZYKŁADY KODU DO POBRANIA

Pliki z kodem źródłowym przykładów z tego rozdziału znajdują się w archiwum, które można pobrać z serwera FTP wydawnictwa Helion pod adresem: <ftp://ftp.helion.pl/przyklady/javeez.zip>. Pliki znajdują się w folderze *r05*, a ich nazwy odzwierciedlają kolejne fragmenty rozdziału.

Wstrzykiwanie zależności (ang. *Dependency Injection* — DI) to jeden z nielicznych powszechnie znanych i stosowanych wzorców projektowych, które *nie* zostały opisane w książce Bandy Czworga¹. Ale obecnie wzorzec ten jest wykorzystywany w nowoczesnych językach programowania zarówno do implementacji mechanizmów wewnętrznych, jak i jako środek do rozluźniania powiązań między klasami.

Technologia J2EE miała służyć do budowania najbardziej złożonych systemów, ale spisała się marnie, ponieważ tylko nadmiernie komplikowała proces tworzenia nawet prostszych systemów. Pierwotny projekt J2EE był oparty na wysokim poziomie złożoności i ścisłych powiązaniach między klasami,

¹ Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku*, Helion, 2010.

co stało się bodźcem do powstania i spopularyzowania różnych systemów szkieletowych, takich jak Spring czy PicoContainer. W 2004 r. Martin Fowler opublikował artykuł na temat kontenerów odwróconego sterowania (ang. *Inversion of Control* — IoC) i wzorca wstrzykiwania zależności². Większość producentów oprogramowania nie zachęcała programistów do używania kontenera J2EE. Wkrótce kontrolę przejęły lekkie kontenery, które zaczęto oficjalnie obsługiwać, i na dodatek Spring stał się w zasadzie nieoficjalnym standardem, doprowadzając do tego, że Javę biznesową postanowiono zaprojektować od nowa.

Na czym polega wstrzykiwanie zależności

Wzorzec Wstrzykiwanie Zależności opiera się na pomysłcie odwrócenia tego, do kogo należy kontrola. Zamiast tworzyć zależności i nowe obiekty za pomocą słowa kluczowego `new` lub przy użyciu operacji wyszukiwania, potrzebne zasoby wstrzykuje się do obiektu docelowego. Podejście to ma wiele zalet:

- Klient nie musi wiedzieć o różnych implementacjach wstrzykiwanych zasobów, co ułatwia wprowadzanie zmian w projekcie.
- Znacznie łatwiej implementuje się testy jednostkowe z wykorzystaniem atrap obiektów.
- Konfigurację można przenieść na zewnątrz, redukując w ten sposób skutki zmian.
- Architektura oparta na luźnych powiązaniach ułatwia tworzenie systemów wtyczkowych.

Podstawową ideą techniki wstrzykiwania zależności jest zmiana miejsca tworzenia obiektów oraz wykorzystanie wtryskiwacza (ang. *injector*) do wstrzykiwania w odpowiednim momencie specyficznych implementacji do obiektów docelowych. Może się to wydawać podobne do implementacji wzorca Fabryka (opisanego w rozdziale 6., „Wzorzec Fabryka”), ale koncepcja ta jest znacznie bardziej zaawansowana niż proste tworzenie obiektu. Odwrócenie sterowania całkowicie wywraca do góry nogami relacje między obiektami i przekazuje całą pracę do wtryskiwacza (który w większości przypadków działa magicznie). Zamiast wywoływać fabrykę, aby dostarczyć implementację wywołującemu, wtryskiwacz aktywnie sprawdza, kiedy obiekt docelowy potrzebuje obiektu źródłowego, i dokonuje wstrzyknięcia w odpowiedni sposób.

Implementacja wstrzykiwania zależności w zwykłym kodzie

Standardowa implementacja wstrzykiwania zależności w Javie poza kontenerem EJB jest dostępna dopiero od czasu wprowadzenia CDI (ang. *Context and Dependency Injection*). Choć istnieją różne systemy szkieletowe, jak choćby Spring i Guice, nietrudno jest samodzielnie zaimplementować podstawowe rozwiązanie.

Najprostsza implementacja wzorca Wstrzykiwanie Zależności to fabryka tworząca zależność na żądanie przy użyciu metody `getInstance()`. Poniżej przedstawiamy taką implementację, aby pokazać, jak to się robi w zwykłym kodzie.

W implementacji tej powinno się oddzielić rozwiązywanie zależności od zachowania klasy. Oznacza to, że klasa powinna mieć określoną funkcjonalność bez definiowania tego, jak pozyskuje referencje do klas, od których zależy. W ten sposób rozłącza się operację tworzenia obiektu i miejsce jego użycia — a to jest esencją wstrzykiwania zależności.

² Martin Fowler, *Inversion of Control Containers and the Dependency Injection Pattern*, 2004, <http://martinfowler.com/articles/injection.html>.

Najpierw na listingach: 5.1, 5.2, 5.3 i 5.4 przedstawimy przykłady silnie powiązanych klas, a następnie zmienimy je zgodnie z zasadami wzorca Wstrzykiwanie Zależności.

Listing 5.1. Klasa `UserService` tworząca nową zależność w konstruktorze

```
package com.devchronicale.di;

class UserService {

    private UserDataRepository udr;

    UserService() {
        this.udr = new UserDataRepositoryImpl();
    }

    public void persistUser(User user) {
        udr.save(user) ;
    }
}
```

Listing 5.2. Interfejs `UserDataRepository`

```
package com.devchronicale.di;

public interface UserDataRepository {
    public void save(User user);
}
```

Listing 5.3. Konkretna implementacja interfejsu `UserDataRepository`

```
package com.devchronicale.di;

public class UserDataRepositoryImpl implements UserDataRepository {
    @Override
    public void save(User user) {
        // zapisywanie danych
    }
}
```

Listing 5.4. Klasa `User`

```
package com.devchronicale.di;

public class User {
    // kod dotyczący użytkownika
}
```

Przedstawiona na listingu 5.1 klasa `UserService` dostarcza usługi logiki biznesowej do zarządzania użytkownikami, np. do zapisywania informacji o użytkownikach w bazie danych. W tym przykładzie obiekt jest tworzony przez konstruktor, co wiąże logikę biznesową (zachowanie klasy) z tworzeniem obiektu.

