

Popraw wydajność
aplikacji bazodanowych w Javie

Hibernate w akcji

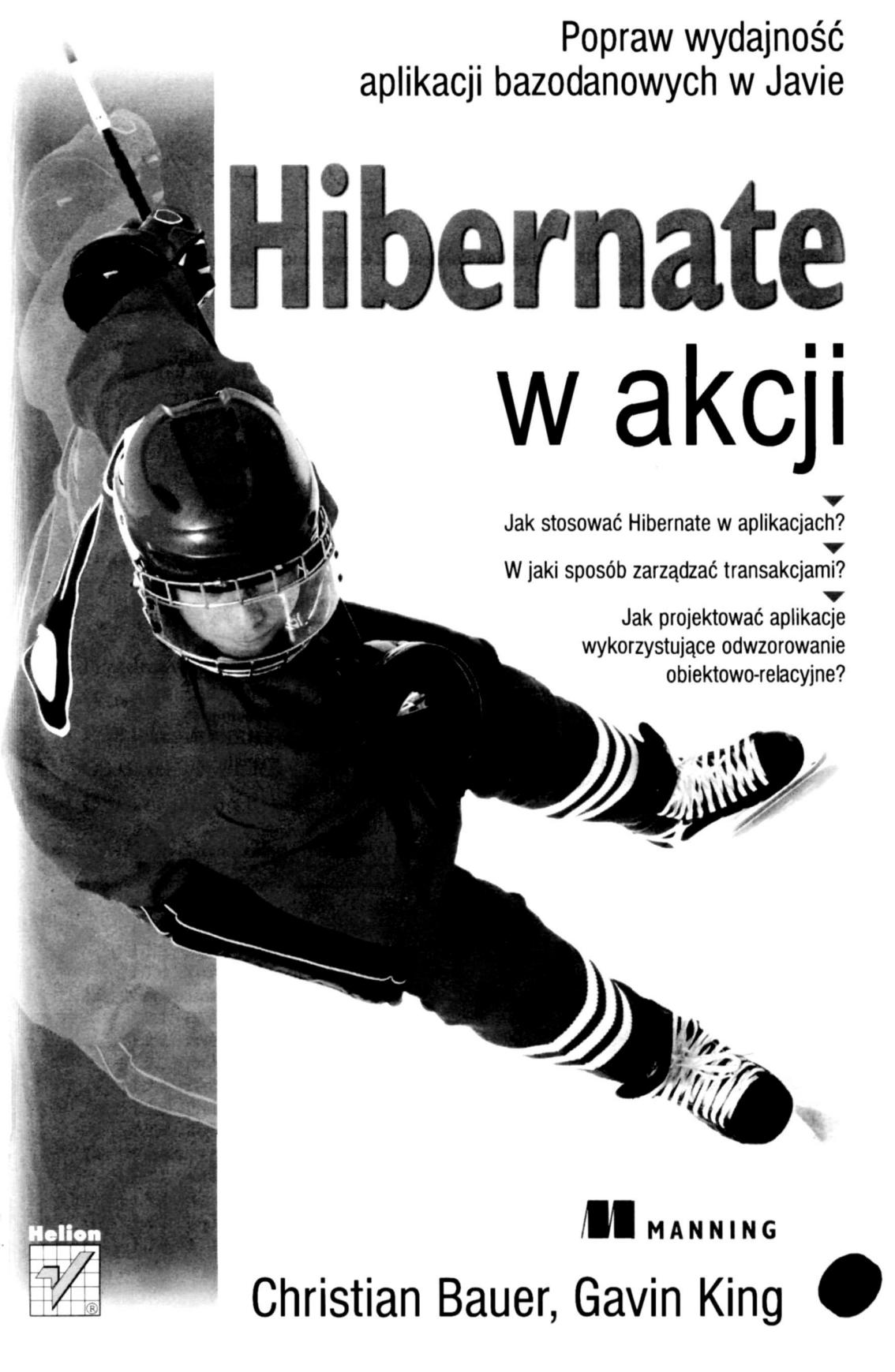
▼ Jak stosować Hibernate w aplikacjach?

▼ W jaki sposób zarządzać transakcjami?

▼ Jak projektować aplikacje
wykorzystujące odwzorowanie
obiektowo-relacyjne?



Christian Bauer, Gavin King



Popraw wydajność
aplikacji bazodanowych w Javie

Hibernate w akcji

▼ Jak stosować Hibernate w aplikacjach?

▼ W jaki sposób zarządzać transakcjami?

▼ Jak projektować aplikacje
wykorzystujące odwzorowanie
obiektowo-relacyjne?



M MANNING

Christian Bauer, Gavin King

Tytuł oryginału: **Hibernate in Action**

Tłumaczenie: Rafał Jońca

ISBN: 978-83-246-0527-9

Original edition copyright 2005 by Manning Publications Co as set forth in copyright notice of Proprietor's edition. All rights reserved.

Polish edition copyright 2007 by Helion S.A. 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 kopowanie 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 właścicielami.

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. 032 231 22 19, 032 230 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?hibakc>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

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

Printed in Poland.

Spis treści

Przedmowa	9
Wstęp	11
Podziękowania	13
O książce i autorach	15
O Hibernate 3 i EJB 3	19
1. Trwałość dzięki odwzorowaniu obiektowo-relacyjnemu	21
1.1. Czym jest trwałość?	23
1.1.1. Relacyjne bazy danych	23
1.1.2. Język SQL	24
1.1.3. Korzystanie z SQL w Javie	24
1.1.4. Trwałość w aplikacjach obiektowych	25
1.2. Niedopasowanie paradygmatów	27
1.2.1. Problem szczegółowości	28
1.2.2. Problem podtypów	29
1.2.3. Problem identyczności	30
1.2.4. Problemy dotyczące asocjacji	32
1.2.5. Problem nawigacji po grafie obiektów	33
1.2.6. Koszt niedopasowania	34
1.3. Warstwy trwałości i alternatywy	35
1.3.1. Architektura warstwowa	35
1.3.2. Ręczne tworzenie warstwy trwałości za pomocą SQL i JDBC	37
1.3.3. Wykorzystanie serializacji	38
1.3.4. A może ziarenka encyjne EJB?	39

1.3.5. Obiektowe systemy bazodanowe	40
1.3.6. Inne rozwiązania	41
1.4. Odwzorowanie obiektowo-relacyjne	41
1.4.1. Czym jest ORM?	42
1.4.2. Ogólne problemy ORM	44
1.4.3. Dlaczego ORM?	45
1.5. Podsumowanie	47
2. Wprowadzenie i integracja Hibernate	49
2.1. „Witaj świecie” w stylu Hibernate	50
2.2. Podstawy architektury Hibernate	55
2.2.1. Interfejsy podstawowe	56
2.2.2. Interfejsy wywołań zwrotnych	58
2.2.3. Typy	58
2.2.4. Interfejsy rozszerzeń	59
2.3. Konfiguracja podstawowa	59
2.3.1. Tworzenie obiektu SessionFactory	60
2.3.2. Konfiguracja w środowisku niezarządzanym	62
2.3.3. Konfiguracja w środowisku zarządzanym	66
2.4. Zaawansowane ustawienia konfiguracyjne	68
2.4.1. Konfiguracja bazująca na pliku XML	69
2.4.2. Obiekt SessionFactory dowiązany do JNDI	70
2.4.3. Dzienniki	71
2.4.4. Java Management Extensions	73
2.5. Podsumowanie	75
3. Odwzorowanie klas trwałości danych	77
3.1. Aplikacja CaveatEmptor	78
3.1.1. Analiza dziedziny biznesowej	79
3.1.2. Model dziedzinowy CaveatEmptor	79
3.2. Implementacja modelu dziedzinowego	82
3.2.1. Kwestia przesyłania zadań	82
3.2.2. Trwałość automatyczna i przezroczysta	83
3.2.3. Tworzenie klas POJO	84
3.2.4. Implementacja asocjacji POJO	86
3.2.5. Dodanie logiki do metod dostępowych	90
3.3. Definicja metadanych odwzorowujących	92
3.3.1. Metadane w pliku XML	92
3.3.2. Podstawowe odwzorowania właściwości i klas	95
3.3.3. Programowanie zorientowane na atrybuty	101
3.3.4. Modyfikacja metadanych w trakcie działania aplikacji	102
3.4. Identyczność obiektów	104
3.4.1. Identyczność a równość	104
3.4.2. Tożsamość bazodanowa w Hibernate	104
3.4.3. Wybór kluczów głównych	106
3.5. Szczegółowe modele obiektów	108
3.5.1. Encje i typy wartości	109
3.5.2. Stosowanie komponentów	109
3.6. Odwzorowanie dziedziczenia klas	113
3.6.1. Tabela na klasę konkretną	113
3.6.2. Tabela na każdą hierarchię klas	115

3.6.3. Tabela na każdą podklasę	117
3.6.4. Wybór strategii	120
3.7. Asocjacje	121
3.7.1. Asocjacje zarządzane?	121
3.7.2. Krotność	122
3.7.3. Najprostsza możliwa asocjacja	122
3.7.4. Tworzenie asocjacji dwukierunkowej	123
3.7.5. Związek rodzic-potomek	126
3.8. Podsumowanie	127
4. Stosowanie obiektów trwałych	129
4.1. Cykl życia obiektu trwałego	130
4.1.1. Obiekty ulotne	131
4.1.2. Obiekty trwałe	132
4.1.3. Obiekt odłączony	133
4.1.4. Zasięg identyczności obiektów	134
4.1.5. Poza zasięgiem identyczności	135
4.1.6. Implementacja equals() i hashCode()	136
4.2. Zarządcia trwałosci	140
4.2.1. Czynienie obiektu trwałym	140
4.2.2. Aktualizacja stanu trwałego obiektu odłączonego	141
4.2.3. Pobranie obiektu trwałego	142
4.2.4. Aktualizacja obiektu trwałego	143
4.2.5. Zmiana obiektu trwałego na ulotny	143
4.2.6. Zmiana obiektu odłączonego na ulotny	144
4.3. Trwałość przechodnia w Hibernate	144
4.3.1. Przechodniość przez osiągalność	145
4.3.2. Trwałość kaskadowa w Hibernate	146
4.3.3. Zarządzanie kategoriami przedmiotów	147
4.3.4. Rozróżnienie obiektów ulotnych i odłączonych	151
4.4. Pobieranie obiektów	152
4.4.1. Pobieranie obiektów na podstawie identyfikatora	153
4.4.2. Wprowadzenie do HQL	154
4.4.3. Zapytania przez określenie kryteriów	155
4.4.4. Zapytanie przez przykład	155
4.4.5. Strategie sprowadzania danych	156
4.4.6. Wybór strategii sprowadzania w odwzorowaniach	158
4.4.7. Optymalizacja pobierania obiektów	163
4.5. Podsumowanie	164
5. Transakcje, współbieżność i buforowanie	167
5.1. Transakcje bazodanowe	169
5.1.1. Transakcje JDBC i JTA	170
5.1.2. Interfejs Transaction	171
5.1.3. Opróżnianie sesji	173
5.1.4. Poziomy izolacji	174
5.1.5. Wybór poziomu izolacji	176
5.1.6. Ustawianie poziomu izolacji	177
5.1.7. Blokada pesymistyczna	177
5.2. Transakcje aplikacyjne	180
5.2.1. Wersjonowanie zarządzane	181

5.2.2. Szczegółowość sesji	184
5.2.3. Inne sposoby implementacji blokady optymistycznej	185
5.3. Buforowanie — teoria i praktyka	186
5.3.1. Strategie i zasięgi buforowania	187
5.3.2. Architektura buforów Hibernate	190
5.3.3. Buforowanie w praktyce	195
5.4. Podsumowanie	204
6. Zaawansowane zagadnienia odwzorowań	205
6.1. System typów Hibernate	206
6.1.1. Wbudowane typy odwzorowań	207
6.1.2. Zastosowania typów odwzorowań	210
6.2. Odwzorowywanie kolekcji typów wartości	220
6.2.1. Zbiory, pojemniki, listy i odwzorowania	220
6.3. Odwzorowanie asocjacji encyjnych	228
6.3.1. Asocjacja jeden-do-jednego	229
6.3.2. Asocjacje wiele-do-wielu	232
6.4. Odwzorowanie asocjacji polimorficznych	241
6.4.1. Polimorficzna asocjacja wiele-do-jednego	241
6.4.2. Kolekcje polimorficzne	243
6.4.3. Asocjacje polimorficzne i jedna tabela na klasę konkretną	244
6.5. Podsumowanie	246
7. Wydajne pobieranie obiektów	247
7.1. Wykonywanie zapytań	249
7.1.1. Interfejsy zapytań	249
7.1.2. Dowiązywanie parametrów	251
7.1.3. Zapytania nazwane	254
7.2. Proste zapytania dotyczące obiektów	255
7.2.1. Najprostsze zapytanie	256
7.2.2. Zastosowanie aliasów	256
7.2.3. Zapytania polimorficzne	257
7.2.4. Ograniczenia	257
7.2.5. Operatory porównań	258
7.2.6. Dopasowywanie tekstów	260
7.2.7. Operatory logiczne	261
7.2.8. Kolejność wyników zapytań	262
7.3. Złączanie asocjacji	262
7.3.1. Złączenia w Hibernate	264
7.3.2. Pobieranie asocjacji	265
7.3.3. Aliasy i złączenia	266
7.3.4. Złączenia niejawne	270
7.3.5. Złączenia w stylu theta	271
7.3.6. Porównywanie identyfikatorów	272
7.4. Tworzenie zapytań raportujących	273
7.4.1. Projekcja	274
7.4.2. Agregacja	276
7.4.3. Grupowanie	277
7.4.4. Ograniczanie grup klauzulą having	278
7.4.5. Poprawa wydajności zapytań raportujących	279

7.5.	Techniki tworzenia zaawansowanych zapytań	279
7.5.1.	Zapytania dynamiczne	280
7.5.2.	Filtры коллекций	282
7.5.3.	Подзапросы	284
7.5.4.	Zapytania SQL	286
7.6.	Optymalizacja pobierania obiektów	288
7.6.1.	Rozwiązywanie problemu n+1 pobrań danych	288
7.6.2.	Zapytania iterate()	291
7.6.3.	Buforowanie zapytań	292
7.7.	Podsumowanie	294
8.	Tworzenie aplikacji stosujących Hibernate	295
8.1.	Projektowanie aplikacji warstwowych	296
8.1.1.	Użycie Hibernate w systemie serwletowym	297
8.1.2.	Stosowanie Hibernate w kontenerze EJB	311
8.2.	Implementacja transakcji aplikacyjnych	319
8.2.2.	Trudny sposób	321
8.2.3.	Odlączone obiekty trwałe	322
8.2.4.	Długa sesja	323
8.2.5.	Wybór odpowiedniej implementacji transakcji aplikacyjnych	327
8.3.	Obsługa specjalnych rodzajów danych	328
8.3.1.	Starsze schematy baz danych i klucze złożone	328
8.3.2.	Dziennik audytowy	337
8.4.	Podsumowanie	343
9.	Narzędzia Hibernate	345
9.1.	Procesy tworzenia aplikacji	346
9.1.1.	Podejście z góry na dół	347
9.1.2.	Podejście z dołu do góry	347
9.1.3.	Podejście od środka	347
9.1.4.	Spotkanie w środku	347
9.1.5.	Ścieżka przejścia	348
9.2.	Automatyczne generowanie schematu bazy danych	348
9.2.1.	Przygotowanie metadanych odwzorowania	349
9.2.2.	Utworzenie schematu	351
9.2.3.	Aktualizacja schematu bazy danych	353
9.3.	Generowanie kodu klas POJO	354
9.3.1.	Wprowadzenie metaatributów	355
9.3.2.	Metody odnajdujące	357
9.3.3.	Konfiguracja hbm2java	358
9.3.4.	Uruchamianie narzędzia hbm2java	359
9.4.	Istniejące schematy i Middlegen	360
9.4.1.	Uruchomienie Middlegen	360
9.4.2.	Ograniczanie tabel i związków	362
9.4.3.	Dostosowanie generowania metadanych	363
9.4.4.	Generowanie metadanych hbm2java i XDoclet	365
9.5.	XDoclet	366
9.5.1.	Ustawianie atrybutów typu wartości	367
9.5.2.	Odwzorowanie asocjacji encyjnych	368
9.5.3.	Uruchomienie XDoclet	369
9.6.	Podsumowanie	371

A Podstawy języka SQL	373
B Strategie implementacji systemów ORM	377
B.1. Właściwości czy pola?	378
B.2. Strategie sprawdzania zabrudzenia	379
B.2.1. Dziedziczenie po wygenerowanym kodzie	379
B.2.2. Przetwarzanie kodu źródłowego	379
B.2.3. Przetwarzanie kodu bajtowego	380
B.2.4. Introspekcja	380
B.2.5. Generowanie kodu bajtowego „w locie”	381
B.2.6. Obiekty „uogólnione”	382
C Powrót do świata rzeczywistego	383
C.1. Dziwna kopia	384
C.2. Im więcej, tym lepiej	385
C.3. Nie potrzebujemy kluczy głównych	385
C.4. Czas nie jest liniowy	386
C.5. Dynamicznie niebezpieczne	386
C.6. Synchronizować czy nie synchronizować?	387
C.7. Naprawdę gruby klient	388
C.8. Wznawianie Hibernate	388
Bibliografia	391
Skorowidz	393

Przedmowa

Relacyjne bazy danych bezsprzecznie stanowią rdzeń najnowszej generacji systemów biznesowych.

Choć nowoczesne języki programowania, na przykład Java, zapewniają intuicyjny i obiektowy widok elementów biznesowych z poziomu aplikacji, dane, z których korzystają procesy biznesowe, są silnie relacyjne. Co więcej, główną siłą modelu relacyjnego — w porównaniu ze wcześniejszym modelem nawigacyjnym i nowszym modelem obiektowym — jest to, że ze swojej natury ciężko poddaje się modyfikacjom programowym i nie ułatwia spojrzenia na dane z punktu widzenia aplikacji z nich korzystających.

Podejmowano wiele prób połączenia technologii relacyjnej i bazodanowej, zastąpienia jednej drugą, ale należy pogodzić się z istnieniem znacznej przepaści między nimi w obecnych zastosowaniach biznesowych. Pojawiło się prawdziwe wyzwanie — zapewnienie połączenia relacyjnych danych i obiektów Javy — któremu stara się sprostać Hibernate, oferując odwzorowanie obiektowo-relacyjne (ORM — ang. *Object-Relational Mapping*). Hibernate dostarcza rozwiązaniami bardziej pragmatyczne, bezpośrednie i realistyczne.

Christian Bauer i Gavin King przedstawiają w niniejszej książce wszystkie aspekty wydajnego stosowania technologii ORM. Warto zdawać sobie sprawę, iż nawet najprostsze środowiska biznesowe wymagają odpowiedniej wiedzy i konfiguracji, by zapewnić odpowiedni pomost między obiektami i danymi relacyjnymi. Wymaga to od programisty wiedzy na temat aplikacji i jej wymagań związanych z danymi, znajomości języka SQL, sposobu relacyjnego zapisu danych i potencjalnych optymalizacji oferowanych przez technologię relacyjną.

Hibernate nie tylko zapewnia gotowe rozwiązanie, które spełnia wymienione wymagania, ale jest również elastyczne i konfigurowalne. Twórcy Hibernate dołożyli starań, by ich dzieło było modułowe, rozszerzalne i umożliwiało dostosowywanie do potrzeb konkretnego projektu. Już po kilku latach od pierwszego wydania Hibernate stał się jedną z wiodących technologii ORM kierowanych do twórców aplikacji biznesowych. Zasługuje na to wyróżnienie.

Niniejsza książka stanowi dokładne omówienie wszystkich aspektów Hibernate. Omawia, jak korzystać z odwzorowania typów, jak modelować powiązania i dziedziczenie, jak wydajnie pobierać obiekty, stosując język zapytań Hibernate, jak skonfigurować Hibernate do działania w środowiskach nadzorowanych i nie-nadzorowanych, jak stosować narzędzia dodatkowe. Poza aspektami ściśle związanymi z Hibernate książka prezentuje wszystkie najważniejsze elementy technologii ORM i wybory projektowe. W ten sposób Czytelnik dowiaduje się, jak wydajnie korzystać z ORM, pisząc aplikacje biznesowe.

Czytelnik dostaje do rąk doskonały przewodnik po Hibernate i odwzorowaniu obiektowo-relacyjnym ze szczególnym uwzględnieniem rozwiązań biznesowych.

Linda DeMichiel
główny architekt, Enterprise JavaBeans
Sun Microsystems

Wstęp

To, że można popychać dwie gałązki leżące na ziemi nosem, nie oznacza od razu, że jest to najlepszy sposób zbierania drewna na ognisko.

— Anthony Berglas

Dziś wielu programistów zajmuje się tworzeniem biznesowych systemów informacyjnych (EIS — ang. *Enterprise information Systems*). Aplikacje tego typu tworzą, zarządzają i przechowują informacje strukturalne i zapewniają współdzielenie danych przez wielu użytkowników w licznych fizycznych lokalizacjach.

Zapisywanie i pobieranie danych dla EIS wymaga intensywnego korzystania z systemów bazodanowych komunikujących się przy użyciu języka SQL. Każda firma, z którą mieliśmy do czynienia w trakcie pracy zawodowej, korzystała przynajmniej z jednego systemu relacyjnej bazy danych. Większość firm jest całkowicie uzależniona od technologii relacyjnych baz danych, gdyż leżą u samych podstaw ich biznesu.

W ostatnich pięciu latach wzrost liczby implementacji stosujących język Java rósł między innymi z powodu ogromnej chęci skorzystania z obiektowych zasad tworzenia oprogramowania. Programiści są obecnie niezmiernie mocno związani z obiektowością. Z drugiej strony, wiele firm również jest mocno związanych, ale z długoterminowymi inwestycjami w kosztowne relacyjne systemy bazodanowe. Często problemem nie jest samo starsze oprogramowanie, ale zawarte w nim dane, które nadal muszą być odstępne dla użytkowników — nawet jeśli stosują lśniącą, w pełni obiektową aplikację.

Tabelaryczna reprezentacja danych w systemach relacyjnych jest fundamentalnie inna od sieci obiektów używanych w aplikacjach Javy. Różnicę niejednokrotnie

nazywa się niedopasowaniem **paradygmatów relacyjno-objektowych**. Tradycyjnie niedoszacowuje się istotności i kosztu tego niedopasowania. Narzędzia stające się go rozwiązać często okazują się chybione. Gdy programiści Javy za brak dopasowania obwiniają technologię relacyjną, osoby zajmujące się przetwarzaniem danych winią technologię objektową.

Odwzorowanie relacyjno-objektowe (ORM) to nazwa nadana zautomatyzowanym rozwiązaniom starającym się znieść niedopasowanie. Programiści znużeni pisaniem kodu dostępu do danych zapewne ucieśszą się z faktu, iż ORM pozwala uniknąć części żmudnego tworzenia kodu. Aplikacje tworzone z wykorzystaniem ORM okazują się tańsze, niejednokrotnie bardziej wydajne, mniej uzależnione od konkretnego systemu bazodanowego i łatwiej podatne na zmiany w reprezentacji objektowej lub schemacie bazy danych. Co zadziwiające, wszystkie przedstawione zalety dostępne są dla programistów Javy całkowicie za darmo.

Gavin King rozpoczął tworzenie Hibernate pod koniec 2001 roku, kiedy stwierdził, że popularne w tamtym okresie rozwiązanie zapewniające trwałość danych — CMP Entity Beans — nie skaluje się dobrze do dużych aplikacji ze złożonymi modelami danych. Hibernate rozpoczęło swoją karierę jako niezależny, niekomercyjny projekt typu *open source*.

Zespół Hibernate (włączając w to autorów) uczył się ORM w wyjątkowo trudny sposób — słuchał prośb użytkowników i realizował je. Dzięki temu finalne rozwiązanie jest bardziej praktyczne, zwiększa produktywność i przoduje technicznie. Hibernate jest stosowane przez dziesiątki tysięcy użytkowników w dziesiątkach tysięcy aplikacji.

Gdy spełnianie prośb użytkowników stało się zbyt czasochłonne, zespół Hibernate uznał, że przyszły sukces projektu (i zdrowia psychicznego Gavina) wymaga profesjonalnych programistów zatrudnionych na pełny etat pracujących nad Hibernate. Projekt dołączył do jboss.org pod koniec 2003 i obecnie ma aspekt bardziej komercyjny — można wykupić wsparcie techniczne i szkolenia w JBoss Inc. Komercyjne szkolenia to nie jedyny sposób, by poznać Hibernate.

Oczywistym jest, że wiele, o ile nie większość, projektów Javy mogłyby skorzystać na zastosowaniu rozwiązania ORM typu Hibernate, choć jeszcze kilka lat temu nikt o tym nie myślał! Ponieważ technologia ORM staje się coraz popularniejsza, dokumentacja projektu taka jak bezpłatny podręcznik użytkownika systemu Hibernate przestaje wystarczać. Okazało się, że społeczność i nowi użytkownicy mogą potrzebować dłuższej książki opisującej nie tylko poszczególne aspekty Hibernate, ale również omawiającej sam sposób działania podejścia relacyjno-objektowego i wyjaśniającej decyzje projektowe podjęte w trakcie prac nad Hibernate.

Włożyliśmy naprawdę wiele wysiłku w napisanie książki, którą trzymasz w ręku. Poświęcaliśmy jej niemal cały wolny czas przez ponad rok. Prace nad nią zaowocowały wieloma dyskusjami i eksperymentami. Mamy nadzieję, że stanie się ona doskonałym przewodnikiem po Hibernate (a nawet „biblią Hibernate”, jak nazwała ją jeden z recenzentów) i pierwszym pełnym omówieniem odwzorowania objektowo-relacyjnego. Sądzimy, iż czytanie jej sprawi każdemu programiści ogromną przyjemność.

Podziękowania

Pisanie książki nie mogłoby się odbywać bez pomocy. Przede wszystkich chcielibyśmy podziękować społeczności Hibernate za mobilizację; gdyby nie Wasze prośby, być może nie zebrałibyśmy się za jej tworzenie tak szybko.

Książka jest tak dobra, jak jej recenzenci, a my mieliśmy najlepszych: J. B. Rain-sberger, Matt Scarpino, Ara Abrahamian, Mark Eagle, Glen Smichm Patrick Peak, Max Rydahl Anderson, Peter Eisentraut, Matt Raible i Michael A. Koziarski. Dziękujemy Wam za te wszystkie godziny spędzane na czytaniu na wpół skończonego rękopisu. Dziękujemy również Emmanuel Bernard za ocenę techniczną i Nickowi Heudeckerowi za pomoc przy tworzeniu pierwszych rozdziałów.

Zespół wydawnictwa Manning okazał się bezcenny. Clay Andres umożliwił rozpoczęcie prac nad projektem, Jackie Carter była z nami w dobrych i złych chwilach oraz podpowiadała, jak ciekawie pisać. Marjan Bace zarażała swoją pewnością, że wszystko się uda. Tiffany Taylor i Liz Welch znajdowały wiele błędów gramatycznych i stylistycznych. Mary Piergies organizowała produkcję niniejszej książki. Dziękujemy Ci za ciężką pracę. Pozostałym osobom z Manning również serdecznie dziękujemy — bez Was powstanie książki nie byłoby możliwe.

O książce i autorach

W niniejszej książce opisujemy odwzorowanie obiektowo-relacyjne. Z lotu ptaka omawiamy istniejące rozwiązania tego bardzo złożonego zagadnienia. Przedstawiamy, w jaki sposób wykorzystać Hibernate jako warstwę trwałości danych dla dziedzinowego modelu obiektowego pełnego różnych typów danych zawartego w jednej aplikacji. Implementacja warstwy trwałości opisuje wszystkie powiązania encyjne, dziedziczenie klas i specjalne strategie odwzorowania typów.

Uczymy, w jaki sposób dostroić zapytania obiektowe Hibernate i system transakcyjny, by działał z maksymalną wydajnością w aplikacjach z wieloma współużytkownikami. Elastyczny dwuwarstwowy system buforowania również stanowi istotny element książki. Omawiamy integrację z Hibernate w różnych sytuacjach. Przedstawiamy typowe problemy dotyczące dwu- lub trójwarstwowych aplikacji bazodanowych. Jeśli zachodzi konieczność stosowania istniejącej relacyjnej bazy danych, interesujący może okazać się opis funkcji Hibernate związanych z integracją ze starszymi bazami danych oraz zestaw dodatkowych narzędzi programistycznych.

Zawartość poszczególnych rozdziałów

Rozdział 1. zajmuje się trwałością obiektów. Wyjaśniamy w nim, dlaczego relacyjne bazy danych z interfejsem SQL stanowią obecnie podstawową warstwę trwałości danych w aplikacjach i dlaczego zakodowana na stałe w kodzie Javy warstwa trwałości stosująca JDBC oraz zapytania SQL jest żmudna i podatna na błędy.

Poszukując różnych rozwiązań przedstawionego problemu, wprowadzamy odwzorowanie obiektowo-relacyjne wraz z omówieniem jego wad i zalet.

Rozdział 2. stanowi opis Hibernate od strony architektury i przedstawia najbardziej istotne interfejsy programistyczne. Demonstrujemy konfigurację Hibernate w zarządzanych i niezarządzanych środowiskach J2EE i J2SE, wykorzystując prostą aplikację „Witaj świecie”.

Rozdział 3. wprowadza przykładową aplikację oraz wszystkie rodzaje encji i związków występujących w schemacie bazy danych, włączając w to jedno- i dwukierunkowe asocjacje, dziedziczenie klas oraz kompozycje. Omawiamy, w jaki sposób pisać pliki odwzorowań Hibernate i jak projektować klasy trwałości danych.

Rozdział 4. uczy interfejsów związanych z operacjami odczytu i zapisu. Opisuje również działanie trwałości przechodniej (trwałości przez osiągalność) w Hibernate. Rozdział skupia się na możliwe efektywnym odczytem i zapisem obiektów.

Rozdział 5. omawia wspólnie dostępny dostęp do danych, włączając w analizę transakcje bazodanowe oraz te związane z dugo działającymi aplikacjami. Opisuje ogólne podejście do systemów buforujących i szczegółowe rozwiązanie zastosowane w Hibernate. Jest ono wyjątkowo mocno związane ze wspólnie dostępnym dostępem do danych.

Rozdział 6. uzupełnia podstawową wiedzę na temat odwzorowań Hibernate, omawiając bardziej zaawansowane pojęcia, na przykład typy użytkownika, kolekcje wartości, a także odwzorowywanie relacji typu jeden-do-jednego i wiele-do-wielu. Dodatkowo zajmuje się opisem w pełni polimorficznego działania Hibernate.

Rozdział 7. wprowadza język zapytań Hibernate (HQL) i inne metody pobierania obiektów, na przykład zapytania przez określanie kryteriów (QBC), które stanowi bezpieczny pod kątem typów sposób wyrażenia zapytania o obiekt. Przedstawiamy, w jaki sposób przekształcać złożone okna dialogowe wyszukiwania w aplikacji na zapytania przez przykład (QBE). Siła drzewiasta w Hibernate objawia się po połączeniu wszystkich trzech wymienionych elementów. Opisujemy również, w jaki sposób używać bezpośrednich poleceń SQL i jak optymalizować zapytania w celu uzyskania największej wydajności.

Rozdział 8. omawia pewne standardowe podejścia do projektowania aplikacji stosującej Hibernate. Dotyczy to między innymi obsługi SessionFactory, popularnego wzorca ThreadLocal Session oraz hermetyzacji funkcjonalności warstwy trwałości w obiektach dostępu do danych (DAO, ang. *Data Access Object*) i poleceń J2EE. Przedstawiamy, w jaki sposób projektować dugo działające transakcje aplikacji i jak korzystać z innowacyjnej obsługi obiektów odłączonych w Hibernate. Zajmujemy się dodatkowo tworzeniem dzienników dla audytów i korzystaniem z istniejących schematów baz danych.

Rozdział 9. wprowadza kilka innych scenariuszy postępowania w trakcie projektowania oraz narzędzia mogące mieć zastosowanie w takich sytuacjach. Przedstawiamy często popełniane w tych rozwiązaniach błędy. Opisujemy zestaw narzędzi Hibernate (hbm2dd, hbm2java) oraz integrację z innymi bezpłatnymi narzędziami: XDoclet i Middlegen.

Do kogo kierujemy książkę?

Czytelnik niniejszej książki powinien posiadać podstawową wiedzę na temat tworzenia oprogramowania obiektowego i mieć pewne doświadczenie w stosowaniu tej wiedzy w praktyce. Aby zrozumieć prezentowane fragmenty aplikacji, trzeba znać język programowania Java oraz diagramy UML.

Książkę kierujemy przede wszystkim do programistów języka Java korzystających z relacyjnych systemów bazodanowych. Przedstawiamy, w jaki sposób dzięki ORM zwiększyć swoją produktywność.

Osobom projektującym bazy danych niniejsza książka może pomóc w zrozumieniu podstaw tworzenia oprogramowania w sposób obiektowy.

Osoby administrujące bazami danych powinny być zainteresowane wpływem ORM na wydajność oraz sztuczkami poprawiającymi wydajność dzięki odpowiedniemu zgraniu systemu bazodanowego z warstwą trwałości. Ponieważ dostęp do danych stanowi wąskie gardło większości aplikacji Javy, książka kładzie szczególny nacisk na utrzymanie wysokiej wydajności. Wielu administratorów baz danych staje się nerwowych, gdy ma powierzyć utrzymanie wydajności narzędziu automatycznie generującemu kod SQL. Chcemy obalić narosłe w tej kwestii mity i omówić w przypadki, w których lepiej nie korzystać z automatycznych generatorów. Administratorów powinno uspokoić stwierdzenie, iż nie sądzimy, by ORM było najlepszym rozwiązaniem dla każdego typu dostępu do bazy danych.

Konwencje dotyczące kodu i przykładów

Z niniejszą книгą związany jest przykładowy kod zawierający wszystkie niezbędne elementy: kod Javy, pliki konfiguracyjne Hibernate i pliki metadanych odwzorowania XML. Kod źródłowy w tekście oraz w osobnych akapitach pisany jest czcionką o stałej szerokości, aby różnił się od pozostałego tekstu. Dodatkowo nazwy metod Javy, nazwy parametrów, właściwości obiektów i elementów XML również pisane są czcionką o stałej szerokości.

Java, XML i HTML potrafią być bardzo rozwlekłe. W wielu sytuacjach oryginalny kod źródłowy (dostępny do pobrania pod adresem <ftp://helion.pl/przykłady/hibakc.zip>) został przeformatowany — dodawaliśmy przejścia do kolejnych wierszy i modyfikowaliśmy wcięcia, by dostosować je do miejsca dostępnego w książce. W rzadkich przypadkach, gdy formatowanie nie wystarczało, listingi zawierają znaczniki kontynuacji w kolejnym wierszu. Oprócz tego z kodu prezentowanego w książce usunięto komentarze.

Dodatkowe notatki uzupełniające kod źródłowy informują o istotnych pojęciach lub fragmentach. Czasem ponumerowane odnośniki łączą kilka powiązanych ze sobą fragmentów kodu.

Hibernate to projekt typu *open source* udostępniany na licencji *LGPL (Lesser GNU Public License)*. Opis sposobu pobrania pakietu w formie kodu źródłowego lub skompilowanych bibliotek dostępny jest na witrynie www.hibernate.org.

Kod źródłowy dla wszystkich przykładów CaveatEmptor jest dostępny na witrynie <http://caveatemptor.hibernate.org>. Kod aplikacji dostępny jest w różnych odmianach: jako serwlety i komponenty EJB oraz z lub bez warstwy prezentacji. Najbardziej odpowiedni dla niniejszej książki pakiet kodu ze standardową warstwą prezentacji jest dostępny pod adresem <ftp://helion.pl/przykłady/hibakc.zip>.

O autorach

Christian Bauer jest członkiem zespołu projektowego Hibernate. Odpowiada również za witrynę i dokumentację projektu. Interesuje się systemami relacyjnymi baz danych i zarządzaniem dźwiękiem w aplikacjach Javy. Pracuje jako programista i konsultant dla JBoss Inc. Mieszka we Frankfurcie nad Menem w Niemczech.

Gavin King to założyciel projektu Hibernate i główny programista. Jest entuzjastycznym zwolennikiem zwinnego tworzenia oprogramowania (ang. *agile development*) oraz kreowania projektów *open source*. Pomaga integrować technologię ORM ze standardami J2EE jako członek EJB 3 Expert Group. Pracuje jako programista i konsultant dla JBoss Inc. Mieszka w Melbourne w Australii.

Kontakt z autorem

Zakupienie niniejszej książki umożliwia bezpłatny dostęp do prywatnego forum internetowego¹, w którym można zamieszczać komentarze na temat książki, zadawać pytania techniczne oraz otrzymywać pomoc od autorów i innych czytelników. Dokładniejsze informacje na temat dostępu do forum i rejestracji zawarte zostały pod adresem www.manning.com/bauer. Na witrynie wyjaśniono, jak korzystać z forum po uzyskaniu subskrypcji, jakiego rodzaju pomocy można oczekwać i jakie zasady na nim obowiązują. Dodatkowo zawiera łącza internetowe do kodu źródłowego przedstawianego w książce, erraty i innych materiałów.

Wydawnictwo Manning dokłada wszelkich starań, by zapewnić możliwie dobrą komunikację między czytelnikami i autorami książek. Nie jest jednak określony rozmiar pomocy udzielanej przez autorów, ponieważ ich wkład w forum jest dobrowolny (nie otrzymują za to dodatkowego wynagrodzenia). Zalecamy więc, by zadawać autorom takie pytania, które pozwolą rozbudzić w nich chęć pomocy.

¹ Forum dostępne jest w języku angielskim — *przyp. tłum.*

O Hibernate 3 i EJB 3

Świat nie zatrzyma się tylko dlatego, że ktoś czyta niniejszą książkę. Poszczególne fazy tworzenia książki aż do jej umieszczenia na półce zajmują więcej czasu, niż się wydaje. Oznacza to, że niektóre informacje techniczne mogą szybko stać się nieaktualne, szczególnie jeśli na horyzoncie pojawiają się nowe standardy i wersje oprogramowania.

Hibernate 3, nowa, zmodyfikowana wersja Hibernate, w trakcie pisania niniejszego tekstu weszła w fazę planowania i projektowania. Wydaje się, że nowa wersja może zagościć na rynku niewiele później po ukazaniu się naszej publikacji. Warto podkreślić, że informacje zawarte w książce dotyczą również Hibernate 3, ponieważ jest ona w dużej mierze zgodna z Hibernate 2.1. Zaznajomienie się z nowymi funkcjami z kolejnych wersji nie powinno zająć dużo czasu po przeczytaniu książki.

Inspirowana sukcesem Hibernate grupa EJB 3 Expert Group wykorzystała kilka pomysłów i interfejsów programistycznych z tego systemu w tworzonych ziarenkach encji (ang. *entity beans*). W momencie pisania tekstu dostępne były jedynie pierwsze szkice nowej specyfikacji EJB 3, więc nie zostały omówione w książce. Niemniej przeczytanie książki i zapoznanie się z Hibernate pozwala szybko zrozumieć nowy model proponowany w EJB 3.

Najbardziej aktualne informacje zawsze znajdują się na tak zwanej mapie drogowej Hibernate: <http://www.hibernate.org/357.html>.

Trwałość dzięki odwzorowaniu obiektowo-relacyjnemu

W rozdziale:

- ◆ Trwałość obiektów wykorzystujących relacyjne bazy danych
- ◆ Niedopasowanie paradygmatu relacyjnego i obiektowego
- ◆ Warstwy trwałości w aplikacjach obiektowych
- ◆ Podstawy odwzorowania obiektowo-relacyjnego

Sposób zarządzania trwałymi danymi był kluczową decyzją projektową w każdym projekcie, nad którym pracowaliśmy. Wiedząc o tym, iż trwałość danych nie jest niczym nowym lub niezwykłym w aplikacjach Javy, wydawać by się mogło, że wybór będzie dokonywać się między podobnymi rozwiązaniami o ugruntowanej pozycji. Wystarczy pomyśleć o szkieletach aplikacji internetowych (Jakarta Struts kontra WebWork), rozwiązaniami interfejsu graficznego aplikacji (Swing kontra SWT) lub systemach szablonów (JSP kontra Velocity). Każde z rywalizujących rozwiązań ma swoje wady i zalety, ale ogólne podejście i zasięg ich działania jest podobne. Niestety, taka sytuacja nie występuje jeszcze w technologach trwałości danych, gdzie spotyka się wyjątkowo różnorodne podejścia do rozwiązania tego samego problemu.

Od kilku lat trwałość danych jest tematem gorącej dyskusji w społeczności Javy. Wielu programistów i projektantów nie zgadza się nawet co do zasięgu trwałości. Czy „trwałość” to problem rozwiązyany już dawno dzięki technologii relacyjnej i rozszerzeniom typu procedury przechowywane czy raczej bardziej rozległy temat, który musi zostać wykonany z użyciem specjalnego modelu komponentów Javy, na przykład ziarenek encji EJB? Czy należy na stałe w kodzie zapisywać nawet najprostsze operacje typu CRUD (tworzenie, odczyt, aktualizacja, usunięcie), używając języka SQL i bibliotek JDBC czy raczej warto cały proces zautomatyzować? W jaki sposób uzyskać wysoką przenośność rozwiązania, skoro każdy system zarządzania bazą danych stosuje własny dialect SQL? Czy warto całkowicie porzucić SQL i skorzystać z nowych, obiektowych technologii bazodanowych. Choć debata trwa nadal, obecnie dużym powodzeniem cieszy się rozwiązanie stosujące odwzorowanie obiektowo-relacyjne (ORM, ang. *object-relational mapping*). Hibernate jest implementacją zasad ORM w postaci biblioteki typu *open source*.