Teraz zdejmujemy obowiązek tworzenia obiektu z naszej klasy i przeniesiemy go do fabryki.

Na listingu 5.5 tworzona jest implementacja klasy `UserDataRepository`, którą przekazujemy do konstruktora klasy `UserService`. Trzeba było zmienić konstruktor klasy `UserService`, aby przyjmował nowy parametr.

Listing 5.5. Klasa UserServiceFactory tworząca obiekty klasy UserService

```
package com.devchronicale.di;

public class UserServiceFactory {
    public UserService getInstance() {
        return new UserService(new UserDataRepositoryImpl());
    }
}
```

Na listingu 5.6 konstruktor UserService żąda „wstrzyknięcia” egzemplarza klasy UserDataRepository. Klasa UserService nie jest sprzężona z klasą UserDataRepositoryImpl. Teraz to zadaniem fabryki jest tworzenie obiektu i „wstrzykiwanie” implementacji do konstruktora klasy UserService. W ten sposób oddzieliliśmy logikę biznesową od operacji tworzenia obiektów.

Listing 5.6. Zmieniona klasa UserService

```
package com.devchronicale.di;

class UserService {

    private UserDataRepository udr;

    UserService(UserDataRepository udr) {
        this.udr = udr;
    }

    public void persistUser(User user) {
        udr.save(user) ;
    }
}
```

OPOWIADANIE WOJENNE

Gdy powierzono mi zadanie napisania aplikacji na Androida, postanowiłem poszukać systemów szkieletowych ze wstrzykiwaniem zależności dla platform mobilnych. Jako programista z doświadczeniem w sektorze biznesowym myślałem, że to najlepsze rozwiązanie. Interfejs użytkownika Androida wykorzystuje przecież strukturę przypominającą wstrzykiwanie zależności, wiążącą składniki interfejsu zdefiniowane w XML-u z kodem Javy, więc wydawało mi się, że implementacja kompletnego systemu wstrzykiwania zależności jest dobrym pomysłem, który pozwoli osiągnąć oszałamiające rezultaty.

Opracowałem piękną architekturę, w której wszystkie obiekty i zasoby były ze sobą powiązane. Wstrzykiwanie działało pięknie, ale aplikacja... nie. Uruchamiała się o wiele dłużej niż inne podobne aplikacje i szwankowała też w niej nawigacja. Wszyscy zakładaliśmy, że elegancki system złożony z luźno powiązanych składników da się utworzyć tylko przy użyciu wstrzykiwania zależności, więc nawet nie przyszło nam do głowy szukać źródła problemów właśnie w tej technice. Stworzyliśmy piękny i lekki interfejs użytkownika oraz wykorzystaliśmy asynchroniczne zadania działające w tle, aby nie blokować aplikacji niektórymi operacjami oraz zmniejszyć ilość pracy podczas uruchamiania programu. Jednak wszystko na próżno.

W końcu dotarło do nas, że problemem jest wstrzykiwanie zależności. Aplikacja podczas uruchamiania i wykonywania niezbędnych czynności początkowych wyszukiwała wszystkie zasoby do wstrzykiwania i referencje. W serwerze, który uruchamia się nieczęsto, ma wielu użytkowników, jest rzadko restartowany i dysponuje gigantyczną ilością pamięci, może i byłoby to dobre. Ale w przypadku urządzenia przenośnego, które ma jednego użytkownika, jest często restartowane i dysponuje niewielką ilością pamięci, ponieśliśmy sromotną klęskę.

Rozwiązanie polegało na powiązaniu ze sobą zasobów. Choć kod zrobił się „brzydszy”, to aplikacja stała się szybka jak błyskawica, co zakończyło nasze problemy z wydajnością.

Morał z tej historii nie jest taki, że wstrzykiwanie zależności nie jest odpowiednią techniką do stosowania w urządzeniach przenośnych, tylko taki, że jeśli się ją nieprawidłowo zaimplementuje (nieważne, w jakim urządzeniu) w nieodpowiednim kontekście, to można mieć poważne kłopoty.

Implementacja wstrzykiwania zależności w Javie EE

Standardowo w J2EE nie było wstrzykiwania zależności aż do Javy EE 5. Zamiast tego w tamtych czasach dostęp do ziaren i zasobów był realizowany przy użyciu interfejsu wyszukiwania kontekstowego (ang. *Java Naming and Directory Interface* — JNDI). Wadą tej metody było zacieśnianie powiązań między składnikami i wykorzystywanie ciężkiego serwerowego kontenera, przez co testowanie wcale nie było łatwiejsze od pisania właściwego kodu.

Od Javy EE 5 i EJB 3 wstrzykiwanie zależności jest już integralną częścią platformy Enterprise Java. W celu pozbycia się konfiguracji XML wprowadzono kilka adnotacji do wykonywania wstrzyknięć:

- `@Resource` (JSR 250) — służy do wstrzykiwania takich zasobów, jak: dane, JMS (ang. *Java Message Service*), URL, poczta oraz zmienne środowiskowe.
- `@EJB` (JSR 220) — służy do wstrzykiwania ziaren EJB.
- `@WebServiceRef` — służy do wstrzykiwania usług sieciowych.

Od pojawienia się Javy EE 6, CDI oraz EJB 3.1 technika wstrzykiwania zależności stała się znacznie bardziej przydatna, a więc też i bardziej interesująca dla programistów Javy EE.

W EJB 3.1 ziarna nie muszą już mieć interfejsów. Ponadto wprowadzono nowy interfejs sieciowy EJB zapewniający uproszczony i lżejszy kontener EJB. Dodano też nową i ulepszoną adnotację wstrzykiwania `@Inject` (JSR 229 i JSR 330), która stanowiła wspólny interfejs do wstrzykiwania dla różnych systemów z królestwa Javy.

Wstrzykiwanie przy użyciu adnotacji `@Inject` jest bezpieczne pod względem typów, ponieważ odbywa się na podstawie typu referencji do obiektu. Gdybyśmy chcieli dostosować do nowych zasad kod z listingu 5.1, usunęlibyśmy z niego konstruktor i dodalibyśmy adnotację `@Inject` do pola `UserDataRepository`. Wyglądałoby to tak jak na listingu 5.7.

Listing 5.7. Klasa `UserService` z użyciem adnotacji `@Inject`

```

package com.devchronicale.di;

import javax.inject.Inject;

class UserService {

    @Inject
    private UserDataRepository udr;

    public void persistUser(User user) {
        udr.save(user);
    }
}

```

Kontener CDI tworzy jeden egzemplarz klasy `UserRepositoryImpl` jako ziarno zarządzane przez kontener i wstrzykuje go wszędzie tam, gdzie znajdzie adnotację `@Inject` przy polu typu `UserDataRepository`.

Ziarna zarządzane przez kontener można wstrzykiwać do konstruktorów, metod i pól bez względu na modyfikator dostępu, ale pole nie może być finalne, a metoda nie może być abstrakcyjna.

Powstają pewne ważne pytania. Co się stanie, jeśli będzie więcej niż jedna implementacja interfejsu `UserDataRepository`? Jak kontener CDI zidentyfikuje implementację do wstrzyknięcia? Aby rozróżnić konkretne implementacje interfejsu `UserDataRepository`, dla konkretnych klas można oznaczyć klasę samodzielnie zdefiniowanym kwalifikatorem.

Wyobraź sobie, że są dwie implementacje interfejsu `UserDataRepository` — jedna dla kolekcji Mongo DB (dokumentowa baza danych), a druga dla bazy danych MySQL (relacyjna baza danych). Należałoby utworzyć dwa kwalifikatory (jeden dla implementacji Mongo i drugi dla implementacji MySQL) i odpowiednio oznaczać nimi konkretne klasy oraz znajdujące się w nich pola.

Weźmy np. klasę `UserService` z listingu 5.7. Gdybyśmy chcieli użyć implementacji interfejsu `UserDataRepository` dla Mongo, dodalibyśmy adnotację `@Mongo` do pola `udr`:

```

@Inject @Mongo
private UserDataRepository udr;

```

Bardziej szczegółowy opis kwalifikatorów znajduje się poniżej i w rozdziale 6.

Adnotacja `@Named`

Kolejnym wielkim wydarzeniem było wprowadzenie adnotacji `@Named` w miejsce kwalifikatorów łańcuchowych. Wieloznaczności w zależnościach EJB rozstrzygano przy użyciu łańcucha w atrybucie `beanName` adnotacji `@EJB` określającej implementację do wstrzyknięcia — `@EJB(beanName="UserDataRepository")`. Adnotacja `@Named` umożliwia również rozstrzygnięcie niejednoznaczności przy użyciu atrybutu łańcuchowego. Na listingu 5.8 implementacja Mongo interfejsu `UserDataRepository` jest wstrzykiwana do pola `udr`.

Listing 5.8. Wykorzystanie adnotacji @Named do rozstrzygnięcia wieloznaczności

```

package com.devchronicale.di;

import javax.inject.Inject;

import javax.inject.Named;

class UserService {

    @Inject
    @Named("UserDataRepositoryMongo")
    private UserDataRepository udr;

    public void persistUser(User user) {
        udr.save(user) ;
    }
}

```

Jawna adnotacja implementacji Mongo jest wymagana przez odpowiadającą jej adnotację @Named. Na listingu 5.9 implementacja Mongo interfejsu `UserDataRepository` jest oznaczona adnotacją z taką samą nazwą, jaka została użyta do rozstrzygnięcia wieloznaczności na listingu 5.8.

Listing 5.9. Konkretna implementacja wymaga adnotacji @Named

```

package com.devchronicale.di;

import javax.inject.Named;

@Named("UserDataRepositoryMongo")
public class UserDataRepositoryMongo implements UserDataRepository {

    @Override
    public void save(User user) {
        // zapisywanie danych
    }
}

```

Wykorzystywanie łańcuchów do identyfikowania zależności to technika przestarzała. Jest ona niebezpieczna dla typów i w specyfikacji CDI JSR 299 odradza się jej stosowanie. Ale da się też użyć adnotacji @Named w taki sposób, aby nie wykorzystywać identyfikatorów łańcuchowych w miejscu wstrzykiwania.

```

@Inject @Named
private UserDataRepository UserDataRepositoryMongo;

```

Na listingu 5.9 nazwa implementacji do wstrzyknięcia jest określana na podstawie nazwy pola `UserDataRepositoryMongo`. W rzeczywistości adnotacja @Named zostaje zamieniona na @Named("UserDataRepositoryMongo").

Wstrzykiwanie zależności i kontekst (CDI)

Technika CDI (ang. *Context and Dependency Injection* — kontekst i wstrzykiwanie zależności) wniosła do platformy Java EE kompletny mechanizm wstrzykiwania zależności, który wcześniej był ściśle związany z EJB i o wiele bardziej ograniczony. Po pojawieniu się EJB 3 w JBoss wprowadzono Seam

(system szkieletowy do budowy aplikacji sieciowych), który zdobył całkiem sporą popularność dzięki obsłudze bezpośrednich interakcji między JSF (ang. *JavaServer Faces*) i JavaBeans oraz EJB. Sukces systemu Seam doprowadził do powstania specyfikacji JSR 299 — WebBeans. Podobnie jak Hibernate, znany system szkieletowy do zapisywania danych dla Javy, był bodźcem do standaryzacji interfejsu API Java Persistence (JPA), Seam stał się inspiracją rdzenia implementacji CDI.