Hibernate to ambitny projekt, który stara się być kompletnym rozwiązaniem problemu zarządzania trwałością danych w Javie. Stanowi pomost między interakcjami aplikacji a relacyjną bazą danych, co pozwala programistom skoncentrować się na zagadnieniu biznesowym. Hibernate jest rozwiązaniem nieinwazyjnym. Oznacza to, że nie trzeba ściśle trzymać się jego zasad i wzorców projektowych w trakcie pisania własnej logiki biznesowej i klas trwałości danych. Biblioteka doskonale integruje się z większością nowych i istniejących aplikacji. Nie wymaga wielu znaczących zmian w pozostałe części aplikacji.

Niniejsza książka stanowi opis Hibernate. Zajmiemy się podstawowymi i bardziej zaawansowanymi funkcjami. Przedstawimy zalecane podejście do projektowania nowych aplikacji wykorzystujących bibliotekę. W zasadzie większość zaleceń nie dotyczy wyłącznie Hibernate — są to raczej ogólne wskazówki, w jaki sposób najlepiej wykonywać pewne zadania, gdy obsługuje się trwałość danych. Nie zaprzeczamy jednak, że zostaną one przedstawione z punktu widzenia Hibernate. Zanim zaczniemy właściwy opis biblioteki, musimy dobrze omówić problemy trwałości obiektów i odwzorowania obiektowo-relacyjnego. Niniejszy rozdział wyjaśnia, dlaczego w ogóle zrodziła się potrzeba istnienia rozwiązań takich jak Hibernate.

Najpierw zajmiemy się zdefiniowanym zarządzania trwałymi danymi w kontekście aplikacji obiektowych. Omówimy związek między SQL, JDBC i Java, czyli technologiami i standardami, na których bazuje Hibernate. Następnie przejdziemy do omówienia **niedopasowania paradygmatów obiektowego i relacyjnego** oraz typowych problemów spotykanych w trakcie tworzenia oprogramowania obiektowego korzystającego z relacyjnych baz danych. Ponieważ lista problemów z czasem staje się coraz dłuższa, jest też oczywiste, że potrzebne byłyby narzędzia i wzorce minimalizujące czas potrzebny na tworzenie kodu związanego z trwałością realizowaną relacyjnie. Po przyjrzeniu się wielu alternatywnym narzędziom i mechanizmom trwałości okaże się, że w wielu przypadkach najlepszym podejściem okazuje się ORM. Dyskusja na temat wad i zalet ORM zapewnia pełną wiedzę o dostępnych rozwiązańach, by można było wybrać najlepsze dla własnego projektu.

Najwłaściwszy sposób nauki niekoniecznie jest liniowy. Domyślamy się, że Czytelnik zapewne chce od razu wypróbować Hibernate. Jeśli jesteś taką osobą, polecamy przejście do rozdziału 2., a dokładniej do podrozdziału 2.1, w którym to przechodzimy do wykonania niewielkiej aplikacji stosującej Hibernate. Rozdział 2. można zrozumieć bez czytania niniejszego, ale zalecamy powrót do pierwszego elementu książki na pewnym etapie czytania. W ten sposób uzyskasz ogólną wiedzę i poznać podstawowe koncepcje niezbędne do przyswojenia dalszych parti materiału.

1.1. Czym jest trwałość?

Niemalże wszystkie aplikacje wymagają pewnej trwałości danych. Jest to więc jedno z podstawowych pojęć dotyczących tworzenia aplikacji. Jeśli system informacyjny nie zachowywałby danych wpisanych przez użytkowników w momencie wyłączenia zasilania, niewielki byłby z niego pożytek. Gdy mówi się o trwałości w języku Java, najczęściej ma się na myśli przechowywanie danych w **relacyjnej bazie danych** wykorzystującej język SQL. Przyjrzyjmy się najpierw dostępnej technologii i sposobach jej użycia w Javie. Uzbrojeni w tę wiedzę będziemy mogli kontynuować opis trwałości i jej implementacji w aplikacjach obiektowych.

1.1.1. Relacyjne bazy danych

Większość profesjonalnych programistów miała styczność z relacyjnymi bazami danych. W zasadzie większość z nas stosuje tego rodzaju bazy codziennie. Technologia relacyjna jest powszechna i dobrze znana. To wystarczy, by wybierała ją większość firm. Pozostanie przy tym opisie nie nadałoby relacyjności odpowiedniej wagi. Relacyjne bazy danych stały się tak popularne nie przez przypadek — są wyjątkowo elastyczne i szybkie w zarządzaniu danymi.

System zarządzania relacyjną bazą danych nie dotyczy wyłącznie języka Java. Same relacyjne bazy danych nie są związane z konkretną aplikacją. Technologia relacyjna umożliwia współdzielenie danych przez wiele aplikacji lub też w jednej

aplikacji wykorzystującej wiele różnych technologii (na przykład niezależne mechanizmy transakcyjne i raportowania). Relacyjne bazy danych stanowią wspólny mianownik wielu systemów i platform technologicznych. W zasadzie relacyjny model danych jest najczęściej podstawową reprezentacją elementów biznesowych.

Systemy zarządzania relacyjnymi bazami danych wykorzystują interfejs programistyczny — język SQL. Z tego powodu w wielu sytuacjach zamiast mówić „relacyjna baza danych oparta na SQL”, mówi się **baza danych SQL**.

1.1.2. Język SQL

Aby dobrze zrozumieć Hibernate, warto wcześniej zaznajomić się z modelem relacyjnym i językiem SQL. Wiedza ta przyda się później do zwiększenia wydajności aplikacji korzystających z Hibernate. Biblioteka Hibernate automatyzuje wiele wielokrotnie powtarzanych zadań programistycznych, ale wiedza na temat technologii trwałości danych musi wykracać poza tę bibliotekę, by móc w pełny wykorzystać potencjał nowoczesnych baz danych SQL. Przypomnijmy, że naszym celem jest elastyczny i wydajny system zarządzania trwałością danych.

Przedstawmy kilka terminów języka SQL używanych w dalszej książce. Fragmentu języka SQL nazwanego **językiem definicji danych** (DDL, ang. *Data Definition Language*) używa się do tworzenia schematu bazy danych — polecenia CREATE i ALTER. Po wykonaniu tabel (indeksów, sekwencji itp.) korzysta się z **języka modyfikacji danych** (DML, ang. *Data Modification Language*), który zapewnia edycję i pobieranie danych. Przez edycję rozumie się **wstawianie, uaktualnianie i usuwanie** danych. Przez pobieranie danych rozumie się wykonywanie zapytań obejmujących **ograniczenia, projekcje i złączenia** (z uwzględnieniem **iloczynu kartezańskiego**). Przy tworzeniu raportów często korzysta się z operacji grupowania, sortowania i agregacji danych na wiele sposobów. Można nawet zagnieździć jedne instrukcje SQL wewnętrz innych — powstają wtedy tak zwane **podzapytania**. Czytelnik zapewne wykorzystuje język SQL od wielu lat i doskonale zna jego polecenia. Jeśli tak, wie również, że pewne jego elementy trudno zapamiętać a użycie niektórych poleceń zmienia się w zależności od systemu bazodanowego. Zalecamy zajrzenie do dodatku A niniejszej książki, jeśli przedstawione terminy są nowe lub niejasne.

Wiedza na temat SQL jest niezbędna do tworzenia w Javie aplikacji bazodanowych. Dodatkową wiedzę zdobyć można, czytając polecaną przez nas książkę *SQL Tuning* autorstwa Dana Towa [Tow 2003]. Warto również przeczytać książkę *An Introduction to Database System* [Date 2004], by zaznajomić się z teorią, pojęciami i ideałami systemów bazodanowych (głównie relacyjnych). Relacyjna baza danych stanowi jedną z części ORM — drugą są obiekty aplikacji Javy przechowujące swoje dane w bazie danych.

1.1.3. Korzystanie z SQL w Javie

Gdy korzysta się z bazy danych SQL w aplikacji Javy, kod Javy przesyła instrukcje SQL do bazy danych, stosując interfejs programistyczny JDBC (*Java DataBase Connectivity*). Kod SQL można napisać ręcznie i dołączyć do kodu lub też gene-

rować przy użyciu dowolnego kodu Javy. Interfejs JDBC pozwala dołączyć argumenty do parametrów zapytań, zainicjować wykonanie zapytania, przejść przez zwróconą listę wyników, pobrać poszczególne wynikowe wartości itp. Są to zadania niskopoziomowego dostępu do danych. Jako programiści aplikacji jesteśmy raczej zainteresowani problemem biznesowym, który tylko wymaga dostępu do danych. Większość osób uważa, że programista aplikacji nie powinien zajmować się takimi żmudnymi, mechanicznymi szczegółami.

Tak naprawdę potrzebny jest system, który zapisuje i pobiera złożone obiekty — egzemplarze klas — do i z bazy danych, odciążając tym samym programistę od zadań niskopoziomowych.

Ponieważ zadania dostępu do danych są bardzo często żmudne, musimy zapytać: czy model relacyjny, a w szczególności język SQL, to dobry wybór rozwiązania zapewniającego trwałość danych aplikacjach obiektowych? Na to pytanie warto udzielić błyskawicznej odpowiedzi: tak! Istnieje wiele powodów, dla których bazy danych SQL dominują w komputerowym świecie. Relacyjne systemy bazodanowe to jedyna sprawdzona technologia zarządzania danymi i nieomal zawsze wymagany składnik każdego projektu Javy.

Przez ostatnich 15 lat programiści mówili o **niedopasowaniu paradygmatów**. Tłumaczy ono dlaczego w każdym projekcie programistycznym związanym z biznesem tak dużo czasu poświęca się na sprawy związane z trwałością danych. Paradygmatami są modele obiektowy i relacyjny, które przekładają się na programowanie obiektowe i język SQL. Zaczniemy wyjaśnianie problemu niedopasowania od zadania sobie pytania, co oznacza **trwałość** w kontekście tworzenia aplikacji obiektowej. Najpierw rozszerzymy uproszoną definicję trwałości przedstawioną na początku rozdziału na szersze, bardziej dojrzałe wyjaśnienie kroków związanych z utrzymaniem i wykorzystaniem trwałych danych.

1.1.4. Trwałość w aplikacjach obiektowych

W aplikacjach obiektowych trwałość pozwala obiektowi żyć dłużej niż proces, który go utworzył. Stan obiektu zostaje zapisany na dysku twardym. Po pewnym czasie obiekt powstaje ponownie z użyciem wcześniejszego stanu.

To rozwiązanie nie jest ograniczone do pojedynczych obiektów — całe grafy powiązanych ze sobą obiektów mogą zostać utrwalone na stałe, a następnie ponownie utworzone w przyszłości. Większość obiektów nie jest trwała; obiekty **ulotne** (zwane też tymczasowymi) mają ograniczony czas życia ściśle związany z tworzącym je procesem. Niemal wszystkie aplikacje Javy zawierają mieszankę obiektów trwałych i ulotnych. Z tego powodu potrzebny jest podsystem zarządzający danymi trwałymi.

Nowoczesne, relacyjne bazy danych zapewniają strukturalną reprezentację trwałych danych oraz możliwość sortowania, wyszukiwania i agregacji danych. Systemy bazodanowe zajmują się obsługą dostępu wspólnego i integralnością danych, czyli zapewniają poprawne współdzielenie danych przez wielu użytkowników i aplikacji. System bazodanowy dodatkowo chroni dane, dostarczając odpowiednich zabezpieczeń. Gdy w niniejszej książce mamy na myśli trwałość, myślimy o wszystkich wymienionych poniżej elementach:

- ◆ zapamiętywanie, organizacja i pobieranie strukturalnych danych,
- ◆ współbieżność i integralność danych,
- ◆ współdzielenie danych.

W szczególności mamy na myśli wymienione zagadnienia w kontekście aplikacji obiektowej stosującej **model dziedzinowy**.

Aplikacja z modelem dziedzinowym nie działa bezpośrednio na tabelarycznej reprezentacji bytów biznesowych; stosuje raczej własny, obiektowy model tych bytów. Jeśli baza danych zawiera tabele ITEM i BID, aplikacja Javy stosuje klasy Item i Bid.

Następnie zamiast bezpośrednio pracować na kolumnach i wierszach zbioru wyników SQL, logika biznesowa wchodzi w interakcję ze wspomnianym obiektowym modelem dziedzinowym i jego realizacją w trakcie wykonywania programu (graf powiązanych ze sobą obiektów). Logika biznesowa nigdy nie jest wykonywana w bazie danych (jako procedura zapamiętana SQL) — stanowi element kodu napisanego w Javie. W ten sposób logika biznesowa potrafi wykorzystać zaawansowane pojęcia obiektowe, między innymi dziedziczenie i polimorfizm. Nic nie stoi na przeszkodzie, by zastosować dobrze znane wzorce projektowe, na przykład strategię (*Strategy*), mediator (*Mediator*) lub złożenie (*Composite*) [GOF 1995], które w dużej mierze polegają na polimorficznych wywołaniach metod. Pojawia się pewien problem — nie wszystkie aplikacje są i muszą być projektowane w ten właśnie sposób. Prostsze aplikacje niejednokrotnie okazują się lepsze bez modelu dziedzinowego. Język SQL i interfejs programistyczny JDBC doskonale radzą sobie z danymi czysto tabelarycznymi. Nowy element JDBC o nazwie RowSet (Sun JPC, JSR 114) czyni operacje CRUD jeszcze prostszymi. Korzystanie z trwałych danych w postaci tabeli jest łatwe i nie wymaga dużej wiedzy.

Gdy aplikacja stosuje bardziej wyrafinowaną logikę biznesową, model dziedzinowy poprawia wielokrotne wykorzystanie kodu i pomaga zachować wysoką elastyczność. W książce skupimy się na aplikacjach wykorzystujących model dziedzinowy, ponieważ to przede wszystkim do nich kierowany jest ORM i Hibernate.

Jeśli ponownie rozważymy SQL i relacyjne bazy danych, bez problemu zauważymy niedopasowanie obu paradygmatów.

Operacje SQL takie jak projekcja i złączenie zawsze powodują uzyskanie tabelarycznej reprezentacji wynikowych danych. Nie jest to równoważne z grafem powiązanych obiektów wykonujących logikę biznesową w aplikacji Javy! Są to całkowicie różne modele, nie tylko inny sposób wizualizacji tego samego modelu.

Uzmysławiając sobie różnicę, widzimy istniejące problemy — niektóre dobrze zrozumiałe, inne nieco mniej — które muszą zostać rozwiązane przez aplikację korzystającą z obu reprezentacji danych: obiektowego modelu dziedzinowego i trwałego modelu relacyjnego. Przyjrzymy się bliżej tej różnicy.

1.2. Niedopasowanie paradygmatów

Niedopasowanie paradygmatów można podzielić na kilka części. Każdą zajmiemy się osobno. Zaczniemy od przedstawienia prostego przykładu, który nie jest problematyczny. Gdy zaczniemy go rozbudowywać, niedopasowanie powoli się uwidocznii.

Przypuśćmy, że projektujemy implementację aplikacji sklepu internetowego. Aplikacja potrzebuje klasy reprezentującej informacje o użytkowniku systemu oraz klasy reprezentującej informacje o szczegółach płatności (patrz rysunek 1.1).



Rysunek 1.1. Prosty diagram UML z encjami użytkownika i szczegółów płatności

Zgodnie z diagramem klasa User może być związana z wieloma klasami BillingDetails. Przechodzenie przez związek odbywa się w dwóch kierunkach. Zaczniemy od sytuacji, w której klasy reprezentujące encje są bardzo proste.

```
public class User {  
    private String userName;  
    private String name;  
    private String address;  
    private Set billingDetails;  
    // metody ustawiania i pobierania, metody biznesowe itp.  
    ...  
}  
public class BillingDetails {  
    private String accountNumber;  
    private String accountName;  
    private String accountType;  
    private User user;  
    // metody ustawiania i pobierania, metody biznesowe itp.  
    ...  
}
```

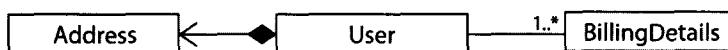
Warto zauważyć, że jesteśmy zainteresowani tylko stanem encji dotyczącej trwałych danych. Pomijamy implementacje metod dostępowych właściwości i metod biznesowych (na przykład `getUserName()` lub `billAuction()`). Dla przedstawionej sytuacji nietrudno wymyślić poprawny schemat bazy danych opisany poniższymi poleceniami SQL.

```
create table USER (  
    USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,  
    NAME VARCHAR(50) NOT NULL,  
    ADDRESS VARCHAR(100)  
)  
create table BILLING_DETAILS (  
    ACCOUNT_NUMBER VARCHAR(10) NOT NULL PRIMARY KEY,  
    ACCOUNT_NAME VARCHAR(50) NOT NULL,  
    ACCOUNT_TYPE VARCHAR(2) NOT NULL,  
    USERNAME VARCHAR(15) FOREIGN KEY REFERENCES USER  
)
```

Związek między dwoma encjami reprezentuje klucz obcy, USERNAME, z BILLING_DETAILS. W tak prostym modelu obiektowym niedopasowanie obiektowo-relacyjne jest prawie niewidoczne. Napisanie kodu JDBC wstawiającego, aktualizującego i usuwającego informacje o użytkowniku i płatnościach nie sprawia problemów.

Zastanówmy się, co się stanie, gdy rozważymy nieco bardziej realistyczny przypadek. Niedopasowanie uwidocznii się wraz z dodaniem kolejnych encji i ich związków.

Przedstawiona implementacja zawiera dosyć oczywisty błąd, ponieważ modeluje adres jako pojedynczą wartość typu String. W większości systemów konieczne jest przechowywanie ulicy, miasta, województwa, kraju i kodu pocztowego w osobnych polach. Oczywiście nic nie stoi na przeszkodzie, by dodać te informacje bezpośrednio do klasy User. Warto jednak postąpić inaczej, gdyż inne klasy aplikacji również mogą przechowywać adresy. Utwórzmy osobną klasę Address. Uaktualniony model obiektów przedstawia rysunek 1.2.



Rysunek 1.2. Klasa User zawiera klasę Address

Czy także powinniśmy dodać tabelę ADDRESS? Niekoniecznie. Najczęściej dane adresowe przechowuje się w tabeli USER, stosując kilka kolumn. Rozwiązanie to zapewne będzie działać wydajniej, bo nie trzeba dokonywać złączenia, by w jednym zapytaniu pobrać dane użytkownika i jego adres. Najlepsze rozwiązanie mogłoby wykorzystywać zdefiniowany przez siebie typ danych SQL, by w tabeli zamiast kilku kolumn stosować tylko jedną.

Wybór sprowadza się do określenia, czy należy zastosować kilka kolumn czy lepiej użyć jednej (nowy typ danych SQL). Wszystko zależy od wymaganego poziomu szczegółowości.

1.2.1. Problem szczegółowości

Szczegółowość dotyczy względnego rozmiaru obiektów, nad którymi się pracuje. Gdy mowa o obiektach Javy i tabelach bazy danych, problem szczegółowości oznacza obiekty trwałe, które mogą w różny sposób odnosić się do szczegółowości tabel i kolumn (jest ona mocno ograniczona).

Powróćmy do przykładu. Dodanie nowego typu danych do przechowywania obiektów Address w pojedynczej kolumnie bazy danych brzmi bardzo dobrze. Przecież nowy typ Address (klasa) w Javie i nowy typ danych SQL o nazwie ADDRESS powinny gwarantować dobre współdziałanie. Niestety, dokładne przyjrzenie się obsłudze definiowanych przez siebie kolumn (UDT) w obecnych systemach bazodanowych spowoduje natknęcie się na kilka problemów.

Obsługa UDT to jedno z tak zwanych rozszerzeń obiektowo-relacyjnych rozbudowujących język SQL. Niestety, obsługę UDT w wielu systemach bazodanowych potraktowano po macoszemu, a co gorsza, nie jest ona przenośna między różnymi systemami. Standard SQL obsługuje zdefiniowane przez użytkownika typy danych wyjątkowo marnie. Z tego powodu (i być może wielu innych), obsługa

UDT nie jest obecnie często spotykana w aplikacjach. Jeszcze mniejsze są szanse spotkania jej w starszych schematach baz danych. Wynika stąd, że przechowanie nowej klasy Address w jednej kolumnie odpowiedniego typu danych SQL jest bardzo niepraktyczne. Pozostaje podejście standardowe, czyli zastosowanie kilku kolumn o typach danych zdefiniowanych przez twórcę systemu bazodanowego (typy numeryczne, logiczne i tekstowe). Po uwzględnieniu szczegółowości tabel, nowa tabela USER ma postać przedstawioną poniżej.

```
create table USER (
    USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,
    NAME VARCHAR(50) NOT NULL,
    ADDRESS_STREET VARCHAR(50),
    ADDRESS_CITY VARCHAR(15),
    ADDRESS_STATE VARCHAR(15),
    ADDRESS_ZIPCODE VARCHAR(5),
    ADDRESS_COUNTRY VARCHAR(15)
)
```

Prowadzi to do następującej obserwacji: klasy w dziedzinowym modelu obiektowym będą miały różny zakres szczegółowości — od bardzo ogólnych klas encyjnych jak User przez bardziej szczegółowe typu Address po specjalistyczne jak właściwość typu String z kodem pocztowym.

W bazie danych widoczne są tylko dwa poziomy szczegółowości: tabele typu USER i kolumny skalarne typu ADDRESS_ZIPCODE. Rozwiążanie to nie jest tak elastyczne jak system typów Javy. Wiele prostszych mechanizmów trwałości nie rozpoznaje poprawnie tego niedopasowania i tym samym wymusza mniej elastyczną reprezentację modelu obiektowego. Widzieliśmy niezliczoną ilość klas User z właściwościami dotyczącymi kodu pocztowego!

Okazuje się, że problem szczegółowości nie jest szczególnie trudny do rozwiązania. W zasadzie nawet byśmy go nie wspominali, gdyby nie to, że występuje w tak wielu istniejących systemach. Rozwiążanie tego problemu prezentujemy w rozdziale 3., a konkretnie w podrozdziale 3.5.

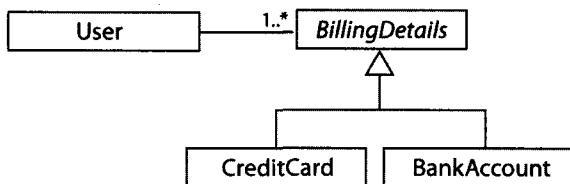
Znacznie trudniejszy i interesujący problem pojawia się, gdy rozważymy dziedzinowe modele obiektowe używające **dziedziczenia**, czyli cechy projektowania obiektowego dającej całkowicie nowe możliwości rozliczania użytkowników w aplikacji sklepu internetowego.

1.2.2. Problem podtypów

W Javie dziedziczenie implementuje się, stosując klasy bazowe i podklasy. Aby zilustrować, dlaczego może to prowadzić do problemu niezgodności, rozwiniemy poprzedni przykład. Zmodyfikujmy go tak, by obsługiwał nie tylko płatności przelewem, ale także karty kredytowe i debetowe. Uzyskujemy w ten sposób kilka sposobów zapłaty. Najbardziej naturalnym sposobem wprowadzenia modyfikacji będzie zastosowanie dziedziczenia dla klasy BillingDetails.

Możemy zastosować abstrakcyjną klasę bazową BillingDetails z kilkoma konkretnymi podklasami: CreditCard, DirectDebit, Cheque itp. Każda z podklas

definiuje nieco inne dane (i przy okazji całkowicie inną funkcjonalność związaną z nowym typem płatności). Nowy model ilustruje rysunek 1.3.



Rysunek 1.3.
Zastosowanie dziedziczenia dla różnych rodzajów płatności

Osoby znające SQL wiedzą, że w żaden sposób nie obsługuje on dziedziczenia. Nie można wskazać, że tabela CREDIT_CARD_DETAILS jest podtypem tabeli BILLING_DETAILS, pisząc wprost CREDIT_CARD_DETAILS EXTENDS BILLING_DETAILS (...).

W podrozdziale 3.6 z rozdziału 3. omawiamy, w jaki sposób mechanizmy odwzorowań obiektowo-relacyjnych (między innymi Hibernate) radzą sobie z przenoszeniem hierarchii klas na tabele baz danych. Problem ten został dosyć dobrze zrozumiany przez społeczność, więc większość rozwiązań oferuje zbliżoną funkcjonalność. Dziedziczenie to jeszcze nie koniec. Wraz z jego wprowadzeniem do modelu pojawia się również **polimorfizm**.

Klasa User jest powiązana z klasą bazową BillingDetails. Jest to **asocjacja polimorficzna**. W trakcie działania programu obiekt User może zawierać referencję do dowolnego egzemplarza jednej z podklas BillingDetails. Nic nie stoi na przeszkodzie, by kreować zapytania odnoszące się do klasy BillingDetails, które zwracają podklasy klasy bazowej. Mówiąc się wtedy o zapytaniach **polimorficznych**.

Ponieważ bazy danych SQL nie oferują żadnej notacji związanej z dziedziczeniem, nie powinno dziwić, że nie obsługują również asocjacji polimorficznych. Standardowy klucz obcy dotyczy tylko jednej konkretnej tabeli i nie łatwo określić klucz związanego z wieloma tabelami. Można to wyjaśnić, mówiąc, iż Java (i inne języki obiektowe) jest mniej konserwatywna w kwestii typów niż SQL. Na szczęście dwa rozwiązania odwzorowań przedstawiane w rozdziale 3. zostały tak zaprojektowane, by obsługiwać reprezentację asocjacji polimorficznych i wydajnie wykonywać zapytania polimorficzne.

Brak obsługi podtypów to sytuacja, w której struktura dziedziczenia modelu Javy musi zostać trwale zawarta w bazie danych SQL, gdyż nie oferuje ona (baza danych) żadnej strategii dziedziczenia. Kolejnym aspektem problemu niedopasowania jest kwestia **tożsamości** obiektu. Jako klucz główny tabeli USER wybraliśmy we wcześniejszych przykładach pole USERNAME. Czy był to dobry wybór? Raczej nie, o czym się wkrótce przekonasz.

1.2.3. Problem identyczności

Choć problem **identyczności obiektu** nie od razu rzuca się w oczy, zetkniami się z nim w momencie dalszej rozbudowy systemu sklepu internetowego. Problem pojawi się, gdy zaczniemy rozważać dwa obiekty (na przykład klasy User) i testować, czy są identyczne. Istnieją trzy rozwiązania: dwa ze świata Javy i jedno zwią-

zane z bazą danych SQL. Jak się nietrudno domyślić, będą działać wspólnie dopiero przy odrobinie pomocy.

Obiekty Javy definiują dwa różne pojęcia **równości**:

- ◆ identyczność obiektów (w zasadzie równoważne tej samej lokalizacji w pamięci; sprawdzane za pomocą `a==b`),
- ◆ równości testowanej na podstawie odpowiedniej implementacji metody `equals()`; mówiąc wtedy o **równości na podstawie wartości**.

Z drugiej strony, identyczność wierszy bazy danych wyraża wartość klucza głównego. W podrozdziale 3.4 udowodnimy, że ani `equals()`, ani `==` nie są w sposób naturalny równoważne wartości klucza głównego. Często zdarza się, że kilka (nie-identycznych) obiektów istniejących jednocześnie reprezentuje ten sam wiersz bazy danych. Co więcej, pojawiają się pewne subtelne trudności z poprawną implementacją metody `equals()` dla trwałych klas.

Ponownie posłużymy się przykładem do zobrazowania problemu. W definicji tabeli USER jako klucz główny wybraliśmy pole USERNAME. Taka decyzja znacząco utrudnia zmianę nazwy użytkownika: w takiej sytuacji musielibyśmy zmienić nie tylko zawartość pola USERNAME tabeli USER, ale także zawartość klucza obcego z tabeli BILLING_DETAILS. W dalszej części książki zalecamy stosowanie **kluczy czysto identyfikacyjnych**, czyli takich, które nie mają żadnego znaczenia dla użytkownika końcowego. Warto zmienić definicje tabel na następujące:

```
create table USER (
    USER_ID BIGINT NOT NULL PRIMARY KEY,
    USERNAME VARCHAR(15) NOT NULL UNIQUE,
    NAME VARCHAR(50) NOT NULL,
    ...
)
create table BILLING_DETAILS (
    BILLING_DETAILS_ID BIGINT NOT NULL PRIMARY KEY,
    ACCOUNT_NUMBER VARCHAR(10) NOT NULL UNIQUE,
    ACCOUNT_NAME VARCHAR(50) NOT NULL,
    ACCOUNT_TYPE VARCHAR(2) NOT NULL,
    USER_ID BIGINT FOREIGN KEY REFERENCES USER
)
```

Kolumny `USER_ID` i `BILLING_DETAILS_ID` zawierają wartości wygenerowane przez system. Zostały wprowadzone tylko po to, by wspomóc relacyjny model danych. W jaki sposób (jeśli w ogóle) wartości te powinny być reprezentowane w modelu obiektowym? Zadamy to pytanie raz jeszcze w podrozdziale 3.4 i przedstawimy rozwiązanie stosujące odwzorowanie obiektowo-relacyjne.

W kontekście trwałości identyczność jest ściśle związana ze sposobami, w jakie system obsługuje buforowanie i transakcje. Różne rozwiązania trwałości wybrały inne podejścia. Pojawiło się wiele niezgodności i nieporozumień. W rozdziale 5. przedstawimy ten interesujący temat i wszystko, co jest z nim związane.

Szkielet aplikacji sklepu internetowego projektowany i implementowany w kolejnych podrozdziałach spełnił swe zadanie. Dzięki niemu wskazaliśmy problemy niedopasowania związane z poziomami szczegółowości, podtypami i identycznością obiektu. Jesteśmy prawie gotowi, by przejść do innych części aplikacji.

Wcześniej jednak zajmiemy się ważnym pojęciem **asocjacji** — sposobem, w jaki obsługiwane są związki międzykласowe. Czy wystarczy klucz obcy z bazy danych?

1.2.4. Problemy dotyczące asocjacji

W modelu obiektowym asocjacje reprezentują związki między encjami. Przypomnijmy, że klasy User, Address i BillingDetails są ze sobą powiązane. W odróżnieniu od Address, BillingDetails może występować w pełni niezależnie. Dane tego obiektu znajdują się w osobnej tabeli. Odwzorowanie asocjacji i zarządzanie nimi to niezwykle istotny element każdego rozwiązania trwałości obiektów.

Języki obiektowe reprezentują asocjacje, używając referencji do obiektów lub kolekcji tych referencji. W świecie relacyjnym asocjację reprezentuje kolumna klucza obcego, która zawiera kopię wartości klucza głównego innej tabeli. Istnieją subtelne różnice między tymi dwiema reprezentacjami.

Referencje obiektowe są bez wątpienia kierunkowe — asocjacja biegnie od jednego obiektu do drugiego. Jeśli asocjacja między obiektemi ma być dwukierunkowa, trzeba zdefiniować ją **dwukrotnie** — raz w każdej z klas. To podejście zostało przedstawione w przykładowym modelu klas.

```
public class User {
    private Set billingDetails;
    ...
}
public class BillingDetails {
    private User user;
    ...
}
```

Z drugiej strony, klucze obce nie są ze swojej natury dwukierunkowe. W gruncie rzeczy **nawigacja** nie ma żadnego znaczenia w relacyjnym modelu danych, gdyż **łączenia i projekcje** pozwalają tworzyć dowolne asocjacje danych.

W zasadzie nie można określić krotkości asocjacji jednokierunkowych, przyglądając się jedynie klasom Javy. Asocjacje te mogą być typu **wiele-do-wielu**. Nic nie stoi na przeszkodzie, by model obiektowy wyglądał następująco:

```
public class User {
    private Set billingDetails;
    ...
}
public class BillingDetails {
    private Set users;
    ...
}
```

Asocjacje w bazie danych zawsze są typu **jeden-do-wielu lub jeden-do-jednego**. Krotkość łatwo określić. Wystarczy spojrzeć na definicję klucza obcego. Poniższy fragment stanowi asocjację jeden-do-wielu (lub jak kto woli wiele-do-jednego):

```
USER_ID BIGINT FOREIGN KEY REFERENCES USER
```

Istnieją również asocjacje jeden-do-jednego:

```
USER_ID BIGINT UNIQUE FOREIGN KEY REFERENCES USER  
BILLING_DETAILS_ID BIGINT PRIMARY KEY FOREIGN KEY REFERENCES USER
```

Aby w relacyjnej bazie danych zatrzymać związek wiele-do-wielu, trzeba wprowadzić nową tabelę nazywaną **tabelą łączącą lub asocjacyjną**. Tabela ta nie pojawia się nigdzie w modelu obiektowym. Jeśli chcielibyśmy w przedstawianym przykładzie zastosować związek wiele-do-wielu między użytkownikiem a sposobami płatności, tabela łącząca wyglądałaby tak:

```
CREATE TABLE USER_BILLING_DETAILS  
    USER_ID BIGINT FOREIGN KEY REFERENCES USER,  
    BILLING_DETAILS_ID BIGINT FOREIGN KEY REFERENCES BILLING_DETAILS  
    PRIMARY KEY (USER_ID, BILLING_DETAILS_ID)  
)
```

Odwzorowania asocjacji znacznie bardziej szczegółowo omawiamy w rozdziałach 3. i 6.

Do tej pory omawiane kwestie dotyczyły struktury. Rozpoznajemy je, analizując czysto statyczny obraz systemu. Prawdopodobnie najtrudniejszym aspektem trwałości obiektów jest dynamizm. Dotyczy on asocjacji. Wskazaliśmy go w momencie rozróżniania nawigacji po grafie obiektów i złączeń tabel w podrozdziale 1.1.4. Przyjrzyjmy się dokładniej temu istotnemu niedopasowaniu.

1.2.5. Problem nawigacji po grafie obiektów

Istnieje fundamentalna różnica w sposobie dostępu do obiektów w Javie i w relacyjnych bazach danych. W Javie dostęp do informacji o sposobach płatności użytkownika odbywa się dzięki wywołaniu `aUser.getBillingDetails().getAccountNumber()`. Jest to najbardziej naturalny sposób dostępu do danych obiektowych. Często nazywa się go **chodzeniem po grafie obiektów**. Przechodzi się z jednego obiektu do drugiego, podążając za asocjacjami między kolejnymi egzemplarzami klas. Niestety, nie jest to wydajny sposób pobierania informacji z bazy danych SQL.

Gdy chce się osiągnąć wysoką wydajność kodu dostępu do danych, warto **minimalizować liczbę żądań kierowanych do bazy danych**. Najłatwiej osiągnąć to, zmniejszając liczbę zapytań SQL (inne rozwiązania to zastosowanie procedur zapamiętanych lub interfejsu wsadowego JDBC).

Oznacza to, że wydajny dostęp do danych w języku SQL wymaga stosowania złączeń dla pobieranych tabel. Liczba złączonych tabel określa głębię grafu obiektów, po którym będzie można się poruszać. Jeśli zachodzi potrzeba pobrania obiektu User, ale nie są potrzebne dane zawarte w obiekcie BillingDetails, wystarczy proste zapytanie SQL.

```
select * from USER u where u.USER_ID = 123
```

Z drugiej strony, gdy należy pobrać tego samego użytkownika, a następnie przejść przez szczegóły zawarte w obiekcie BillingDetails, lepiej użyć innego zapytania.

ROZDZIAŁ 1.***Trwałość dzięki odwzorowaniu obiektowo-relacyjnemu***

```
select *
from USER u
left outer join BILLING_DETAILS bd on bd.USER_ID = u.USER_ID
where u.USER_ID = 123
```

Nietrudno zauważyc, że musimy znać fragment grafu obiektów, przez który planujemy przejść, w momencie pobierania danych podstawowego obiektu User, czyli przed rozpoczęciem właściwej nawigacji po grafie!

Z drugiej strony, dowolne rozwiązywanie trwałości obiektów zapewnia pobieranie danych powiązanych obiektów dopiero wtedy, gdy po raz pierwszy chce się skorzystać z obiektu. Niestety, ten rozłożony na wiele kroków dostęp do danych jest wyjątkowo mało wydajny w kontekście relacyjnych baz danych, wymaga bowiem wykonania jednego zapytania pobierającego na każdy węzeł grafu obiektów. Pojawia się problem **n+1 pobrań**.

To niedopasowanie sposobu dostępu do obiektów Javy i danych relacyjnej bazy danych stanowi najpowszechniejsze źródło problemów z wydajnością w aplikacjach Javy. Choć niezliczona ilość książek i artykułów zaleca, by do łączenia tekstów stosować klasę StringBuffer, wręcz niemożliwym wydaje się odnalezienie dobrego rozwiązania problemu *n+1 pobrań*. Hibernate stosuje zaawansowane algorytmy, by wydajnie pobierać z bazy danych graf obiektów. Czyni to w sposób całkowicie niewidoczny dla aplikacji korzystającej z grafu. Algorytmy te omawiamy w rozdziałach 4. i 7.

Zebrala się już stosunkowo obszerna lista problemów niedopasowania obiektowo-relacyjnego. Znalezienie odpowiednich rozwiązań jest kosztowne. Niejedna osoba przekonała się o tym na własnej skórze. Co więcej, niejednokrotnie niedoszacowuje się tego kosztu, co jak sądzimy, staje się przyczyną upadku wielu projektów informatycznych.

1.2.6. Koszt niedopasowania

Rozwiązywanie wszystkich wspomnianych problemów niedopasowania w sposób ogólny może wymagać ogromnego wysiłku i mnóstwa czasu. Z własnego doświadczenia wiemy, że głównym celem do 30% kodu aplikacji języka Java jest obsługa bazy danych w języku SQL z użyciem JDBC, a także ręczne kreowanie pomostu między modelem relacyjnym i obiektowym. Pomimo tego ogromnego wysiłku końcowy efekt niejednokrotnie nie zadowala. Nie raz widzieliśmy projekty, które tonęły z powodu złożoności i niskiej elastyczności stosowanych warstw abstrakcji baz danych.

Największy koszt dotyczy fazy modelowania. Modele obiektowy i relacyjny muszą uwzględniać te same byty biznesowe. Osoby w pełni oddane projektowaniu obiektowemu inaczej zamodelują te byty niż doświadczony projektant relacyjnych baz danych. Typowo tak długo rozciąga się i skręca model obiektowy, aż nie dopasuje się do wykorzystywanej technologii relacyjnej.

Zadanie to niejednokrotnie udaje się wykonać kosztem niektórych zalet podejścia obiektowego. Warto pamiętać o ograniczeniach modelu relacyjnego związanych z teorią relacyjną. Podejście obiektowe nie stosuje tak rygorystycznych definicji matematycznych ani teorii. Nie warto więc spoglądać w stronę mate-

matyki, poszukując rozwiązań mogących zatkać dziurę między dwoma paradygmatami — nie istnieje eleganckie przekształcenie czekające na odkrycie. (Odrzucenie Javy i SQL-a w celu rozpoczęcia pracy od podstaw nie uważa się za eleganckie podejście).

Problem niedopasowania modeli nie jest jedynym źródłem niskiej elastyczności i utraty produktywności prowadzącej nieubłaganie ku wyższym kosztom. Kolejnym powodem problemów jest sam interfejs programistyczny JDBC. JDBC i SQL korzystają z **instrukcyjnego** (poleceńowego) podejścia do przesyłania danych do i z bazy danych. Związek strukturalny trzeba określić przynajmniej trzy razy (INSERT, UPDATE, SELECT), dodając go do czasu wymaganego na projekt i implementację. Różnice w dialektach SQL poszczególnych systemów bazodanowych nie ułatwiają zadania.

Ostatnio modne stało się traktowanie modeli architektonicznych lub wzorcowych jako częściowego rozwiązania problemu niedopasowania. Z tego powodu powstał model komponentowy z ziarenkami encyjnymi, wzorzec obiektu dostępu do danych (DAO) i inne próby nowej implementacji dostępu do danych. Podejścia te pozostawiają programistom do rozwiązania większość z wcześniej wymienionych problemów. Aby dopełnić opis trwałości obiektów, musimy jeszcze omówić architekturę aplikacji i rolę warstwy trwałości w projektowaniu typowej aplikacji.

1.3. Warstwy trwałości i alternatywy

W dużej i średniej wielkości aplikacjach najczęściej ma sens organizacja klas według zajęć. Trwałość jest jednym z takich zajęć. Pozostałymi są: prezentacja, przepływ zadań i logika biznesowa. Istnieją także zajęcia „podstawowe”, które dają się zaimplementować w sposób ogólny, na przykład jako kod szkieletowy. Typowymi zajęciami podstawowymi są: dziennik zdarzeń, uwierzytelnianie i działania transakcyjne.

Typowa architektura obiektowa składa się z warstw dotyczących konkretnych zajęć. Powszechną i wysoce zalecaną praktyką jest grupowanie wszystkich klas i komponentów odpowiedzialnych za trwałość na osobnej warstwie trwałości w warstwowej architekturze systemowej.

W tym podrozdziale zajmiemy się najpierw warstwom tego rodzaju architektury i powodom ich stosowania. Następnie skupimy się na warstwie, której jesteśmy szczególnie zainteresowani — warstwie trwałości danych — oraz sposobach jej implementacji.

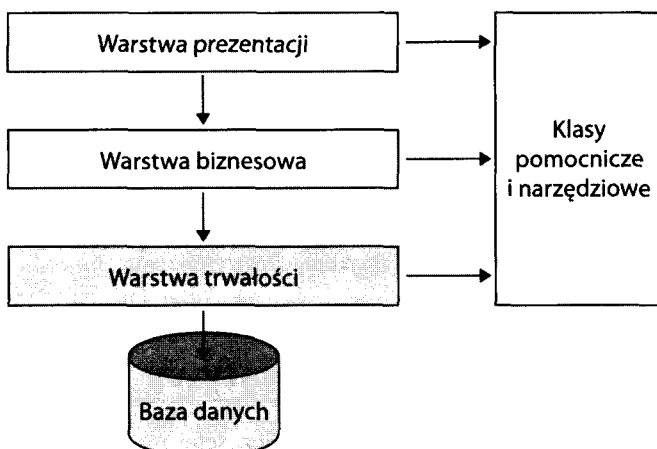
1.3.1. Architektura warstwowa

Architektura warstwowa definiuje interfejs między kodem, który implementuje różne zajęcia, by zmiana w jednym z zajęć i sposobie jego wykonania, w jaki najmniejszym stopniu wpływała na kod związany z pozostałymi zajęciami.