CDI współpracuje z każdym zwykłym obiektem Javy (POJO) przez tworzenie i wstrzykiwanie obiektów do innych obiektów. Wstrzykiwać można następujące rodzaje obiektów:

- POJO;
- zasoby biznesowe, np. dane i kolejki;
- zdalne referencje EJB;
- ziarna sesyjne;
- obiekty typu `EntityManager`;
- referencje do usług sieciowych;
- pola producenckie i obiekty zwracane przez metody producenckie.

CDI a EJB

Choć może się wydawać, że CDI i EJB to technologie konkurencyjne, w rzeczywistości egzystują one w harmonii. CDI może działać bez kontenera EJB. W istocie CDI może być podstawą aplikacji desktopowej lub dowolnej aplikacji sieciowej, która nie korzysta z kontenera EJB. CDI dostarcza fabrykę i wstrzykiwanie do dowolnego ziarna.

Natomiast ziarna EJB wymagają kontenera EJB. Nawet uproszczona architektura EJB jest bardziej złożona niż obiekty POJO i dlatego ziarna EJB potrzebują kontenera EJB. Kontener ten zapewnia dodatkowe przydatne usługi, takie jak: zabezpieczenia, transakcje i współbieżność.

Mówiąc krótko: kontener CDI jest lżejszym i potężniejszym, ale też mniej funkcjonalnym kontenerem dla obiektów POJO. Jednak oba kontenery są na tyle dobrze ze sobą zintegrowane, że adnotacje CDI mogą służyć jako brama i standardowy interfejs do interakcji z kontenerem EJB. Na przykład adnotacja `@Inject` może być używana zarówno z POJO, jak i EJB oraz może wstrzykiwać dowolną kombinację tych obiektów, wywołując odpowiedni kontener.

Ziarna CDI

Ziarno zarządzane przez kontener to trochę więcej niż tylko obiekt POJO spełniający pewne proste wymagania:

- Musi mieć bezargumentowy konstruktor lub konstruktor deklarujący adnotację `@Inject`.
- Klasa musi być konkretna i znajdować się na najwyższym poziomie hierarchii albo być opatrzona adnotacją `@Decorate`. Nie może być to niestatyczna klasa wewnętrzna.
- Ziarno nie może być zdefiniowane jako EJB.
- Jeśli ziarno jest zdefiniowane jako zarządzane przez inną technologię Javy EE, np. JSF, to również będzie zarządzane przez ten kontener.

Obiekty każdej klasy spełniającej te wymagania są tworzone i zarządzane przez kontener i mogą być wstrzykiwane. Nie trzeba żadnej specjalnej adnotacji, aby oznaczyć klasę jako ziarno zarządzane.

Kontener szuka ziaren w archiwach ziaren. Wyróżnia się dwa typy takich archiwów — jawne i niejawne. Archiwum jawne zawiera deskryptor wdrożenia *bean.xml*, który z reguły pozostaje pusty. CDI skanuje klasy w archiwum w poszukiwaniu klas spełniających opisane powyżej wymagania stawiane ziarnom oraz przejmuje do zarządzania i wstrzykiwania wszystkie te z nich, które nie mają adnotacji `@Vetoed`. Adnotacja ta wyklucza klasę z grupy klas zarządzanych przez kontener.

W niektórych przypadkach nie jest pożądane zezwolenie kontenerowi na zarządzanie wszystkimi znalezionymi ziarnami, które spełniają warunki. Jeśli trzeba ograniczyć grupę klas, które zostaną przyjęte przez kontener CDI do zarządzania, można zdefiniować własność `bean-discovery-mode` w deskrytorze wdrożenia *bean.xml*. Na listingu 5.10 pokazano fragment tego pliku zawierający własność `bean-discovery-mode` z wartością `all`.

Listing 5.10. Tryb wykrywania ziaren ustawia się w pliku *bean.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">
...
</beans>
```

Własności `bean-discovery-mode` można przypisać jedną z trzech wartości: `all`, `none` lub `annotated`. Ustawienie `all` oznacza dla kontenera CDI, że ma zarządzać wszystkimi znalezionymi w archiwum ziarnami. Jest to wartość domyślna. Ustawienie `none` oznacza, że kontener CDI ma w ogóle nie zarządzać ziarnami, a `annotated` sprawia, że archiwum zachowuje się jak archiwum niejawne. W takim przypadku kontener szuka ziaren z adnotacjami oznaczającymi ich zakres.

Niejawne archiwum ziaren nie zawiera deskryptora wdrożenia *bean.xml*. Stanowi to dla kontenera CDI sygnał, że powinien zarządzać tylko ziarnami z określonym zakresem. Więcej informacji na temat zakresów ziaren znajduje się w punkcie „Konteksty i zakres”.

Adnotacja `@Inject`

Właściwości adnotacji `@Inject` zostały już opisane. Zanim wprowadzono technologię CDI do Javy EE, każdy system szkieletowy wstrzykiwania zależności działał na swój sposób. Kiedy w Javie EE zastosowano kontener CDI, który miał pracować obok kontenera EJB, adnotacja `@Inject` stała się jedynym i abstrakcyjnym interfejsem dla prawie wszystkich operacji wstrzykiwania. Dzięki niej można używać każdego kontenera lub systemu szkieletowego wstrzykiwania zależności odpowiedniego w danym przypadku.

Konteksty i zakres

Kontekst jest tym, co odróżnia kontenery EJB od CDI. Cykl życia ziarna CDI jest powiązany z zakresem kontekstowym. Istnieją cztery zakresy CDI:

- `@RequestScoped` — zakres obejmuje żądanie HTTP użytkownika.
- `@SessionScoped` — zakres obejmuje sesję HTTP użytkownika.
- `@ApplicationScoped` — stan jest współdzielony przez wszystkich użytkowników w aplikacji.
- `@ConversationScoped` — zakres jest kontrolowany przez programistę.

Ziarno opatrzone adnotacją określającą zakres przechowuje stan przez cały ten zakres i współdzieli ten stan z każdym klientem działającym w tym samym zakresie. Na przykład ziarno o zakresie żądania przechowuje stan przez cały czas istnienia żądania HTTP, a ziarno o zakresie sesji przechowuje stan przez cały czas istnienia sesji HTTP. Ziarno z zakresem jest automatycznie tworzone w razie potrzeby i niszczone na końcu kontekstu, w którym bierze udział.

Adnotacje zakresowe są często używane do określania zakresu ziaren wykorzystywanych przez język EL (ang. *Expression Language*) w faceletach.

Nazewnictwo i EL

Ziarno z adnotacją `@Named` jest dostępne poprzez język EL. Domyślnie w wyrażeniu należy użyć nazwy klasy, tylko zmienić pierwszą literę na małą. W odniesieniach do metod dostępowych zaczynających się od przedrostka `get` lub `is` należy opuścić tę cząstkę. Na listingu 5.11 pokazano stosowny przykład.

Listing 5.11. Adnotacja `@Named` sprawia, że ziarno staje się widoczne dla EL

```
package com.devchronicale.di;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named // Defining that this is a managed bean
@RequestScoped // Defines the scope
public class User {

    private String fullName;

    public String getFullName() {
        return this.fullName;
    }

    // dalsze metody usunięte dla uproszczenia
}
```

Jest to prosta implementacja nazwanego ziarna zwracającego łańcuch, gdy zostanie wywołana metoda `getFullName()`. W facelecie do metody tej należałoby odnieść się za pomocą nazwy `user.fullName`.

```
<h:form id="user">
    <p><h:outputText value="#{user.fullName}"/></p>
</h:form>
```

Ziarna CDI jako wsparcie dla JSF

Jak w poprzednim przykładzie, ziarna CDI mogą służyć jako ziarna wspierające dla stron JSF. Dostęp do nazwanych ziaren można uzyskać poprzez ich nazwę z pierwszą literą zmienioną na małą. Dostęp do pól i metod dostępowych na stronach JSF można uzyskać zgodnie z konwencjami Javy. Technologia JSF nie jest tematem tej książki, ale na listingu 5.11 pokazano przykład wykorzystania ziaren CDI z JSF.

Kwalifikatory

W tym podrozdziale opisujemy sposoby tworzenia własnych klas kwalifikatorów.

Na listingu 5.12 tworzymy kwalifikator o nazwie `Mongo`, przy użyciu którego można dodawać adnotacje do pól. Jeśli chcesz zastosować tę adnotację do *metody*, *parametru* lub klasy albo interfejsu (*typu*), to możesz ją dodać do adnotacji `@Target`.

Listing 5.12. Tworzenie kwalifikatora o nazwie `@Mongo`

```
package com.devchronicale.di;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({ FIELD })
public @interface Mongo {}
```

Szerzej na temat różnych zastosowań adnotacji piszemy w rozdziale 6.

Alternatywy

W przedstawionych przykładach pokazaliśmy, jak za pomocą kwalifikatorów odróżnić od siebie dwie różne implementacje interfejsu `UserDataRepository`. Takich wyborów implementacji dokonuje się z reguły w czasie pisania programu, wprowadzając odpowiednie zmiany w kodzie. Ale w razie potrzeby można też zrobić to podczas wdrażania programu za pomocą adnotacji `@Alternative` i kilku linijek konfiguracji w deskrytorze wdrożenia *bean.xml*.

Korzystając z dotychczasowych przykładów, oznaczymy nasze dwie implementacje interfejsu `UserDataRepository` adnotacją `@Alternative` oraz zdefiniujemy odpowiednią konfigurację w pliku *bean.xml*. W niej zdecydujemy, którą implementację należy wstrzyknąć.

```
@Alternative
public class UserDataRepositoryMongo implements UserDataRepository { ... }
```

```
@Alternative
public class UserDataRepositoryMySQL implements UserDataRepository { ... }
```

Implementację wykorzystywaną w aplikacji deklarujemy w pliku *bean.xml*:

```
<beans ...>
  <alternatives>
    <class>com.devchronicale.di.UserDataRepositoryMongo</class>
  </alternatives>
</beans>
```

Alternatywy często są wykorzystywane w fazie testowania programu do tworzenia atrap obiektów.

Stereotypy

Stereotypy można sobie wyobrazić jako szablony definiujące cechy typu ziarna. Na przykład ziarno wykorzystywane na poziomie modelu w aplikacji zbudowanej według wzorca Model – Widok – Kontroler (MVC) do działania wymaga pewnych adnotacji. Mogą to być np. te:

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
```

Do zdefiniowania ziarna modelowego niezbędne są tylko adnotacje `@Named` i `@RequestScoped`. Pozostałe są potrzebne do utworzenia adnotacji o nazwie `@Model`.

W razie potrzeby do każdego ziarna można przypisać wszystkie te adnotacje albo można zdefiniować stereotyp o nazwie `@Model` i tylko jego używać w przypadku ziaren. To drugie rozwiązanie znacznie upraszcza kod i ułatwia jego obsługę serwisową.

Aby utworzyć stereotyp, definiuje się nową adnotację i stosuje się wymagane adnotacje, jak pokazano na listingu 5.13.