Tworzenie warstw uwidacznia bardzo dobrze istniejące powiązania między warstwowe. Zasady są następujące:

- ◆ Warstwy komunikują się ze sobą z góry na dół. Warstwa jest zależna jedynie od warstwy znajdującej się tuż poniżej.
- ◆ Każda warstwa jest nieświadoma innych warstw poza tą, która znajduje się poniżej.

Różne aplikacje grupują zajęcia w różny sposób, więc nie stosują identycznych warstw. Typowa, sprawdzona architektura aplikacji wysokiego poziomu stosuje trzy warstwy: prezentacji, logiki biznesowej i trwałości danych (patrz rysunek 1.4).



Rysunek 1.4. Warstwa trwałości stanowi standardowy element architektury warstwowej

Przyjrzyjmy się bliżej warstwom i innym elementom z rysunku.

- ◆ **Warstwa prezentacji** — logika interfejsu użytkownika znajduje się na samej górze. Zawiera kod odpowiedzialny za nawigację między stronami, wyświetlanie formularzy i tabel.
- ◆ **Warstwa biznesowa** — dokładna postać kolejnej warstwy bywa w aplikacjach mocno zróżnicowana. Panuje ogólna zgodna, że warstwa ta powinna zawierać implementację reguł biznesowych i wymagań systemowych, które zostały przez użytkowników uznane za dziedzinę rozwiązywanego problemu. W pewnych systemach warstwa ta stosuje własną, wewnętrzną reprezentację elementów dziedziny biznesowej. W innych wykorzystuje model definiowany przez warstwę trwałości. Tym tematem zajmiemy się w rozdziale 3.
- ◆ **Warstwa trwałości** — ta warstwa stanowi grupę klas i komponentów odpowiedzialnych za zapamiętywanie danych i odczytywanie ich z wielu źródeł. Warstwa koniecznie zawiera model elementów dziedziny biznesowej, nawet jeśli są to tylko metadane.

- ◆ **Baza danych** — baza danych istnieje poza aplikacją Javy. Stanowi rzeczywistą, trwałą reprezentację stanu systemu. W przypadku korzystania z bazy danych SQL poza danymi zawiera ona schemat bazy i ewentualnie procedury zapamiętane.
- ◆ **Klasy pomocnicze i narzędziowe** — każda aplikacja wykorzystuje zestaw klas narzędziowych i pomocniczych stosowanych we wszystkich warstwach (na przykład klasę `Exception` do obsługi wyjątków). Elementy infrastruktury nie kreują nowej warstwy, gdyż nie spełniają zasad zależności międzywarstwowej.

Przyjrzymy się różnym sposobom implementacji warstwy trwałości w aplikacjach Javy. Wkrótce dojdziemy również do ORM i Hibernate. Wiele można się dowiedzieć, przeglądając wcześniej rozwiązania alternatywne.

1.3.2. Ręczne tworzenie warstwy trwałości za pomocą SQL i JDBC

Programiści aplikacji bardzo często w celu zapewnienia trwałości danym w Javie decydują się na bezpośrednią pracę z SQL i JDBC. Wynika to z faktu, iż znają relacyjne systemy bazodanowe oraz język SQL i wiedzą, jak korzystać z tabel i kluczy obcych. Co więcej, w tym podejściu zawsze mogą skorzystać z dobrze znanego i powszechnie stosowanego wzorca projektowego DAO w celu ukrycia złożoności kodu JDBC oraz z nieprzenośnego kodu SQL w celu wykonania logiki biznesowej.

Wzorzec DAO jest dobry — nawet tak dobry, że polecamy jego stosowanie razem z ORM (patrz rozdział 8.). Nakład pracy potrzebny do ręcznego napisania trwałości dla każdej klasy dziedziny bywa znaczący, szczególnie gdy trzeba obsłużyć kilka dialektów języka SQL. Często się okazuje, że element ten zabiera znaczącą część czasu poświęconego projektowi. Gdy następuje zmiana wymagań, ręcznie zakodowane rozwiązanie na ogół wymaga większych nakładów na konserwację.

Dlaczego więc nie zaimplementować prostego szkieletowego rozwiązania ORM, by spełnić konkretne wymagania danego projektu? Wynik tej pracy najczęściej będzie można zastosować również w innych projektach. Wielu programistów tak właśnie postąpiło. Wiele działających obecnie systemów wykorzystuje obiektowo-relacyjne warstwy prezentacji tworzone przez twórców tych systemów. Nie zalecamy takiego podejścia. Doskonałe rozwiązania już istnieją. Są to nie tylko komercyjne narzędzia (często drogie), ale i projekty typu *open source* z bezpłatnymi licencjami. Jesteśmy pewni, że każdy znajdzie rozwiązanie spełniające jego potrzeby, zarówno biznesowe, jak i techniczne. Jest wielce prawdopodobne, że to rozwiązanie będzie potrafiło wykonać więcej zadań, niejednokrotnie lepiej, od własnego, naprędce tworzonego kodu.

Wykonanie systemu ORM o znacznych możliwościach potrafi zająć wielu programistom całe miesiące. Przykładowo, Hibernate zawiera 43 tysiące wierszy kodu (niektóre jego fragmenty są znacznie trudniejsze od typowego kodu

aplikacji) oraz 12 tysięcy wierszy kodu testów jednostkowych. Wiele osób niepotrzebnie skupia się przy wyborze na mało istotnych szczegółach — z doświadczenia wiedzą o tym doskonale również autorzy! Nawet jeśli istniejące narzędzie nie implementuje w pełni dwóch lub trzech bardziej egzotycznych wymagań, nie oznacza to, że należy od razu tworzyć własne. Dowolny ORM z pewnością obsługuje najbardziej żmudne przypadki, czyli te, które potrafią znaczco spowolnić pracę programistów. Nic nie szkodzi, gdy pewne specjalne przypadki trzeba będzie napisać ręcznie od podstaw — niewiele aplikacji składa się wyłącznie z przypadków szczególnych!

Nie popadaj w syndrom „tego nie wymyślono tutaj” i nie zaczynaj tworzyć własnego odwzorowania obiektowo-relacyjnego, by tylko uniknąć dodatkowej nauki wiążącej się z poznaniem zewnętrznych narzędzi. Jeśli zdecydujesz, że cały ORM to szaleństwo, i chcesz pracować możliwie blisko bazy danych SQL, istnieją inne szkielety trwałości nieimplementujące w pełni zasad ORM. Przykładem może być warstwa bazy danych iBATIS, która jest bezpłatną warstwą obsługującą pewne żmudne zadania związane z JDBC, nie ograniczając jednocześnie programiście dostępu do poleceń SQL.

1.3.3. Wykorzystanie serializacji

Java ma wbudowany mechanizm trwałości: **serializacja** zapewnia zapis całego grafu obiektów (stanu aplikacji) do strumienia bajtów, który można zapisać trwale na dysku twardym w postaci pliku lub wpisu w bazie danych. Z serializacji korzysta również mechanizm zdalnego wywoływania metod (RMI — *Remote Method Invocation*), by uzyskać semantykę operującą na zwykłym przekazywaniu wartości dla złożonych obiektów. Kolejnym przykładem zastosowań serializacji jest powielenie stanu aplikacji na wielu węzłach klastra.

Dlaczego nie używa się serializacji jako warstwy trwałości? Niestety, ze zserializowanego grafu powiązanych ze sobą obiektów można korzystać tylko jako jednej całości. Nie istnieje sposób, by pobrać ze strumienia dowolne dane bez odbudowywania całej jego zawartości. Oznacza to, że strumień bajtów nie nadaje się do dowolnych operacji wyszukiwania i agregacji. Co więcej, nie można w sposób niezależny uzyskać dostępu ani zaktualizować pojedynczego obiektu lub fragmentu grafu. Ciągłe wczytywanie i zapisywanie całego grafu obiektów w każdej transakcji jest wykluczone w systemach projektowanych do obsługi dużej współprzejności.

Z przedstawionego opisu jednoznacznie wynika, że na obecnym poziomie technologicznym wykorzystanie serializacji jako mechanizmu trwałości w aplikacjach internetowych i biznesowych o dużej współprzejności nie jest możliwe. Nadaje się jedynie jako wygodny mechanizm trwałości dla jednostanowiskowych aplikacji klienckich.

1.3.4. A może ziarenka encyjne EJB?

W ostatnich latach zalecano stosowanie komponentów EJB (*Enterprise JavaBeans*) w celu uzyskania trwałości danych. Jeśli kiedykolwiek pracowałeś nad aplikacjami biznesowymi pisanyymi w Javie, zapewne korzystałeś z EJB, a w szczególności z ziarenek encyjnych. Jeśli nie, nie przejmuj się — ich popularność drastycznie spada, choć większość uwag programistów zostanie uwzględniona w specyfikacji EJB 3.0.

Ziarenka encyjne (określone w specyfikacji EJB 2.1) są interesujące, bo w odróżnieniu od innych wspomnianych rozwiązań zostały wymyślone przez komitet projektowy. Pozostałe rozwiązania (wzorzec DAO, serializacji i ORM) stanowią wynik wieloletniego doświadczenia — reprezentują podejścia, które zostały sprawdzone w praktyce. Nie powinno więc dziwić, że ziarenka encyjne EJB 2.1 w rzeczywistych rozwiązań okazały się prawdziwą katastrofą. Błędy projektowe w specyfikacji uniemożliwiają wydajne działanie ziarenkom encyjnym z trwałością zarządzaną przez ziarenko (BMP — *Bean Managed Persistence*). Lepszym rozwiązaniem jest trwałość zapewniania przez kontener (CMP — *Container Managed Persistence*), bo przynajmniej wyciągnięto tu pewnie wnioski z wcześniejszej specyfikacji EJB 1.1.

Niestety, CMP nie nadaje się do rozwiązyania problemu niedopasowania obiektowo-relacyjnego. Oto sześć powodów:

- ◆ Ziarenka CMP stanowią idealne odwzorowanie tabel z modelem relacyjnego. Są więc zbyt **mało szczegółowe**, bo nie potrafią skorzystać z bogactwa typów danych zapewnianego przez Javę. CMP wymusza, by model dziedzinowy znajdował się w pierwszej postaci normalnej.
- ◆ Z drugiej strony ziarenka CMP są **zbyt szczegółowe**, bo muszą sprostać założeniom EJB — definicji komponentów wielokrotnego użytków. Komponent wielokrotnego stosowania powinien być obiektem o bardzo małej szczegółowości z zewnętrznym interfejsem na tyle stabilnym, by pozostawać stałym po niewielkich zmianach schematu bazy danych. (Tak, właśnie wskazaliśmy, iż ziarenka CMP są jednocześnie zbyt szczegółowe w pewnych miejscach i za mało szczegółowe w innych).
- ◆ Choć EJB może korzystać z dziedziczenia implementacyjnego, ziarenka encyjne nie obsługują asocjacji i zapytań polimorficznych, czyli jednych z głównych cech „rzeczywistych” systemów ORM.
- ◆ Ziarenka encyjne, mimo ich konkretnych zadań określonych w specyfikacji EJB, w praktyce nie są przenośne. Możliwości mechanizmów CMP różnych producentów znacząco się różnią, w szczególności w kwestii odwzorowanie metadanych. Pewne projekty wybrały Hibernate tylko dlatego, że okazał się on bardziej przenośny od ziarenek między różnymi serwerami aplikacji.
- ◆ Ziarenka encyjne nie poddają się serializacji. Okazuje się, że trzeba definiować dodatkowe **obiekty transferu danych** (DTO — *Data Transfer Object*, nazywane również **obiektami wartości**), by przenieść dane do warstwy zdalnego klienta. Użycie szczegółowych wywołań metod

kierowanych od klienta do zdalnego egzemplarza ziarenka encyjnego nie jest skalowalne — to DTO zapewnia wsadowy dostęp do danych po zdalnej stronie. Wzorzec DTO powoduje powiększenie równoległej hierarchii klas, bo encja modelu dziedzinowego jest reprezentowana zarówno przez ziarenko encyjne, jak i DTO.

- ◆ EJB to model mocno narzucający własną wersję — wymusza nienaturalny styl programowania w języku Java i niezwykle utrudnia ponowne wykorzystanie kodu poza konkretnym kontenerem. To ogromna bariera dla programowania opartego na testach. Co więcej, utrudnia działanie aplikacji wymagających przetwarzania wsadowego lub innych działań harmonogramowanych.

Nie będziemy poświęcać więcej czasu na opis zalet i wad ziarenek encyjnych EJB 2.1. Przyglądając się ich możliwościom w kwestii trwałości, łatwo możemy dojść do wniosku, że nie nadają się do pełnego odwzorowywania obiektowego. Choć specyfikacja EJB 3.0 poprawia sytuację, przejdźmy do innych rozwiązań trwałości zasługujących na większą uwagę.

1.3.5. Obiektowe systemy bazodanowe

Ponieważ Java jest językiem w pełni obiektowym, idealnie byłoby, gdyby udało się przechowywać w bazie danych same obiekty bez potrzeby jakiegokolwiek konwersji. W połowie lat 90-tych XX wieku dużym zainteresowaniem zaczęły cieszyć się obiektowe systemy bazodanowe.

System zarządzania obiektową bazą danych przypomina rozszerzenie środowiska aplikacji, a nie zewnętrzne źródło danych. System taki jest w zasadzie implementacją wielowarstwową składającą się z mechanizmu przechowywania danych, bufora obiektów i aplikacji klienckiej. Elementy te są ze sobą ściśle powiązane i komunikują się dzięki własnemu protokołowi sieciowemu.

Tworzenie obiektowych baz danych rozpoczęło się od zdefiniowania w języku macierzystym takich powiązań, by dodać możliwość trwałego zapisywania danych do języka programowania. Dzięki temu obiektowa baza danych doskonale integruje się z aplikacją obiektową. Różni się to od stosowanego obecnie modelu z relacyjnymi bazami danych, w którym to interakcja z bazą danych odbywa się dzięki językowi pośredniemu (SQL).

Nie tylko relacyjne bazy danych posiadają swój ustandaryzowany język ANSI SQL — odpowiednik istnieje również dla obiektowych baz danych. Specyfikacja ODMG (*Object Data Management Group*) definiuje interfejs programistyczny, język zapytań, język metadanych i dowiązania języka macierzystego dla języków C++ SmallTalk i Java. Większość obiektowych systemów bazodanowych implementuje w pewnym stopniu standard ODGM, ale zgodnie z naszą wiedzą nie istnieje jego pełna implementacja. Kilka lat po swoim opublikowaniu (nawet w wersji 3.0) specyfikacja wydaje się niedojrzała. Brakuje w niej wielu użytecznych funkcji, szczególnie w kontekście środowiska języka Java. Grupa ODMG od dłuższego czasu nie jest aktywna. Jakiś czas temu (kwiecień 2002) specyfikacja JDO (*Java Data Object*) otwarła nowe możliwości. JDO powstało dzięki spo-

łeczności związanej z obiektowymi bazami danych. Obecnie stanowi podstawowy interfejs wielu obiektowych baz danych będący alternatywą dla ODMG. W przyszłości okaże się, czy ten ruch na rzecz obiektowych baz danych rozwinię się poza zastosowania CAD/CAM (wspomagane komputerowo projektowanie i modelowanie), obliczenia naukowe i inne wąskie dziedziny.

Nie będziemy się zastanawiać, dlaczego technologia obiektowych baz danych nie stała się bardziej popularna — wystarczy zaobserwować, że obecnie nie cieszą się dużą popularnością i nie wydaje się, by w bliskiej przyszłości miałyby się to zmienić. Mamy pewność, że jeszcze przez długi czas programiści zmuszeni będą korzystać z technologii relacyjnej, choćby z racji amortyzacji poniesionych już przez firmy kosztów związanych z implementacją rozwiązań bazodanowych.

1.3.6. Inne rozwiązania

Oczywiście istnieją inne rodzaje warstw trwałości. Trwałość XML to w zasadzie odmiana trwałości przez serializację — pozwala ominąć pewne ograniczenia serializacji do strumienia bajtów, ułatwiając określonym narzędziom łatwiejszy dostęp do struktury danych (niemniej sama cierpi na niedopasowanie obiektowo-hierarchiczne). XML nie przynosi żadnych innych zalet, gdyż jest po prostu formatem tekstem. Można użyć procedur zapamiętanych (pisząc je chociażby w Javie za pomocą SQLJ), by przenieść problem do warstwy bazy danych. Wydaje się, że liczba podejść jest wprost nieograniczona, ale wszystkie one mają małe szanse stać się dominującym w przyszłości.

Ograniczenia polityczne (związane z długoterminowymi inwestycjami w bazy danych SQL) i potrzeba dostępu do bardzo istotnych danych zapisanych w starszy sposób wymaga innego podejścia. ORM wydaje się najbardziej praktycznym rozwiązaniem przedstawionych wcześniej problemów.

1.4. Odwzorowanie obiektowo-relacyjne

Po przyjrzeniu się alternatywnym technikom uzyskiwania trwałości obiektów możemy wreszcie wprowadzić rozwiązanie, które wydaje się nam najlepsze i jest stosowane przez Hibernate — ORM. Pomimo jego długiej historii (pierwsze publikacje naukowe powstały pod koniec lat 80-tych XX wieku), cały czas stosuje się różne nazewnictwo ORM. Niektórzy piszą mapowanie relacyjno-obiektowe, a inni mówią po prostu **odwzorowanie obiektowe**. W niniejszej książce stosujemy wyłączenie określenie **odwzorowanie obiektowo-relacyjne** i jego angielski skrót **ORM**.

W podrozdziale przyjrzymy się najpierw, czym jest ORM. Następnie wyliczymy problemy, z którymi powinien sobie radzić dobry ORM. Na końcu wspomnimy o ogólnych zaletach tego podejścia i powodach jego zalecania przez nas.

1.4.1. Czym jest ORM?

W dużym skrócie odwzorowanie obiektowo-relacyjne jest automatyczną (i niewidoczną dla użytkownika) trwałością obiektów w aplikacji Javy zamieniającą je na tabele relacyjnej bazy danych na podstawie metadanych opisujących odwzorowanie obiektu na dane. ORM działa jako dwukierunkowy translator danych z jednej reprezentacji w inną.

Oczywiście translacja wpływa na wydajność. Jeśli jednak ORM implementuje się jako oprogramowanie pośredniczące, istnieje wiele okazji do optymalizacji niedostępnych w momencie ręcznego kreowania warstwy trwałości. Dodatkowy nakład czasu programisty trzeba poświęcić na przygotowanie i aktualizację metadanych zarządzających przekształceniami (translacji). Czas poświęcony temu zadaniu jest nieporównywalnie mniejszy od czasu związanego z konserwacją własnego, na stałe zakodowanego podejścia. Nawet obiektowa baza danych ODMG wymaga obszernego opisu w postaci metadanych.

Informacja Czy ORM to przypadkiem nie plugin do programu Visio? ORM może równie dobrze być skrótem od *object role modelling*. Termin ten powstał przed wymyśleniem odwzorowania obiektowo-relacyjnego. Opisuje metodę analizy informacyjnej używaną w trakcie projektowania baz danych. Obecnie jest obsługiwany przez Microsoft Visio — narzędzie do graficznego modelowania. Specjalisci od baz danych stosują je jako zastępstwo lub uzupełnienie bardziej popularnego **modelowania związków encji**. Jeśli w tekstuach dotyczących języka Java mówi się o ORM, najczęściej ma się na myśli odwzorowanie obiektowo-relacyjne.

Rozwiązań ORM składa się z czterech elementów:

- ◆ interfejsu do przeprowadzania podstawowych operacji CRUD na obiektach klas zapewniających trwałość;
- ◆ języka lub interfejsu programistycznego do określania zapytań związanych z klasami lub ich właściwościami;
- ◆ narzędzia do określania metadanych;
- ◆ technik takiej implementacji ORM, by poprawnie współgrała z obiektami transakcyjnymi, wykonując sprawdzenia brudnych danych, leniwe pobieranie asocjacji lub inne funkcje optymalizacyjne.

W zasadzie terminu ORM można użyć dla dowolnej warstwy trwałości, która kod SQL generuje automatycznie na podstawie opisu w postaci metadanych. Nie obejmuje on warstw trwałości, w których problem odwzorowania obiektowo-relacyjnego został rozwiązany ręcznie przez programistę stosującego własny kod SQL i interfejs JDBC. W ORM aplikacja korzysta tylko z interfejsu ORM i klas modelu dziedzinowego — w ogóle nie zajmuje się niskopoziomowymi SQL i JDBC. W zależności od dostępnych funkcji lub konkretnej implementacji system wykonawczy ORM może przejąć odpowiedzialność za takie elementy jak optymistyczne blokowanie i buforowanie. W ten sposób programista aplikacji może skupić się na innych kwestiach.

Przyjrzyjmy się różnym sposobom implementacji ORM. Mark Fussel [Fussel 1997], badacz zagadnień ORM, zdefiniował cztery poziomy jakości ORM.

Pełna relacyjność

Cała aplikacja, włączając w to interfejs użytkownika, jest zaprojektowana wokół modelu relacyjnego i podstawowych operacji języka SQL. To podejście, jeśli pominię się jego wady w dużych systemach, potrafi być doskonałym rozwiązaniem dla prostych aplikacji, w których trudności z ponownym wykorzystaniem niskopoziomowego kodu nie są problemem. Bezpośredni kod SQL można dopasować do własnych potrzeb, ale często brakuje mu przenośności, a konserwacja bywa czasochłonna, szczególnie w dłuższej perspektywie czasu. Aplikacje z tej kategorii często w znacznym stopniu korzystają z procedur zapamiętanych, przekierowując część zadań z warstwy biznesowej do warstwy bazy danych.

Lekkie odwzorowanie obiektów

Encje są reprezentowane przez ręcznie napisane klasy odwzorowujące tabele relacyjne. Wykonany przez programistę kod SQL i JDBC zostaje ukryty przed logiką biznesową dzięki powszechnie znanym wzorcom projektowym. To podejście bywa powszechnie stosowane i osiąga wiele sukcesów w aplikacjach z niewielką liczbą encji lub z aplikacjami stosującymi uogólniony model danych. Nic nie stoi na przeszkodzie, by w tym podejściu stosować procedury zapamiętane.

Średnie odwzorowanie obiektów

Aplikację projektuje się, stosując model obiektowy. Kod SQL powstaje w osobnym etapie przed właściwą komplikacją dzięki specjalnym narzędziom lub też jest generowany dynamicznie przez szkielet systemu. Mechanizm trwałości obsługuje asocjacje między obiektami, natomiast zapytania tworzy się w języku wykorzystującym techniki obiektowe. Warstwa trwałości buforuje obiekty. Większość produktów ORM i tworzonych niezależnie na własne potrzeby warstw trwałości obsługuje przynajmniej te funkcje. Podejście to nadaje się dla średniej wielkości aplikacji z pewnymi złożonymi transakcjami. Okazuje się szczególnie dobre, gdy produkt musi działać z różnymi systemami baz danych. Aplikacje tego typu najczęściej nie stosują procedur zapamiętanych.

Pełne odwzorowanie obiektów

Pełne odwzorowanie obiektowo-relacyjne obsługuje wyrafinowane modele obiektowe stosujące kompozycję, dziedziczenie, polimorfizm i „trwałość przez osiągalność”. Warstwa trwałości w sposób niezauważalny implementuje zapis i odczyt danych. Klasy trwałości nie dziedziczą po żadnych szczególnych klasach bazowych ani nie implementują specjalnych interfejsów. Wydajne strategie pobierania danych (leniwe oraz wstępne) i buforowania działają w sposób niewidoczny dla aplikacji. Ten poziom funkcjonalności trudno osiągnąć warstwom prezentacji tworzonym na własne potrzeby, bo wymaga to miesięcy, a nawet lat poświęconych

na tworzenie kodu. Ten poziom jakości osiągnęło kilka komercyjnych i bezpłatnych systemów ORM dla języka Java. W niniejszej książce zajmujemy się wszystkimi wymienionymi w akapicie aspektami. Przyjrzyjmy się problemom, które mają zostać rozwiązane przez narzędzie dostarczające pełne odwzorowanie obiektów.

1.4.2. Ogólne problemy ORM

Poniższa lista problemów nazwanych przez nas **problemami odwzorowania obiektowo-relacyjnego** zostaje rozwiązana przez narzędzia pełnego odwzorowania obiektów dostępne dla języka Java. Poszczególne systemy ORM mogą oferować dodatkową funkcjonalność (na przykład buforowanie agresywne), ale podana lista wyczerpuje podstawowe zagadnienia specyficzne dla odwzorowania obiektowo-relacyjnego.

- 1. Jak wyglądają klasy trwałości?** Czy są szczegółowe jak JavaBeans? Czy raczej stanowią egzemplarze mniej szczegółowego modelu komponentowego jakim jest EJB? Na ile niewidoczny dla aplikacji jest system trwałości? Czy trzeba dostosować do niego sposób programowania i klasy dziedziny biznesowej?
- 2. Jak definiuje się metadane? Ponieważ przekształcanie obiektowo-relacyjne polega jedynie na metadanych, ich format i definicja to niezwykle istotna kwestia. Czy narzędzie ORM powinno zapewniać graficzny interfejs użytkownika do generowania metadanych? A może istnieją inne, lepsze podejścia do definiowania metadanych?**
- 3. W jaki sposób odwzorowuje się hierarchię dziedziczenia?** Istnieje kilka podstawowych podejść. Co z asocjacjami polimorficznymi, klasami abstrakcyjnymi i interfejsami?
- 4. W jaki sposób identyczność i równość obiektu wiąże się z tożsamością (kluczem głównym) w bazie danych?** W jaki sposób odwzorować egzemplarze konkretnych klas na wiersze tabeli bazy danych?
- 5. Jak logika trwałości współgra w trakcie działania aplikacji z obiekta**mi dziedziny biznesowej? Jest to problem programowania szablonowego. Istnieje wiele rozwiązań: generowanie kodu, refleksja w trakcie wykonywania aplikacji, generowanie kodu bajtowego w czasie rzeczywistym i wz bogacanie kodu bajtowego po komplikacji. Rozwiązanie tej kwestii wpływa na proces budowania wykonywalnej aplikacji, ale najczęściej nie ma znaczenia dla końcowego użytkownika.
- 6. Jaki jest cykl życia obiektu trwałego?** Czy cykl życia niektórych obiektów zależy od życia innych, powiązanych z nim obiektów? W jaki sposób przekształcić cykl życia obiektu w cykl życia wiersza bazy danych?
- 7. Jakkie są możliwości w kwestiach wyszukiwania, sortowania i agregacji?** Aplikacja powinna niektóre z tych zadań wykonywać w pamięci. Z drugiej strony, wydajne korzystanie z technologii relacyjnej wymaga przeniesienia niektórych działań na bazę danych.

8. W jaki sposób wydajnie pobierać dane z asocjacjami? Wydajny dostęp do relacyjnych danych najczęściej odbywa się dzięki złączeniom tabel. Aplikacje obiektowe wykorzystują raczej nawigację po grafie obiektów. Należy unikać dwóch wzorców dostępu do danych: problemu **n+1** pobrań i jego przeciwnieństwa, czyli problemu **iloczynu kartezjańskiego** (pobierania zbyt wielu danych).

Dwie dodatkowe kwestie dotyczą dowolnej technologii dostępu do danych. W znaczący sposób wpływają i ograniczają architekturę systemów ORM:

- ◆ transakcje i współbieżność,
- ◆ zarządzanie buforowaniem (i współbieżnością).

Jak nietrudno zauważyc, pełny system odwzorowania obiektów musi rozwiązywać bardzo długą listę problemów. Sposób obsługi tych zagadnień przez Hibernate oraz sam dostęp do danych omawiamy w rozdziałach od 3. do 5. Rozwijamy go również w rozdziałach kolejnych.

Już w tym miejscu powinny ukazywać się zalety systemów ORM. W kolejnym podrozdziale przyjrzymy się kilku innym zaletom stosowania ORM.

1.4.3. Dlaczego ORM?

Implementacja ORM jest wyjątkowo złożona — mniej złożona niż serwer aplikacji, ale bardziej niż szkielety aplikacji internetowych typu Struts lub Tapestry. Dlaczego mielibyśmy wprowadzać do aplikacji nowy element? Czy rzeczywiście przyniesie to korzyści?

Większość niniejszej książki zajmie pełna odpowiedź na te pytania. Ten podrozdział kierujemy dla osób niecierpliwych, gdyż w bardzo zwięzły sposób podsumowuje większość zalet. Zanim przejdziemy do zalet, wspomnijmy o pewnej istotnej kwestii.

Zakłada się, iż podstawową zaletą ORM jest „ochrona” programisty przed „czasołonnym” SQL-em. Taki punkt widzenia może sugerować, że programiści języków obiektowych niekoniecznie muszą dobrze rozumieć SQL i relacyjne bazy danych (bo na przykład jest to przestarzałe podejście). My jednak wierzymy, że każdy programista języka Java powinien przynajmniej w pewnym stopniu znać model relacyjny i język SQL, by wydajnie pracować z ORM. ORM to zaawansowana technologia, która powinna być stosowana przez programistów doświadczonych w budowaniu własnych połączeń bazodanowych. Aby wydajnie korzystać z Hibernate, trzeba umieć przejrzeć i zrozumieć instrukcje SQL, które generuje i wysyła do bazy danych. Tylko tak zrozumie się ich wpływ na wydajność systemu.

Przyjrzymy się zaletom ORM i Hibernate.

Produktywność

Kod dotyczący trwałości to w aplikacjach Javy chyba najbardziej niewdzięczny element programowania. Hibernate eliminuje większość żmudnych zadań (więcej niż można się spodziewać) i pozwala skoncentrować się na problemie biznesowym.

Niezależnie do preferowanej strategii rozwoju aplikacji: z góry na dół, od modelu dziedzinowego, z dołu do góry, od istniejącego schematu bazy danych — Hibernate w połączeniu z odpowiednimi narzędziami pozwala znaczaco zredukować czas poświęcony na tworzenie kodu.

Konserwacja

Mniej wierszy kodu czyni system bardziej zrozumiałym, bo dzięki temu uwidacznia się logika biznesowa, a nie zbędne „zapychacze”. Co ważniejsze, im kod krótszy, tym łatwiej go przekształcić. Automatyczna trwałość obiektowo-relacyjna znaczaco zmniejsza liczbę wierszy kodu. Oczywiście liczenie wierszy to dyskusyjny sposób określania złożoności aplikacji.

Istnieją też inne powody, dzięki którym aplikację stosującą Hibernate łatwiej się konserwuje. W systemach z ręcznie pisana trwałością istnieje ogromne tarcie między istniejącą reprezentacją relacyjną i modelem obiektowym implementującym dziedzinę biznesową. Zmiany w jednym niemal na pewno wiążą się ze zmianami w drugim. Często jedna z reprezentacji stanowi kompromis pozwalający korzystać z drugiej części (w zdecydowanej większości przypadków to **model obiektowy** dostosowuje się do ograniczeń modelu relacyjnego). ORM stanowi strefę buforową między dwoma modelami, co pozwala zastosować bardziej eleganckie rozwiązanie obiektowe po stronie Javy i dostosować się do drobnych zmian po drugiej stronie.

Wydajność

Powszechnie uważa się, że ręcznie napisana trwałość zawsze potrafi być co najmniej tak szybka, a niejednokrotnie znacznie szybsza, od trwałości automatycznej. Jest to stwierdzenie prawdziwe w takim samym sensie, że kod w języku assemblera zawsze może być co najmniej tak szybki jak kod Javy lub że kod analizatora składniowego napisany ręcznie bywa szybszy od analizatora wygenerowanego narzędziami YACC lub ANTLR. Innymi słowy, jest to kwestia niepodlegająca dyskusji. Po cichu zakłada się, że ręcznie napisana trwałość rzeczywiście będzie działać równie dobrze w rzeczywistej aplikacji. Założenie będzie prawdziwe tylko wtedy, gdy wysiłek poświęcony na ręczne napisanie trwałości będzie przynajmniej podobny do wysiłku związanego wykorzystaniem podejścia automatycznego. Najbardziej interesujący nie jest sam czas działania, ale raczej końcowy efekt uwzględniający również ramy czasowe i budżet.

Zadanie trwałości danych pozwala zastosować wiele optymalizacji. Niektóre (na przykład podpowiedź zapytań) znacznie łatwiej uzyskać, samemu pisząc kod SQL i JDBC. Pozostałe prościej wykonać globalnie w automatycznym ORM. W projekcie o ścisłe określonym czasie zakończenia ręcznie tworzona trwałość umożliwia przeprowadzenie pewnych optymalizacji w niektórych miejscach. Hibernate pozwala korzystać ze znacznie większej liczby optymalizacji przez cały czas. Co więcej, automatyczna trwałość na tyle zwiększa produktywność, że pozostaje jeszcze czas na dokonanie kilku najbardziej istotnych poprawek optymalizacyjnych.

Osoby implementujące system ORM mają znacznie więcej czasu na sprawdzenie wszystkich możliwych ścieżek optymalizacji niż osoba zajmująca się wykonaniem własnego rozwiązania. Czy wiesz, że utworzenie puli egzemplarzy PreparedStatement znacząco zwiększa wydajność sterownika JDBC dla DB2, ale całkowicie nie nadaje się do użycia ze sterownikiem dla InterBase? Czy wiesz, że aktualizacja jedynie zmienionych kolumn bywa szybsza w jednych bazach danych, a potencjalnie wolniejsza w innych? Na jak dużo testów można sobie pozwolić, kreując własne rozwiązanie trwałości danych?

Niezależność od dostawcy

Abstrakcja ORM uwalnia od szczegółów języka SQL i różnych jego odmian w poszczególnych systemach bazodanowych. Jeśli narzędzie obsługuje różne bazy danych (większość tak czyni), uzyskuje się znacznie większą przenośność niż w przypadku własnego rozwiązania. Niekoniecznie jest to od razu przenośność typu „napisz raz, uruchamiaj wszędzie”, bo to podejście wymagałoby poświęcenia wielu zalet niektórych rozwiązań szczegółowych. Niemniej wykonanie wieloplatformowej aplikacji jest znacznie prostsze, gdy używa się ORM. Nawet jeśli nie wymaga się obsługi wielu platform, ORM pomaga uniknąć uwięzania na stałe do jednego dostawcy. Co więcej, niezależność od bazy danych pomaga wykorzystywać podejścia, w których programiści używają lekkich, lokalnych baz danych, ale wdrażają system działający na innej, bardziej złożonej bazie danych.

1.5. Podsumowanie

W niniejszym rozdziale zajęliśmy się koncepcją trwałości obiektów i istotnością ORM jako techniki jej implementacji. Trwałość obiektów oznacza, że czas życia niektórych obiektów jest dłuższy od czasu życia procesu aplikacji. Obiekt zapisuje się, a następnie przywraca w momencie ponownego uruchamiania aplikacji. Niedopasowanie obiektowo-relacyjne przeszkadza w wykonaniu trwałości, gdy dane przechowuje się w relacyjnej bazie danych. Grafu obiektów nie da się tak po prostu zapisać w tabeli bazy danych — musi on zostać rozłożony na części i zamieniony na kolumny dotyczące przenośnych typów danych SQL. Dobrym rozwiązaniem niedopasowania jest system ORM. Staje się szczególnie cenny w modelach dziedzinowych stosujących bogactwo typów danych Javy.

Model dziedzinowy reprezentuje byty biznesowe stosowane w aplikacji Javy. W architekturze warstwowej model dziedzinowy służy do wykonywania logiki biznesowej na warstwie biznesowej (czyli w Javie zamiast w bazie danych). Warstwa biznesowa komunikuje się z niższą warstwą trwałości, by wczytać lub zapisać trwałe obiekty modelu dziedzinowego. ORM to oprogramowanie pośredniczące znajdujące się na warstwie prezentacji i zarządzające trwałością.

ORM nie jest złotym środkiem na wszystkie zadania trwałości. Jego zadaniem jest odciążenie programisty w 95% zadań związanych z trwałością obiektów, na przykład tworzeniem złożonych instrukcji SQL z wieloma złączami tabel i kopowania wartości ze zbiorów wyników JDBC do odpowiednich obiektów.

Rozbudowane oprogramowanie ORM zapewnia przenośność między różnymi bazami danych, techniki optymalizacyjne jak buforowanie i inne istotne funkcje, które niełatwo wykonać ręcznie w SQL i JDBC przy ograniczonym czasie i środkach.

Wydaje się, że pewnego dnia powstanie rozwiązań lepszych niż ORM. Zarówno my, jak i wiele innych osób, musi dobrze przemyśleć wszystkie sprawy związane z SQL, standardami interfejsów trwałości danych i integracją baz danych z aplikacjami. Sposobu ewolucji dzisiejszych systemów w rzeczywiście relacyjne systemy bazodanowe z doskonale zintegrowaną obsługą pozostaje się tylko domyślać. Większość projektów nie może czekać, a nie wydaje się, by sytuacja uległa natychmiastowej poprawie (wielomiliardowy przemysł jest skostniały i niesklonny do szybkich zmian). ORM to najlepsze dostępne obecnie rozwiązanie. Pozwala wielu programistom zaoszczędzić mnóstwo czasu związanego z niedopasowaniem obiektowo-relacyjnym.

Wprowadzenie i integracja Hibernate

W rozdziale:

- ◆ Działanie Hibernate na przykładzie aplikacji „Witaj świecie”
- ◆ Podstawowe interfejsy programistyczne Hibernate
- ◆ Integracja z zarządzanymi i niezarządzanymi środowiskami
- ◆ Zaawansowane opcje konfiguracyjne

Dobrze zdawać sobie sprawę z potrzeby odwzorowania obiektowo-relacyjnego w aplikacjach Javy, ale jeszcze lepiej zobaczyć takie odwzorowanie w akcji. Zaczniemy od prostego przykładu obrazującego zalety Hibernate.

Wiele osób wie, że niemal każda książka o programowaniu rozpoczyna się od przykładu „Witaj świecie”. Nie zamierzamy się wyłamywać z tej konwencji i również przedstawimy prostą aplikację tego typu używającą Hibernate. Niestety, proste wyświetlenie komunikatu tekstowego na konsoli nie uwidoczniłoby żadnych zalet Hibernate. Z tego powodu tworzony program będzie tworzył nowe obiekty w bazie danych, aktualizował je i wykonywał zapytania, by je pobrać.

Niniejszy rozdział stanowi podstawę dla pozostałych rozdziałów. Poza przedstawieniem typowego dla książek informatycznych przykładu wprowadzimy również interfejs programistyczny Hibernate i omówimy konfigurację systemu w różnych środowiskach wykonawczych, na przykład serwerach aplikacji J2EE lub aplikacjach klienckich.

2.1. „Witaj świecie” w stylu Hibernate

Aplikacje Hibernate definiują klasy trwałe, które są odwzorowywane na tabele bazy danych. Pierwszy przykład zawiera jedną klasę i jeden plik opisu odwzorowania. Przyjrzyjmy się, jak wygląda prosta klasa trwałości, jak określa się odwzorowanie i jak wykonuje się podstawowe operacje dotyczące trwałości danych.

Celem pierwszej aplikacji jest zapamiętywanie w bazie danych komunikatów, a następnie ich pobieranie w celu wyświetlania. Aplikacja stosuje prostą klasę trwałości, Message, reprezentującą komunikaty do wyświetlenia. Klasę przedstawia listing 2.1.

Listing 2.1. Plik Message.java - prosta klasa trwałości

```
package hello;
public class Message {
    private Long id;           ← Atrybut identyfikatora
    private String text;        ← Treść komunikatu
    private Message nextMessage; ← Referencja do innego komunikatu
    private Message() {}
    public Message(String text) {
        this.text = text;
    }
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    public String getText() {
        return text;
    }
    private void setText(String text) {
        this.text= text;
    }
}
```

```
public Long getNextMessage() {  
    return nextMessage;  
}  
private void setnNextMessage(Long nextMessage) {  
    this.nextMessage= nextMessage;  
}
```

Klasa Message ma trzy atrybuty: identyfikator, treść komunikatu i referencję do innego komunikatu. Atrybut identyfikacyjny umożliwia aplikacji dostęp do identyfikatora bazodanowego — wartości klucza głównego — obiektu trwałego. Jeśli dwa egzemplarze klasy Message zawierają ten sam identyfikator, dotyczą tego samego wiersza w tabeli bazy danych. Jako typ identyfikatora wybraliśmy Long, choć nie jest to wymóg. Hibernate dopuszcza stosowane dla identyfikatorów dowolnych typów, co wkrótce przedstawimy.

Wszystkie atrybuty klasy Message stosują metody dostępowe do właściwości w stylu ziarenek JavaBeans. Klasa zawiera także bezparametrywy konstruktor. Klasy trwałości stosowane w kolejnych przykładach wyglądają bardzo podobnie do przedstawionej.

Egzemplarze klasy Message mogą być zarządzane (w sensie zapewnienia trwałości) przez Hibernate, ale nie jest to **przymus**. Ponieważ obiekt Message nie implementuje żadnych klas i interfejsów specyficznych dla Hibernate, można go używać jak dowolnej innej klasy Javy.

```
Message message = new Message("Witaj świectie");  
System.out.println(message.getText());
```

Przedstawiony fragment wykonuje właśnie to zadanie, którego oczekuje się po aplikacji „Witaj świectie” — wyświetla na konsoli napis Witaj świectie. Niektórym wydaje się zapewne, że w tym momencie jesteśmy wyjątkowo uprzejmi — w rzeczywistości ten prosty przykład demonstruje istotną cechę wyróżniającą Hibernante od innych rozwiązań trwałości, na przykład ziarenek encyjnych EJB. Klasę trwałości można stosować w dowolnym kontekście, bo nie jest konieczny żaden szczególny kontener. Ponieważ książka dotyczy Hibernate, zapiszmy obiekt Message do bazy danych.

```
Session session = getSessionFactory().openSession();  
Transaction tx = session.beginTransaction();  
Message message = new Message("Witaj świectie");  
session.save(message);  
tx.commit();  
session.close();
```

Przedstawiony kod korzysta z interfejsów Session i Transaction systemu Hibernate (wkrótce zajmiemy się wywołaniem getSessionFactory()). Przedstawiony kod skutkuje przekazaniem do bazy danych polecenia SQL podobnego do następującego:

```
insert into MESSAGE (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID) values  
(1, 'Witaj świectie', null)
```

Chwileczkę — kolumna MESSAGE_ID zostaje zainicjalizowana dziwną wartością. Nigdzie nie ustawialiśmy w kodzie właściwości id, więc oczekujemy dla niej wartości null, prawda? W rzeczywistości właściwość id jest szczególna — jest właściwością identyfikującą, która przechowuje wygenerowaną, unikatową wartość (w dalszej części tekstu wyjaśnimy sposób generowania wartości). Wartość zostaje wprowadzona do obiektu Message w momencie wykonania metody save().

W przykładzie zakładamy wcześniejsze istnienie tabeli MESSAGE. W rozdziale 9. omówimy wykorzystanie Hibernate do automatycznego tworzenia tabel wymaganych przez aplikację dzięki informacjom zawartym w plikach odwzorowań (w ten sposób unikamy pisania jeszcze większej ilości kodu SQL). Chcemy, by program wyświetlił komunikat na konsoli. Skoro komunikat znalazł się w bazie danych, nie powinno to sprawić żadnych trudności. Kolejny przykład pobiera z bazy danych wszystkie komunikaty w kolejności alfabetycznej i wyświetla ich zawartość.

```
Session newSession = getSessionFactory().openSession();
Transaction newTransaction = newSession.beginTransaction();
List messages = newSession.find("from Message as m order by m.text asc");
System.out.println("Znalezionych komunikatów: " + messages.size());
for (Iterator iter = messages.iterator(); iter.hasNext();) {
    Message message = (Message) iter.next();
    System.out.println(message.getText());
}
newTransaction.commit();
newSession.close();
```

Ciąg znaków "from Message as m order by m.text asc" to zapytanie Hibernate zapisane w specjalnym, obiektowym języku zapytań Hibernate (HQL — *Hibernate Query Language*). Zapytanie zostaje wewnętrznie przekształcone na następujący kod SQL w momencie wywołania metody find().

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
order by m.MESSAGE_TEXT asc
```

Przedstawiony kod powoduje wyświetlenie na konsoli poniższego tekstu:

```
Znalezionych komunikatów: 1
Witaj Świecie
```

Jeśli ktoś nigdy wcześniej nie korzystał z narzędzi ORM, spodziewa się ujrzeć kod SQL w metadanych lub też kodzie Javy. Nie ma go tam. Cały kod SQL zostaje wygenerowany dynamicznie w trakcie działania aplikacji (ewentualnie w momencie jej uruchamiania dla wszystkich instrukcji SQL wielokrotnego użytku).

Aby wystąpiła cała magia, Hibernate potrzebuje więcej informacji na temat zapewniania trwałości klasie Message. Informację tę najczęściej umieszcza się **dokumencie odwzorowania** zapisywany w formacie XML. Dokument ten definiuje, poza innymi kwestiami, w jaki sposób przełożyć właściwości klasy Message na kolumny tabeli MESSAGES. Przyjrzyjmy się fragmentowi dokumentu odwzorowania — listing 2.2.

Listing 2.2. Prosty plik odwzorowania Hibernate w formacie XML

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN" <
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
    <class
        name="hello.Message"
        table="MESSAGES">
        <id
            name="id"
            column="MESSAGE_ID">
            <generator class="increment"/>
        </id>
        <property
            name="text"
            column="MESSAGE_TEXT"/>
        <many-to-one
            name="nextMessage"
            cascade="all"
            column="NEXT_MESSAGE_ID"/>
    </class>
</hibernate-mapping>
```

Warto zauważyć,
że Hibernate 2.0
i 2.1 stosują ten sam
schemat DTD!

Dokument odwzorowania informuje Hibernate, że klasa Message przekłada się na tabelę MESSAGES. Informuje również, że właściwość identyfikatora ma trafić do kolumny nazwanej MESSAGE_ID, właściwość komunikatu do kolumny o nazwie MESSAGE_TEXT, natomiast właściwość nextMessage jest asocjacją z **krotnością wielo-dojrzałego** i powinna trafić do kolumny o nazwie NEXT_MESSAGE_ID (pozostałymi szczegółami na tym etapie nie będziemy się zajmować).

Dokument XML nie jest trudny do zrozumienia. Można go bez problemów ręcznie tworzyć i modyfikować. W rozdziale 3. omówimy sposób generowania tego pliku na podstawie komentarzy zawartych w kodzie źródłowym aplikacji. Niezależnie od wybranego podejścia Hiberante uzyskuje wystarczająco dużo informacji, by poprawnie generować wszystkie polecenia SQL dotyczące wstawiania, aktualizacji, usuwania i pobierania egzemplarzy klasy Message. Nie trzeba ręcznie pisać wszystkich wymaganych poleceń SQL.

Uwaga

Wielu programistów Javy narzeka na tak zwane „piekło metadanych”, które towarzyszy tworzeniu oprogramowania J2EE. Niektórzy sugerują nawet rezygnację z metadanych XML i powrót do zwykłego kodu Javy. Choć w pełni popieramy te dążenia w niektórych dziedzinach, ORM wydaje się być zagadnieniem, w którym metadane naprawdę są potrzebne. Hibernate stosuje sensowne wartości domyślne, by zminimalizować długosć plików konfiguracyjnych. Wykorzystuje dojrzałą definicję typu dokumentu (DTD), więc możliwe jest wykorzystanie edytorów z uzupełnianiem składni i walidacją. Istnieją nawet narzędzia zapewniające automatyczne generowanie metadanych.

Zmieńmy pierwszy komunikat i dodatkowo utwórzmy kolejny powiązany z pierwszym. Kod wykonujący to zadanie przedstawia listing 2.3.

Listing 2.3. Aktualizacja komunikatu

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();

// 1 to identyfikator wygenerowany dla pierwszego komunikatu
Message message = (Message) session.load(Message.class, new Long(1));
message.setText("Witajcie Ziemanie");
Message nextMessage = new Message("Proszę zabrać mnie do waszego przywódcy");
message.setNextMessage(nextMessage);
tx.commit();
session.close();
```

Powyższy kod powoduje wykonanie w jednej transakcji trzech poleceń SQL.

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
where m.MESSAGE_ID = 1

insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
values (2, 'Proszę zabrać mnie do waszego przywódcy', null)

update MESSAGES
set MESSAGE_TEXT = 'Witajcie Ziemanie', NEXT_MESSAGE_ID = 2
where MESSAGE_ID = 1
```

Zauważ, że Hibernate automatycznie rozpoznał modyfikację właściwości `text` i `nextMessage` dla pierwszego komunikatu i zapisał nowe wartości w bazie danych. Skorzystaliśmy z cechy Hibernate nazywanej **automatycznym wykrywaniem zabrudzenia**. Pozwala ona uniknąć jawnego proszenia Hibernate o aktualizację bazy danych, gdy zmienia się stan obiektu wewnątrz transakcji. Warto zwrócić uwagę, iż nowy komunikat również stał się trwałym, gdy tylko powstała referencja do niego z pierwszego komunikatu. Jest to tak zwany zapis kaskadowy — unika się w ten sposób jawnego wywoływanego metody `save()` dla nowego obiektu trwałego, o ile tylko jest on osiągalny z poziomu innego obiektu trwałego. Kolejność wykonania instrukcji SQL nie jest taka sama jak kolejność modyfikacji wartości właściwości. Hibernate używa wyrafinowanego algorytmu do określenia wydajnej kolejności zapisów, by uniknąć zerwania ograniczeń klucza obcego i jednocześnie pozostać odpowiednio przewidywalnym. Jest to tak zwany **transakcyjny zapis opóźniony**.

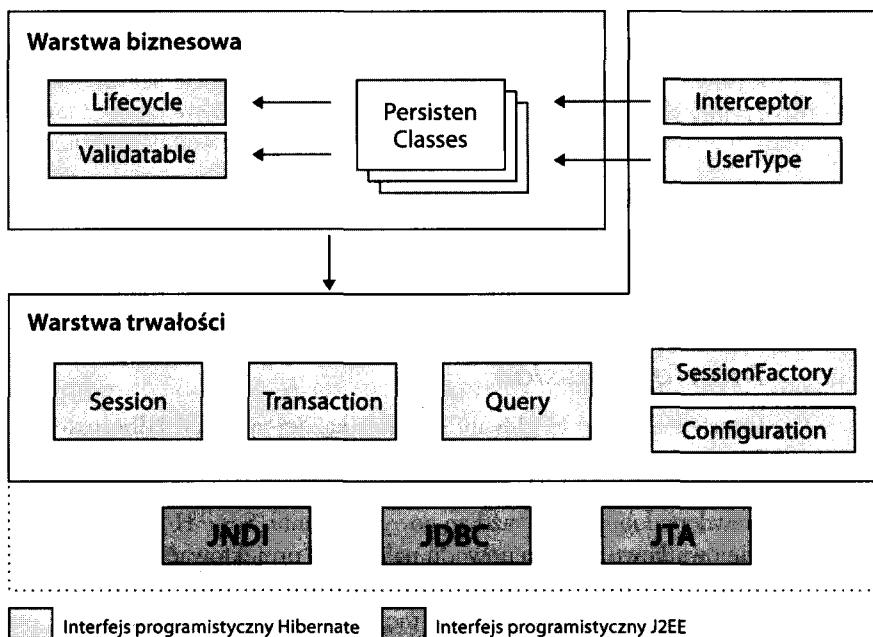
Ponownie uruchomienie programu spowoduje wyświetlenie poniższego wyniku:

```
Znalezionych komunikatów: 2
Witajcie Ziemanie
Proszę zabrać mnie do waszego przywódcy
```

Na tym zakończymy przygodę z aplikacją „Witaj świecie”. Gdy poznaliśmy działający kod, możemy się cofnąć i dokładniej omówić poszczególne elementy głównego interfejsu programistycznego Hibernate.

2.2. Podstawy architektury Hibernate

Interfejs programistyczny to pierwszy element Hibernate, który trzeba poznać, by móc skorzystać z warstwy trwałości w aplikacji. Głównym celem projektowym interfejsu była minimalizacja współzależności między poszczególnymi komponentami oprogramowania. W praktyce interfejs programistyczny ORM nie należy do najmniejszych. Na szczęście nie wszystkie interfejsy trzeba poznać od razu, by korzystać z Hibernate. Rysunek 2.1 przedstawia zadania najważniejszych interfejsów w warstwie biznesowej i trwałości. Na rysunku warstwa biznesowa znajduje się powyżej warstwy trwałości, gdyż w typowych aplikacjach warstwowych działa jako klient warstwy trwałości. Niektóre proste aplikacje niezbyt dobrze separamają logikę biznesową od trwałości — nie będziemy się tym jednak zajmować, bo zalecane podejście przedstawia rysunek.



Rysunek 2.1. Wysokopoziomowy przegląd interfejsów programistycznych Hibernate z uwzględnieniem architektury warstwowej

Interfejsy Hibernate przedstawione na rysunku 2.1 można podzielić następująco:

- ◆ Interfejsy wywoływanie przez aplikacje w celu wykonania prostych operacji CRUD i wyszukiwania. Interfejsy te stanowią główny punkt styku logiki biznesowej z Hibernate. Zawierają interfejsy: Session, Transaction i Query.
- ◆ Interfejsy wywoływanie przez kod infrastruktury aplikacji w celu konfiguracji Hibernate. Jest to przede wszystkim klasa Configuration.
- ◆ Interfejsy wywołań zwrotnych pozwalające aplikacji reagować na zdarzenia zachodzące wewnętrzHibernate. Są to interfejsy: Interceptor, Lifecycle i Validatable.
- ◆ Interfejsy zapewniające rozszerzanie rozbudowanej funkcjonalności odwzorowania w Hibernate. Są to interfejsy: UserType, CompositeUserType i IdentifierGenerator. Interfejsy te implementuje kod infrastruktury aplikacji (jeśli to konieczne).

Hibernate wykorzystuje istniejące interfejsy programistyczne Javy, włączając w to JDBC, JTA (*Java Transaction API*) oraz JNDI (*Java Naming and Directory Interface*). JDBC zapewnia odpowiedni poziom abstrakcji funkcji wspólnych dla różnych systemów relacyjnych baz danych, co oznacza, że Hibernate potrafi skorzystać z dowolnej bazy danych ze sterownikiem JDBC. JNDI i JTA umożliwiają Hibernate integrację z serwerami aplikacji J2EE.

W niniejszym podrozdziale nie będziemy omawiać szczegółowej semantyki metod interfejsów Hibernate. Zajmiemy się raczej podstawowymi rolami interfejsów. Większość interfejsów znajduje się w pakiecie net.sf.hibernate. Przyjrzyjmy się pokrótko każdemu z nich.

2.2.1. **Interfejsy podstawowe**

Pięć interfejsów podstawowych pojawia się w niemal każdej aplikacji wykorzystującej Hibernate. Dzięki nim możliwe staje się pobieranie i zapamiętywanie trwałych obiektów oraz sterowanie transakcjami.

Interfejs Session

Interfejs Session to główny interfejs każdej aplikacji Hibernate. Egzemplarze Session są lekkie — koszt ich utworzenia i zniszczenia bywa niewielki. Jest to niezwykle istotne, bo każda aplikacja cały czas tworzy i usuwa coraz to nowe sesje, prawdopodobnie przy każdym nowym żądaniu. Sesje Hibernate nie są bezpieczne wielowątkowo i z tego powodu należy tak projektować aplikację, by używać ich w danym momencie w tylko jednym wątku.

Znaczenie sesji Hibernate można opisać jako coś pomiędzy połączeniem i transakcją. Najprościej traktować sesję jako bufor lub kolekcję załadowanych obiektów powiązanych z jedną jednostką zadaniową. Hibernate potrafi wykryć zmiany w obiektach tej jednostki zadaniowej. Często interfejs Session nazywa się zarządcą trwałości, gdyż zapewnia dostęp do podstawowych operacji trwałości takich jak zapis i pobieranie obiektów. Warto podkreślić, że sesja Hibernate nie ma nic wspólnego z sesją warstwy połączenia internetowego, na przykład

z HttpSession. Gdy w książce używamy słowa sesja, mamy na myśli sesję Hibernate. Gdy mówimy o sesji użytkownika, mamy na myśli obiekt HttpSession.

Interfejs Session został dokładnie omówiony w podrozdziale 4.2.

Interfejs SessionFactory

Aplikacja pobiera egzemplarze Session z SessionFactory. W porównaniu z Session obiekt ten jest znacznie bardziej interesujący.

Interfejs SessionFactory pewnością nie jest lekki! Został zaprojektowany z myślą o współdzieleniu przez wiele wątków aplikacji. Najczęściej na całą aplikację występuje tylko jeden obiekt SessionFactory tworzony w momencie jej inicjalizacji. Jeśli jednak aplikacja korzysta z kilku baz danych w połączeniu z Hibernate, potrzeba osobnego obiektu SessionFactory dla każdej bazy danych.

Obiekt SessionFactory buforuje wygenerowane polecenia SQL i inne metadane odwzorowania stosowane przez Hibernate w trakcie działania aplikacji. Dodatkowo buforuje dane, które zostały odczytane w jednej jednostce zadaniowej i mogłyby przydać się w innej jednostce zadaniowej (ten drugi poziom buforowania działa tylko wtedy, gdy odwzorowania klas i kolekcji sugerują jego użycie).

Interfejs Configuration

Obiekt Configuration służy do konfiguracji i uruchomienia Hibernate. Aplikacja używa egzemplarza Configuration do wskazania położenia dokumentów odwzorowań i właściwości specyficznych dla Hibernate. Podrozdział 2.3 ze średnim stopniem złożoności opisuje konfigurację systemu Hibernate.

Interfejs Transaction

Interfejs Transaction jest opcjonalny. Aplikacje Hibernate nie muszą z niego korzystać, jeśli chcą zarządzać transakcjami we własnym zakresie. Interfejs stara się ukryć szczegóły implementacji konkretnych mechanizmów transakcyjnych: JDBC, klasy UserTransaction z JTA lub nawet transakcji CORBA (*Common Object Request Broker Architecture*). W ten sposób zapewnia aplikacji jednorodny sposób ich obsługi. Ułatwia to zachowanie przenośności aplikacji Hibernate między różnymi środowiskami wykonywania i kontenerami.

W niniejszej książce stosujemy interfejs Transaction. Został on dokładnie wyjaśniony w rozdziale 5.

Interfejsy Query i Criteria

Interfejs Query umożliwia wysyłanie zapytań do bazy danych i sterowanie procesem ich wykonywania. Zapytania pisze się w języku HQL lub też w odpowiednim dla danej bazy danych dialekcie SQL. Egzemplarz Query odpowiada za dowiązanie parametrów zapytania, ograniczenie liczby zwracanych wyników i wykonanie zapytania.

Interfejs Criteria jest bardzo podobny. Umożliwia utworzenie i wykonanie obiektowych kryteriów wyszukiwania.

Aby tworzony kod był możliwie krótki, Hibernate dostarcza kilka skrótów w interfejsie Session, by móc w jednym wierszu wykonać zapytanie. W książce nie stosujemy tych skrótów — zawsze używamy interfejsu Query.

Egzemplarz Query jest lekki nie daje się zastosować poza egzemplarzem Session, dla którego powstał. Opis funkcji interfejsu Query znajduje się w rozdziale 7.

2.2.2. Interfejsy wywołań zwrotnych

Interfejsy wywołań zwrotnych umożliwiają aplikacji otrzymywanie powiadomień, gdy coś ciekawego stanie się ze sprawdzanym obiektem — na przykład zostanie załadowany, zapisany lub usunięty. Aplikacje stosujące Hibernate nie muszą korzystać z wywołań zwrotnych, ale niejednokrotnie przydają się one do tworzenia ogólnej funkcjonalności, na przykład automatycznego tworzenia zapisów audytorskich.

Interfejsy Lifecycle i Validatable zapewniają trwałym obiektom reakcję na zdarzenia związane z własnym cyklem życia trwałego. Cykl życia ma nierozerwalny związek z operacjami CRUD obiektu. Zespół projektowy Hibernate był pod silnym wpływem innych rozwiązań ORM posiadających podobne wywołania zwrotne. Później zdano sobie sprawę, że kasy trwałości implementujące interfejsy specyficzne dla Hibernate nie są najlepszym pomysłem, gdyż ich stosowanie utrudnia przenośność kodu. Ponieważ ich używanie obecnie nie jest zalecane, nie są one szerzej opisywane w niniejszej książce.

Interfejs Interceptor wprowadzono, by aplikacje mogły przetwarzać wywołania zwrotne bez potrzeby wprowadzania implementacji interfejsów Hibernate do klas trwałości. Implementacje interfejsu Interceptor wprowadza się do egzemplarzy trwałych klas jako parametry. Przykład ich użycia pojawi się w rozdziale 8.

2.2.3. Typy

Podstawowym i niezwykle użytecznym elementem architektury jest sposób wykonania obiektu Type w Hibernate. Obiekt ten odwzorowuje typ Java na typ kolumny tabeli bazy danych (w rzeczywistości typ może dotyczyć wielu kolumn). Wszystkie trwałe właściwości klas trwałych, włączając w to asocjacje, posiadają odpowiedni typ Hibernate. Ten sposób obsługi czyni Hibernate wyjątkowo elastycznym i rozszerzalnym.

Istnieje bogaty zbiór typów wbudowanych zawierający wszystkie typy podstawowej Java oraz wiele klas ze standardowej biblioteki JDK, włączając w to typy: `java.util.Currency`, `java.util.Calendar`, `byte[]` i `java.io.Serializable`.

Co więcej, Hibernate obsługuje typy zdefiniowane przez użytkownika. Interfejsy UserType i CompositeUserType dają możliwość kreowania własnych typów. W ten sposób klasy wykorzystywane w wielu różnych aplikacjach, na przykład Address, Name i MonetaryAmount udaje się obsłużyć wygodnie i elegancko. Własne typy uważa się za jedną z najistotniejszych cech Hibernate. Zalecamy z nich korzystać w coraz to nowy sposób!

Szczegóły dotyczące typów Hibernate zostały omówione w podrozdziale 6.1.

2.2.4. Interfejsy rozszerzeń

Większość funkcjonalności Hibernate można dostosować do własnych potrzeb, włączając w to wybór wbudowanych strategii. Gdy okazują się one niewystarczające, Hibernate dopuszcza zastosowanie własnej implementacji przez zastosowanie odpowiedniego interfejsu. Możliwe rozszerzenia są następujące:

- ◆ generacja klucza głównego (interfejs IdentifierGenerator),
- ◆ obsługa dialekta języka SQL (klasa abstrakcyjna Dialect),
- ◆ strategia buforowania (interfejsy Cache i CacheProvider),
- ◆ zarządzanie połączeniem JDBC (interfejs ConnectionProvider),
- ◆ zarządzanie transakcjami (interfejsy TransactionFactory, Transaction i TransactionManagerLookup),
- ◆ strategia ORM (hierarchia interfejsów ClassPersister),
- ◆ strategia dostępu do właściwości (interfejs PropertyAccessor),
- ◆ tworzenie pośredników (interfejs ProxyFactory).

Hibernate zawiera co najmniej jedną implementację każdego z wymienionych interfejsów, więc najczęściej nie trzeba zaczynać od podstaw, gdy chce się jedynie rozszerzyć pewną wbudowaną funkcjonalność. Co więcej, dostępny kod źródłowy stanowi doskonały przykład w sytuacji, gdy tworzy się własne rozwiązanie.

Zanim zacznie się tworzyć jakikolwiek kod związany z Hibernate, warto zadać sobie następujące pytanie: w jaki sposób zmusić do działania obiekt Session?

2.3. Konfiguracja podstawowa

Przyjrzelismy się przykładowej aplikacji oraz podstawowym interfejsom Hibernate. Aby móc skorzystać z automatycznej trwałości, trzeba poznać sposób jej konfiguracji. Hibernate udaje się tak skonfigurować, by działało poprawnie w niemalże dowolnej aplikacji Javy i środowisku programistycznym. Na ogół jednak stosuje się go w dwu- lub trójwarstwowych aplikacjach klient-serwer, przy czym Hibernate zostaje wdrożony tylko po stronie serwera. Aplikacją kliencką najczęściej jest przeglądarka internetowa, ale zdarzają się również aplikacje SWT i Swing. Choć w książce skupiamy się na wielowarstwowych aplikacjach internetowych, przedstawiane opisy mają zastosowanie również w dowolnej innej architekturze, na przykład aplikacjach wiersza poleceń. Istotnym jest, by zrozumieć różnicę w konfiguracji Hibernate w środowisku zarządzanym i niezarządzanym.

- ◆ **Środowisko zarządzanie** — tworzy pule zasobów, na przykład połączeń z bazą danych, i dopuszcza deklaratywne (w postaci metadanych) określanie zakresu transakcji oraz zasad bezpieczeństwa. Serwery aplikacji J2EE, na przykład JBoss, BEA WebLogic lub IBM WebSphere, implementują w Javie standardowe (zgodne z J2EE) środowisko zarządzane.
- ◆ **Środowisko niezarządzane** — zapewnia proste zarządzanie współbieżnością dzięki puli wątków. Pojemnik serwletowy taki jak Jetty lub Tomcat

zapewnia niezarządzane środowisko serwerowe dla aplikacji internetowych pisanych w Javie. Samowystarczalne aplikacje kliencie i aplikacje wiersza poleceń również uważa się za niezarządzane. Środowiska te nie udostępniają automatycznego zarządzania transakcjami i zasobami oraz nie zapewniają infrastruktury bezpieczeństwa. Sama aplikacja zarządza połączeniami z bazą danych i określa granice transakcji.

Hibernate stara się w sposób abstrakcyjny traktować środowisko, w którym przychodzi mu pracować. W środowisku niezarządzanym sam obsługuje transakcje i połączenia JDBC (lub deleguje ich wykonanie do odpowiedniego kodu aplikacji). W środowisku zarządzanym integruje się z transakcjami i połączeniami bazodanowymi zarządzanymi przez kontener. Hibernate potrafi poprawnie działać w obu środowiskach.

W obu środowiskach pierwszym zadaniem jest uruchomienie Hibernate. W praktyce nie jest to trudne — wystarczy utworzyć obiekt SessionFactory, używając klasy Connection.

2.3.1. Tworzenie obiektu SessionFactory

Aby uzyskać obiekt SessionFactory, należy najpierw utworzyć pojedynczy egzemplarz klasy Configuration w momencie inicjalizacji aplikacji i skonfigurować ścieżki do plików odwzorowań. Po konfiguracji obiekt Configuration pozwala tworzyć egzemplarze SessionFactory. Po ich wykonaniu obiekt Configuration staje się zbędny.

Poniższy kod uruchamia Hibernate:

```
Configuration cfg = new Configuration();
cfg.addResource("hello/Message.hbm.xml");
cfg.setProperties(System.getProperties());
SessionFactory sessions = cfg.buildSessionFactory();
```

Lokalizację pliku odwzorowania, *Message.hbm.xml*, określa się względem podstawowej ścieżki klas aplikacji. Jeśli na przykład ścieżka klas jest aktualnym folderem, plik *Message.hbm.xml* musi znajdować się wewnątrz folderu *hello*. Pliki odwzorowania XML muszą znajdować się w ścieżce wyszukiwania klas. W przykładzie wykorzystujemy również właściwości systemowe maszyny wirtualnej, by ustawić pozostałe opcje konfiguracyjne (można je inicjalizować wcześniej w samej aplikacji lub przy uruchamianiu systemu maszyny wirtualnej).

Łańcuch wywołań metod

Tworzenie łańcuchów wywołań metod to styl programowania obsługiwany przez wiele interfejsów Hibernate. Jest on bardziej popularny w języku Smalltalk niż w Javie. Niektóre osoby uważają go za mniej czytelny i trudniejszy do debugowania niż typowy styl programowania w Javie. W większości sytuacji jest jednak wyjątkowo wygodny.

Większość programistów Javy tworzy metody ustawiające lub dodające zwracające typ *void*, czyli brak wartości. Język SmallTalk nie posiada typy *void*, więc metody ustawiające lub dodające zwracają obiekt otrzymujący dane. Takie podejście umożliwia zmianę wcześniejszego kodu na następujący.

```
SessionFactory sessions = new Configuration()
    .addResource("hello/Message.hbm.xml")
    .setProperties(System.getProperties())
    .buildSessionFactory();
```

Zauważ, że nie potrzeba deklarować lokalnej zmiennej dla obiektu Configuration. Styl ten będzie stosowany w niektórych przykładach, ale gdy ktoś go nie lubi, nie wymuszały jego stosowania. Jeśli ktoś go lubi, zalecamy pisanie każdego wywołania metody w osobnym wierszu. W przeciwnym razie znacznie trudniej testować utworzony kod debugerem.

Przyjęło się, że pliki odwzorowań Hibernate w formacie XML zapisuje się w plikach z rozszerzeniem *.hbm.xml*. Dodatkowo przyjęło się, by tworzyć osobny plik odwzorowania dla każdej klasy, zamiast umieszczać cały opis odwzorowań w jednym pliku (jest to możliwe, ale nie jest zalecane). Przedstawiony przykład „Witaj świecie” stosuje tylko jedną klasę trwałości, ale założymy, że jest ich wiele i każde odwzorowanie znajduje się w osobnym pliku. Gdzie należy umieścić pliki odwzorowań?

Dokumentacja Hibernate zaleca, by plik odwzorowania dla każdej klasy trwałości umieszczać w tym samym folderze co klasę. Przykładowo, plik odwzorowania dla klasy Message należy umieścić w folderze *hello* pod nazwą *Message.hbm.xml*. Dla innej klasy trwałej należałoby wykonać osobny plik odwzorowania. Zalecamy stosować się do przedstawionej sugestii. Monolityczne pliki metadanych narzucane przez niektóre systemy szkieletowe, na przykład *struts-config.xml* z Struts, są głównym czynnikiem powodującym powstanie hasła „piekło metadanych”. Wiele plików odwzorowań wczytuje się, wielokrotnie wywołując metodę addResource(). Ewentualnie, gdy programista stosuje się do przedstawionej powyżej konwencji, może użyć metody addClass(), przekazując jako parametr klasę trwałą.

```
SessionFactory sessions = new Configuration()
    .addClass(org.hibernate.auction.model.Item.class)
    .addClass(org.hibernate.auction.model.Category.class)
    .addClass(org.hibernate.auction.model.Bid.class)
    .setProperties(System.getProperties())
    .buildSessionFactory();
```

Metoda addClass() zakłada, że nazwa pliku odwzorowania kończy się rozszerzeniem *.hbm.xml* i znajduje się w tym samym folderze co odwzorowywany plik klasy.

Przedstawimy tworzenie pojedynczego obiektu SessionFactory, ponieważ takie podejście występuje w aplikacjach najczęściej. Jeśli potrzebny jest kolejny obiekt, bo na przykład istnieje kilka baz danych, cały proces powtarza się od nowa. W ten sposób zawsze istnieje jeden obiekt SessionFactory na bazę danych gotowy do tworzenia obiektów Session działających z tą bazą danych i zestawem odwzorowań klas.

Konfiguracja Hibernate to nie tylko wskazanie dokumentów odwzorowań. Trzeba również wskazać sposób uzyskania połączenia z bazą danych oraz określić zachowanie Hibernate w różnych sytuacjach. Mnogość dostępnych właściwości konfiguracyjnych potrafi przytłoczyć (pełna ich lista znajduje się w dokumentacji

Hibernate). Na szczęście większość parametrów stosuje sensowne wartości domyślne, więc najczęściej potrzeba zmienić tylko ich niewielką część.

Aby określić opcje konfiguracyjne, korzysta się z jednej z wymienionych technik.

- ◆ Przekazuje się egzemplarz klasy `java.util.Properties` jako parametr metody `Configuration.setProperties()`.
- ◆ Ustawia się właściwości systemowe za pomocą konstrukcji `java -Dwłaściwość=wartość`.
- ◆ Umieszcza się w ścieżce wyszukiwania klas plik o nazwie *hibernate.properties*.
- ◆ Dołącza się elementy `<property>` do pliku *hibernate.cfg.xml* znajdującego się w ścieżce wyszukiwania klas.

Techniki pierwszą i drugą stosuje się naprawdę rzadko — jedynie w trakcie testów lub szybkiego prototypowania. Większość aplikacji potrzebuje stałego pliku konfiguracyjnego. Oba pliki, *hibernate.properties* i *hibernate.cfg.xml*, pełną tę samą funkcję — konfigurują Hibernate. Wybór pliku zależy tak naprawdę od własnych preferencji co do składni właściwości. Możliwe jest nawet mieszanie obu technik i posiadanie osobnych ustawień dla środowiska testowego i wdrożeniowego. Przedstawimy to zagadnienie w dalszej części rozdziału.

Bardzo rzadko stosowanym rozwiązaniem alternatywnym jest przekazywanie przez aplikację obiektu `Connection` z JDBC w momencie tworzenia obiektu `Session` dzięki obiekowi `SessionFactory` (powstaje wtedy kod typu `sessions.openSession(myConnection)`). Opcja ta pozwala nie podawać w trakcie konfiguracji żadnych opcji związanych z połączeniem z bazą danych. Nie polecamy takiego podejścia dla nowych aplikacji, które mogą skorzystać z infrastruktury połączeń bazodanowych całego środowiska (na przykład wykorzystując pulę połączeń JDBC lub źródła danych serwera aplikacji).

Ze wszystkich opcji konfiguracyjnych najważniejsze są ustawienia połączenia bazodanowego. Różnią się w zależności od tego, czy stosuje się środowisko zarządzane czy niezarządzane. Z tego względu opis rozbiujemy na dwa przypadki. Zaczniemy od środowiska niezarządzanego.

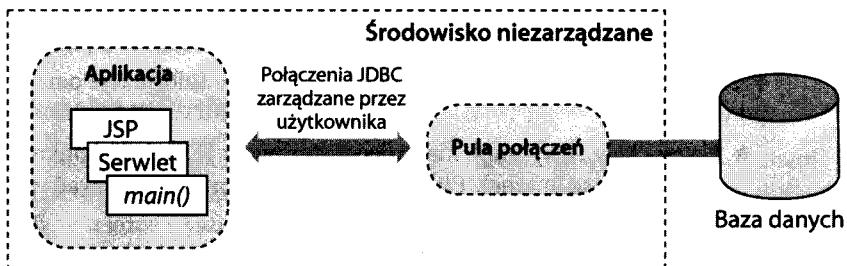
2.3.2. Konfiguracja w środowisku niezarządzanym

W środowisku niezarządzanym, na przykład kontenerze serwletów, aplikacja odpowiada za uzyskanie połączeń JDBC. Hibernate jest częścią aplikacji, więc również może uczestniczyć w ich uzyskaniu. Plik konfiguracyjny informuje, w jaki sposób Hibernate może uzyskać (lub utworzyć nowe) połączenia JDBC. Ogólnie nie zaleca się tworzenia połączeń za każdym razem, gdy tylko chce się skorzystać z bazy danych. Aplikacja Java powinna zatem używać puli połączeń JDBC. Istnieją trzy powody przemawiające ze stosowaniem puli:

- ◆ Uzyskiwanie nowych połączeń jest kosztowne.
- ◆ Utrzymywanie wielu niewykorzystywanych połączeń jest kosztowne.

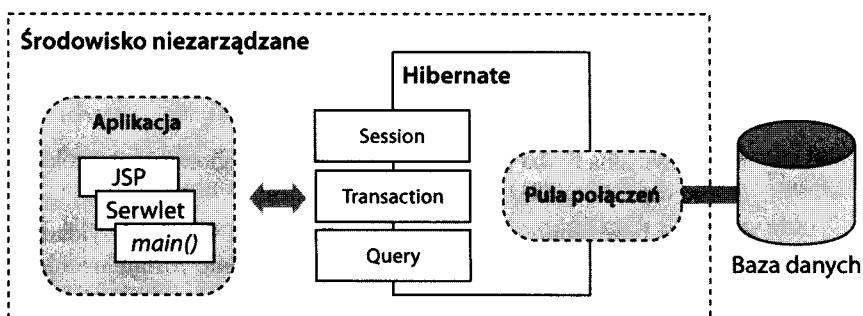
- ◆ Tworzenie instrukcji przygotowanych dla niektórych sterowników również jest kosztowne.

Rysunek 2.2 przedstawia rolę puli połączeń JDBC w środowisku wykonawczym aplikacji internetowej. Ponieważ samo środowisko nie udostępnia puli połączeń bazodanowych, aplikacja musi wprowadzić własny algorytm tworzenia puli lub stosować w tym celu niezależną bibliotekę, na przykład udostępnianą na zasadach *open source* bibliotekę C3P0. Aplikacje bez Hibernate najczęściej wywołuje metody puli połączeń, by uzyskać połączenie JDBC i wykonać polecenie SQL.



Rysunek 2.2. Pula połączeń JDBC w środowisku niezarządzanym

Po dodaniu Hibernate schemat ulega zmianie — ORM działa jako klient puli połączeń JDBC, co przedstawia rysunek 2.3. Kod aplikacji używa interfejsów programistycznych Session i Query z Hibernate do wykonywania operacji trwałości danych. Zajmuje się również zarządzaniem transakcjami, używając interfejsu Transaction z Hibernate.



Rysunek 2.3. Hibernate z pulą połączeń w środowisku niezarządzanym

Wykorzystanie puli połączeń

Hibernate stosuje architekturę modułów rozszerzających, więc potrafi zintegrować się z dowolną pulą połączeń. Obsługa C3P0 została wbudowana, więc z niej skorzystamy. Hibernate same utworzy pulę po podaniu wymaganych parametrów. Przykład pliku `hibernate.properties` stosującego C3P0 przedstawia listing 2.4.

Listing 2.4. Plik hibernate.properties z ustawieniami puli połączeń C3P0

```

hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/auctiondb
hibernate.connection.username = auctionuser
hibernate.connection.password = secret
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000

```

Przedstawiony fragment pliku określa następujące informacje omawiane wiersz po wierszu:

- ◆ Nazwę klasy Javy implementującej sterownik JDBC dla konkretnej bazy danych (klasa `Driver`). Plik JAR sterownika musi znajdować w ścieżce wyszukiwania klas aplikacji.
- ◆ Adres URL w formacie JDBC określający adres serwera i nazwę bazy danych dla połączeń JDBC.
- ◆ Nazwa użytkownika bazy danych.
- ◆ Hasło do bazy danych dla wskazanego użytkownika.
- ◆ Klasa dialekta języka SQL. Pomimo wysiłków standaryzacyjnych organizacji ANSI język SQL został zaimplementowany inaczej przez poszczególnych twórców baz danych. Należy więc określić klasę Dialekt dla stosowanych połączeń. Hibernate zawiera wbudowaną obsługę najpopularniejszych baz danych SQL. Nowe dialekty można łatwo dodawać.
- ◆ Minimalna liczba oczekujących na działania połączeń JDBC utrzymywana przez C3P0.
- ◆ Maksymalna liczba połączeń w puli. Zostanie zgłoszony wyjątek wykonania, gdy wartość ta okaże się niewystarczająca.
- ◆ Okres bezczynności podawany w sekundach, więc w przykładzie jest to 5 minut. Po tym okresie nieużywane połączenie zostaje usunięte z puli.
- ◆ Maksymalna liczba zbuforowanych poleceń przygotowanych. Buforowanie poleceń przygotowanych pozwala znaczco zwiększyć szybkość działania Hibernate.
- ◆ Czas bezczynności w sekundach, po którym połączenie jest automatycznie poddawane validacji.

Określanie właściwości w postaci `hibernate.c3p0.*` powoduje automatyczne wybranie C3P0 jako rozwiązania puli połączeń dla Hibernate (nie potrzeba żadnej dodatkowej opcji w celu włączenia C3P0). Biblioteka C3P0 ma znacznie więcej funkcji i opcji niż zostało przedstawionych w przykładzie. Zalecamy zajrzeć do dokumentacji interfejsu programistycznego Hibernate. Dokumentacja dla klasy

net.sf.hibernate.cfg.Environment informuje o wszystkich właściwościach konfiguracyjnych, włączając w to opcje C3P0 i innych puli połączeń obsługiwanych w sposób bezpośredni przez Hibernate.

Pozostałymi obsługiwanyimi pulami połączeń są: Apache DBCP i Proxool. Warto wypróbować we własnym środowisku wszystkie trzy rozwiązania przed podjęciem decyzji. Społeczność Hibernate najczęściej wybiera C3P0 i Proxool.

Hibernate zawiera dodatkowo własny mechanizm puli połączeń, ale nadaje się on jedynie do testów i eksperymentów z Hibernate — nie należy go stosować w systemie produkcyjnym. Nie był projektowany z uwzględnieniem wydajnej obsługi wielu współbieżnych żądań. Brakuje mu również funkcji wykrywania niepowodzeń istniejących w wyspecjalizowanych pulach połączeń.

Uruchamianie Hibernate

W jaki sposób uruchomić Hibernate z przedstawionymi właściwościami? Jeśli właściwości zadeklarowane się w pliku *hibernate.properties*, wystarczy umieścić ten plik w ścieżce wyszukiwania klas aplikacji. Zostanie automatycznie wykryty i wczytany przez Hibernate w momencie tworzenia obiektu Configuration.

Podsumujmy poznane do tej pory kroki konfiguracyjne. Najwyższy czas pobrać i zainstalować Hibernate, jeśli chce się go użyć w środowisku niezarządzanym.

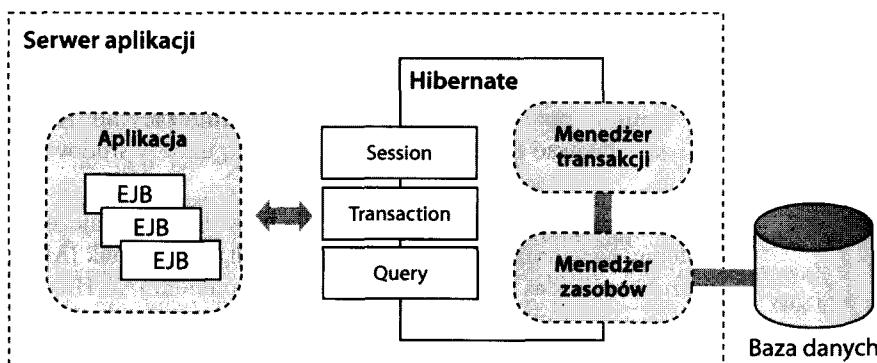
1. Pobierz i rozpakuj sterownik JDBC dla bazy danych. Najczęściej jest dostępny na witrynie producenta bazy danych. Umieść plik JAR w ścieżce wyszukiwania klas aplikacji. To samo zrobić z plikiem JAR Hibernate.
2. Dodaj zależności Hibernate do ścieżki wyszukiwania klas. Są one dołączone do Hibernate w folderze *lib*. Warto zajrzeć do pliku *lib/README.txt*, aby poznać listę wymaganych i opcjonalnych bibliotek.
3. Wybierz pulę połączeń JDBC obsługiwana przez Hibernate i skonfiguruj ją, używając pliku właściwości. Nie zapomnij o wskazaniu dialekta języka SQL.
4. Niech klasa Configuration potrafi odnaleźć plik właściwości. Nadaj mu nazwę *hibernate.properties* i umieść go w ścieżce wyszukiwania klas.
5. Utwórz egzemplarz klasy Configuration w aplikacji i załaduj pliki odwzorowania w formacie XML, używając metod `addResource()` lub `addClass()`. Utwórz obiekt SessionFactory na podstawie Configuration, wywołując metodę `buildSessionFactory()`.