Listing 5.13. Adnotacja stereotypowa

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Model {}
```

Każde ziarno z adnotacją `@Model` ma zakres żądania (`@RequestScoped`) i jest widoczne dla EL (`@Named`). Na szczęście kontener CDI z tym stereotypem został już zdefiniowany.

Adnotacji stereotypowych najczęściej używa się w kombinacji z adnotacją alternatywną do oznaczania obiektów atrapowych.

Inne wzorce związane z CDI

Technologia CDI dała programistom Javy EE wiele nowych możliwości. Nie jest ona tylko prostym systemem szkieletowym wstrzykiwania zależności, lecz znacznie ułatwia implementowanie różnych wzorców.

W kolejnych rozdziałach znajduje się szczegółowy opis tych wzorców projektowych. Poniżej zamieściliśmy krótkie wprowadzenie na zaostrenie apetytu.

W rozdziale 7., „Wzorec Dekorator”, znajduje się opis wzorca Dekorator. Dekoratory opakowują obiekty docelowe, aby dynamicznie dodać nowe obowiązki. Każdy dekorator można opakować w inny dekorator, co teoretycznie umożliwia utworzenie nieskończonej liczby dekorowanych obiektów docelowych w czasie działania programu. Wzorec Dekorator wykorzystuje adnotacje `@Decorator` i `@Delegate`. Kolejność dekorowania określa się w pliku *bean.xml*.

W rozdziale 6. opisany jest wzorec Fabryka. Fabryki ograniczają do minimum wykorzystanie słowa kluczowego `new` i mogą zawierać proces inicjacji oraz różne konkretne implementacje. Wzorec Fabryka wykorzystuje adnotację `@Produces` do oznaczania metod producenckich. Obiekt docelowy może wstrzyknąć lub obserwować wyprodukowane obiekty.

W rozdziale 11. znajduje się opis wzorca Obserwator i zdarzeń. Wzorzec ten zmienia kierunek przepływu wiadomości, czyli kolejność wywołującego i wywoływanego. Przy użyciu tego wzorca nie trzeba agresywnie sprawdzać zasobu, tylko można subskrybować zmiany zachodzące w zasobie. Wzorzec Obserwator w Javie EE wykorzystuje adnotację `@Observes` i zdarzenia. Obserwatory docelowe mogą obserwować wszystkie zdarzenia.

Tematem rozdziału 8. są aspekty i interceptory. Przy ich użyciu można zmienić sposób wykonywania kodu w czasie działania programu. Każdy aspekt lub interceptor może zatrzymać wykonywanie i włączyć się w wybranym miejscu. Umożliwia to dynamiczne wprowadzanie zmian nawet w dużych programach.

Podsumowanie

W tym rozdziale przedstawiliśmy techniki wstrzykiwania zależności w Javie EE. Koncepcja ta umożliwia rozluźnienie więzi między składnikami systemu łatwiej, niż można by się było spodziewać. Pokazaliśmy, jak wstrzykiwanie zależności umożliwia pozbycie się słowa kluczowego `new`, a więc uniknięcie ręcznego tworzenia obiektów.

Sporo miejsca poświęciliśmy też technologii CDI, która dzięki wykorzystaniu nowego kontenera stwarza całkiem nowe możliwości. Przy jej użyciu wstrzykiwanie zależności można stosować do wszystkich obiektów, a implementacja innych wzorców opisanych w tej książce jest znacznie łatwiejsza.

ĆWICZENIA

1. Zaprojektuj klasę usługową zwracającą do klienta dowolny łańcuch.
2. Zaimplementuj czytnik plików i wstrzyknij go do wcześniej utworzonej usługi.
3. Zaimplementuj obiekt odczytujący jako łańcuch treść HTML z określonego na stałe adresu URL.
4. Zastanów się, co musisz zmienić w klasie usługowej, aby móc wstrzykiwać obu dostawców danych przy użyciu tej samej referencji.
5. Czy da się dynamicznie wstrzyknąć wybraną implementację w zależności od pewnych warunków? Na przykład czy można sprawić, aby czytnik plików był wstrzykiwany podczas pracy nad programem, a czytnik HTTP w produkcji?

Skorowidz

A

abstrakcyjny dekorator, 107
adnotacja, 37
 @DependsOn, 65
 @GeneratedValue, 176
 @Inject, 76, 79, 94
 @Interceptor, 124
 @Interceptors, 124
 @Lock, 67
 @MessageEvent, 166
 @Named, 76–80, 96, 207
 @Observes, 166
 @Path, 193
 @Produce, 101
 @RequestScope, 207
 @Schedule, 150, 156
 @Secure, 129
 @Timeout, 151, 156
 @Transient, 176
 stereotypowa, 82
agencja informacyjna, 162
alternatywy, 81
antywzorce, 35, 220
AOP, aspect-oriented programming, 117
API JPA, 174
architektura
 Lazani, 221
 mikrousługowa, 215, 219
 monolityczna, 215, 216, 220
 wielowarstwowa, 38

aspekty, 122
asynchroniczne
 serwlety, 140
 ziarna, 138
asynchroniczność, 133

B

baza danych typu NoSQL, 171
bezpieczeństwo, 32
bezstanowość, 184

C

CDI, Context and Dependency Injection, 34, 43, 71, 77
CMP, Container-Managed Persistence, 174
cykl życia interceptora, 125
czasomierz, 155
 automatyczny, 150
 programowy, 151
czas dostępu współbieżnego, 68

D

DAO, data access object, 172
definicja filtru serwletu, 121
dekompozycja na usługi, 218
dekorator
 BlackFridayDiscountDecorator, 111
 dodający dodatki, 107
 PriceDiscountDecorator, 110
dekoratory bez konfiguracji XML, 113
dekorowanie, 109
delegacja, 143