Niestety, obecnie nie są zdefiniowane żadne pliki odwzorowań. Jeśli chcesz, możesz skorzystać z przykładu „Witaj świecie” lub pominąć pozostałą część tego rozdziału i przejść do rozdziału 3. omawiającego klasy trwałości oraz odwzorowania. Czytaj dalej, jeśli chcesz poznać konfigurację Hibernate w środowisku zarządzanym.

2.3.3. Konfiguracja w środowisku zarządzanym

Środowisko zarządzane zajmuje się kilkoma najczęściej potrzebnymi zagadnieniami, na przykład bezpieczeństwem aplikacji (uwierzytelnianie i autoryzacja), pulami połączeń i zarządzaniem transakcjami. Typowym środowiskiem zarządzanym jest serwer aplikacji J2EE. Choć serwery aplikacji najczęściej projektuje się do obsługi EJB, można skorzystać z wielu udostępnianych przez nie usług, nawet jeśli nie stosuje się ziarenek encyjnych EJB.

Hibernate najczęściej stosuje się w połączeniu z EJB sterowanym sesyjnie lub przez komunikaty, co przedstawia rysunek 2.4. EJB wykorzystuje te same interfejsy Hibernate co serwlet, JSP lub samowystarczalna aplikacja: Session, Transaction i Query. Kod dotyczący Hibernate można bez problemów przenosić między środowiskami zarządzanymi i niezarządzanymi. Sam system w niewidoczny sposób zajmuje się różnymi strategiami połączeń i transakcji.



Rysunek 2.4. Hibernate w środowisku zarządzanym przez serwer aplikacji

Serwer aplikacji udostępnia pulę połączeń jako **źródło danych** dowiązywane dzięki JNDI — egzemplarz javax.jdbc.DataSource. Należy poinformować Hibernate, gdzie ma szukać źródła danych w JNDI, podając pełną nazwę JNDI. Listing 2.5 przedstawia przykład pliku konfiguracyjnego dla wspomnianej sytuacji.

Listing 2.5. Przykładowy plik hibernate.properties dla źródła danych zapewnianego przez kontener

```

hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class =
net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class =
net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect

```

Plik najpierw podaje nazwę JNDI źródła danych. Źródło danych należy skonfigurować w deskryptorze wdrożenia aplikacji J2EE — sposób ustawiania zależy od dostawcy serwera aplikacji. Następnie włączona zostaje integracja Hibernate

z JTA. W następnym wierszu wskazujemy Hibernate lokalizację klasy TransactionManager serwera aplikacji, by zapewnić pełną integrację z transakcjami kontenera. Standard J2EE nie definiuje jednego konkretnego podejścia, ale Hibernate zawiera wbudowaną obsługę wszystkich popularnych serwerów aplikacji. Na końcu ponownie pojawia się informacja o dialekcie języka SQL.

Po poprawnej konfiguracji wszystkich elementów sposób korzystania z Hibernate w środowisku zarządzanym nie różni się znacząco od korzystania z wersji niezarządzanej — tworzy się obiekt Configuration z odwzorowaniami i na jego podstawie tworzy obiekt SessionFactory. Dodatkowej uwagi wymagają ustawienia związane z transakcyjnością środowiska.

Java ma standardowy interfejs programistyczny dla transakcji, JTA, który służy do sterowania transakcjami w zarządzanym środowisku J2EE. Są to tak zwane **transakcje zarządzane przez kontener (CMT — Container Managed Transaction)**. Jeśli menedżer transakcji JTA występuje, połączenia JDBC są z nim związane i są pod jego pełną kontrolą. W środowisku niezarządzanym sytuacja jest zupełnie inna, ponieważ aplikacja (lub pula) bezpośrednio zarządza połączonymi i transakcjami JDBC.

Zarządzane i niezarządzane środowiska stosują inne podejście do transakcji. Ponieważ Hibernate w miarę możliwości chce zapewnić przenośność między oboma środowiskami, sam definiuje interfejs programistyczny do sterowania transakcjami. Interfejs Transaction z Hibernate stanowi abstrakcję ukrywającą różnice między transakcjami JTA i JDBC (a nawet transakcjami CORBA). Konkretną strategię transakcyjną ustawia właściwość hibernate.connection.factory_class. Przyjmuje ona jedną z dwóch wartości:

- ◆ `net.sf.hibernate.transaction.JDBCTransactionFactory`, która deleguje wykonanie zadań do transakcji JDBC; strategię tę stosować należy z pulami połączeń w środowisku niezarządzanym (staje się domyślna, gdy nie określi się żadnej strategii).
- ◆ `net.sf.hibernate.transaction.JTATransactionFactory`, która deleguje wykonanie zadań do transakcji JTA; jest to poprawna strategia dla CMT, gdzie połączenia są powiązane z JTA. Warto pamiętać, że jeśli właśnie trwa transakcja JTA w momencie wywołania `beginTransaction()`, kolejne zadania zostają wykonane w tej transakcji (w przeciwnym razie powstaje nowa transakcja JTA).

Bardziej szczegółowo wprowadzenie do interfejsu programistycznego Transaction i wpływu wybranych scenariuszy transakcji zostało opisane w podrozdziale 5.1. Należy pamiętać o dwóch krokach koniecznych do przeprowadzenia w serwerze aplikacji J2EE: ustawieniu klasy fabryki dla interfejsu programistycznego Transaction na JTA w opisany wcześniej sposób i zadeklarowaniu wyszukiwania menedżera transakcji w sposób specyficzny dla serwera aplikacji. Strategia wyszukiwania potrzebna jest tylko wtedy, gdy korzysta się w Hibernate z buforowania dwupoziomowego. Nie zaszkodzi ustawiać jej także wtedy, gdy nie używa się takiego buforowania.

Hibernate z serwerem Tomcat

Tomcat nie jest pełnym serwerem aplikacji, ale kontenerem serwletów. Z drugiej strony zawiera pewne funkcje spotykane tylko w serwerach aplikacji. Hibernate może skorzystać z jednej z tych dodatkowych funkcji — puli połączeń Tomcata. Tomcat wewnętrznie korzysta z puli połączeń DBCP, ale udostępnia ją jako źródło danych JNDI (podobnie jak serwery aplikacji). Aby skonfigurować źródło danych Tomcata, dokonaj edycji pliku server.xml zgodnie z instrukcjami podanymi w dokumentacji JNDI/JDBC Tomcata. Hibernate odnajdzie źródło danych po ustaleniu właściwości hibernate.connection.datasource. Warto pamiętać, że Tomcat nie zawiera menedżera transakcji, więc cała sytuacja jest nadaj bardziej podobna do środowiska niezarządzanego.

W tej chwili powinieneś mieć działający system Hibernate, niezależnie od tego, czy używasz prostego kontenera serwletów czy serwera aplikacji. Utwórz i skompiluj klasę trwałości (na przykład klasę Message z początku rozdziału), skopiuj Hibernate i wymagane przez niego biblioteki do ścieżki wyszukiwania klas aplikacji wraz z plikiem `hibernate.properties`. Na końcu utwórz obiekt `SessionFactory`.

Kolejny podrozdział opisuje zaawansowane opcje konfiguracyjne Hibernate. Niektóre z nich polecamy. W szczególności możliwość tworzenia dla celów testowych dziennika wykonanych polecen SQL lub stosowania wygodnych plików konfiguracyjnych XML zamiast prostych plików tekstowych. Można bezpiecznie pominąć kolejny podrozdział i powrócić do niego dopiero po zdobyciu w rozdziale 3. szczegółowej wiedzy na temat klas trwałych.

2.4. Zaawansowane ustawienia konfiguracyjne

Gdy aplikacja z Hibernate działa poprawnie, warto zainteresować się wszystkimi parametrami konfiguracyjnymi Hibernate. Dzięki nim nierzadko udaje się zoptymalizować działanie Hibernate, w szczególności przez odpowiednie dostrojenie współpracy z JDBC (na przykład zastosowanie aktualizacji wsadowych JDBC).

Nie chcemy na razie przynudzać wszystkimi szczegółami — najlepszym źródłem informacji na temat wszystkich opcji konfiguracyjnych jest dokumentacja Hibernate. W poprzednim podrozdziale przedstawiliśmy jedynie opcje pozwalające uruchomić aplikację.

Istnieje pewien parametr, na który już w tym momencie musimy kłaść duży nacisk. Przydaje się wielokrotnie, gdy tworzy się oprogramowanie. Ustawienie właściwości `hibernate.show_sql` na wartość `true` włącza wyświetlanie wszystkich utworzonych polecen SQL na konsoli. Warto korzystać z tej opcji w momencie poszukiwania błędów, poszukiwania wąskiego gardła lub po prostu analizy działania Hibernate. Warto wiedzieć, czym tak naprawdę zajmuje się warstwa ORM, więc system nie ukrywa szczegółów polecen SQL przed programistą.

Do tej pory zakładaliśmy przekazywanie parametrów konfiguracyjnych dzięki plikowi `hibernate.properties` lub programowo dzięki egzemplarzowi `java.util.Properties`. Trzecie z rozwiązań polega na użyciu pliku konfiguracyjnego w formacie XML.

2.4.1. Konfiguracja bazująca na pliku XML

Warto użyć pliku konfiguracyjnego XML (patrz listing 2.6), by w pełni skonfigurować obiekt SessionFactory. W odróżnieniu od pliku *hibernate.properties*, który zawiera tylko parametry konfiguracyjne, plik *hibernate.cfg.xml* może również określać lokalizacje dokumentów odwzorowań. Wiele osób preferuje centralizację konfiguracji Hibernate właśnie w ten sposób, by uniknąć dodawania parametrów do obiektu Configuration w kodzie aplikacji.

Listing 2.6. Przykładowy plik konfiguracyjny hibernate.cfg.xml

```
?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
  <session-factory name="java:/hibernate/HibernateFactory">
    <property name="show_sql">true</property>
    <property name="connection.datasource">
      java:/comp/env/jdbc/AuctionDB
    </property>
    <property name="dialect">
      net.sf.hibernate.dialect.PostgreSQLDialect
    </property>
    <property name="transaction.manager_lookup_class">
      net.sf.hibernate.transaction.JBossTransactionManagerLookup
    </property>
    <mapping resource="auction/Item.hbm.xml"/>
    <mapping resource="auction/Category.hbm.xml"/>
    <mapping resource="auction/Bid.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

- ① Deklaracja typu dokumentu pomaga analizatorowi XML dokonać validacji dokumentu, czyli sprawdzić zgodność jego formatu z wytycznymi zawartymi w DTD pliku konfiguracyjnego Hibernate.
- ② Opcjonalny atrybut name jest równoważny właściwości `session_factory_name`. Służy do dowiązania JNDI dla SessionFactory, co zostanie omówione w dalszej części rozdziału.
- ③ Właściwości hibernate można określać bez przedrostka hibernate. Poza tym przedrostkiem nazwy i wartości są dokładnie takie same jak we właściwościach ustawianych programowo.
- ④ Dokumenty odwzorowań określa się jako zasoby aplikacji lub nawet jako na stałe zakodowane nazwy plików. Pliki użyte w przykładzie pochodzą z aplikacji systemu akcyjnego opisywanego w rozdziale 3.

Do inicjalizacji Hibernate wystarczy teraz użycie następującego kodu:

```
SessionFactory sessions = new Configuration()
  .configure().buildSessionFactory();
```

Chwileczkę — skąd Hibernate wie, gdzie znajduje się plik konfiguracyjny?

W momencie wywołania metody `configure()` Hibernate poszukuje pliku o nazwie `hibernate.cfg.xml` w ścieżce wyszukiwania klas. Jeśli chce się dla pliku konfiguracyjnego zastosować inną nazwę lub też plik znajduje się w niestandardowej lokalizacji, do metody `configure()` należy przekazać ścieżkę.

```
SessionFactory sessions = new Configuration()
    .configure("/hibernate-config/auction.cfg.xml")
    .buildSessionFactory();
```

Stosowanie plików konfiguracyjnych okazuje się bardziej wygodne od plików właściwości lub programowego ustawiania opcji. Możliwość usunięcia z kodu lokalizacji plików odwzorowań klas (nawet jeśli miałby się znaleźć w pomocniczej klasie inicjalizującej) stanowi ogromną zaletę przedstawionego podejścia. Nic nie stoi na przeszkodzie, by stosować różne zbiory plików odwzorowań (i inne opcje konfiguracyjne) w zależności od bazy danych i środowiska (produkcyjnego lub testowego). Co więcej, można je przełączać programowo.

Jeśli ścieżka wyszukiwania klas zawiera pliki `hibernate.properties` i `hibernate.cfg.xml`, ustawienia występujące w pliku XML nadpiszą ustawienia z pliku właściwości. Rozwiązywanie bywa użyteczne, gdy pewne podstawowe ustawienia znajdują się w pliku właściwości i modyfikuje dla każdego wdrożenia plikiem XML.

Warto zwrócić uwagę na nadanie parametru `name` dla obiektu `SessionFactory` w pliku konfiguracyjnym XML. Hibernate używa tej nazwy do automatycznego dowiązania obiektu do JNDI po jego utworzeniu.

2.4.2. Obiekt `SessionFactory` dowiązany do JNDI

W większości aplikacji stosujących Hibernate obiekt `SessionFactory` powinien zostać utworzony tylko jeden raz w momencie inicjalizacji aplikacji. Cała aplikacja powinna stosować ten pojedynczy egzemplarz — wszystkie obiekty `Session` należą tworzyć na jego podstawie. Często pojawia się pytanie, gdzie umieścić obiekt fabryki, by był bez trudu dostępny z każdego miejsca aplikacji.

W środowisku J2EE obiekt `SessionFactory` warto dowiązać do JNDI, by łatwo udostępnić go wielu wątkom i komponentom wykorzystującym Hibernate. Oczywiście JNDI to nie jedyny sposób, w jaki komponenty aplikacji mogą uzyskać obiekt `SessionFactory`. Istnieje wiele możliwych implementacji wzorca rejestru, włączając w to użycie obiektu `ServletContext` lub zmiennej `static final` w obiekcie singletonu. Szczególnie eleganckim wydaje się być rozwiązanie stosujące komponent szkieletowy odwrócenia sterowania (IoC — *Inversion of Control*) o zasięgu całej aplikacji. JNDI jest popularnym rozwiązaniem udostępnianym jako usługi JMX, co wkrótce pokażemy. Niektóre alternatywy przedstawimy w podrozdziale 8.1.

Obiekt `SessionFactory` automatycznie dowiąże się do JNDI, gdy właściwość `hibernate.session_factory_name` będzie zawierała nazwę węzła katalogu. Jeśli środowisko wykonawcze nie zapewnia domyślnego kontekstu JNDI (lub też domyślna implementacja JNDI nie obsługuje egzemplarzy interfejsu Reference-

able), należy wskazać kontekst JNDI, używając właściwości hibernate.jndi.url i hibernate.jndi.class.

Uwaga Interfejs programistyczny JNDI umożliwia zapisywanie i odczyt obiektów w strukturze hierarchicznej (drzewiastej). JNDI implementuje wzorzec rejestru. Z JNDI można powiązać obiekty infrastruktury (konteksty transakcji, źródła danych), ustawienia konfiguracyjne (opcje środowiska, rejestrzy użytkownika) lub nawet obiekty aplikacji (referencje do EJB, fabryki obiektów).

Poniżej przedstawiamy przykładową konfigurację Hibernate, która dowiązuje obiekt SessionFactory do nazwy hibernate/HibernateFactory, wykorzystując bezpłatną implementację JNDI firmy Sun (*fscontext.jar*) bazującą na systemie plików.

```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class =
net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.session_factory_name = hibernate/HibernateFactory
hibernate.jndi.class = com.sun.jndi.fscontext.RefFSContextFactory
hibernate.jndi.url = file:/auction/jndi
```

Do określenia parametrów równie dobrze można skorzystać z pliku konfiguracyjnego XML. Przykład jest mało realistyczny, bo większość serwerów aplikacji udostępniających pule połączeń dzięki JNDI stosuje również implementację JNDI z domyślnym kontekstem dopuszczającym zapis. Opisane podejście stosuje serwer JBoss, w którym można opuścić dwie ostatnie właściwości i określić tylko nazwę dla obiektu SessionFactory. Do inicjalizacji dowiązania wystarczy wtedy wykonanie kodu Configuration().configure().buildSessionFactory().

Uwaga Tomcat zawiera wbudowany kontekst JNDI tylko do odczytu — nie można w nim nic zapisać z poziomu aplikacji po uruchomieniu kontenera serwletów. Z tego powodu Hibernate nie potrafi skorzystać z tego kontekstu — należy stosować pełną implementację kontekstu (wykorzystując wspomniany kontekst firmy Sun) lub wyłączyć dowiązanie JNDI dla obiektu SessionFactory przez pominięcie w konfiguracji właściwości session_factory_name.

Przyjrzyjmy się innym istotnym opcjom konfiguracyjnym Hibernate dotyczącym tworzenia dziennika operacji.

2.4.3. Dzienniki

Hibernate (i wiele innych implementacji ORM) wykonuje polecenia SQL w sposób **asynchroniczny**. Polecenie INSERT zazwyczaj nie jest wykonywane zaraz po wywołaniu przez aplikację metody Session.save(). Podobnie polecenie UPDATE

nie wykona się od razu po wywołaniu `Item.addBid()`. Instrukcje SQL zostają wykonane na końcu transakcji. Takie zachowanie nazywa się **zapisem opóźnionym**.

Takie podejście znacząco utrudnia śledzenie i debugowanie kodu ORM. W teorii możliwe jest traktowanie przez aplikację systemu Hibernate jako czarnej skrzynki i ignorowanie jego zachowania. Aplikacja nie potrafi wykryć asynchroniczności działań Hibernate (a przynajmniej nie bez posiłkowania się bezpośrednimi wywołaniami JDBC). Gdy pojawią się problemy, warto byłoby się dowiedzieć, co tak naprawdę wykonuje Hibernate i w jakiej kolejności. Ponieważ Hibernate jest projektem typu *open source*, można przyjrzeć się jego kodowi źródłowemu. Czasem to naprawdę pomaga! Z drugiej strony z powodu asynchronicznego działania Hibernate przy jego debugowaniu łatwo się „stracić”. Zamiast korzystać z debugera, lepiej użyć dziennika operacji.

Wspomnieliśmy wcześniej o parametrze konfiguracyjnym `hibernate.show_sql`, który warto włączyć za każdym razem, gdy pojawią się problemy. Czasem polecenia SQL nie wystarczają i trzeba sięgnąć głębiej.

Hibernate tworzy dziennik wszystkich interesujących operacji, używając biblioteki commons-logging Apache. Jest to cienka warstwa abstrakcji kierująca dane do biblioteki log4j Apache (jeśli plik `log4j.jar` znajduje się w ścieżce wyszukiwania klas) lub do systemu dziennika z JDK 1.4 (jeśli używa się Javy 1.4 lub nowsze i brakuje log4j). Zalecamy stosowanie log4j, gdyż jest bardziej rozwinięty, bardziej popularny i stale rozwijany.

Aby zobaczyć jakiekolwiek wyniki działania log4j, w ścieżce wyszukiwania klas potrzeba pliku `log4j.properties` (podobnie jak plików `hibernate.properties` i `hibernate.cfg.xml`). Poniższa przykładowa zawartość tego pliku powoduje wyświetlanie wszystkich komunikatów na konsoli.

```
### kieruj komunikaty bezpośrednio na standardowe wyjście ####
log4j.appenders.stdout=org.apache.log4j.ConsoleAppender
log4j.appenders.stdout.Target=System.out
log4j.appenders.stdout.layout=org.apache.log4j.PatternLayout
log4j.appenders.stdout.layout.ConversionPattern=%d{ABSOLUTE}%5p
    %c{1}:%L - %m%n
### opcje dziennika podstawowego ####
log4j.rootLogger=warn, stdout
### opcje dziennika Hibernate ####
log4j.logger.net.sf.hibernate=info
### uwzględniaj parametry dowiezanych JDBC ####
log4j.logger.net.sf.hibernate.type=info
### uwzględniaj aktywność bufora poleceń przygotowanych ####
log4j.logger.net.sf.hibernate.ps.PreparedStatementCache=info
```

Przy tej konfiguracji w trakcie działania systemu nie pojawi się zbyt wiele komunikatów. Wystarczy jednak zamielić wartość `info` na `debug` w kategorii `log4j.logger.net.sf.hibernate`, by poznać szczegóły działania Hibernate. Warto się upewnić, że tego rodzaju konfiguracja nie trafi na serwer produkcyjny — zapis do dziennika będzie wolniejszy od dostępu do bazy danych.

Konfiguracja zawiera się obecnie w trzech plikach: `hibernate.properties`, `hibernate.cfg.xml` i `log4j.properties`.

Istnieje jeszcze jeden sposób konfiguracji Hibernate, jeśli serwer aplikacji obsługuje JMX (*Java Management Extensions*).

2.4.4. Java Management Extensions

Świat Javy pełen jest specyfikacji, standardów i ich implementacji. Stosunkowo nowym i istotnym już od pierwszej wersji jest standard **Java Management Extensions** (JMX). Dotyczy on zarządzania komponentami systemowymi, a w zasadzie usługami systemu.

Gdzie w tym całym obrazie mieści się Hibernate? Gdy umieści się go na serwerze aplikacji, korzysta z innych usług, na przykład zarządzania transakcjami i pulą połączeń z bazą danych. Dlaczego nie uczynić samego Hibernate zarządzalną usługą, której mogą używać inne elementy systemu? Integracja z JMX czyni z Hibernate zarządzalny komponent JMX.

Specyfikacja JMX definiuje następujące komponenty:

- ◆ **JMX MBean** — komponent wielokrotnego użytku (najczęściej infrastrukturalny), który udostępnia interfejs zarządzania (administracji),
- ◆ **kontener JMX** — zapewnia uogólniony dostęp (lokalny lub zdalny) do MBean,
- ◆ **klient JMX** (najczęściej uogólniony) — umożliwia administrację dowolnym MBean dzięki kontenerowi JMX.

Serwer aplikacji obsługujący JMX (na przykład JBoss) działa jak kontener JMX, umożliwiając konfigurację i inicjalizację komponentów MBean jako części procesu uruchamiania serwera aplikacji. Można monitorować i administrować MBean, stosując konsolę administracyjną serwera aplikacji (dzieli się na klient JMX).

Komponent MBean daje się utworzyć jako usługę JMX, która nie tylko jest przenośna między różnymi serwerami aplikacji z wbudowaną obsługą JMX, ale również zapewnia wdrożenie w działającym systemie (tak zwane „wdrożenie na gorąco”).

Hibernate można administrować jako JMX MBean. Usługa JMX dopuszcza inicjalizację Hibernate w momencie uruchamiania serwera aplikacji i sterowania (konfigurowania) go dzięki klientowi JMX. Z drugiej strony komponenty JMX nie są automatycznie zintegrowane z transakcjami zarządzanymi przez kontener. Z tego powodu opcje konfiguracyjne z listingu 2.7 (deskryptory wdrożenia usługi w JBoss) wyglądają podobnie do ustawień Hibernate dla środowiska zarządzanego.

Listing 2.7. Deskryptory wdrożenia Hibernate jako JMX w JBoss - plik jboss-service.xml

```
<server>
<mbean code="net.sf.hibernate.jmx.HibernateService"
       name="jboss.jca:service=HibernateFactory, name=HibernateFactory">
  <depends>jboss.jca:service=RARDeployer</depends>
  <depends>jboss.jca:service=LocalTxCM, name=DataSource</depends>
  <attribute name="MapResources">
    auction/Item.hbm.xml, auction/Bid.hbm.xml
  </attribute>
</mbean>
```

ROZDZIAŁ 2.

Wprowadzenie i integracja *Hibernate*

```

</attribute>
<attribute name="JndiName">
    java:/hibernate/HibernateFactory
</attribute>
<attribute name="Datasource">
    java:/comp/env/jdbc/AuctionDB
</attribute>
<attribute name="Dialect">
    net.sf.hibernate.dialect.PostgreSQLDialect
</attribute>
<attribute name="TransactionStrategy">
    net.sf.hibernate.transaction.JTATransactionFactory
</attribute>
<attribute name="TransactionManagerLookupStrategy">
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
</attribute>
<attribute name="UserTransactionName">
    java:/UserTransaction
</attribute>
</mbean>
</server>
```

Usługa `HibernateService` zależy od dwóch innych usług JMX — `service=RARDeployer` i `service=LocalTxCM, name=DataSource` — znajdujących się w domenie usługowej `jboss.jca`.

MBean dla `Hibernate` znajduje się w pakiecie `net.sf.hibernate.jmx`. Niestety, metody zarządzania cyklem życia JMX nie stanowią części specyfikacji JMX 1.0. Z tego powodu metody `start()` i `stop()` z `HibernateService` są specyficzne dla serwera aplikacji JBoss.

Uwaga

Jeśli jesteś zainteresowany zaawansowanymi użyciami JMX, warto zająć się kodu serwera JBoss. Wszystkie usługi (nawet kontener EJB) z JBoss są zaimplementowane jako MBean i mogą być zarządzane dzięki interfejsowi konsolowemu.

Zalecamy konfigurację `Hibernate` w sposób programowy (z zastosowaniem obiektu `Configuration`) przez próbę jego uruchomienia jako usługa JMX. Niektóre funkcje (na przykład wdrażanie na gorąco aplikacji z `Hibernate`) jest możliwe tylko z JMX. Największą zaletą `Hibernate` działającego z JMX jest automatyczny rozruch. Nie trzeba ręcznie tworzyć obiektów `Configuration` i `SessionFactory`. Wystarczy korzystać z obiektu `SessionFactory` przez JNDI zaraz po uruchomieniu `HibernateService`.

2.5. Podsumowanie

W rozdziale przyjrzaliśmy się Hibernate z nieco wyższego poziomu. Dodatkowo przedstawiliśmy jego architekturę, stosując przykład „Witaj świecie”. Pokazaliśmy, jak konfigurować Hibernate w różnych środowiskach z zastosowaniem różnego sposobów przekazywania konfiguracji, włączając w to JMX.

Interfejsy Configuration i SessionFactory to punkty początkowe dla aplikacji chcących używać Hibernate zarówno w środowisku zarządzanym, jak i niezarządzanym. Hibernate stosuje dodatkowe interfejsy programistyczne, na przykład interfejs Transaction, by zakryć różnice środowisk i zapewnić przenośność kodu trwałości danych.

Hibernate można zintegrować z niemal dowolnym środowiskiem Javy: serwetem, apletem lub trójwarstwową aplikacją klient-serwer. Najważniejszymi elementami konfiguracji Hibernate są: zasoby bazy danych (konfiguracja połączeń), strategie transakcyjne i oczywiście metadane odwzorowań zawarte w plikach XML.

Interfejsy konfiguracyjne Hibernate zostały tak zaprojektowane, by obsłużyć możliwe dużo różnych sposobów użycia przy zachowaniu ich dużej zrozumiałosci. Najczęściej jeden plik o nazwie *hibernate.cfg.xml* i jeden wiersz kodu w programie wystarczają do uruchomienia Hibernate.

Konfiguracja nie miałaby jednak większego sensu bez klas trwałych i dokumentów odwzorowań. Kolejny rozdział został w całości poświęcony pisaniu i odwzorowywaniu klas trwałych. Wkrótce się przekonasz, jak w rzeczywistych aplikacjach dzięki nietrywialnemu odwzorowaniu obiektowo-relacyjnemu zapewnić trwałość obiektów.

Odwzorowanie klas trwałości danych

W rozdziale:

- ◆ Podstawy POJO dla rozbudowanych modeli dziedzinowych
- ◆ Odwzorowanie POJO dzięki metadanym Hibernate
- ◆ Odwzorowanie dziedziczenia klas i modele szczegółowe
- ◆ Wprowadzenie do odwzorowań asocjacji klas

Przykład „Witaj świecie” z rozdziału 2. wprowadził podstawy działania Hibernate. Niestety, z powodu jego prostoty trudno zrozumieć wszystkie wymagania stawiane zaawansowanym aplikacjom z rzeczywistego świata korzystającym ze złożonych modeli danych. W pozostałej części książki skorzystamy z bardziej wyrafinowanej aplikacji przykładowej — systemu aukcji internetowych — by zademonstrować poszczególne elementy Hibernate.

W niniejszym rozdziale rozpoczęliśmy od wprowadzenia modelu programistycznego klas trwałości danych. Projektowanie i implementacja tych klas jest procesem wieloetapowy, który wkrótce szczegółowo omówimy.

Najpierw zajmiemy się sposobami rozpoznawania **encji biznesowych** problemu dziedzinowego. Utworzmy model koncepcyjny encji i ich atrybutów noszący nazwę **modelu dziedzinowego**. Zaimplementujemy model w Javie, tworząc klasę trwałą dla każdej encji. (Sporo czasu poświęcimy na opis sposobu, w jaki powinny wyglądać klasy Javy dotyczące trwałości danych.)

Następnie zdefiniujemy **metadane odwzorowania**, by poinformować Hibernate, w jaki sposób klasy i ich właściwości przekładają się na tabele i kolumny bazy danych. Wymaga to napisania lub wygenerowania dokumentów XML, które zostaną wdrożone wraz ze skompilowanymi klasami Javy, by Hibernate mógł je wczytać w trakcie działania aplikacji. Opis metadanych odwzorowania wraz technikami dotyczącymi szczegółowości, określania identyczności, dziedziczenia i asocjacji, stanowi rdzeń rozdziału. Rozdział omawia podstawy rozwiązań dla pierwszych czterech ogólnych problemów ORM wymienionych w podrozdziale 1.4.2.

Zacznijmy od przedstawienia przykładowej aplikacji.

3.1. Aplikacja CaveatEmptor

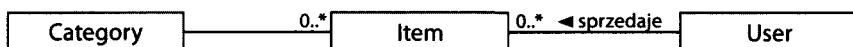
Internetowy system aukcyjny CaveatEmptor przedstawia techniki ORM i funkcjonalność Hibernate. Kod źródłowy w pełni działającej aplikacji znajduje się do pobrania na witrynie <http://caveatemptor.hibernate.org>, natomiast sama warstwa trwałości opisywana w książce jest dostępna pod adresem <ftp://ftp.helion.pl/przyklady/troyaid.zip>. Aplikacja ma interfejs oparty na stronach WWW i działa z wykorzystaniem serwera serwletów typu Tomcat. Nie będziemy zwracać dużej uwagi na sposób wyświetlania; skupimy się na dostępie do danych. W rozdziale 8. opisujemy zmiany, które należałyby dokonać, gdyby całą logikę biznesową i dostęp do danych trzeba było przeprowadzać na osobnej warstwie biznesowej zaimplementowanej jako ziarenko sesyjne EJB.

Zacznijmy od samego początku. Aby dobrze poznać zagadnienia projektowe związane z ORM, załączmy, że aplikacja CaveatEmptor jeszcze nie istnieje i dopiero zaczynamy ją tworzyć od podstaw. Pierwszym zadaniem będzie analiza.

3.1.1. Analiza dziedziny biznesowej

Tworzenie oprogramowania najczęściej rozpoczyna się od analizy problemu dziedzinowego (jeśli nie istnieje żaden starszy kod ani schemat bazy danych, do którego należy się dostosować).

Na tym etapie projektant wraz z ekspertami dziedzinowymi identyfikuje podstawowe encje (byty) istotne dla systemu komputerowego. Encje to najczęściej elementy zrozumiałe dla użytkowników systemu: płatność, klient, przedmiot, zamówienie itp. Pewne encje mogą być abstrakcjami mniej konkretnych bytów, o których myśli użytkownik aplikacji (na przykład algorytm określania cen). Nadal jednak zagadnienia te powinny być zrozumiałe dla użytkownika. Wszystkie informacje niejednokrotnie znajdują się w koncepcyjnym widoku biznesowym nazywanym modelem biznesowym. Projektanci oprogramowania obiektowego analizują model biznesowy i tworzą model obiektowy. Wszystko odbywa się jak na razie na poziomie koncepcyjnym (bez kodu Javy). Model obiektowy może być prostym obrazem w głowie projektanta lub bardzo złożonym diagramem klas UML (patrz rysunek 3.1) tworzonym z wykorzystaniem narzędzi CASE (*Computer-Aided Software Engineering*) typu ArgoUML lub TogetherJ.



Rysunek 3.1. Diagram klas modelu obiektowego typowego systemu aukcji internetowych

Ten prosty model zawiera encje, które spodziewamy się odnaleźć w każdym systemie aukcji internetowych: Category (kategoria), Item (przedmiot) i User (użytkownik). Encje i ich związki (a niejednokrotnie również atrybuty) zostają przedstawione na modelu problemu dziedzinowego. Ten rodzaj modelu obiektowego uwzględniającego problem dziedzinowy z punktu widzenia jego użytkownika nazywamy najczęściej modelem dziedzinowym.

To abstrakcyjny widok rzeczywistego świata. Odniesiemy się do tego modelu w momencie implementacji w Javie klas trwałości danych.

3.1.2. Model dziedzinowy CaveatEmptor

Witryna aukcyjna CaveatEmptor zawiera różnego rodzaju przedmioty: od sprzętu elektronicznego po bilety lotnicze. Aukcje odbywają się w następujący sposób: użytkownicy zgłaszają ceny ofertowe tak długo, jak długo trwa aukcja. Po jej zakończeniu wygrywa użytkownik dający za przedmiot najwyższą kwotę.

W każdym sklepie produkty kategoryzuje się według typu i grupuje z innymi podobnymi produktami na jednej półce. Katalog aukcyjny również wymaga hierarchii kategorii. Kupujący może przeszukiwać kategorie lub dokonywać wyszukiwania dowolnego przedmiotu po podaniu jego atrybutów. Lista przedmiotów pojawia się w przeglądarce kategorii i na liście wyników. Wybór elementu z listy wyświetla stronę ze szczegółami przedmiotu.

Akcja składa się z ciągu ofert. Jedna z ofert staje się ofertą wygrywającą. Szczegółowe dane użytkownika zawierają: imię i nazwisko, login, adres zamieszkania, adres e-mail oraz informacje o płatności.

Sieć zaufania to istotny element każdego internetowego systemu aukcyjnego. Użytkownicy budują swoją reputację na komentarzach pozytywnych lub negatywnych od innych osób. Komentarz wystawia zarówno sprzedający, jak i kupujący. Komentarze użytkownika są widoczne przez wszystkich zainteresowany przed zgłoszeniem oferty.

Dokładniejszy model dziedzinowy przedstawia rysunek 3.2. Omówmy pokrótkę różnice jego elementy.

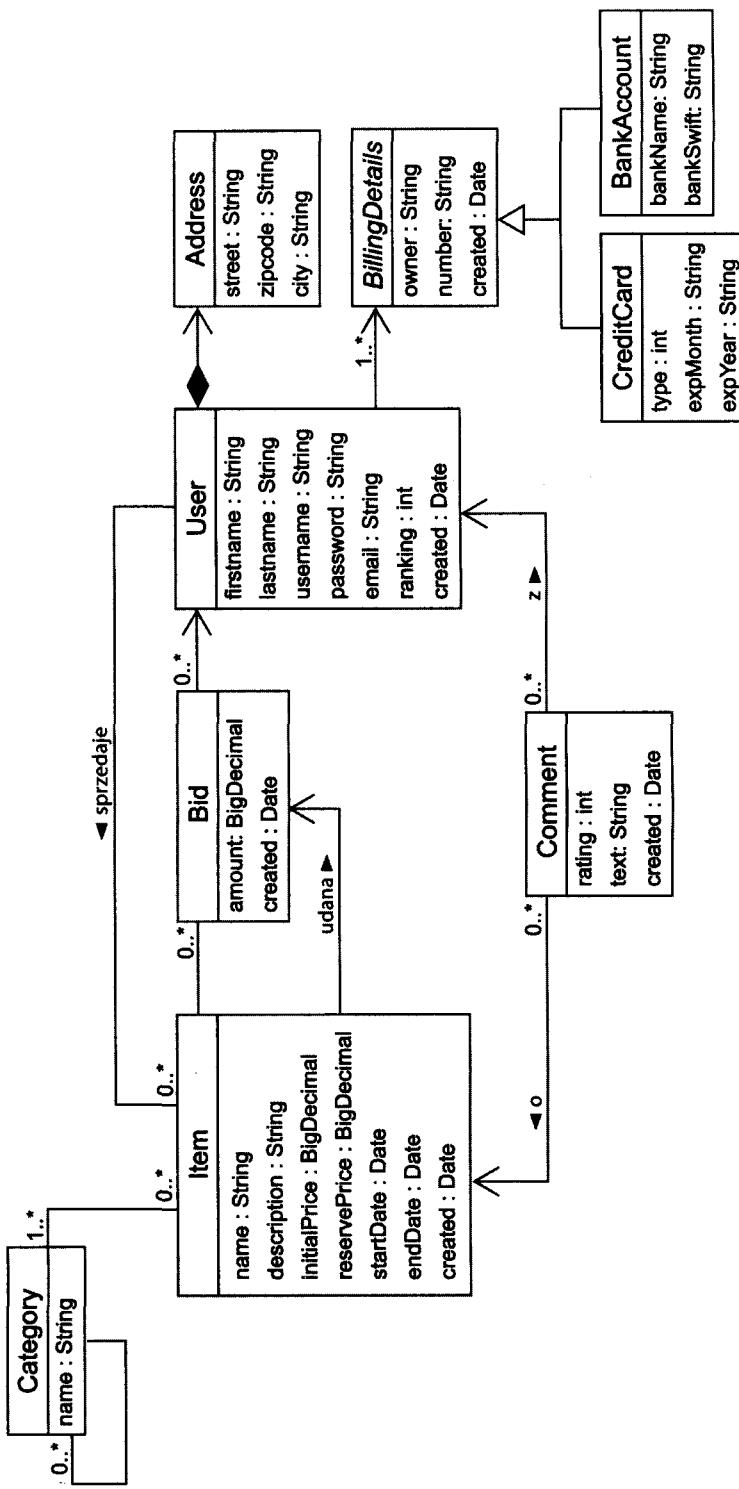
Każdy przedmiot można wystawić tylko raz, więc nie trzeba wyróżniać elementów Item i Auction. Wystarczy pojedynczy element aukcyjny o nazwie Item. Obiekt Bid (oferta) jest związany bezpośrednio z Item. Użytkownicy mogą pisać komentarze (Comment) o innych użytkownikach tylko w kontekście konkretnej aukcji. Istnieje więc powiązanie między Item i Comment. Informacje adresowe zostały zamodelowane jako osobna klasa Address, choć użytkownik może mieć tylko jeden adres. Z użytkownikiem związane są różne rodzaje płatności (BillingDetails). Poszczególne sposoby płatności reprezentują klasy szczegółowe dziedziczące po klasie abstrakcyjnej, co umożliwia przyszłe rozszerzanie systemu.

Kategorię (Category) można zagnieździć wewnętrz innej kategorii. Wyraża to asocjacja rekurencyjna od kategorii do kategorii. Pojedyncza kategoria może posiadać wiele kategorii podrzędnych, ale tylko jedną kategorię nadrzędną. Każdy przedmiot przynależy przynajmniej do jednej kategorii.

Elementy modelu dziedzinowego powinny określać stan i zachowanie. Przykładowo, element User powinien definiować nazwisko użytkownika i jego adres, a także logikę wymaganą do wyliczenia kosztu przesyłki wygranej przedmiotu (do tego konkretnego odbiorcy). Przedstawiony model dziedzinowy jest bogatym modelem obiektowym ze złożonymi asocjacjami, interakcjami i związkami dziedziczenia. Szczegółowy opis różnych technik tworzenia modeli dziedzinowych dla obiektowych aplikacji biznesowych zawierają między innymi książki *Patterns of Enterprise Application Architecture* [Fowler 2003] i *Domain-Driven Design* [Evans 2004].

W niniejszej książce niewiele powiemy na temat reguł biznesowych i zachowania modelu biznesowego. Nie dlatego, że uważamy te aspekty za nieistotne, ale raczej dlatego, że nie mają one bezpośredniego związku z problemem trwałości. Stan elementów jest trwałym. Z tego powodu w opisach skupimy się na reprezentacji stanu w modelu dziedzinowym. Na przykład w książce nie będziemy się interesować, w jaki sposób liczyć podatek od sprzedaży oraz w jaki sposób potwierdzać nowe konta użytkowników. Zajmiemy się raczej związkami między użytkownikami, sprzedawanymi przedmiotami i wszystkim, co czyni te elementy trwałymi.

Po określeniu modelu dziedzinowego możemy przystąpić do jego implementacji w Javie. Rozważmy jednak wcześniej kilka istotnych kwestii.



Rysunek 3.2. Klasę trwałe modelu obiektów systemu CaveatEmptor i związki między nimi

Odpowiedzi na pytania

Czy można korzystać z ORM bez modelu dziedzinowego? Chcielibyśmy podkreślić, iż trwałość obiektów wykorzystująca pełną postać ORM najlepiej nadaje się dla aplikacji bazujących na bogatym modelu dziedzinowym. Jeśli aplikacja nie implementuje złożonych reguł biznesowych lub zaawansowanych interakcji między encjami (lub też zawiera tylko kilka encji), model dziedzinowy niekoniecznie jest potrzebny. Wiele prostych i kilka nieco bardziej skomplikowanych problemów doskonale sprawdza się w rozwiązaniach bazujących na tabelach, w których aplikację projektuje się na podstawie modelu tabel bazy danych zamiast na modelu obiektowym. Często w uproszczonym modelu stosuje się logikę wykonywaną przez bazę danych (procedury zapamiętane). Im bardziej złożony i ekspresyjny model dziedzinowy, tym więcej zalet stosowania Hibernate. Jego zalety objawiają się przy bardzo złożonej trwałości obiektowo-relacyjnej.

3.2. Implementacja modelu dziedzinowego

Gdy implementuje się model dziedzinowy w Javie, należy rozważyć kilka kwestii — na przykład tę, w jaki sposób oddzielić kwestie biznesowe od kwestii ogólnych (na przykład transakcji lub nawet trwałości?). Jakiego rodzaju trwałość jest potrzebna: czy oczekujemy trwałości automatycznej czy przezroczystej? Czy do jego osiągnięcia wymagane jest użycie konkretnego modelu programistycznego? W podrozdziale zastanowimy się nad tymi zagadnieniami i sposobami ich rozwiązywania w kontekście typowej aplikacji stosującej Hibernate.

Zacznijmy od kwestii, z którą musi sobie poradzić każda implementacja — separacji zadań. Implementacja modelu dziedzinowego jest najczęściej centralizowanym komponentem organizującym — korzysta się z niej wielokrotnie, pisząc nową funkcjonalność aplikacji. Z tego powodu warto zabezpieczyć się przed przesiąknięciem do implementacji modelu dziedzinowego innych zadań niż aspekty biznesowe.

3.2.1. Kwestia przesiąkania zadań

Implementacja modelu dziedzinowego jest tak istotnym fragmentem kodu, że nie powinna zależeć od innych interfejsów programistycznych Javy. Przykładowo, kod modelu dziedzinowego nie powinien zajmować się wyszukiwaniami JNDI lub wywoływaniem bazy danych za pomocą JDBC. Dzięki temu model dziedzinowy będzie można zastosować niemal wszędzie. Co ważniejsze, ułatwi on tworzenie testów jednostkowych dla modelu dziedzinowego (na przykład testów JUnit) poza serwerem aplikacji i jakimkolwiek innym środowiskiem zarządzanym.

Mówiąc inaczej, model dziedzinowy powinien zajmować się wyłącznie modelowaniem dziedziny biznesowej. Pojawiają się niestety inne kwestie, na przykład trwałość, zarządzanie transakcjami i autoryzacja. Nie warto umieszczać tych ogólnych zadań w klasach implementujących model dziedzinowy. Gdy wymienione aspekty jednak pojawiają się w klasach modelu dziedzinowego, mówimy wtedy o przesiąkaniu zadań.

Standard EJB stara się rozwiązań problem przesyłania. Gdy model dziedzinowy zaimplementuje się przy użyciu ziarenek encyjnych, kontener zajmie się za nas niektórymi zadaniami (lub przynajmniej przekaże je do określenia w deskryptorze wdrożenia). Kontener EJB zapobiega przesyłaniu niektórych zadań ogólnych, używając **przechwytywania (intercepcji)**. EJB jest **komponentem zarządzanym** zawsze wykonywanym wewnątrz kontenera EJB. Kontener przechwytuje wywołania kierowane do ziarenek i wykonuje je we własnym zakresie, na przykład zlecając ich wykonanie mechanizmom CMP zajmującym się trwałością. To podejście umożliwia kontenerowi implementację predefiniowanych zadań ogólnych — bezpieczeństwa, współbieżności, trwałości, transakcji i zdalności — w ogólny sposób.

Niestety, specyfikacja EJB wprowadza wiele reguł i ograniczeń co do tworzenia modelu dziedzinowego. W zasadzie można w tym momencie mówić o przesyłaniu zadań — w tym momencie elementem powodującym przesyłanie jest kontener! Hibernate nie jest serwerem aplikacji i nie implementuje wszystkich ogólnych zadań wymienionych w specyfikacji EJB. Stanowi rozwiązanie jedynie dla jednego z zadań ogólnych — trwałości danych. Gdy aplikacja wymaga deklaratywnego określania bezpieczeństwa i zarządzania transakcjami, warto korzystać z modelu dziedzinowego poprzez ziarenka sesyjne, wykorzystując tym samym implementację tych zadań zawartą w kontenerze EJB. Hibernate stosuje się często w połączeniu z dobrze znany wzorcem J2EE nazywanym fasadą sesyjną.

Istnieje mnóstwo dyskusji na temat trwałości. Tematem tym zajmuje się zarówno Hibernate, jak i ziarenko encyjne EJB. Hibernate oferuje jednak coś, czego nie ma konkurent — **trwałość przezroczystą**.

3.2.2. Trwałość automatyczna i przezroczysta

Mechanizm CMP serwera aplikacji implementuje **trwałość automatyczną**. Zajmuje się nudnymi szczegółami obsługi klas ResultSet i PreparedStatement z JDBC. Podobnie czyni Hibernate, ale jest w tej kwestii dużo bardziej zaawansowany. Co więcej, Hibernate wykonuje trwałość w taki sposób, iż jest ona **przezroczysta** (niewidoczna) dla modelu dziedzinowego.

Słowo „przezroczysty” oznacza, że istnieje pełna separacja klas trwałych modelu dziedzinowego od logiki trwałości. Odbywa się ona w taki sposób, by klasy trwałe nie były świadome i nie posiadały żadnych zależności związanych z mechanizmami trwałości danych.

Przedstawiona klasa Item nie ma żadnych zależności na poziomie kodu związanych z interfejsem programistycznym Hibernate. Oto kilka dodatkowych zagadnień:

- ◆ Hibernate nie wymusza stosowania żadnych specjalnych klas bazowych i interfejsów dziedziczonych lub implementowanych przez klasy trwałe. Nie istnieją również specjalne klasy dotyczące implementacji właściwości i asocjacji. Trwałość przezroczysta zwiększa czytelność kodu.
- ◆ Nic nie stoi na przeszkodzie, by klasy trwałe stosować w kontekście innym niż trwałości, na przykład w testach jednostkowych lub kodzie interfejsu

użytkownika. Łatwość testowania to jeden z kluczowych elementów dla aplikacji stosujących bogaty model dziedzinowy.

- ◆ W systemie z trwałością przezroczystą obiekty nie są świadome rzeczywistej organizacji danych w bazie danych. Co więcej, nie wiedzą nawet, że są zapamiętywane. Cała obsługa trwałości znajduje się w osobnym interfejsie **menedżera trwałości** — w Hibernate są to w zasadzie dwa interfejsy: Session i Query.

Trwałość przezroczysta zwiększa przenośność kodu. Bez istnienia specjalnych interfejsów klasy trwałe łatwo oddzielić od konkretnego rozwiązania trwałości. Logikę biznesową można bez problemów zastosować w dowolnym innym kontekście aplikacyjnym. Również nic nie stoi na przeszkodzie, by zastąpić jeden mechanizm trwałości innym.

Analizując przedstawioną definicję trwałości przezroczystej, można dojść do wniosku, że niektóre nieautomatyczne warstwy trwałości są przezroczyste (na przykład wzorzec DAO), ponieważ oddzielają kod trwałości, stosując abstrakcyjne interfejsy programistyczne. Logika biznesowa korzysta z prostych klas Javy bez żadnych zależności. Z drugiej strony niektóre automatyczne warstwy trwałości (włączając w to ziarenka encyjne i część rozwiązań ORM) nie są przezroczyste, bo wymagają specjalnych interfejsów lub narzucają pewien model programowania.

Wydaje nam się, że przezroczystość powinna być wymogiem. Powinna stanowić jeden z głównych celów dowolnego rozwiązania ORM. Niestety, żadne automatyczne rozwiązanie trwałości nie jest w 100% przezroczyste — każda automatyczna warstwa trwałości, również Hibernate, wprowadza pewne wymagania dotyczące klas trwałych. Hibernate wymaga na przykład, by właściwości będące kolekcjami były typu interfejsowego, czyli java.util.Set lub java.util.List, zamiast typu konkretnej implementacji, na przykład java.util.HashSet (w zasadzie jest to dobra praktyka programistyczna). Powody takiego stanu rzeczy opisujemy w dodatku B.

Wskazaliśmy, dlaczego mechanizm trwałości powinien mieć minimalny wpływ na sposób implementacji modelu dziedzinowego oraz że obecnie powinno się wymagać trwałości automatycznej i przezroczystej. EJB nie jest przezroczyste, więc jaki model programistyczny należy zastosować? Czy w ogóle potrzebny jest specjalny model programistyczny? W teorii: nie; w praktyce warto rygorystycznie trzymać się jednego modelu programistycznego uznanego przez społeczność programistów języka Java. Przedstawmy ten model i opiszmy jego współpracę z Hibernate.

3.2.3. Tworzenie klas POJO

Programiści odkryli, że ziarenka encyjne są zmudne w tworzeniu, nienaturalne i zmniejszają produktywność. Jako ich przeciwieństwo zaczęli wskazywać klasy POJO (*Plain Old Java Objects*), czyli powrót do podstaw przez ponowne ożywienie komponentów JavaBeans znanych z kreowania graficznych interfejsów użytko-

kownika i zastosowanie ich w warstwie biznesowej (obecnie większość programistów wymiennie stosuje pojęcia JavaBean i POJO)¹.

Hibernate działa najlepiej w połączeniu z modelem biznesowym implementowanym jako POJO. Kilka wymagań nakładanych przez Hibernate na implementację modelu dziedzinowego w POJO jest jednocześnie zalecanymi praktykami w samym modelu programistycznym POJO. Większość klas POJO jest zgodna z Hibernate bez żadnych przeróbek. Przedstawiany w książce model programistyczny jest mieszanką szczegółów specyfikacji JavaBeans, najlepszych praktyk POJO i wymagań Hibernate. POJO deklaruje **metody biznesowe**, które definiują zachowanie oraz **właściwości**, które reprezentują stan. Niektóre właściwości reprezentują asocjacje do innych obiektów POJO.

Listing 3.1 przedstawia prostą klasę POJO stanowiącą implementację encji User z modelu dziedzinowego.

Listing 3.1. Implementacja klasy User zgodna z zasadami POJO

```
public class User implements Serializable {  
    private String username;  
    private Address address;  
  
    public User() {} ← ❶ Konstruktor klasy  
  
    public String getUsername() {  
        return username;  
    }  
  
    public void setUsername(String username) {  
        this.address = address;  
    }  
  
    public Address getAddress() {  
        return address;  
    }  
  
    public void setAddress(Address address) {  
        this.address = address;  
    }  
  
    public MonetaryAmount calcShippingCosts(Address from Location) {  
        ...  
    } ← ❷ Metoda biznesowa  
}
```

¹ POJO czasem pisze się również jako Plain Ordinary Java Objects. Termin ten został wymyślony w 2002 roku przez Martina Fowlera, Rebbeccę Parsons i Josha Mackenzine — *przyp. aut.*

- ❶ Hibernate nie wymaga, by klasy trwałe implementowały interfejs Serializable. Jeśli jednak obiekty mają być przechowywane w HttpSession lub przekazywane jako wartość w RMI, serializacja okazuje się konieczna. (Jest wielce prawdopodobne, że okaże się potrzebna w aplikacji Hibernate.)
- ❷ W odróżnieniu od specyfikacji JavaBeans, która nie wymaga żadnej konkretnej wersji konstruktora, Hibernate oczekuje konstruktora bezparametrowego w każdej klasie trwałej. Hibernate tworzy egzemplarze klas trwałych metodą Constructor.newInstance() znajdującą się w interfejsie programowym Reflection API. Konstruktor nie musi być publiczny, ale powinien być przynajmniej widoczny w pakiecie, jeśli do optymalizacji wydajności chce się wykorzystać pośredników generowanych w czasie działania aplikacji (patrz rozdział 4.).
- ❸ Właściwości POJO implementują atrybuty encji biznesowych — na przykład nazwę użytkownika dla User. Właściwości są najczęściej zmiennymi egzemplarza i zawierają również metody dostępowe (metodę do pobierania wartości zmiennej i metodę do jej ustawiania). Mówią się często odpowiednio o metodzie **pobierającej** i **ustawiającej**. Przykładowa klasa POJO implementuje wymienione metody dla zmiennych egzemplarza username i address.

Specyfikacja JavaBean określa zasady nazewnictwa dla metod dostępowych. Dzięki nim odpowiednio napisane narzędzia, na przykład Hibernate, potrafią poznać i wykorzystać wspomniane metody. Metoda pobierająca zaczyna się od słowa get, po której występuje nazwa zmiennej (pierwszy znak nazwy to wielka litera). Metoda ustawiająca zaczyna się od słowa set i nazwy. Dodatkowo metody pobierające zwracające zmienną logiczną mogą zamiast słowa get stosować is.

Hibernate nie wymaga, by metody dostępowe były publiczne. Bez problemów potrafi korzystać z metod prywatnych do zarządzania właściwościami.

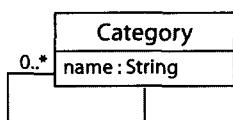
Niektóre metody dostępowe są bardziej złożone od prostego ustawienia lub pobrania wartości (mogą na przykład dokonywać walidacji). Najprostsze wersje spotyka się najczęściej.

- ❹ Klasa POJO definiuje również metodę biznesową wyliczającą koszt dostawy przedmiotu do konkretnego użytkownika (nie przedstawiliśmy jej implementacji).

Po zapoznaniu się z zaletami stosowania klas trwałych POJO w modelu programistycznym przyjrzyjmy się obsłudze asocjacji między tymi klasami.

3.2.4. Implementacja asocjacji POJO

Do wyrażenia związków między klasami POJO stosuje się właściwości. Poruszanie się po grafie obiektów zapewniają metody dostępowe. Rozważmy związek (asocjację) zdefiniowany dla klasy Category. Przedstawia go rysunek 3.3.



Rysunek 3.3.
Diagram klasy Category z asocjacją

We wszystkich przedstawianych diagramach pomijamy atrybuty związane z asocjacjami (`parentCategory` i `childCategories`), gdyż zaciemniałoby to ilustrację. Atrybuty te i metody do ich modyfikacji nazywa się często kodem rusztującym (ang. *scaffolding code*).

Zastosujmy taki kod dla asocjacji **jeden-do-wielu** do samego siebie dotyczącej klasy `Category`.

```

public class Category implements Serializable {
    private String name;
    private Category parentCategory;
    private Set childCategories = new HashSet();

    public Category() { }

    ...
}
  
```

Aby zapewnić dwukierunkową nawigację po asocjacji, potrzebujemy dwóch atrybutów. Atrybut `parentCategory` implementuje **zakończenie z pojedynczą wartością**, więc jest typu `Category`. Atrybut `childCategories` implementujący **zakończenie z wieloma wartościami** musi być kolekcją. Wybraliśmy kolekcję `Set`, ponieważ duplikaty nie są dopuszczalne. Zainicjalizowaliśmy również zmienną egzemplarza nowym obiektem `HashSet`.

Hibernate wymaga interfejsów dla atrybutów będących kolekcjami, więc trzeba użyć `java.util.Set` zamiast `HashSet`. W momencie działania aplikacji Hibernate otacza egzemplarz `HashSet` egzemplarzem własnej klasy (ta specjalna klasa nie jest widziana przez kod aplikacji). Zaleca się korzystać z interfejsów kolekcji zamiast z konkretnych implementacji, więc wspomniane ograniczenia nie powinno stanowić dużego problemu.

Na razie istnieją pewne zmienne egzemplarza, ale nie ma publicznego interfejsu zapewniającego dostęp do nich z poziomu kodu biznesowego lub kodu Hibernate. Dodajmy kilka metod dostępowych do klasy `Category`.

```

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public Set getChildCategories() {
    return childCategories;
}

public void setChildCategories(Set childCategories) {
    this.childCategories = childCategories;
}
  
```

```

public String getParentCategory() {
    return parentCategory;
}

public void setParentCategory(String parentCategory) {
    this.parentCategory= parentCategory;
}

```

Metody dostępowe muszą być publiczne tylko wtedy, gdy stanowią część zewnętrznego interfejsu klasy trwałej, czyli są używane przez logikę aplikacji.

Podstawowa procedura dodawania kategorii podrzędnej do kategorii głównej wygląda następująco:

```

Category aParent = new Category();
Category aChild = new Category();
aChild.setParentCategory(aParent);
aParent.getChildCategories().add(aChild);

```

Za każdym razem, gdy powstaje asocjacja między kategorią nadzczną i podrzędną, muszą zaistnieć dwie akcje.

- ◆ Należy ustawić parentCategory elementu podrzecnego, kończąc tym samym asocjację między potomkiem a starym rodzicem (zawsze może istnieć tylko jeden rodzic danego potomka).
- ◆ Element podrzędny trzeba dodać do kolekcji childCategories nowego rodzica.

Zarządzane asocjacje w Hibernate

Hibernate nie „zarządza” asocjacjami trwałymi. Jeśli chce się obsłużyć asocjację, trzeba zapisać kod dokładnie tak samo jak w sytuacji bez Hibernate. Gdy asocjacja jest dwukierunkowa, trzeba rozważyć obie strony związku. Modele programistyczne typu ziarenka encyjne EJB wprowadzają związki zarządzane przez kontener. Kontener automatycznie modyfikuje jedną z części związku, gdy druga ulegnie zmianie w skutek działań aplikacji. Jest to jeden z powodów, dla którego ziarenka encyjne nie stosuje się poza kontenerem.

Jeśli kiedykolwiek zaczną pojawiać się problemy z określeniem zachowania asocjacji w Hibernate, zadaj sobie pytanie: „Co bym zrobił, gdybym nie stosował Hibernate?”. Hibernate nie zmienia najczęściej stosowanej semantyki Javy.

Warto do klasy Category dodać metodę pomocniczą, która grupuje wcześniejsze operacje. Pozwoli to uniknąć ewentualnych niedopatrzeń.

```

public void addChildCategory(Category childCategory) {
    if (childCategory == null)
        throw new IllegalArgumentException("Nieokreślona kategoria
potomna!");
    if (childCategory.getParentCategory() != null)

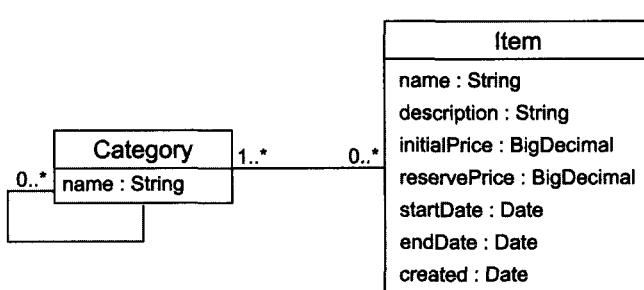
    childCategory.getParentCategory().getChildCategories().remove(childCategory);
    childCategory.setParentCategory(this);
    childCategories.add(childCategory);
}

```

Metoda `addChildCategory()` nie tylko zmniejsza liczbę wierszy kodu związanych z obróbką obiektów `Category`, ale również wymusza poprawną krotność asocjacji. Unika się błędów spowodowanych opuszczeniem jednej z wymaganych akcji. Tego rodzaju grupowanie operacji warto wykonać dla asocjacji, jeśli tylko jest możliwe.

Ponieważ chcemy, by `addChildCategory()` było jedynym dostępnym sposobem zmiany kategorii potomnych, możemy zmienić widoczność metody `setChildCategories()` na prywatną. Hibernate nie zwraca uwagi na widoczność metod, więc można skupić się na jak najlepszym zaprojektowaniu interfejsu programistycznego klas.

Inny rodzaj związku występuje między klasami `Category` i `Item` — dwukierunkowa asocjacja wiele-do-wielu (patrz rysunek 3.4).



Rysunek 3.4.
Klasa `Category` i powiązana z nią klasa `Item`

W przypadku asocjacji wiele-do-wielu obie strony, korzystając z atrybutów będących kolekcjami. Dodajmy nowe atrybuty i metody dostępu do obiektów klasy `Item` z poziomu klasy `Category` (patrz listing 3.2).

Listing 3.2. Kod rusztujący wiążący klasę `Category` z klasą `Item`

```

public class Category {

    ...
    private Set items = new HashSet();
    ...

    public Set getItems() {
        return items;
    }

    public void setItems(Set items){
        this.items = items;
    }
}
  
```

Kod dla klasy `Item` (druga strona asocjacji wiele-do-wielu) jest bardzo podobny do kodu klasy `Category`. Dodaliśmy atrybut kolekcji, standardowe metody dostępowe i metodę upraszczającą zarządzanie związkami (można ją po drobnych modyfikacjach dodać również do klasy `Category`). Kod przedstawia listing 3.3.

Listing 3.3. Kod rusztujący wiążący klasę Item z klasą Category

```
public class Item {
    private String name;
    private String description;
    ...
    private Set categories = new HashSet();
    ...

    public Set getCategories() {
        return categories ;
    }

    public void setCategories(Set categories ){
        this.categories = categories ;
    }

    public void addCategory(Category category) {
        if (category == null)
            throw new IllegalArgumentException("Nieokreślona kategoria!");
        category.getItems().add(this);
        categories.add(category);
    }
}
```

Metoda `addCategory()` klasy `Item` przypomina metodę `addChildCategory()` klasy `Category`. Służy do modyfikacji związku między kategorią i przedmiotem przez kod aplikacji. W celu zwiększenia czytelności nie będziemy przedstawiać tego typu metod pomocniczych w dalszej części przykładów. Zakładamy, że ich dodanie nie sprawi nikomu większych trudności.

Metody pomocnicze obsługi asocjacji to nie jedyny sposób usprawnienia implementacji modelu dziedzinowego. Do metod dostępowych można dodać bardziej wyrafinowaną logikę.

3.2.5. Dodanie logiki do metod dostępowych

Jeden z powodów, dla których lubimy metody dostępowe JavaBean, wynika z tworzonej przez nie hermetyzacji — zmiany w ukrytej wewnętrznej implementacji właściwości można przeprowadzać bez modyfikacji publicznego interfejsu. Dzięki temu wewnętrzna struktura danych klasy — zmienne egzemplarza — może być inna niż projekt bazy danych.

Jeśli baza danych przechowuje imię i nazwisko użytkownika w jednej kolumnie NAME, ale klasa User stosuje właściwości `firstname` i `lastname`, warto w klasie zastosować następujący kod metod dostępowych:

```
public class User {
    private String firstname;
    private String lastname;
    ...
}
```

```
public String getName() {
    return firstname + ' ' + lastname;
}

public void setName(String name) {
    StringTokenizer t = new StringTokenizer(name);
    firstname = t.nextToken();
    lastname = t.nextToken();
}
...
```

Wkrótce wyjaśnimy, dlaczego własny typ Hibernate jest zapewne lepszym rozwiązaniem przedstawionego powyżej problemu. Zawsze warto wiedzieć o istnieniu alternatyw.

Metody dostępowe mogą również zajmować się walidacją. W kolejnym przykładzie metoda `setFirstName()` sprawdza, czy pierwsza litera imienia została zapisana wielką literą.

```
public class User {
    private String firstname;
    ...

    public String getFirstname() P
        return firstname;
    }

    public void setFirstname(String firstname) throws InvalidNameException {
        if (!StringUtil.isCapitalizedName(firstname))
            throw new InvalidNameException (firstname);
        this.firstname = firstname;
    }
    ...
}
```

Hibernate stosuje metody dostępowe do wypełnienia stanu obiektu w momencie pobierania informacji z bazy danych. Czasem **nie** chce się przeprowadzać walidacji, gdy Hibernate inicjalizuje wczytywany obiekt. W takiej sytuacji warto poinformować system, by w sposób bezpośredni zainicjalizował zmienną (w odzworowaniu właściwości w pliku metadanych umieszcza się atrybut `access="field"`), pomijając metodę ustawiającą. Inną kwestią wartą rozważenia jest brudne sprawdzenie. Hibernate automatycznie wykrywa zmiany stanu obiektu w celu synchronizacji bazy danych ze stanem faktycznym. Najczęściej w pełni bezpiecznie można zwracać inny obiekt z metody pobierającej do obiektu przekazywanego przez Hibernate do metody ustawiającej. Hibernate porówna wartości obiektów — a nie ich identyczność — w celu sprawdzenia, czy trwały stan wymaga aktualizacji. Poniższa metoda pobierająca nie spowoduje zastosowania zbędnej aktualizacji SQL:

```
public String getFirstname() {
    return new String(firstname);
}
```

Istnieje jeden niezwykle istotny wyjątek od tej reguły. Hibernate porównuje kolekcje na zasadzie identyczności!

Gdy właściwość jest trwałą kolekcją, należy zwracać metodę pobierającą dokładnie ten sam egzemplarz kolekcji, który został przekazany przez Hibernate do metody ustawiającej. W przeciwnym razie Hibernate uaktualni bazę danych, nawet jeśli nie jest to konieczne, **za każdym razem**, gdy będzie przeprowadzać synchronizację stanu przechowywanego w pamięci z tym w bazie danych. Poniższego kodu należy niemal zawsze unikać w metodach dostępowych.

```
public void setNames(List namesList) {
    names = (String[]) namesList.toArray();
}

public List getNames() {
    return Array.asList(names);
}
```

Okazuje się, że Hibernate nie ogranicza modelu programistycznego JavaBean (POJO), gdy nie jest to konieczne. Nic nie stoi na przeszkodzie, by implementować w metodach dostępowych dowolną logikę (o ile tylko metoda pobierająca i ustawiająca będzie zwracała dokładnie tę samą kolekcję). W uzasadnionych przypadkach można poinformować Hibernate, by korzystał z innej metody dostępu do stanu właściwości (na przykład przez bezpośredni odczyt pola). Zastosowany rodzaj przezroczystości gwarantuje niezależną i elastyczną implementację modelu dziedzinowego.

Po utworzeniu klas trwałych modelu dziedzinowego przystępujemy do definiowania ORM.

3.3. Definicja metadanych odwzorowujących

Narzędzia ORM wymagają podania formatu metadanych, by móc poprawnie odwzorować klasę na tabelę, właściwości na kolumny, asocjacje na klucze obce i typy Javy na typy SQL. Informacje te noszą nazwę **metadanych odwzorowania obiektowo-relacyjnego**. Definiują przekształcenia między różnymi typami danych i reprezentacjami związków.

Programista odpowiada za zdefiniowanie i utrzymanie tych metadanych. W podrozdziale zajmiemy się różnymi podejściami do tworzenia i edycji metadanych.

3.3.1. Metadane w pliku XML

Każdy system ORM powinien dopuszczać stosowanie czytelnego dla człowieka i łatwego w edycji formatu odwzorowania. Stosowanie narzędzia graficznego nie powinno być wymagane. Obecnie najpopularniejszym formatem zapisu metadanych odwzorowania obiektowo-relacyjnego jest dokument XML. Dokumenty te są stosunkowo krótkie, czytelne dla człowieka, łatwo poddają się wersjonowa-

niu i edycji w dowolnym edytorze oraz mogą być modyfikowane w momencie wdrożenia (a nawet działania aplikacji dzięki bibliotekom XML).

Czy metadane bazujące na dokumentach XML to rzeczywiście najlepsze podejście? W społeczności Javy pojawiła się pewna niechęć do XML z powodu jego nadużywania. Każdy szkielet i serwer aplikacji wymaga własnego formatu deskryptorów XML.

Z naszego punktu widzenia istnieją trzy główne powody niechęci do formatu XML:

- ◆ Wielu istniejących formatów metadanych nie zaprojektowano do dużej czytelności i łatwej modyfikacji w sposób ręczny. W szczególności bolesny staje się brak sensownych wartości domyślnych dla atrybutów i wartości, co wymaga znacznie dłuższego opisu, niż mógłby być potrzebny.
- ◆ Rozwiązania stosujące metadane często wykorzystują je nieodpowiednio. Metadane ze swojej natury nie są łatwiejsze w edycji i bardziej elastyczne od zwykłego kodu Javy.
- ◆ Dobre edytory XML, szczególności w zintegrowanych środowiskach programistycznych, niejednokrotnie nie są tak dobre jak środowiska nakierowane tylko na Javę. Często narzędzia nie obsługują deklaracji typu dokumentu (DTD), choć umożliwia ona automatyczne uzupełnianie i validację. Z drugiej strony czasem zdarzają się zbyt ogólne dokumenty DTD, w których każdą deklarację otacza uogólnione „rozszerzenie” elementu „meta”.

Nie istnieje rozwiązanie pozwalające obejść potrzebę stosowania tekstowych plików metadanych w ORM. Hibernate zaprojektowano tak, by uniknąć typowych problemów trapiących twórców metadanych. Gdy brakuje wartości atrybutów, Hibernate stara się określić wartości domyślne, używając interfejsu refleksji dla odwzorowywanej klasy. Co więcej, system zawiera kompletny i w pełni udokumentowany DTD. Obsługa plików XML w zintegrowanych środowiskach programistycznych uległa w ostatnim czasie znaczącej poprawie — nowoczesne IDE dla Javy zawiera dynamiczny validator XML, a nawet automatyczne uzupełnianie składni. Jeśli to nie wystarcza, w rozdziale 9. przedstawimy narzędzia mogące generować pliki XML z odwzorowaniami klas.

Przyjrzyjmy się sposobowi korzystania z metadanych XML w Hibernate. W poprzednim podrozdziale wykonaliśmy klasę Category. Trzeba ją teraz odwzorować na tabelę CATEGORY w bazie danych. W tym celu używamy dokumentu odwzorowania z listingu 3.4.

Listing 3.4. Odwzorowanie XML klasy Category w Hibernate

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
      PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>           • Deklaracja DTD
    <class name="Category">          • Klasa Category odwzorowana
        <table name="CATEGORY">       na tabelę CATEGORY
            <!-- odwzorowania -->
```

ROZDZIAŁ 3.**Odwzorowanie klas trwałości danych**

```

name="org.hibernate.auction.model.Category"
table="CATEGORY"
<id>
    name="id"
    column="CATEGORY_ID"
    type="long"
    <generator class="native"/>
</id>
    ● Odwzorowanie identyfikatora
<property name="name" column="NAME" type="string"/>
    ● Odwzorowanie właściwości
        Namena kolumnę NAME
</class>
</hibernate-mapping>

```

- ❶ Informację o deklaracji DTD odwzorowania Hibernate warto umieszczać w każdym pliku odwzorowania. Zapewnia sprawdzenie poprawności syntaktycznej dokumentu XML.
- ❷ Odwzorowania znajdują się wewnętrz elementu `<hibernate-mapping>`. Można dodać dowolną liczbę odwzorowań klas oraz dodatkowe deklaracje szerzej opisywane w dalszej części książki.
- ❸ System ma za zadanie odwzorować klasę Category (z pakietu `org.hibernate.auction.model`) na tabelę CATEGORY. Każdy wiersz tabeli reprezentuje jeden egzemplarz klasy.
- ❹ Nie omawialiśmy jeszcze dokładnie **identyczności** (lub **tożsamości**) obiektów, więc ten element odwzorowania może zaskakiwać. Ten złożony temat opisujemy dokładniej w podrozdziale 3.4. Aby zrozumieć to odwzorowanie, wystarczy wiedzieć, że każdy rekord tabeli CATEGORY zawiera wartość klucza głównego pasującą do tożsamości obiektu znajdującego się w pamięci. Element odwzorowania o nazwie `<id>` określa szczegółowy definiowanie identyczności obiektów.
- ❺ Właściwość `name` typu `String` zostaje odwzorowana na kolumnę `NAME` bazy danych. Warto zauważyć, że typem dla odwzorowania jest wbudowany typ Hibernate (`string`), a nie typ Javy lub SQL. Warto go traktować jako „odwzorowujący typ danych”. Dokładniej typom tym przyjrzymy się w podrozdziale 6.1.

Celowo w przykładzie pominęliśmy odwzorowanie asocjacji. Jest bardziej złożone, więc zajmiemy się nim dopiero w podrozdziale 3.7.

Wypróbuj

Uruchom **Hibernate** z pierwszą klasą trwałą. Po napisaniu kodu POJO dla klasy Category i zapisaniu odwzorowania Hibernate w pliku XML można uruchomić system ORM i wykonać dowolne operacje. Przedstawiony wcześniej kod POJO dla Category nie był kompletny. Trzeba dodać właściwość o nazwie `id` i typie `java.lang.Long` oraz dodać dla niej metody dostępowe, by włączyć obsługę identyczności obiektów opisy-

waną w dalszej części rozdziału. Utworzenie schematu bazy danych z tabelą dla tak prostej klasy nie powinno dla nikogo stanowić dużego wyzwania. Obserwuj dziennik operacji aplikacji, by potwierdzić poprawne uruchomienie i utworzenie obiektu SessionFactory na podstawie obiektu Configuration (patrz rozdział 2.).

Jeśli chcesz przetestować trwałość, sprawdź metody save(), load() i delete() obiektu Session otrzymanego z SessionFactory. Pamiętaj o zarządzaniu transakcjami. Najprościej pobrać nowy obiekt Transaction wywołaniem Session.beginTransaction() i zatwierdzić go metodą commit() po wykonaniu wszystkich operacji. Podrozdział 2.1 zawiera przykładowy kod testowania trwałości.

Choć możliwa jest deklaracja w jednym pliku odwzorowań dotyczących wielu klas (elementy <class>), zaleca się stosować jeden plik odwzorowania na każdą klasę. Co więcej, takiego podejścia oczekują niektóre narzędzia. Najczęściej plikowi nadaje się tę samą nazwę co klasie trwałej i dołącza do niej przyrostek *hbm*. Powstaje nazwa typu *Category.hmb.xml*.

Omówmy podstawowe właściwości klasy i odwzorowania w Hibernate. Pamiętaj jednak, że temat asocjacji w plikach odwzorowań poruszamy dopiero w dalszej części rozdziału.

3.3.2. Podstawowe odwzorowania właściwości i klas

Typowe odwzorowanie właściwości Hibernate definiuje nazwę właściwości Java-Beans, nazwę kolumny bazy danych i nazwę typu Hibernate. Innymi słowy, określa sposób odwzorowania właściwości klasy na kolumnę tabeli. Podstawowa deklaracja ma wiele odmian i rozszerza się na wiele sposobów. Często można pominąć nazwę typu. Jeśli właściwość description jest typu java.lang.String, Hibernate domyślnie skorzysta z wbudowanego w siebie typu string (rozdział 6. dokładnie omawia system typów Hibernate). Gdy samemu nie określi się typu, Hiernate postara się go ustalić, stosując refleksję. Dwa poniższe wpisy są sobie równoważne:

```
<property name="description" column="DESCRIPTION" type="string"/>  
<property name="description" column="DESCRIPTION"/>
```

Co więcej, jeśli nazwa kolumny jest taka sama jak nazwa właściwości (bez uwzględniania wielkości liter), ją również można pominąć. (To właśnie jest doskonały przykład wspomnianych wcześniej sensownych wartości domyślnych.)

Niektóre sytuacje wymagają użycia elementu <column> zamiast atrybutu column. Element zapewnia większą elastyczność — ma więcej atrybutów opcjonalnych i może występować kilkukrotnie. Dwa poniższe wpisy są sobie równoważne:

```
<property name="description" column="DESCRIPTION" />  
  
<property name="description" type="string">  
    <column name="DESCRIPTION"/>  
</property>
```

Element <property> (a w szczególności element <column>) definiuje kilka atrybutów związanych głównie z automatycznym tworzeniem schematu bazy danych. Jeśli nie używasz narzędzia hbm2ddl (patrz podrozdział 9.2) do generowania schematu bazy danych, możesz je pominąć. Nawet bez korzystania z automatycznej generacji warto jednak dodać do definicji przynajmniej atrybut not-null, gdyż w ten sposób Hibernate potrafi od razu zgłosić błąd w wartości właściwości bez wykonywania polecenia SQL.

```
<property name="initialPrice" column="INITIAL_PRICE" not-null="true"/>
```

Wykrywanie niepoprawnych wartości null przydaje się do generowania odpowiednich wyjątków w trakcie projektowania aplikacji. Jego celem nie jest zastąpienie pełnej walidacji danych, która wykracza poza trwałość oferowaną przez Hibernate.

Niektórym właściwościom w ogóle nie odpowiada kolumna bazy danych. W szczególności właściwość **wyliczana** bierze swą wartość z wyrażenia SQL.

Właściwości wyliczane

Wartość właściwości wyliczanej system liczy w trakcie swojej pracy na podstawie wskazanego wyrażenia. Wyrażenie definiuje atrybut formula. Na przykład obiekt może zawierać właściwość totalIncludingTax choć w bazie danych nie istnieje ani jedna kolumna z pełną ceną.

```
<property name="totalIncludingTax" formula="TOTAL + TAX_RATE * TOTAL"
          type="big_decimal"/>
```

Hibernate określa wartość wyrażenia przy każdym pobieraniu encji z bazy danych. Właściwość nie zawiera atrybutu column (ani elementu column). Nie pojawia się też w poleceniach INSERT lub UPDATE. Istnieje tylko w poleceniach SELECT. Wyrażenie może odnosić się do wielu kolumn bazy danych, wywoływać funkcję SQL lub wykonywać podzapytanie SQL.

Poniższa przykładowa właściwość wyliczana klasy Item używa powiązanego podzapytania w celu wyliczenia średniej wartości wszystkich ofert dla wskazanego przedmiotu:

```
<property name="averageBidAmount"
          formula="( select AVG(b.AMOUNT) from BID b where b.ITEM_ID = ITEM_ID )"
          type="big_decimal"/>
```

Zauważ, że skrócone nazwy kolumn odnoszą się do kolumn tabeli związanej z klasą, do której przynależy właściwość wyliczana.

Jak wspomnieliśmy wcześniej, Hibernate nie wymaga metod dostępowych dla klas POJO, jeśli tylko zdefiniuje się nową strategię dostępu do właściwości.

Strategie dostępu do właściwości

Atrybut access określa, w jaki sposób Hibernate powinien pobierać i ustawać wartości właściwości obiektów POJO. Domyślna strategia o nazwie property stosuje metody dostępowe (czyli metodę ustawiającą i pobierającą wartość). Przedstawione odwzorowanie właściwości nie wymaga pary metod dostępowych:

```
<property name="name"  
         column="NAME"  
         type="string"  
         access="field"/>
```

Dostęp do właściwości przy użyciu metod dostępowych społeczność użytkowników Hibernate uważa za najlepsze rozwiązanie. Zapewnia dodatkowy poziom abstrakcji między modelem dziedzinowym w języku Java a modelem danych, który może wykraczać znacznie poza rozwiązania oferowane przez Hibernate. Właściwości są bardziej elastyczne. Definicje metod dostępowych łatwo zmieniać w podklasach.

Jeśli ani metody dostępowe, ani bezpośredni dostęp do zmiennej nie są odpowiednie, warto zdefiniować własną strategię dostępu do właściwości, implementując interfejs net.sf.hibernate.property.PropertyAccessor i podając jego nazwę w atrybucie access.

Sterowanie wstawieniami i aktualizacjami

Dla właściwości odwzorowywanych na kolumny można określić, czy powinny wystąpić w poleceniu INSERT (atribut insert) i czy powinny wystąpić w poleceniu UPDATE (atribut update).

Stan zdefiniowanej poniżej właściwości nigdy nie jest zapisywany do bazy danych.

```
<property name="name"  
         column="NAME"  
         type="string"  
         insert="false"  
         update="false"/>
```

Właściwość name obiektu JavaBean staje się **niezmienna** — jest odczytywana z bazy danych, ale w żaden sposób nie uda się jej zmienić. Jeśli cała klasa ma być niezmienna, warto użyć atrybutu immutable="false" w jej odwzorowaniu.

Dodatkowo atrybut dynamic-insert informuje Hibernate, czy dodawać wartość niezmodyfikowanej właściwości do polecenia INSERT. Podobnie atrybut dynamic-update informuje, czy dodawać wartość niezmodyfikowanej właściwości do polecenia UPDATE.

```
<class name="org.hibernate.auction.model.User"  
      dynamic-insert="true"  
      dynamic-update="true">  
    ...  
</class>
```

Oba ustawienia dotyczą całej klasy. Włączenie ich spowoduje generowanie niektórych fragmentów kodu SQL w trakcie pracy systemu. Nie będzie on umieszczany w buforze kodu SQL w momencie uruchamiania systemu. Utrata wydajności jest najczęściej niewielka. Co więcej, opuszczenie niektórych zbędnych kolumn we wstawianiu (a szczególnie w aktualizacji) potrafi niejednokrotnie podnieść wydajność, jeśli tabela składa się z wielu kolumn.

Stosowanie identyfikatorów SQL z apostrofami

Domyślnie Hibernate nie otacza nazw kolumn i tabel specjalnymi apostrofami w trakcie generowania kodu SQL. Czyni to kod bardziej czytelny i ma tę zaletę, że większość systemów bazodanowych nie zwraca uwagi na wielkość liter w nazwach, jeśli nie są one zawarte w apostrofach. Od czasu do czasu, szczególnie w starszych bazach danych, spotyka się w identyfikatorach nietypowe znaki i spacje. W takiej sytuacji, lub gdy zachodzi potrzeba wymuszenia sprawdzania wielkości liter, warto posłużyć się apostrofami.

Jeśli tabelę lub nazwę kolumny umieści się w dokumencie odwzorowania między specjalnymi znakami ` , Hibernate będzie za każdym w instrukcjach z tym identyfikatorem stosował apostrofy. Przedstawiona poniżej deklaracja właściwości wymusza umieszczenie nazwy kolumny w apostrofach "Opis przedmiotu". Hibernate poprawnie obsługuje serwer Microsoft SQL wymagający zapisu [Opis przedmiotu] i serwer MySQL wymagający zapisu `Opis przedmiotu` .

```
<property name="description" column="`Opis przedmiotu`"/>
```

Nie istnieje żaden inny sposób, poza wskazanym powyżej, by wymusić na Hibernate stosowanie apostrofów dla wszystkich identyfikatorów.

Konwencje nazewnictwa

Niejednokrotnie spotyka się firmy narzucające bardzo rygorystyczne zasady nazewnictwa tabel i kolumn baz danych. Hibernate umożliwia automatyczne wymuszanie stosowania odpowiednich konwencji nazewnictwa.

Przypuśćmy, że wszystkie tabele przykładowego CaveatEmptor muszą stosować wzorzec CE_<nazwa tabeli>.