- DI, Dependency Injection, 71
 - diagram klas
 - dziedziczenie, 28
 - wzorca
 - Dekorator, 105
 - Dostępu do Danych, 172
 - Fabryka Abstrakcyjna, 90
 - Fasady, 51
 - Obserwator, 162
 - Singleton, 59
 - diagram wzorca Model – Widok – Kontroler, 201
 - długotrwałe sondowanie, 212
 - dokument JSR, 41
 - dostęp współbieżny do singletonu, 68
 - DTO, data transfer object, 172, 200
- E**
- EJB, 78
 - EJB, Enterprise JavaBeans, 41
 - EL, 80
 - encja, entity, 174
- F**
- fabryka, 85
 - Abstrakcyjna, 85, 89
 - DAO, 177
 - faceletry, 206
 - filtr serwletu, 120
 - format JSON, 191
- G**
- gniazda sieciowe, 212
- H**
- HATEOAS, 182, 194
- I**
- idempotencja, 187
 - implementacja
 - @Secure, 129
 - AOP, 120
 - asynchroniczności, 136
 - automatycznego czasomierza, 151
 - bezstanowej fasady, 54
 - CoffeMachine, 88
 - czasomierza programowego, 152
 - czasomierza w Javie EE, 150
 - DAO, 179
 - fabryki, 99
 - Fabryki Abstrakcyjnej, 90
 - filtru serwletu, 121
 - interceptora, 122
 - interceptorów klasowych, 123
 - interfejsu AbstractDrinksMachineFactory, 91
 - interfejsu DAO, 177, 179
 - interfejsu UserRepositoryity, 73
 - LongMessage, 98
 - Metody Fabrycznej, 88, 91
 - publikacja-subskrypcja, 215
 - punkt do punktu, 214
 - rady docelowej, 123
 - REST w Javie EE, 191
 - REST-owego interfejsu API, 191
 - ShortMessage, 98
 - SoftDrinksMachine, 88
 - wstrzykiwania zależności, 72
 - wstrzykiwania zależności w Javie EE, 75
 - wzorca
 - DAO w Javie EE, 174
 - Dekorator, 106
 - Dekorator w Javie EE, 109
 - Fabryka w Javie EE, 91
 - Fasada, 52
 - Fasada w Javie EE, 53
 - MVC, 202, 206, 207
 - MVC w Javie EE, 206
 - Obserwator, 162
 - Obserwator w Javie EE, 164
 - Singleton, 59, 62
 - Singleton w Javie EE, 63
 - wzorców projektowych, 47
 - informacje
 - o kontekście, 124
 - o wzorcu DAO, 173
 - interceptor, 44, 117, 122, 125
 - interceptor dziennika, 45
 - interceptory
 - CDI, 128
 - na poziomie domyślnym, 125
 - interfejs
 - AbstractDrinksMachineFactory, 91
 - API, 197
 - DAO, 176
 - dla fabryki abstrakcyjnej, 90

kwalifikacyjny adnotacji, 166
 MessageType, 98
 Observable, 164
 Order, 106
 Product, 109
 Publisher, 163
 Serializable, 176
 Timer, 150
 TimerService, 151, 152
 UserDataRepository, 73

J

J2EE, 31
 Java SE, Java Standard Edition, 31
 jednostka utrwalania, 178
 JPA, Java Persistence API, 174
 JPE, 31
 JSF, 80
 JSR, Java Specification Request, 41

K

klasa

- akcji, 205
- DrinksMachine, 88
- encyjna, 175
- fabryczna, 204
- ListUserAction, 206
- Observable, 164
- User, 73
- UserService, 73–76, 203
- UserServiceFactory, 74

 klient-serwer, 183
 kolejka, 214
 kolejność

- interceptorów, 126
- uruchamiania, 65

 komponent sieciowy, 40
 konfiguracja serwletu facesservlet, 206
 konteksty, 44, 77, 79
 kontener CDI, 97, 101
 kontroler, 200
 konwencja, 43
 kwalifikator, 81, 95

- LongMessage, 94
- niestandardowy, 112
- ShortMessage, 94

L

LDAP, Lightweight Directory Access Protocol, 171
 literał adnotacyjny, 99

M

mapowanie obiektowo-relacyjne, ORM, 174
 metoda

- DELETE, 188
- GET, 187
- executeTask, 150
- Fabryczna, 85, 86
- GET, 194
- getInfo, 150
- getNextTimeout, 153
- getThirdChild, 168
- isCalendarTimer, 153
- POST, 188, 194
- PUT, 188
- serviceTrace, 165
- setRollbackOnly, 169
- setTimer, 151
- startAsync(), 141
- startService, 165

 metody HTTP, 187
 mikrousługi, 219
 model

- dojrzałości Richardsona, 185
- komponentowy, 42
- programowania J2EE, 227

 modyfikowanie czasomierza, 155
 MVC, model-view-controller, 199
 MVP, model, view, presenter, 202

N

nazewnictwo, 80
 nazewnictwo zasobów, 186
 niejednoznaczność, 95
 niestandardowa adnotacja wiadomości, 99
 niestandardowy typ adnotacji, 95

O

obiekt

- dostępu do danych, DAO, 172
- DTO, 173
- JSON, 192
- transferu danych, DTO, 172

obiekt
 typu ManagedThreadFactory, 142
 typu ScheduleExpression, 153
 obserwator, observer, 160
 RadioChannel, 163
 zdarzeń transakcji, 167
 obserwowalne ziarno usługowe, 164
 odbiornik, listener, 161
 odmierzanie czasu, 147
 określanie kolejności uruchamiania, 65, 66
 operacje CRUD, 172
 ORM, Object-Relational Mapping, 174

P

plik persistence.xml, 178
 pliki EJB-JAR, 126
 pobieranie informacji, 124
 podłączanie faz cyklu życia, 125
 podmiot, 160
 POJO, 37
 profil sieciowy Javy EE, 42
 programowanie
 aspektowe, AOP, 117
 asynchroniczne, 133, 138
 przekazywanie wiadomości, 214
 przenośność, 32
 publikacja-subskrypcja, 215
 punkt
 do punktu, 214
 końcowy, 212
 końcowy z adnotacjami, 213

R

repozytorium LDAP, 171
 REST, 181, 182
 bezstanowość, 184
 jednolity interfejs, 184
 klient-serwer, 183
 kod na żądanie, 184
 nazewnictwo zasobów, 186
 posts, 189
 system warstwowy, 184
 topics, 189
 users, 188
 zapisywanie danych w buforze, 184