Jedno z rozwiązań polega na ręcznym określaniu atrybutu table we wszystkich elementach <class> i kolekcjach zawartych w plikach odwzorowań. To podejście jest czasochłonne i łatwo coś przeoczyć. Hibernate dostarcza interfejs NamingStrategy, którego implementację przedstawia listing 3.5.

Listing 3.5. Implementacja interfejsu NamingStrategy

```
public class CENamingStrategy implements NamingStrategy {

    public String classNameToTableName(String className) {
        return tableName(
            StringHelper.unqualify(className).toUpperCase() );
    }

    public String propertyToColumnName(String propertyName) {
        return propertyName.toUpperCase();
    }

    public String tableName(String tableName) {
        return "CE_" + tableName;
    }

    public String columnName(String columnName) {
```

```
        return columnName;
    }

    public String propertyToTableName(String className, String propertyName)
    {
        return classToTableName(className) + '_' +
            propertyToColumnName(propertyName);
    }

}
```

Metoda `classToTableName()` zostaje wywołana tylko wtedy, gdy odwzorowanie `<class>` nie określa jawnie nazwy tabeli. Hibernate wywołuje metodę `propertyToTableName()`, jeśli właściwość nie zawiera jawnie podanej nazwy kolumny. Metody `tableName()` i `columnName()` są wywoływanie, gdy nazwa została określona jawnie.

Po włączeniu strategii nazewnictwa `CENamingStrategy` poniższa deklaracja odwzorowania klasy:

```
<class name="BankAccount">
```

spowoduje stosowanie nazwy `CE_BANKACCOUNT` jako nazwy tabeli. System wywołał metodę `classToTableName()`, podając jako argument pełną nazwę klasy.

Jeśli jednak w odwzorowaniu pojawi się nazwa tabeli:

```
<class name="BankAccount" table="BANK_ACCOUNT">
```

wtedy docelową nazwą tabeli będzie `CE_BANK_ACCOUNT`. W tym przypadku oryginalna nazwa tabeli była argumentem metody `tableName()`.

Najważniejszą cechą `NamingStrategy` jest potencjalnie dynamiczne zachowanie. By uaktywnić konkretną strategię nazewnictwa, wystarczy jej egzemplarz przekazać w momencie uruchamiania systemu do obiektu klasy `Configuration`.

```
Configuration cfg = new Configuration();
cfg.setNamingStrategy( new CENamingStrategy() );
SessionFactory sessionFactory = cfg.configure().buildSessionFactory();
```

W ten sposób można korzystać z wielu egzemplarzy `SessionFactory` bazujących na tych samych dokumentach odwzorowania, ale stosujących różne strategie nazewnictwa. Podejście to okazuje się wyjątkowo przydatne w instalacjach z wieloma klientami, w których każdy klient używa własnych, unikatowych nazw tabel (choć stosujących ten sam model danych).

Lepszym sposobem obsługi przedstawionego wymagania jest koncepcja **schematu SQL** (przypominającego przestrzeń nazw).

Schematy SQL

Domyślny schemat określa opcja konfiguracyjna `hibernate.default_schema`. Alternatywne podejście polega na określaniu schematu w dokumencie odwzorowania. Schemat można określić dla konkretnej klasy lub odwzorowania kolekcji.

ROZDZIAŁ 3.

Odwzorowanie klas trwałości danych

```
<hibernate-mapping>
    <class
        name="org.hibernate.auction.model.Category"
        table="CATEGORY"
        schema="AUCTION">
        ...
    </class>
</hibernate-mapping>
```

Nic nie stoi na przeszkodzie, by schemat zdefiniować dla całego dokumentu.

```
<hibernate-mapping
    default-schema="AUCTION">
    ...
</hibernate-mapping>
```

To nie jedyny powód występowania elementu `<hibernate-mapping>`.

Deklaracja nazw klas

Wszystkie klasy trwale aplikacji CaveatEmptor są zadeklarowane w pakiecie `org.hibernate.auction.model`. Wpiswanie nazwy pakietu za każdym razem, gdy podaje nazwę klasy, byłoby wyjątkowo żmudne i nieefektywne.

Powróćmy do odwzorowania dotyczącego klasy Category (plik `Category.hbm.xml`).

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://ibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
    <class
        name="org.hibernate.auction.model.Category"
        table="CATEGORY">
        ...
    </class>
</hibernate-mapping>
```

Nie chcemy powtarzać pełnej nazwy pakietu za każdym razem, gdy trzeba podać nazwę tej lub innej klasy w asocjacji, podklasie lub odwzorowaniu komponentu, więc wykorzystamy atrybut package.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://ibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping
package="org.hibernate.auction.model">
    <class
        name="Category"
        table="CATEGORY">
        ...
    </class>
</hibernate-mapping>
```

Teraz wszystkie niepełne nazwy klas pojawiające się w dokumencie odwzorowania zostaną uzupełnione zadeklarowaną nazwą pakietu. Zakładamy stosowanie atrybutu package we wszystkich przykładach odwzorowań przedstawianych w książce.

Jeśli tworzenie plików XML ręcznie (choć oczywiście z zastosowaniem automatycznego uzupełniania opartego o DTD) wydaje się zbyt trudne, warto skorzystać z programowania zorientowanego na atrybuty. Pliki odwzorowań Hibernate można automatycznie wygenerować z atrybutów osadzonych w plikach kodu źródłowego Javy.

3.3.3. Programowanie zorientowane na atrybuty

Innowacyjny projekt XDoclet wprowadził do Javy notację programowania zorientowanego na atrybuty. Przed JDK 1.5 Java nie obsługiwała adnotacji, więc XDoclet korzystał ze znacznika Javadoc @attribute do określania atrybutów metadanych metody, klasy lub pola. (Napisano nawet książkę na temat XDoclet, *XDoclet in Action* [Walls/Richards, 2004], wydaną przez Manning Publications.)

XDoclet to narzędzie zaimplementowane jako zadanie Ant, które generuje kod i metadane XML jako część procesu komplikacji. Tworzenie dokumentów odwzorowania XML w XDoclet nie jest trudne. Wszystkie informacje potrzebne do odwzorowania umieszcza się bezpośrednio w kodzie źródłowym Javy w specjalnych znacznikach Javadoc (patrz listing 3.6).

Listing 3.6. Zastosowanie XDoclet do określenia metadanych odwzorowania dla właściwości klas Javy

```
/**  
 * Klasa Category modelu dziedzinowego systemu aukcyjnego CaveatEmptor.  
 *  
 * @hibernate.class  
 * table="CATEGORY"  
 */  
public class Category {  
  
    ...  
  
    /**  
     * @hibernate.id  
     * generator-class="native"  
     * column="CATEGORY_ID"  
     */  
    public Long getId() {  
        return id;  
    }  
  
    ...  
  
    /**  
     * @hibernate.property  
     */
```

```
public String getName() {  
    return name;  
}  
...  
}
```

Stosując przedstawioną klasę z adnotacjami i odpowiednie zadanie Ant, możemy automatycznie wygenerować ten sam dokument XML, który został przedstawiony na listingu 3.4.

Wadą XDoclet jest wymóg stosowania dodatkowego kroku budowania projektu. Większość dużych projektów stosuje Ant, więc najczęściej nie stanowi to problemu. Odwzorowania XDoclet dają mniejsze możliwości konfiguracji w momencie wdrażania. Nic jednak nie stoi na przeszkodzie, by przed samym wdrożeniem ręcznie zmodyfikować odpowiednie pliki. Jako ostatnią wadę XDoclet wymienić można brak walidacji znaczników tego systemu w niektórych środowiskach programistycznych. Dwa z nich, JetBrains IntelliJ IDEA i Eclipse, obsługują przynajmniej automatyczne uzupełnianie nazw znaczników. Zastosowanie XDoclet z Hibernate zostało dokładniej opisane w podrozdziale 9.5.

Uwaga

XDoclet nie jest standardowym podejściem do metadanych opisywanych atrybutami. W momencie korzystania z JDK Java w wersji 1.5 możliwe jest korzystanie z oficjalnego rozszerzenia języka zapewniającego definiowanie adnotacji.

Oba opisywane do tej pory rozwiązania, pliki XML i atrybuty XDoclet, zakładają, że wszystkie informacje odwzorowania są znane w momencie wdrażania. Założymy jednak, że nie znamy wszystkich danych przed uruchomieniem aplikacji. Czy można programowo modyfikować metadane odwzorowań w trakcie działania aplikacji?

3.3.4. Modyfikacja metadanych w trakcie działania aplikacji

Niejednokrotnie użytkownika okazuje się możliwość przeglądania, modyfikacji i kreowania nowych odwzorowań przez aplikację już po jej uruchomieniu. Interfejsy programistyczne związane z XML, na przykład DOM, dom4j i JDOM, dopuszczają bezpośrednią edycję dokumentów XML przez działającą aplikację. Można więc programowo modyfikować i tworzyć dokument XML przed jego przekazaniem do obiektu klasy Configuration.

Hibernate idzie o krok dalej i udostępnia model metadanych w momencie konfiguracji uruchomionego systemu. Model zawiera wszystkie informacje określone w dokumentach odwzorowań. Bezpośrednia, programowa edycja modelu bywa potrzebna, szczególnie w aplikacjach dopuszczających uruchamianie rozszerzeń pisanych przez użytkowników.

Poniższy kod dodaje nową właściwość, motto, do istniejącego odwzorowania klasy User.

```
// Pobierz istniejące odwzorowanie dla klasy User do obiektu Configuration.  
PersistentClass userMapping = cfg.getClassMapping(User.class);  
  
// Zdefiniuj nową kolumnę dla tabeli USER.  
Column column = new Column();  
column.setType(Hibernate.STRING);  
column.setName("MOTTO");  
column.setNullable(false);  
column.setUnique(true);  
userMapping.getTable().addColumn(column);  
  
// Otoń kolumnę obiektem klasy Value.  
SimpleValue value = new SimpleValue();  
value.setTable( userMapping.getTable() );  
value.addColumn(column);  
value.setType(Hibernate.STRING);  
  
// Zdefiniuj nową właściwość klasy User.  
Property prop = new Property();  
prop.setValue(value);  
prop.setName("motto");  
userMapping.addProperty(prop);  
  
// Utwórz fabrykę sesji, korzystając z nowego odwzorowania.  
SessionFactory sf = cfg.buildSessionFactory();
```

Obiekt PersistentClass reprezentuje model metadanych dla pojedynczej klasy trwałej. Obiekt pobieramy z klasy Configuration. Klasy Column, SimpleValue i Property dotyczą modelu metadanych Hibernate i są dostępne w pakiecie net.sf.hibernate.mapping. Warto sobie zdawać sprawę, że dodanie nowej właściwości do istniejącego odwzorowania klasy trwałej (jak w przykładzie) jest proste. Programowe utworzenie całego odwzorowania od podstaw jest znacznie bardziej skomplikowane.

Po utworzeniu obiektu SessionFactory jego odwzorowania stają się niezmienne. W zasadzie obiekt ten stosuje wewnętrznie inny model metadanych od tego używanego na etapie konfiguracji. Nie istnieje żaden sposób, by powrócić do oryginalnego obiektu Configuration na podstawie obiektów SessionFactory lub Session. Aplikacja może jednak odczytać model metadanych zawarty w obiekcie SessionFactory, używając metod getClassMetadata() i getCollectionMetadata(). Oto przykład:

```
Category category = ....;  
ClassMetadata meta = sessionFactory.getClassMetadata(Category.class);  
String[] metaPropertyNames = meta.getPropertyNames();  
Object[] propertyValues = meta.getPropertyValues(category);
```

Przedstawiony fragment kodu pobiera nazwy trwałych właściwości klasy Category i ich wartości dla konkretnego egzemplarza klasy. Dzięki temu można pisać bardziej ogólny kod, wykorzystując zdobyte informacje do generowania interfejsu użytkownika lub tworzenia bardziej rozbudowanego dziennika zdarzeń.

Powróćmy do specjalnego elementu odwzorowania pojawiającego się w wielu wcześniejszych przykładach — **odwzorowania właściwości identyfikującej**. Załączymy opis od wyjaśnienia **identyczności obiektów**.

3.4. Idenyczność obiektów

Niezwylkłe ważne jest poprawne zrozumienie różnic między identycznością i równością obiektu, szczególnie jeśli mówi się jeszcze o tożsamości bazodanowej i zarządzaniu nią przez Hibernate. Potrzebujemy wiedzy na temat tych pojęć, by zakończyć odwzorowywanie klasy trwałych CaveatEmptor i ich powiązań w Hibernate.

3.4.1. Idenyczność a równość

Każdy programista języka Java powinien znać różnicę między identycznością i równością obiektu Javy. Idenyczność obiektu badaną za pomocą `==` definiuje maszyna wirtualna Javy. Dwa obiekty uważa się za idenyczne, jeśli znajdują się w tym samym miejscu w pamięci.

Równość obiektów określa klasa, na podstawie której są tworzone, definiując metodę `equals()`. W tym przypadku niejednokrotnie mówi się o równoważności. Równoważność oznacza, że dwa różne (nieidenyczne) obiekty zawierają tę samą wartość. Dwa różne obiekty `String` uważa się za równe, jeśli zawierają tę samą sekwencję znaków, choć same obiekty najczęściej znajdują się w różnych miejscach w pamięci (przypomnijmy, że dla obiektów `String` nie zawsze jest to prawda).

Trwałość znacznie komplikuje sytuację. W trwałości obiektowo-relacyjnej obiekt trwały jest w zasadzie tylko reprezentacją wiersza bazy danych przeniesionego do pamięci. Z tego powodu poza identycznością i równością obiektów Javy mówimy jeszcze o tożsamości bazodanowej (lokalizacji w trwałym magazynie danych). Uzyskujemy trzy różne metody identyfikacji obiektów:

- ◆ **Idenyczność obiektów** — obiekty są idenyczne, jeśli zajmują te same miejsce w maszynie wirtualnej Javy. Idenyczność sprawdza się operatorem `==`.
- ◆ **Równość obiektów** — obiekty są równe, jeśli zawierają tę samą wartość sprawdzaną metodą `equals(Object o)`. Klassy, które nie przesyłają tej metody dziedziczonej po obiekcie `java.lang.Object`, traktują równość w ten sam sposób co idenyczność obiektów.
- ◆ **Tożsamość bazodanowa** — obiekty przechowywane w relacyjnej bazie danych są idenyczne, jeśli dotyczą tego samego wiersza tabeli, czyli znajdują się w tej samej tabeli i mają ten sam klucz główny.

Warto wiedzieć, w jaki sposób idenyczność bazodanowa przekłada się na idenyczność obiektów w Hibernate.

3.4.2. Tożsamość bazodanowa w Hibernate

Hibernate udostępnia aplikacji tożsamość bazodanową na dwa sposoby:

- ◆ jako wartość właściwości `identyfikującej` trwały egzemplarz,
- ◆ jako wartość zwróconą przez `Session.getIdentifier(Object o)`.

Właściwość identyfikująca jest szczególna — jej wartość odpowiada wartości klucza głównego wiersza bazy danych związanego z obiektem trwałym. Najczęściej właściwości identyfikującej nie przedstawia się na modelu dziedziznowym, bo jest to zagadnienie dotyczące trwałości, a nie problemu biznesowego. W przedstawianych przykładach właściwość zawsze ma nazwę `id`. Jeśli obiekt `myCategory` jest egzemplarzem klasy `Category`, wywołanie `myCategory.getId()` zwraca wartość klucza głównego reprezentującego `myCategory` w bazie danych.

Czy metody dostępowe właściwości identyfikującej powinny być publiczne czy prywatne? Aplikacje niejednokrotnie stosują identyfikatory bazodanowe jako wygodne uchwyty zapewniające dostęp do trwałego egzemplarza nawet poza warstwą trwałości. Aplikacje internetowe często wyświetlają wyniki wyszukiwania jako listę. Gdy użytkownik wybierze konkretną pozycję, aplikacja pobiera wybrany egzemplarz. Do jego wyszukania warto posłużyć się wartością właściwości identyfikującej. To podejście występuje najczęściej w aplikacjach bezpośrednio stosujących interfejs JDBC. Wynika stąd, że warto udostępnić tożsamość bazodanową dzięki metodom dostępowym.

Przedstawiony wywód dotyczy przede wszystkim pobierania wartości identyfikującej. Metodę `setId()` warto uczynić prywatną, by tylko Hibernate mógł nadać jej wartość. Wyjątkiem od tej reguły są tak zwane klucze naturalne, czyli wartości identyfikujące nadawane przez aplikację jeszcze przed trwałym zapisaniem obiektu. Nie są one wtedy generowane przez Hibernate. Opisujemy je dokładniej w kolejnym podrozdziale. Hibernate nie dopuszcza zmiany wartości identyfikującej egzemplarza obiektu trwałego po jej przypisaniu.

Pamiętaj, że część definicji klucza głównego mówi o tym, że nie powinien on nigdy ulegać zmianie. Zaimplementujmy właściwość identyfikującą w klasie `Category`.

```
public class Category {  
    private Long id;  
    ...  
    public Long getId() {  
        return this.id;  
    }  
  
    private void setId(Long id) {  
        this.id = id;  
    }  
    ...  
}
```

Typ właściwości zależy od typu klucza głównego tabeli `CATEGORY` i typu odwzorowującego Hibernate. Wszystkie te informacje określa element `<id>` dokumentu odwzorowania.

```
<class name="Category" table="CATEGORY">  
    <id name="id" column="CATEGORY_ID" type="long">  
        <generator class="native"/>  
    </id>  
    ...  
</class>
```

Właściwość identyfikatora zostaje odwzorowana na klucz główny (kolumnę CATEGORY_ID) tabeli CATEGORY. Typ Hibernate dla tej właściwości to long, co w większości baz danych jest równoznaczne kolumnie typu BIGINT. Typ ten odpowiada wartości generowanej przez wbudowany w Hibernate generator native (w kolejnym podrozdziale szczegółowo zajmiemy się strategiami generowania identyfikatorów). Poza operacjami sprawdzającymi identyczność obiektów Javy (`a == b`) i równość obiektów (`a.equals(b)`) dołącza testowanie tożsamości bazodanowej (`a.getId().equals(b.getId())`).

Alternatywne podejście do obsługi tożsamości bazodanowej polega na braku jawnej implementacji właściwości identyfikującej. Wtedy Hibernate sam zajmuje się zarządzaniem identycznością. W takiej sytuacji w dokumencie odwzorowania pomija się atrybut name elementu id.

```
<id column="CATEGORY_ID">
    <generator class="native"/>
</id>
```

Gdy Hibernate sam wewnętrznie zarządza identyfikatorami, ich wartości dla klas trwałych pobiera się, używając poniższego kodu:

```
Long catId = (Long) session.getIdentifier(category);
```

Podejście to ma poważną wadę — nie można w nim używać Hibernate do wydajnej modyfikacji obiektów odłączonych (patrz podrozdział 4.1.6). Zawsze warto stosować właściwości identyfikujące. Jeśli nie chcesz, by były widoczne przez pozostałą część aplikacji, uzyj metody dostępowe prywatnymi.

Słosowanie identyfikatorów bazodanowych w Hibernate jest proste. Znacznie trudniejszy okazuje się wybór dobrego klucza głównego (a co za tym idzie odpowiedniej strategii jego generowania). Zajmijmy się tym tematem.

3.4.3. Wybór kluczy głównych

Należy poinformować Hibernate o preferowanej metodzie generowania kluczy głównych, ale najpierw trzeba zdefiniować sam **klucz główny**.

Klucz kandydujący to kolumna lub zbiór kolumn jednoznacznie identyfikujący konkretny wiersz tabeli. Klucz kandydujący musi spełniać następujące warunki:

- ◆ jego wartość nigdy nie może być równa null,
- ◆ każdy wiersz posiada inną wartość,
- ◆ wartość dla konkretnego wiersza nigdy się nie zmienia.

Dla każdej tabeli kilka kolumn lub ich kombinacji może spełniać podane właściwości. Jeśli kolumna zawiera tylko jeden atrybut identyfikujący, zazwyczaj jest to **klucz główny**. Jeśli istnieje kilka kluczy kandydujących, należy wybrać jeden z nich (pozostałe klucze kandydujące niewybrane na klucz główny powinny zostać zadeklarowane jako klucze unikalne w schemacie bazy danych). Jeżeli nie istnieją unikatowe kolumny lub ich kombinacje i tym samym nie ma kluczy kandydujących, wtedy tabela z definicji nie jest relacją w świetle modelu relacyjnego (dopuszczającego duplikację wierszy). Należy wtedy raz jeszcze przemyśleć model danych.

Wiele starszych modeli baz danych SQL wykorzystuje naturalne klucze główne. Klucz naturalny najczęściej ma znaczenie biznesowe — jest atrybutem lub zbiorem atrybutów o unikatowych wartościach z punktu widzenia semantyki biznesowej. Przykładami takich kluczy mogą być numery PESEL lub NIP. Rozpoznanie, czy klucz jest kluczem naturalnym, nie nastręcza żadnych trudności: jeśli atrybut klucza kandydującego ma znaczenie poza kontekstem bazodanowym, jest kluczem naturalnym; w przeciwnym razie mamy do czynienia z kluczem wygenerowanym automatycznie.

Doświadczenie pokazuje, że klucze główne w dłuższej perspektywie czasu niemal zawsze sprawiają problemy. Dobry klucz główny zawsze musi być unikatowy, stały i wymagany (nie może mieć wartości null lub być nieznany). Nie wiele atrybutów encji spełnia te wymagania. Niektórych z nich bazy danych SQL nie indeksują wydajnie. Należy się upewnić, że definicja klucza kandydującego z całą pewnością nie zmieni swojej wartości w trakcie życia bazy danych przed jej promocją na klucz główny. Zmiana definicji klucza głównego i wszystkich związanych z nim kluczy obcych nie jest zadaniem łatwym.

Z przedstawionych powodów zalecamy we wszystkich nowych aplikacjach stosować sztuczne identyfikatory (**klucze czysto identyfikacyjne**). Klucze te nie mają żadnego znaczenia biznesowego — są generowane jako unikatowe wartości przez bazę danych lub aplikację. Istnieje kilka dobrze znanych sposobów generowania sztucznych identyfikatorów.

Hibernate ma kilka wbudowanych strategii generowania identyfikatorów. Najbardziej użyteczne wymienia tabela 3.1.

Programista nie jest ograniczony wymienionymi generatorami — zawsze może utworzyć własny, implementując interfejs IdentifierGenerator. Można nawet mieszać różne generatory identyfikatorów dla klas trwały w jednym modelu dziedzinowym. Dla starszych danych zalecamy stosowanie jednego generatora dla wszystkich klas.

Specjalna strategia generowania identyfikatorów assigned nadaje się dla encji z naturalnymi kluczami głównymi. W tej strategii aplikacja sama przypisuje wartości identyfikatorom, ustawiając właściwość identyfikującą obiekt przed jego zapisaniem do bazy danych metodą save(). Strategia ma pewne poważne wady, gdy korzysta się z obiektów odłączanych i trwałości ulotnej (oba pojęcia są dokładnie wyjaśnione w kolejnym rozdziale). Zalecamy omijanie identyfikatorów assigned szerokim łukiem, jeśli tylko można skorzystać z innych rozwiązań. Znacznie łatwiej wprowadzić do aplikacji sztuczny klucz główny generowany przez jedną ze strategii opisanych w tabeli 3.1.

W przypadku korzystania ze starszych danych całe zagadnienie najczęściej się komplikuje. Niejednokrotnie nie ma wyboru i trzeba korzystać z kluczy naturalnych lub, co gorsza, kluczy złożonych (kluczy naturalnych obejmujących kilka kolumn). Ponieważ identyfikatory obejmujące wiele kolumn bywają znacznie bardziej skomplikowane, omówimy je dopiero w podrozdziale 8.3.1.

Kolejny krok to dodanie właściwości identyfikujących do klas aplikacji CaveatEmptor. Czy wszystkie klasy trwałe mają własną tożsamość bazodanową? By odpowiedzieć na to pytanie, musimy dokładniej przedstawić rozróżnianie w Hibernate encji i typów wartości.

Tabela 3.1. Moduły generatorów kluczy wbudowane w Hibernate

Nazwa generatora	Opis
native	Generator native wybiera inny generator, na przykład identity, sequence lub hilo w zależności od możliwości wykorzystywanego systemu bazodanowego.
identity	Generator obsługuje kolumny identyfikujące baz danych DB2, MySQL, MS SQL Server, Sybase, HSQLDB, Informix i HypersonicSQL. Zwracany identyfikator jest typu long, int lub short.
sequence	Generator wykorzystuje sekwencje baz danych DB2, PostgreSQL, Oracle, SAP DB, McKoi, Firebird lub generator wbudowany w bazę InterBase. Zwracany identyfikator jest typu long, int lub short.
increment	W momencie uruchamiania Hibernate generator odczytuje maksymalną wartość z kolumny klucza głównego tabeli i inkrementuje ją przy wstawianiu kolejnych wierszy do bazy danych. Generowany identyfikator jest typu long, int lub short. Generator jest szczególnie wydajny, jeśli jedna aplikacja stosująca Hibernate ma na wyłączność dostęp do bazy danych. W przeciwnym razie nie zaleca się stosowania tego rozwiązania.
hilo	Algorytm nisko-wysoko to wydajny sposób generowania identyfikatorów typu long, int lub short, gdy tabela i kolumna (domyślnie odpowiednio hibernate_unique_key i next_hi) są źródłem wartości wysokich. Algorytm ten generuje identyfikatory unikatowe dla konkretnej bazy danych. Więcej informacji na jego temat znajduje się w pracy [Ampler 2002].
uuid.hex	Generuje 128-bitową wartość UUID (algorytm generowania identyfikatorów tekstowych unikatowych w sieci). Stosuje adres IP w połączeniu ze znacznikiem czasowym. UUID zostaje przekształcony na tekst zawierający cyfry szesnastkowe o łącznej długości 32 znaków. Ta strategia generacji nie cieszy się dużą popularnością, ponieważ klucze główne typu CHAR zajmują w bazie danych więcej miejsca i są wolniejsze.

3.5. Szczegółowe modele obiektów

Główym zadaniem projektu Hibernate jest obsługa *szczegółowych* modeli obiektów, gdyż ten element okazuje się najważniejszym wymogiem stawianym bogatym modelom dziedzinowym. Jest też jednym z powodów wybrania POJO.

W bardzo dużym skrócie można powiedzieć, że „szczegółowy” w tym przypadku oznacza „więcej klas niż tabel”. Dane użytkownika niejednokrotnie zawierają adres domowy i adres płatności. Baza danych może zawierać jedną tabelę USER z kolumnami BILLING_STREET, BILLING_CITY i BILLING_ZIPCODE oraz kolumnami HOME_STREET, HOME_CITY i HOME_ZIPCODE. Istnieją poważne przesłanki, by stosować tego rodzaju nieznormalizowany model relacyjny (na przykład wydajność).

W modelu obiektowym nic nie stoi na przeszkodzie, by zastosować to samo podejście, tworząc sześć właściwości klasy User do przechowywania dwóch adresów. Lepiej jednak zastosować osobną klasę Address, natomiast w klasie User umieścić właściwości billingAddress i homeAddress.

Zaproponowany model okazuje się bardziej zwarty oraz łatwiejszy w ponownym wykorzystaniu i zrozumieniu. Dawniej wiele rozwiązań ORM nie oferowało dobrej obsługi przedstawionego odwzorowania.

Hibernate udostępnia elastyczność i użyteczność szczegółowych klas, by ułatwić osiągnięcie bezpieczeństwa typów i spójnego zachowania. Wiele osób modeluje adres e-mail jako właściwość tekstową klasy User. Sugerujemy bardziej wyrafinowane podejście — zastosowanie klasy EmailAddress dodającej szczegółowe zachowanie i wyższy poziom semantyczny, na przykład przez istnienie metody sendEmail().

3.5.1. Encje i typy wartości

Rozważania prowadzą nas do niezwykle istotnego dla ORM rozróżnienia. W Javie wszystkie klasy są sobie w pewnym sensie równe — wszystkie mają własną tożsamość i cykl życia, wszystkie są przekazywane przez referencję. Jedynie typy prostego są przekazywane przez wartość.

Jesteśmy zwolennikami projektów, w których istnieje więcej klas trwałych niż tabel. Jeden wiersz tabeli w Javie reprezentuje wiele obiektów. Ponieważ tożsamość bazodanowa to tak naprawdę wartość klucza głównego, pewne obiekty trwale nie będą posiadały własnej tożsamości (wybranej kluczem głównym). Mechanizm trwałości implementuje dla niektórych klas semantykę przekazywania przez wartość. Jeden z obiektów reprezentujących wiersz ma własną tożsamość natomiast pozostałe obiekty są od niego uzależnione.

Hibernate dokonuje następującego rozróżnienia:

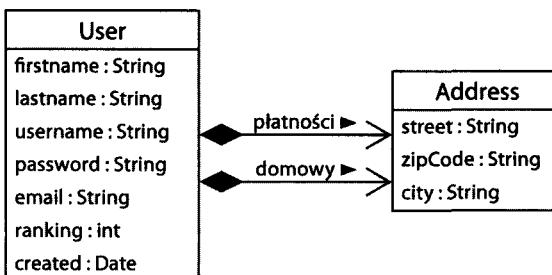
- ◆ Obiekt **typu encyjnego** ma własną tożsamość bazodanową (wartość klucza głównego). Referencja obiektu do encji jest przekazywana jako referencja w bazie danych (wartość klucza obcego). Encja ma własny cykl życia i może istnieć niezależnie od innych encji.
- ◆ Obiekt **typu wartościowego** nie ma własnej tożsamości bazodanowej. Należy do konkretnej encji. Stan jego trwałości jest powiązany wierszem tabeli encji (wyjątkiem są kolekcje, które Hibernate traktuje jako typy wartości; szczegóły znajdują się w rozdziale 6.). Typy wartości nie mają identyfikatorów ani właściwości identyfikujących. Ich cykl życia jest ścisłe powiązany z encją, do której przynależą.

Najbardziej oczywistymi typami wartości są proste obiekty jak String lub Integer. Hibernate dopuszcza traktowanie jako typu wartości klas zdefiniowanych przez użytkownika. Do tego istotnego tematu powrócimy jeszcze w podrozdziale 6.1.

3.5.2. Stosowanie komponentów

Do tej pory klasy modelu obiektów były klasami encyjnymi z własnym cyklem życia i tożsamością. Klasa User zgodnie z rysunkiem 3.5 tworzy specjalny rodzaj asocjacji z klasą Address.

Ten rodzaj asocjacji w modelowaniu obiektowym nazywa się **agregacją**, czyli związkiem „jest częścią”. Agregacja stanowi silniejszą formę asocjacji — ma dodatkową semantykę związaną z cyklem życia obiektów. W zasadzie rysunek



Rysunek 3.5. Związek między klasami User i Address wykorzystujący kompozycję

przedstawia jeszcze silniejszy związek, **kompozycję**, która informuje, że cykl życia części jest zależny od cyklu życia całości.

Eksperci modelowania obiektów i projektanci UML twierdzą, że nie istnieją różnice między kompozycją i innymi słabszymi wersjami asocjacji w kwestii ich implementacji w języku Java. W kontekście ORM różnica jest i to spora — obiekt będący częścią związku kompozycyjnego jest często dobrym kandydatem na typ wartości.

Odwzorujmy więc klasę **Address** jako typ wartości, a klasę **User** jako encję. Czy zmieni to w jakiś sposób implementację klas POJO?

Język Java nie wie nic o pojęciu kompozycji — klasy ani atrybutu nie da się zaznaczyć jako komponentu lub kompozycji. Jedyna różnica polega na identyfikatorze obiektu: komponent nie ma własnej tożsamości, więc pomimo tego, że jest klasą trwałą, nie wymaga właściwości identyfikującej ani odwzorowania identyfikatora. Kompozycję między klasami **User** i **Address** określa się na poziomie metadanych — Hibernate musi wiedzieć, że klasa **Address** jest typem wartości. Informację tę podaje się w dokumentacji odwzorowania.

W Hibernate używa się terminu **komponent** do określenia klasy użytkownika, która dotyczy tej samej tabeli co główna encja. Sytuację powinien wyjaśnić listing 3.7. Stosowane tu słowo komponent nie ma nic wspólnego z architekturą oprogramowania i **komponentami programowymi**.

Listing 3.7. Odwzorowanie klasy User wraz z komponentem Address

```

<class name="User"
      table="USER">

    <id
        name="id"
        column="USER_ID"
        type="long">
        <generator class="native"/>
    </id>

    <property
        name="username"
        column="FIRSTNAME"
        type="string"/>

    <component
        name="homeAddress" Deklaracja
        ❶ atrybutów trwałych
        <!-- implementation -->
    </component>

```

```
class="Address">>

<property name="street"
          type="string"
          column="HOME_STREET"
          not-null="true"/>

<property name="city"
          type="string"
          column="HOME_CITY"
          not-null="true"/>

<property name="zipcode"
          type="short"
          column="HOME_ZIPCODE"
          not-null="true"/>

</component>

<component                  Ponowne użycie
  name="billingAddress"    ❷ klasy komponentów
  class="Address">

  <property name="street"
            type="string"
            column="BILLING_STREET"
            not-null="true"/>

  <property name="city"
            type="string"
            column="BILLING_CITY"
            not-null="true"/>

  <property name="zipcode"
            type="short"
            column="BILLING_ZIPCODE"
            not-null="true"/>

</component>

...
</class>
```

-
- ❶ Trwałe atrybuty klasy Address podaje się wewnątrz elementu <component>. Właściwość klasy User nosi nazwę homeAddress.
 - ❷ Zauważ ponowne użycie tej samej klasy komponentu do odwzorowania innej właściwości o identycznym typie w tej samej tabeli.

Rysunek 3.6 przedstawia sposób, w jaki atrybuty klasy Address są trwale umieszczane w tej samej tabeli co encja User.

<<Table>>	
USER	
USER_ID <<PK>>	
USERNAME	

BILLING_STREET	Komponent adresu płatności
BILLING_ZIPCODE	
BILLING_CITY	
HOME_STREET	Komponent adresu domowego
HOME_ZIPCODE	
HOME_CITY	

Rysunek 3.6. Kolumny tabeli encji Usser i komponentów Address

W przedstawionym przykładzie kompozycja została zamodelowana jako jednokierunkowa. Nie jest możliwe przejście z obiektu Address do User. Hibernate obsługuje również kompozycje dwukierunkowe, ale w praktyce spotyka się je znacznie rzadziej. Oto przykład takiej kompozycji:

```
<component
    name="homeAddress"
    class="Address">
    <parent name="user"/>
    <property name="street" type="string" column="HOME_STREET"/>
    <property name="city" type="string" column="HOME_CITY"/>
    <property name="zipcode" type="short" column="HOME_ZIPCODE"/>
</component>
```

Element `<parent>` odwzorowuje właściwości typu User na encję. W przedstawionym przykładzie właściwość nosi nazwę `user`. Wywołanie `Address.getUser()` spowoduje zwrócenie obiektu użytkownika i tym samym zapewni przejście dwukierunkowe.

Komponent Hibernate może zawierać inne komponenty lub nawet asocjacje do innych encji. Elastyczność związana ze szczegółowymi modelami obiektów to w Hibernate element priorytetowy. W rozdziale 6. opisujemy różne odwzorowania komponentów.

Istnieją dwa bardzo istotne ograniczenia klas odwzorowywanych jako komponenty.

- ◆ Nie są możliwe współdzielone referencje. Komponent `Address` nie ma własnej tożsamości bazodanowej (klucza głównego), więc nie może się do niego odnosić inny obiekt niż zawierający egzemplarz klasy `User`.
- ◆ Nie istnieje elegancki sposób reprezentacji referencji typu `null` dla komponentu. Hibernate stara się być elegancki i reprezentuje komponenty typu `null` jako wartości `null` wpisane we wszystkich kolumnach komponentu. Oznacza to, że jeśli obiekt komponentu będzie zawierał właściwości wyłącznie z wartościami `null`, Hibernate zwróci komponent jako wartość `null` przy pobieraniu całej encji z bazy danych.

Obsługa szczegółowych klas to nie jedyny składnik bogatego modelu dziedzinowego. Dziedziczenie klas i polimorfizm to kolejne nieodłączne cechy modeli obiektowych.

3.6. Odwzorowanie dziedziczenia klas

Prostą strategię odwzorowania klas na tabele baz danych można przeprowadzić przy założeniu: „jedna tabela dla każdej klasy”. To podejście wydaje się proste i rzeczywiście działa dobrze, ale tylko do momentu natknięcia się na dziedziczenie.

Dziedziczenie to chyba najbardziej widoczny aspekt niedopasowania świata obiektowego i relacyjnego. Modele obiektowe stosują związki „jest” i „ma”. Modele relacyjne SQL obsługują jedynie związki typu „ma”.

Istnieją trzy sposoby reprezentacji hierarchii dziedziczenia. Zostały skatalogowane przez Scotta Amblera [Ambler 2002] w pracy „Mapping Objects to Relational Databases”.

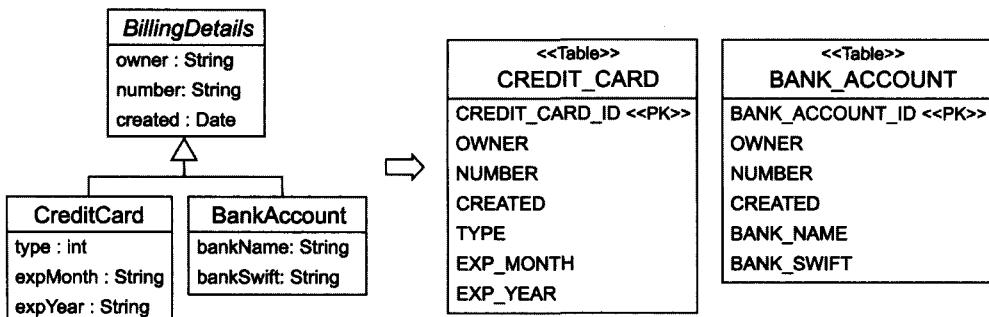
- ◆ **Tabela na każdą klasę konkretną** — całkowicie pomija polimorfizm i dziedziczenie w modelu relacyjnym.
- ◆ **Tabela na każdą hierarchię klas** — umożliwia polimorfizm przez denormalizację modelu relacyjnego i użycie kolumny typu do przechowywania informacji o typie.
- ◆ **Tabela na każdą podklasę** — reprezentuje związki „jest” (dziedziczenie) jako związki „ma” (klucze obce).

Ten podrozdział wykorzysta podejście z góry na dół. Zakłada, że zaczynamy od modelu dziedzinowego i staramy się wykonać nowy schemat bazy danych SQL. Omawiane strategie odwzorowania są tak samo istotne, jeśli używa się podejścia z dołu do góry, czyli zaczyna od istniejącego schematu bazy danych.

3.6.1. Tabela na klasę konkretną

Załóżmy korzystanie z najprostszego podejścia: możemy użyć dokładnie jednej tabeli na każdą (nieabstrakcyjną) klasę. Wszystkie właściwości klasy, włączając w to właściwości dziedziczone, można odwzorować na kolumny tabeli, co przedstawia rysunek 3.7.

Główną wadą tego podejścia jest słaba obsługa asocjacji polimorficznych. W bazie danych asocjacje najczęściej reprezentuje się jako związki z kluczami obcymi. Jeśli na rysunku 3.7 podklasy byłyby odwzorowywane na różne tabele, polimorficzna asocjacja dotycząca klasy bazowej (w przykładzie klasa BillingDetails) nie daje się przedstawić jako prosty związek z kluczem obcym. Pojawiliby się duże trudności z poprawnym działaniem modelu dziedzinowego, bo to właśnie klasa BillingDetails łączy się z klasą User. Obie tabele dla konkretnych klas muszą zawierać referencje klucza obcego do tabeli USER.



Rysunek 3.7. Tabela na każdą klasę konkretną

Zapytania polimorficzne (zapytania zwracające obiekty wszystkich klas pasujących do wskazanego interfejsu) także stwarzają duże problemy. Zapytanie dotyczące klasy bazowej musi zostać rozbite na wiele poleceń SELECT (po jednym na każdą podklasę). W celu poprawy wydajności warto w takiej sytuacji użyć unii (UNION), by zwracane wyniki zostały połączone w jedną całość. Niestety, unie są słabo przenośne i mają pewne ograniczenia. Hibernate w wersji 2.1 nie używa unii i zawsze w takiej sytuacji wykonuje kilka zapytań. Dla zapytania HQL dotyczącego klasy `BillingDetails` ograniczającego zakres czasu utworzenia, Hibernate wykonałby następujące zapytania SQL.

```

select CREDIT_CARD_ID, OWNER, NUMBER, CREATED, TYPE, ...
from CREDIT_CARD
where CREATED = ?
  
```

```

select BANK_ACCOUNT_ID, OWNER, NUMBER, CREATED, BANK_NAME, ...
from BANK_ACCOUNT
where CREATED = ?
  
```

Zauważ potrzebę wykonania osobnego zapytania dla każdej konkretnej klasy. Z drugiej strony zapytanie dotyczące tylko wybranej konkretnej klasy jest proste i wykonuje się szybko.

```

select CREDIT_CARD_ID, TYPE, EXP_MONTH, EXP_YEAR
from CREDIT_CARD where CREATED = ?
  
```

(W tym miejscu, a także w wielu innych w dalszej części książki, przedstawiamy kod SQL będący jedynie w sposób koncepcyjny identyczny z kodem generowanym przez Hibernate. Rzeczywiście wykonywany kod może wyglądać całkowicie inaczej.)