 REST-owy interfejs API, 185, 188
 rozróżnianie ziaren, 96

rozstrzyganie
 niejednoznaczności, 95, 96
 wieloznaczności, 77
 rozwiązania dla biznesu, 31

S

schemat implementacji MVC, 202
 serwer Java EE, 41
 serwlet, 140
 serwlet FacesServlet, 206
 skalowalność, 217
 SOA, Service Oriented Architecture, 33, 215, 228
 stereotypy, 82
 synchronizacja singletonu, 60
 system szkieletowy, 222
 sześcian AKF, 217

T

technika CDI, 77
 technologia J2EE, 32
 tematy, topics, 189, 214
 transakcje, 32, 156
 tworzenie
 obiektu singletonowego, 60, 61
 użytkownika, 189
 typ wyliczeniowy, 62
 typy wzorca MVC, 201

U

Uberklasa, 220
 users, 188
 usługa odmierzania czasu, 147
 usługi sieciowe, 181
 użytkownicy, users, 188
 używanie
 interceptorów, 130
 programowania asynchronicznego, 143
 REST, 196
 wzorca DAO, 180
 wzorca Dekorator, 114
 wzorca Fasada, 55
 wzorca MVC, 208
 wzorca Obserwator, 169
 wzorca Singleton, 69
 wzorców, 227
 wzorców fabrycznych, 100

W

- warstwa
 - EIS, 40
 - kliencka, 39
 - logiki biznesowej, 40
 - pośrednia, 214
- warunki REST, 183
- wiązanie interceptora, 129
- widok, 200
- widok renderujący dane, 208
- wpisy, posts, 189
- współbieżność, 67
- wstrzykiwanie
 - łańcucha, 92
 - zależności, 44, 77
 - zależności, DI, 71
 - ziaren, 94, 96
- wyjątek IndexOutOfBounds, 168
- wykorzystywanie singletonów, 64
- wykrywanie ziaren, 79
- wyłączanie interceptorów, 128
- wyrażenia
 - czasowe, 153
 - kalendarzowe, 154
- wywoływanie singletonu, 64
- wzorce
 - behawioralne, 30
 - biznesowe, 33
 - konstrukcyjne, 30
 - strukturalne, 30
 - związane z CDI, 82

wzorzec

- Asynchroniczność, 134
- Dekorator, 103
- Dostęp do Danych, 171, 172
- Fabryka, 85
- Fasada, 49
- Model – Widok – Kontroler, 199
- MVC, 200
- Obiekt Transferu Danych, 173
- Obserwator, 159
- projektowy, 28
- Singleton, 57

Z

- zakres, 79
- zalety architektury mikrousługowej, 218
- zarządzanie współbieżnością, 67
- zasada Hollywood, 160
- zasady Javy EE, 42
- zdarzenie transakcji, 167
- ziarna
 - asynchroniczne, 138
 - bezstanowe, 53
 - CDI, 78, 80
 - obserwatora, 165
 - singletonowe, 63
 - stanowe, 55
 - usługowe, 164
 - wiadomości, 93
 - wspierające, 206
- ziarno, 38
 - encyjne, entity bean, 174
 - MessageA, 92
 - MessageB, 93

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

 **Helion SA**

Lektura obowiązkowa dla programistów języka Java!

Platforma Java EE to najbardziej zaawansowane rozwiązanie do budowania złożonych systemów informatycznych, jakie oferuje firma Oracle. Jest wykorzystywana wszędzie tam, gdzie wymaga się najwyższej wydajności, bezpieczeństwa oraz niezawodności. Java EE dostarcza kompletny zbiór narzędzi, który pozwoli Ci zbudować aplikację o dowolnym stopniu skomplikowania.

Jeżeli chcesz poznać najlepsze wzorce tworzenia oprogramowania z wykorzystaniem Javy EE, to trafieś na doskonałą książkę. Sięgnij po nią i poznaj podstawy pracy z tą platformą oraz klasyczne wzorce projektowe. Gdy już opanujesz fundamenty, przejdiesz do nauki zaawansowanych mechanizmów. Wstrzykiwanie zależności, porównanie CDI i EJB, budowanie serwisów REST-owych, fabryki czy fasady to tylko niektóre z poruszanych tu tematów. Ponadto przekonasz się, jak tworzyć asynchroniczne ziarna lub serwlety oraz przygotujesz własną usługę odmierzającą czas. Na sam koniec przeanalizujesz kilka antywzorców. Pamiętaj — ich nie warto stosować! Książka ta jest doskonałą lekturą dla wszystkich programistów języka Java chcących pogłębić swoją wiedzę na temat Javy EE oraz najlepszych praktyk.

Dzięki tej książce:

- zaznajomisz się z klasycznymi wzorcami projektowymi
- poznasz platformę Java EE
- stworzysz asynchroniczne serwlety i ziarna
- zbudujesz usługę odmierzającą czas
- poznasz antywzorce, niewarte stosowania
- nauczysz się tworzyć przejrzyste, łatwy w utrzymaniu kod

Murat Yener — pasjonat systemu Android i pracownik firmy Intel, odpowiedzialny za tworzenie aplikacji mobilnych. Ma bogate doświadczenie w pracy w języku Java, z platformą Java EE oraz OSGi. Bierze czynny udział w rozwijaniu zintegrowanego środowiska programistycznego Eclipse. Jest liderem grupy GDG ze Stambułu.

Alex Theedom — starszy programista w Indigo Code Collective. Ma ogromne doświadczenie w tworzeniu rozwiązań opartych na Javie EE oraz frameworku Spring. W swojej karierze pracował przy systemach informatycznych w zakresie mikrousług, tworzył oprogramowanie dla bankomatów oraz platformy e-learningowe.

Helion

35922 numer katalogowy

księgarnia internetowa

<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/nowości>

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

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ



KOD KORZYSCI

ISBN 978-83-283-1315-6



cena: 49,00 zł

Wrox
An Imprint of
WILEY