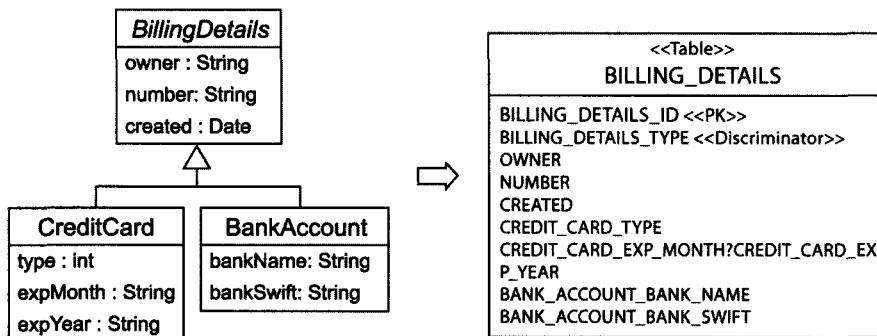
Kolejny problem koncepcyjny tej strategii odwzorowania polega na tym, że wiele różnych kolumn różnych tabel wspólnie dzieli tę samą semantykę. Czyni to zmiany w schemacie znacznie bardziej skomplikowane. Zmiana typu jednej właściwości klasy bazowej pociąga za sobą zmiany wielu kolumn. Co więcej, znacznie bardziej zapisów koniecznych jest zaimplementować dla wielu podklas ograniczenia zapewniające integralność bazy danych.

Przedstawiona strategia odwzorowania nie wymaga żadnej dodatkowej deklaracji w odwzorowaniu Hibernate. Wystarczy wykonać nową deklarację `<class>`

dla każdej konkretnej klasy i wskazać różne tabele w atrybucie table. Zalecamy to podejście (jedynie) dla hierarchii klas wysokiego poziomu, gdzie najczęściej polimorfizm nie jest potrzebny.

3.6.2. Tabela na każdą hierarchię klas

Alternatywne podejście polega na odwozorowaniu całej hierarchii klas w jednej tabeli. Tabela zawiera kolumny dotyczące wszystkich właściwości wszystkich klas występujących w hierarchii. Konkretną podkласę reprezentowaną przez wiersz tabeli określa wartość kolumny dyskryminatora typu. Rysunek 3.8 przedstawia to rozwiązanie.



Rysunek 3.8. Tabela na każdą hierarchię klas

Ta strategia jest zwyciężką zarówno w kwestii wydajności, jak i prostoty. To najlepszy sposób wyrażenia polimorfizmu — żadnego problemu nie stanowi poprawna obsługa polimorficznych i zwykłych zapytań. Ręczna implementacja nie sprawia dużych trudności. Własny system raportujący nie wymaga wyrafinowanych złączeń i unii. Modyfikacja schematu bazy danych nie nastręcza trudności.

Pojawia się jednak inny problem — kolumny właściwości zadeklarowanych w podklassach muszą dopuszczać wstawianie wartości null. Jeżeli każda z tych klas definiuje kilka właściwości, które trzeba bezwzględnie wypełnić, traci się ograniczenie NOT NULL, co stanowi znaczne zagrożenie z punktu widzenia integralności danych.

W Hibernate używamy elementu <subclass>, by wskazać odwozorowanie stosujące jedną tabelę dla całej hierarchii klas (patrz listing 3.8).

Listing 3.8. Odwozorowanie podklaśs elementem <subclass> w Hibernate

```

<hibernate-mapping>
  <class name="BillingDetails" table="BILLING_DETAILS" discriminator-value="BD">
    <id name="id" />
    <!-- Klasa bazowa odwozowywana na tabelę -->
  </class>
</hibernate-mapping>
  
```

```

        column="BILLING_DETAILS_ID"
        type="long">
        <generator class="native"/>
    </id>
    
    • Kolumna dyskryminatora
    <discriminator
        column="BILLING_DETAILS_TYPE"
        type="string"/>
    • Odwzorowanie właściwości
    <property
        name="name"
        column="OWNER"
        type="string"/>
    ...
    • Podklasa CreditCard
    <subclass
        name="CreditCard"
        discriminator-value="CC">
        <property
            name="type"
            column="CREDIT_CARD_TYPE"/>
        ...
    </subclass>
    ...
    </class>
</hibernate-mapping>

```

- ① Klasa bazowa hierarchii dziedziczenia, `BusinessDetails`, zostaje odwzorowana na tabelę `BUSINESS_DETAILS`.
- ② Do odróżniania podklas potrzeba osobnej kolumny — **dyskryminatora**. Nie jest to właściwość żadnej z klas trwałych. Służy wyłącznie do poprawnej obsługi dziedziczenia przez Hibernate. Kolumna nosi nazwę `BILLING_DETAILS_TYPE`. Jej wartościami są krótkie ciągi znaków, na przykład "CC" lub "BA". Hibernate automatycznie ustawia i pobiera wartości dyskryminatora.
- ③ Właściwości klasy bazowej odwzorowuje się w sposób tradycyjny — wykorzystując właściwość `<property>`.
- ④ Każda podklasa ma własny element `<subclass>`. Właściwości podklasy zostają odwzorowane na kolumny tabeli `BILLING_DETAILS`. Nie można dla tych właściwości stosować ograniczenia `not-null`, ponieważ egzemplarz klasy `CreditCard` nie będzie zawierał właściwości `bankSwift` i tym samym kolumna `BANK_ACCOUNT_BANK_SWIFT` pozostanie pusta.

Element `<subclass>` może zawierać inny element `<subclass>` aż do odwzorowania całej hierarchii na tabelę. Element `<subclass>` nie może zawierać elementu

<joined-subclass> (jest on używany przez trzecie rozwiązanie odwozorowywania hierarchii klas opisywane w kolejnym podrozdziale). Hibernate 2.1 nie dopuszcza dowolnego przełączania strategii odwozorowania.

Hibernate zastosuje poniższy kod SQL w momencie pobierania szczegółów klasy BillingDetails.

```
select BILLING_DETAILS_ID, BILLING_DETAILS_TYPE,
       OWNER, .... CREDIT_CARD_TYPE
  from BILLING_DETAILS
 where CREATED = ?
```

W momencie pobierania szczegółów podklas CreditCard Hibernate wykorzysta dyskryminator do ograniczenia wyników do konkretnej klasy.

```
select BILLING_DETAILS_ID, CREDIT_CARD_TYPE, CREDIT_CARD_EXP_MONTH, ...
  from BILLING_DETAILS
 where BILLING_DETAILS_TYPE='CC' AND CREATED = ?
```

Czy istnieje prostsze podejście?

3.6.3. Tabela na każdą podklasę

Trzecie podejście polega na reprezentacji związków dziedziczenia jako asocjacji relacyjnych kluczy obcych. Każda podklasa deklarująca trwałe właściwości — włączając w to klasy abstrakcyjne i interfejsy — korzysta z własnej tabeli.

W odróżnieniu od rozwiązania stosującego tabelę na każdą klasę konkretną, tutaj każda tabela zawiera jedynie niedziedziczone właściwości (czyli te deklarowane przez podklassę) oraz klucz główny będący jednocześnie kluczem obcym tabeli klasy bazowej. To podejście przedstawia rysunek 3.9.

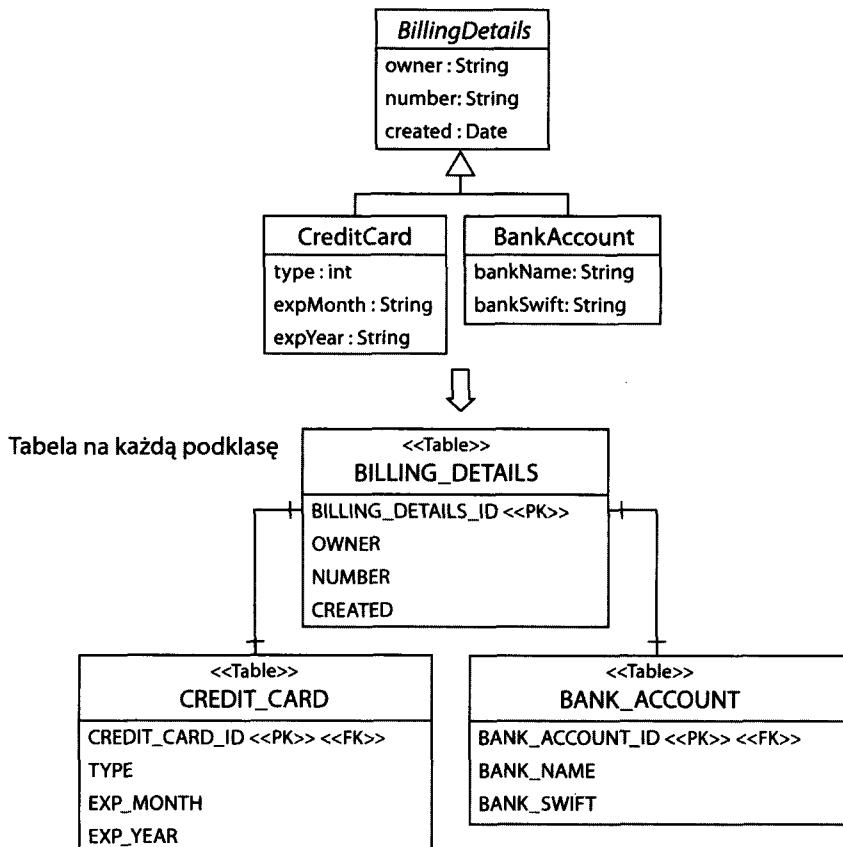
Jeśli aplikacja zapamiętuje egzemplarz podklasy CreditCard, wartości właściwości zadeklarowanych przez klasę bazową BillingDetails trafiają do nowego wiersza tabeli BILLING_DETAILS. Jedynie wartości właściwości zadeklarowanych przez podklasy trafiają do nowego wiersza tabeli CREDIT_CARD. Oba wiersze łączy wspólny klucz główny. W przyszłości egzemplarz podklasy pobiera się z bazy danych, łącząc tabelę podklasy z tabelą klasy bazowej.

Główną zaletą tej strategii jest pełna normalizacja modelu relacyjnego. Ewentualna zmiana schematu i stosowanie ograniczeń zapewniających integralność nie jest trudne. Asocjacje polimorficzne do konkretnej podklasy można przedstawić jako klucz obcy wskazujący na tabelę podklasy.

W Hibernate do określenia opisywanego odwozorowania służy element <joined-subclass> przedstawiony na listingu 3.9.

Listing 3.9. Odwozorowanie podklas elementem <joined-subclass> w Hibernate

```
<?xml version="1.0"?> <hibernate-mapping>
<class
      name="BillingDetails" <!-- Klasa bazowa
      odwozorowana na tabelę
      BILLING_DETAILS -->
```



Rysunek 3.9. Odwzorowanie stosujące tabelę dla każdej podklasy

```

table="BILLING_DETAILS">
<id>
  name="id"
  column="BILLING_DETAILS_ID"
  type="long"
  <generator class="native"/>
</id>

<property>
  name="name"
  column="OWNER"
  type="string"/>

...
<joined-subclass
  name="CreditCard"
  table="CREDIT_CARD">
  <!-- Element -->
</joined-subclass>
  
```

```

<key column="CREDIT_CARD_ID"/> ←
<property
    name="type"
    column="TYPE"/>
    ...
</joined-subclass>
    ...
</class>
</hibernate-mapping>

```

❶ Klucz główny
i jednocześnie klucz obcy

- ❶ Ponownie klasa bazowa `BillingDetails` jest odwzorowana na tabelę `BILLING_DETAILS`. Tym razem jednak nie trzeba podawać dyskryminatora.
- ❷ Nowy element `<joined-subclass>` służy do odwzorowania podklasy na nową tabelę (w przykładzie jest to tabela `CREDIT_CARD`). Wszystkie właściwości zadeklarowane włączanej podklasie trafią do tej tabeli. Celowo pomineliśmy przykład dla `BankAccount`, gdyż jest on praktycznie powieleniem odwzorowania klasy `CreditCard` z innymi danymi.
- ❸ Tabela `CREDIT_CARD` wymaga klucza głównego. Dodatkowo będzie to również klucz obcy zapewniający integralność z kluczem głównym tabeli `BILLING_DETAILS`. Uzyskanie pełnych danych obiektu `CreditCard` wymagałąączenia obu tabel.

Element `<joined-subclass>` może zawierać inne elementy `<joined-subclass>`, ale nie elementy `<subclass>`. Hibernate 2.1 nie obsługuje mieszania dwóch strategii odwzorowań.

Hibernate wykorzystałąączenie zewnętrzne w momencie poszukiwania obiektów `BillingDetails`.

```

select DB.BILLING_DETAILS_ID, BD.OWNER, BD.NUMBER, BD.CREATED,
       CC.TYPE, ..., BA.BANK_SWIFT, ...
  case
    when CC.CREDIT_CARD_ID is not null then 1
    when BA.BANK_ACCOUNT_ID is not null then 2
    when BD.BILLING_DETAILS_ID is not null then 0
  end as TYPE
 from BILLING_DETAILS BD
 left join CREDIT_CARD CC on BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
 left join BANK_ACCOUNT BA on BD.BILLING_DETAILS_ID = BA.BANK_ACCOUNT_ID
 where BD.CREATED = ?

```

Polecenie `case` wykorzystuje istnienie (lub brak istnienia) wierszy w tabelach `CREDIT_CARD` i `BANK_ACCOUNT` do określenia konkretnej podklasy wskazanego wiersza tabeli `BILLING_DETAILS`.

By ograniczyć wyniki zapytania, Hibernate dla podklasy stosujełąączenie wewnętrzne.

```
select DB.BILLING_DETAILS_ID, BD.OWNER, BD.CREATED, CC.TYPE, ...
from CREDIT_CARD CC
inner join BILLING_DETAILS BD on BD.BILLING_DETAILS_ID =
CC.CREDIT_CARD_ID
where CC.CREATED = ?
```

Nietrudno zauważyc, że ręczne korzystanie z tej strategii jest bardzo zagmatwane. Nawet kreowanie własnych raportów nie będzie proste. To istotny czynnik, gdy planuje się wykorzystać kod Hibernate z ręcznie tworzonymi zapytaniami SQL wykorzystującymi JDBC. Tworzenia raportów ad hoc można ułatwić przez wykonanie widoków ukrywających złożoność rozwiązania przez przekształcanie ich na znacznie prostszy model jednej tabeli na hierarchię klas.

Choć sama koncepcja przedstawionej strategii odwzorowania wydaje się kuścąco prosta, doświadczenie pokazuje, że dla bardziej złożonych hierarchii jej wydajność nie jest do zaakceptowania. Zapytania zawsze wymagają albo łączenia wielu tabel, albo wielu sekwencyjnych odczytów. Zastanówmy się, w jaki sposób wybrać odpowiednią kombinację strategii odwzorowań dla hierarchii klas stosowanej w aplikacji. Typowy model dziedzinowy składa się z mieszanki wielu interfejsów i klas abstrakcyjnych.

3.6.4. Wybór strategii

Wszystkie przedstawione strategie można stosować dla klas abstrakcyjnych i interfejsów. Interfejsy nie muszą zawierać stanu, ale zawierają deklaracje metod dostępowych i tym samym mogą być traktowane jak klasy abstrakcyjne. Nic nie stoi na przeszkodzie, by odwzorowywać interfejs elementami <class>, <subclass> lub <joined-subclass>. Element <property> służy do odwzorowywania dowolnej zadeklarowanej lub odziedziczonej właściwości. Hibernate nie utworzy egzemplarza abstrakcyjnej klasy nawet wtedy, gdy wykona się dla niego zapytanie i spróbuje go załadować.

Oto kilka podstawowych zasad:

- ◆ Gdy asocjacje i zapytania polimorficzne nie są potrzebne, kieruj się w stronę strategii tabeli na każdą klasę konkretną.
- ◆ Jeśli korzystasz z zapytań lub asocjacji polimorficznych (asocjacji do klasy bazowej, czyli do wszystkich klas z jej hierarchii z dynamicznym rzutowaniem na konkretną podkласę) i podklasy zawierają niewiele własnych właściwości (w szczególności, gdy różnica między klasami wynika z ich zachowania), kieruj się w stronę strategii tabeli na każdą hierarchię klas.
- ◆ Jeśli korzystasz z zapytań lub asocjacji polimorficznych i podklasy definiują wiele właściwości (podklasy różnią się przede wszystkim przechowywanymi danymi), skorzystaj z podejścia jednej tabeli na podkласę.

Najczęściej podejście z jedną tabelą na hierarchię klas wybiera się dla prostych problemów. Bardziej wyrafinowane przypadki lub namowy projektanta bazy danych związane z wymuszeniem ograniczeń co do wartości null wymagają rozważenia podejścia z jedną tabelą na podklasę. W tym momencie warto zadać sobie

pytanie, czy nie lepiej zamodelować dziedziczenie jako delegację w modelu obiektowym. Złożonego dziedziczenia warto unikać z wielu różnych powodów niezwiązańych z bezpośrednio z trwałością ani ORM. Hibernate działa jak bufor między modelami relacyjnymi i obiektowymi. Nie oznacza to jednak, że należy całkowicie zapomnieć o kwestiach trwałości w momencie projektowania modelu obiektowego.

Elementy `<subclass>` i `<joined-subclass>` można stosować w osobnych plikach odwzorowań (jako elementy głównego poziomu zastępujące `<class>`). Należy jednak wtedy określić rozszerzaną klasę (na przykład `<subclass name="CreditCard" extends="BillingDetails">`) i wczytywać odwzorowanie klasy bazowej przed odwzorowaniem podklasy. W ten sposób rozbudowa hierarchii klas nie pociąga za sobą zmian w plikach odwzorowań klas bazowych.

Do tej pory zajmowaliśmy się przypadkami, w których encje nie wiedziały nic o sobie nawzajem. W kolejnym podrozdziale zajmiemy się problemem odwzorowywania asocjacji, który stanowi poważne wyzwanie w niedopasowaniu paradigmatów obiektowego i relacyjnego.

3.7. Asocjacje

Zarządzanie asocjacjami między klasami i związkami między tabeli to dusza każdego systemu ORM. Większość problemów w implementacji systemu ORM sprawia właśnie zarządzanie asocjacjami.

Model asocjacyjny Hibernate jest wyjątkowo bogaty, ale początkującym sprawia wiele trudności. W niniejszym podrózdziele nie zajmiemy się wszystkimi możliwymi przypadkami, ale raczej najczęściej spotykanyimi. Pełniejszy opis wszystkich elementów odwzorowania asocjacji znajduje się w rozdziale 6.

Zanim jednak przejdziemy do właściwego przepisu, kilka dodatkowych wyjaśnień.

3.7.1. Asocjacje zarządzane?

Osoby korzystające z CMP 2.0 lub 2.1 powinny znać koncepcję **asocjacji zarządzanych** ((lub związków zarządzanych). Asocjacje CMP noszą nazwę związków zarządzanych przez kontener (CRM — *Container Managed Relationship*) nie bez powodu. Asocjacje CMP są dwukierunkowe: zmiana wykonana po jednej stronie asocjacji jest natychmiast wykonywana po drugiej stronie. Jeśli na przykład wywołamy `bid.setItem(item)`, kontener automatycznie wykona za nas wywołanie `item.getBids().add(item)`.

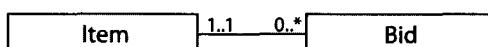
Przezroczyste implementacje trwałości stosujące POJO, na przykład Hibernate, nie implementują asocjacji zarządzanych. W odróżnieniu od CRM asocjacje w Hibernate są zawsze **jednokierunkowe**. Z punktu widzenia Hibernate asocjacja od `Bid` do `Item` jest **inną asocjacją** niż przejście z `Item` do `Bid`.

Jednym osobom wydaje się to dziwne, dla innych jest całkowicie naturalne. Asocjacje w języku Java zawsze są jednokierunkowe, a Hibernate stara się implementować trwałość, wykorzystując standardowe obiekty Javy. Powód podjęcia tego rodzaju decyzji powodowany jest tym, że obiekty Hibernate w odróżnieniu od ziarenek encyjnych nie zakładają istnienia sterującymi. W aplikacji stosującej Hibernate zachowanie egzemplarza nietrwałego jest dokładnie takie samo jak działanie egzemplarza trwałego.

Ponieważ asocjacje są niezwykle istotne, potrzebujemy bardzo precyzyjnego języka do ich klasyfikacji.

3.7.2. Krotność

W opisach i klasyfikacjach asocjacji spotyka się często informację o ich **krotności**. Przyjrzyj się rysunkowi 3.10.



Rysunek 3.10. Związek między klasami Item i Bid

Krotność to tak naprawdę dwie informacje:

- ◆ czy może istnieć więcej niż jeden obiekt Bid dla konkretnego obiektu Item,
- ◆ czy może istnieć więcej niż jeden obiekt Item dla konkretnego obiektu Bid.

Po dokładnym rozpatrzeniu modelu obiektowego możemy stwierdzić, że asocjacja od Bid do Item jest związkiem typu **wiele-do-jednego**. Przypominając sobie o kierunkowości asocjacji w Hibernate, możemy również stwierdzić, że asocjacja odwrotna od Item do Bid jest związkiem typu **jeden-do-wielu**. Istnieją jeszcze dwie możliwości, **wiele-do-wielu** i **jeden-do-jednego**, ale nimi zajmiemy się w rozdziale 6.

W kontekście trwałości obiektu nie interesuje nas, czy „wiele” oznacza w tym przypadku „dwa”, „maksymalnie pięć” czy może „nieograniczoną ilość”.

3.7.3. Najprostsza możliwa asocjacja

Asocjacja od Bid do Item jest przykładem najprostszego rodzaju asocjacji w ORM. Referencję do obiektu zwracanego przez metodę getItem() łatwo odwzorować na kolumnę klucza obcego tabeli BID. Najpierw przedstawmy implementację w Javie klasę Bid:

```

public class Bid {
    ...
    private Item item;

    public void setItem(Item item) {
        this.item = item;
    }

    public Item getItem() {
        return item;
    }
}
  
```

```
    }  
  
    ...  
}  
  
Kolejna odwzorowanie Hibernate dla asocjacji:  
  
<class  
    name="bid"  
    table="BID">  
  
    ...  
    <many-to-one  
        name="item"  
        column="ITEM_ID"  
        class="Item"  
        not-null="true"/>  
</class>
```

Odwzorowanie to nosi nazwę **jednokierunkowej asocjacji wiele-do-jednego**. Kolumna ITEM_ID tabeli BID jest kluczem obcym korzystającym z klucza głównego tabeli ITEM.

Jawnie wskazaliśmy klasę (Item), której dotyczy asocjacja. Jej wskazywanie nie jest wymagane, ponieważ Hibernate potrafi określić docelową klasę dzięki refleksji.

Odwzorowanie zawiera atrybut not-null, ponieważ nie może istnieć oferta bez przedmiotu. Atrybut nie wpływa na zachowanie Hibernate w trakcie pracy aplikacji — ma jedynie sterować poprawną generacją kodu DDL SQL (patrz rozdział 9.).

3.7.4. Tworzenie asocjacji dwukierunkowej

Na razie wszystko idzie zgodnie z planem. Przydałaby się jednak możliwość pobrania wszystkich ofert dla konkretnego przedmiotu. Innymi słowy, potrzebujemy asocjacji dwukierunkowej. Poniżej przedstawiamy podstawowy kod klasy Item.

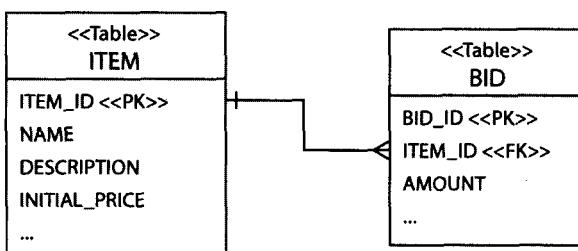
```
public class Item {  
    ...  
    private Set bids = new HashSet();  
  
    public void setBids(Set bids) {  
        this.bids = bids;  
    }  
  
    public Set getBids() {  
        return bids;  
    }  
  
    public void addBid(Bid bid) {  
        bid.setItem(this);  
        bids.add(bid);  
    }  
  
    ...  
}
```

Kod zawarty w metodzie addBid(), będącej metodą pomocniczą, warto traktować jako implementację asocjacji zarządzanych w modelu obiektowym.

Proste odwzorowanie asocjacji jeden-do-wielu wygląda następująco.

```
<class
    name="Item"
    table="ITEM">
    ...
<set name="bids">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
</class>
```

Odwzorowanie kolumny zdefiniowane przez element `<key>` stanowi kolumnę klucza obcego powiązanej tabeli `BID`. Zauważ wskaazywanie tej samej kolumny klucza obcego w tej asocjacji, jak i w asocjacji wiele-do-jednego. Rysunek 3.11 przedstawia strukturę tabel dla przedstawianego przykładu.



Rysunek 3.11.
Związki między tabelami i klucze dla odwzorowania związków jeden-do-wielu i wiele-do-jednego

Otrzymujemy dwie różne asocjacje jednokierunkowe odwzorowane na ten sam klucz obcy, co rodzi pewien problem. W czasie działania aplikacji w pamięci wystąpią dwie reprezentacje tej samej wartości klucza obcego: we właściwości `item` obiektu `Bid` i elemencie kolekcji `bids` przechowywanej przez obiekt `Item`. Założymy, że aplikacja modyfikuje asocjację, dodając ofertę do przedmiotu przy użyciu metody `addBid()`.

```
bid.setItem(this);
bids.add(bid);
```

Kod jest poprawny, ale Hibernate wykrywa w takiej sytuacji dwie zmiany w pamięciowej reprezentacji obiektów trwałych. Z punktu widzenia bazy danych wystarczy uaktualnić tylko jedną wartość, by trwale zapamiętać zmianę — wartość kolumny `ITEM_ID` tabeli `BID`. **Hibernate nie wykrywa automatycznie, że dwie zmiany dotyczą tej samej kolumny bazy danych, bo nie wskazaliśmy w żaden sposób, iż asocjacja jest dwukierunkowa.**

Aby Hibernate wiedział, że asocjacja jest dwukierunkowa, należy dodać atrybut `inverse`. W ten sposób ORM dowiaduje się, iż kolekcja stanowi lustrzane odbicie asocjacji wiele-do-jednego występującej po drugiej stronie związku.

```
<class  
    name="Item"  
    table="ITEM">  
    ...  
  
    <set  
        name="bids"  
        inverse="true">  
        <key column="ITEM_ID"/>  
        <one-to-many class="Bid"/>  
    </set>  
  
</class>
```

Bez tego atrybutu Hibernate starałby się wykonać dwa różne polecenia SQL aktualizujące tę samą kolumnę klucza obcego po modyfikacji asocjacji między dwoma egzemplarzami. Pisząc `inverse="true"`, jawnie informujemy Hibernate, która część asocjacji powinien synchronizować z bazą danych. W tym przykładzie poinformowaliśmy ORM, by propagował do bazy danych jedynie zmiany wykonane po stronie Bid asocjacji i jednocześnie ignorował zmiany dokonywane jedynie w kolekcji bids. Jeśli wywołamy tylko `item.getBids().add(bid)`, modyfikacja nie zostanie trwale zapamiętana. Odpowiada to typowemu zachowaniu Javy bez Hibernate — jeśli asocjacja jest dwukierunkowa, trzeba utworzyć łącze po obu stronach, a nie tylko po jednej.

Uzyskaliśmy działającą, dwukierunkową asocjację wiele-do-jednego (choć nic nie stoi na przeszkodzie, by nazwać ją dwukierunkową asocjacją jeden-do-wielu).

Brakuje jeszcze jednego elementu. W kolejnym rozdziale bardziej szczegółowo zajmiemy się tematem **trwałości ułotnej**. Na razie wprowadzimy tylko pojęcia **kaskadowego usuwania** i **kaskadowego zapisywania**, by z czystym sumieniem zakończyć odwzorowywanie asocjacji.

Gdy utworzymy nowy obiekt Bid i dodamy go do Item, powinien on natychmiast stać się trwałym. Chcemy w ten sposób uniknąć jawnego wywoływanego metody `save()` interfejsu Session dla obiektu Bid.

Dokonajmy ostatniej zmiany w dokumencie odwzorowania, by włączyć usuwanie kaskadowe.

```
<class  
    name="Item"  
    table="ITEM">  
    ...  
  
    <set  
        name="bids"  
        inverse="true"  
        cascade="save-update">  
        <key column="ITEM_ID"/>  
        <one-to-many class="Bid"/>  
    </set>  
  
</class>
```

Atrybut cascade informuje Hibernate, by każdy nowy obiekt Bid czynił trwałym (czyli zapisywał go do bazy danych), jeśli tylko referencja do niego znajdzie się w trwałym obiekcie Item.

Atrybut cascade jest kierunkowy — dotyczy tylko jednej strony asocjacji. Choć nic nie stoi na przeszkodzie, by dodać atrybut cascade="save-update" do asocjacji określonej w odwzorowaniu dla Bid, nie miałoby to dużego sensu, bo obiekty Bid tworzy się dopiero obiektach Item.

Czy to już koniec? Niezupełnie. Pozostaje jeszcze określić cykl życia obu encji asocjacji.

3.7.5. Związek rodzic-potomek

W poprzednim odwzorowaniu asocjacja między obiektami Bid i Item jest bardzo luźna. Można by ją zastosować w rzeczywistym systemie tylko wtedy, gdy cykle życia obu encji byłyby od siebie niezależne, a ich tworzenie i usuwanie dotyczyłyby osobnych procesów biznesowych. Pewne asocjacje są znacznie mocniejsze od wcześniej przedstawionych. Cykle życia obu encji nie są od siebie w pełni niezależne. W przedstawianym przykładzie logiczne wydaje się, że usunięcie przedmiotu aukcji powinno usuwać również wszystkie związane z nim oferty. Konkretna oferta przez cały czas życia dotyczy jednego i tylko jednego przedmiotu. Sens ma więc usuwanie kaskadowe.

W momencie włączenia usuwania kaskadowego asocjacja między Bid i Item staje się związkiem rodzic-potomek. W tym związku encja rodzica odpowiada za cykl życia powiązanych z nią encji potomnych. Ma to samo znaczenie co kompozycja wykorzystująca komponenty Hibernate, choć biorą w niej udział jedynie encje (Bid nie jest typem wartości). Zaletą związku rodzic-potomek względem kompozycji jest możliwość niezależnego wczytywania potomka i tworzenia referencji do niego przez inne encje. Przykładowo, egzemplarz Bid daje się modyfikować bez potrzeby pobierania egzemplarza Item. Co więcej, ten sam egzemplarz Bid może pojawić się w referencji innej właściwości obiektu Item (patrz właściwość successfulBid z rysunku 3.2). Obiektów typów wartościowych nie współdzieli się.

By zamienić asocjację na związek rodzic-potomek, wystarczy dokonać zmian w atrybutie cascade.

```

<class
    name="Item"
    table="ITEM">
    ...
    <set
        name="bids"
        inverse="true"
        cascade="all-delete-orphan">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
</class>
```

Atrybut cascade="all-delete-orphan" wskazuje dwa działania do przeprowadzenia:

- ◆ Dowolny nowo utworzony `Bid` staje się trwałym, jeśli obiekt `Item` zawiera do niego referencję (ta sama sytuacja zachodziła również dla atrybutu `cascade="save-update"`). Dowolny obiekt `Bid` należy usunąć, gdy jest usuwany jedyny obiekt `Item` na niego wskazujący.
- ◆ Dowolny obiekt `Bid` należy usunąć, jeśli zostanie usunięty z kolekcji `bids` trwałego obiektu `Item`. (Hibernate założy, że była to jedyna referencja do obiektu, i stwierdzi, iż obiekt został osierocony).

Przedstawione wcześniej odwzorowanie zapewnia, że oferta zostaje usunięta z bazy danych, gdy tylko znika z kolekcji ofert przedmiotu (lub też sam przedmiot zostaje usunięty).

Kaskadowe wykonywanie operacji dla powiązanych encji to sposób implementacji przez Hibernate tak zwanej **trwałości ulotnej**. Dokładniej przyjrzymy się temu tematowi w podrozdziale 4.3.

Omówiliśmy jedynie niewielki fragment rozwiązań asocjacyjnych dostępnych w Hibernate. Uzyskana wiedza jest jednak na tyle rozległa, by móc tworzyć w pełni działające aplikacje. Pozostałe rozwiązania są albo wyjątkowo rzadkie, albo stanowią odmianę opisanych asocjacji.

Zalecamy jak największą prostotę w odwzorowaniach asocjacji, by bardziej zaawansowane zadania pozostawić zapytaniom SQL.

3.8. Podsumowanie

W rozdziale skupiliśmy się na strukturalnych aspektach niedopasowania obiektowo-relacyjnego i opisaliśmy cztery pierwsze ogólne problemy ORM. Zajęliśmy się modelem programistycznym klas trwałych i metadanymi Hibernate dotyczącymi szczegółowych klas, identyczności obiektów, dziedziczenia oraz asocjacji.

Dowiedziałeś się, że klasy trwałe modelu dziedzinowego warto uwolnić od zadań ogólnych, na przykład transakcji lub bezpieczeństwa. Nawet zagadnienia trwałości nie powinny przenikać do modelu dziedzinowego. Nie polecamy stosowania restrykcyjnych modeli programistycznych, na przykład ziarenek encyjnych EJB, do tworzenia modelu dziedzinowego. Zamiast tego stosujemy trwałość przezroczystą wraz z nierestrykcyjnym modelem programistycznym POJO, który tak naprawdę jest zbiorem zalecanych podejść do poprawnej hermetyzacji typów Javy.

Hibernate wymaga utworzenia metadanych w postaci dokumentu XML. Dokument definiuje strategie odwzorowania dla wszystkich klas trwałych (i tabel). Przyjrzyliśmy się odwzorowaniom klas i właściwości, a także odwzorowaniom asocjacji klas. Przedstawiliśmy sposób implementacji w Hibernate trzech dobrze znanych strategii odwzorowania dziedziczenia.

Zajęliśmy się istotną różnicą między obiektami **encyjnymi** i obiektami **typu wartości**. Encje mają własną tożsamość i cykl życia, natomiast typy wartości są zależne od encji i ich trwałość przypomina przekazywanie przez wartość. Hibernate

dopuszcza szczegółowe modele danych stosujące mniejszą liczbę tabel niż klas trwałych.

Zaimplementowaliśmy i odwzorowaliśmy pierwszą asocjację rodzic-potomek między klasami trwałymi, stosując pola kluczy obcych i operacje kaskadowe.

W następnym rozdziale zajmiemy się dynamicznymi aspektami niedopasowania obiektowo-relacyjnego, bardziej szczegółowo omawiając wprowadzone operacje kaskadowe i cykl życia obiektów trwałych.

Stosowanie obiektów trwałych

W rozdziale:

- ◆ Cykl życia obiektów w aplikacji używającej Hibernate
- ◆ Stosowanie zarządcy trwałości sesyjnej
- ◆ Trwałość przechodnia
- ◆ Efektywne strategie pobierania danych

Wiemy, w jaki sposób Hibernate i inne systemy ORM rozwiążają statyczne problemy niedopasowania obiektowo-relacyjnego. Zdobyta do tej pory wiedza pozwala rozwiązać problem niedopasowania strukturalnego, ale wydajne rozwiązanie problemu wymaga czegoś więcej. Musimy rozpatrzyć strategie dostępu do danych, gdyż są one niezwykle ważne dla wydajności aplikacji, czyli dowiedzieć się, jak wydajnie zapamiętywać i odczytywać obiekty.

Rozdział omawia **behawioralne** aspekty niedopasowania obiektowo-relacyjnego wymienione w rozdziale 1. jako ostatnie cztery zagadnienia podrozdziału 1.4.2. Uważamy te problemy za co najmniej równie istotne jak problemy strukturalne omawiane w rozdziale 3. Z doświadczenia wiemy, że programiści większą uwagę zwracają na niedopasowanie strukturalne i rzadko wykrywają dynamiczne aspekty niedopasowania.

W rozdziale opisujemy cykl życia obiektów — jak obiekt staje się trwały i kiedy przestaje być uznawany za trwały — oraz metody i inne działania wpływające na przechodzenie między wspomnianymi stanami. Zarządcą trwałości w Hibernate, obiekt Session, odpowiada za stan obiektu, więc warto dobrze poznać jego interfejs programistyczny.

Wydajne pobieranie grafu obiektów to kolejna istotna kwestia, więc w tym rozdziale wprowadzimy pewne podstawowe strategie. Hibernate zawiera kilka metod określania zapytań zwracających obiekty bez tracenia siły wyrazu i elastyczności znanej z języka SQL. Ponieważ opóźnienia sieciowe powodowane przez zdalny dostęp do bazy danych niejednokrotnie decydują o ogólnej wydajności aplikacji Javy, warto się dowiedzieć, w jaki sposób pobrać graf obiektów przy minimalnej liczbie zapytań kierowanych do bazy danych.

Zacznijmy od opisu obiektów, ich cyklu życia i zdarzeń wpływających na zmianę stanu trwałego. Podstawa te pozwoli lepiej zrozumieć sposób działania grafu obiektów, by wiedzieć, kiedy i jak pobierać i zapisywać obiekty. Materiał jest dosyć formalny, ale dobre zrozumienie **cyklu życia obiektów trwałych** naprawdę bywa przydatne.

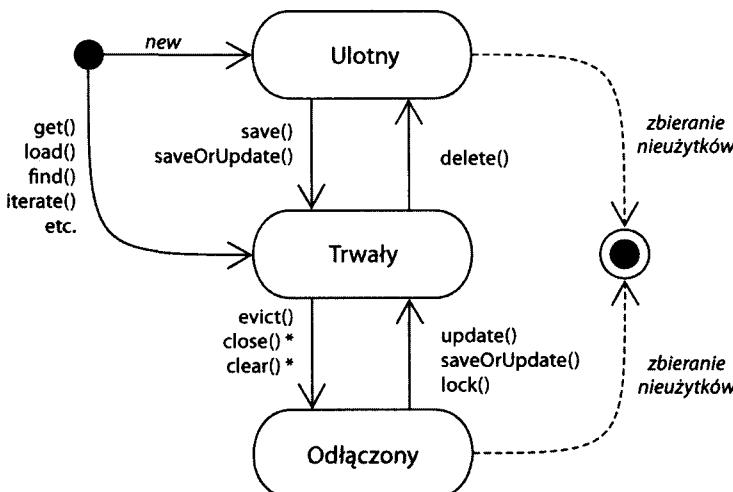
4.1. Cykl życia obiektu trwałego

Ponieważ Hibernate jest przezroczystym mechanizmem trwałości — klasy nie wiedzą o tym, że są trwale pamiętane — można tak pisać logikę aplikacji, by nie wiedziała, czy obiekty, których używa, reprezentują stan trwały czy tymczasowy (istniejący jedynie w pamięci komputera). Aplikacja niekoniecznie musi być zainteresowana tym, że obiekt jest trwał w momencie wywoływania jego metod.

Z drugiej strony aplikacja wykorzystująca trwałość musi informować warstwę trwałości o tym, że chce przenieść obiekt znajdujący się w pamięci do bazy danych (i odwrotnie). W tym celu wywołuje zarządcę trwałości i stosuje interfejsy zapytań. Ten sposób korzystania z mechanizmów trwałości wymaga, by programista piszący aplikację zdawał sobie sprawę ze stanu i cyklu życia obiektu z uwzględnieniem trwałości. Nazwijmy to zagadnienie **cyklem życia obiektu trwałego**.

Różne implementacje ORM stosują inną terminologię i definiują inne stany oraz przejścia w cyklu życia obiektu trwałego. Co więcej, stany wykorzystywane wewnętrznie mogą być inne od tych udostępnianych aplikacji klienckiej. Hibernate definiuje tylko trzy stany, ukrywając złożoność wewnętrznych implementacji przed kodem klienckim. W niniejszym podrozdziale opiszemy wszystkie trzy stany: **ulotny**, **trwały** i **odłączony**.

Przyjrzyjmy się stanom i ich przejściom na diagramie stanów z rysunku 4.1. Diagram zawiera dodatkowo wywołania metod zarządcy trwałości, które powodują przejścia. Diagram omówimy w tej części rozdziału. W przyszłości będziemy się do niego wielokrotnie odwoływać.



* wpływa na wszystkie egzemplarze związane z obiektem Session

Rysunek 4.1. Stany obiektów i przejścia w aplikacji stosującej Hibernate

W swoim cyklu życia obiekt może przejść od stanu przechodniego przez stan trwałego do stanu odłączenia. Przyjrzyjmy się dokładniej każdemu z nich.

4.1.1. Obiekty ulotne

W Hibernate obiekty utworzone operatorem `new` nie stają się automatycznie trwałymi. Ich stan jest ulotny, czyli nie są związane z żadnym wierszem bazy danych. Znikną, gdy tylko przestanie istnieć referencja do nich w całości aplikacji. Cykl życia obiektów tego typu jest ściśle związany z ich dostępnością. Kończy się, gdy przestają być dostępne i mogą zostać usunięte przez mechanizm odzyskiwania pamięci.

Hibernate traktuje wszystkie egzemplarze ulotne jako nietransakcyjne — modyfikacja stanu takiego obiektu nie wykonuje się w kontekście żadnej transakcji. Można stąd wnioskować, że Hibernate nie zapewnia funkcji wycofywania dla obiektów ulotnych (w zasadzie Hibernate nie wycofuje żadnych zmian w obiektach, ale o tym wkrótce).

Obiekty, których referencje znajdują się w innych ulotnych obiektach, domyślnie również są ulotne. Przejście ze stanu ulotności do stanu trwałości wymaga albo wywołania metody `save()` obiektu zarządzającego trwałością, albo też utworzenia do niego referencji w aktualnie trwałym obiekcie.

4.1.2. Obiekty trwałe

Obiekt trwały to dowolny egzemplarz o **tożsamości bazodanowej** zdefiniowanej w podrozdziale 3.4. Oznacza to istnienie ustawionej wartości klucza głównego równej identyfikatorowi bazodanowemu.

Obiekt trwały mógł zostać utworzony przez aplikację, a następnie uczyniony trwałym przez wywołanie metody `save()` zarządcy trwałości (obiektu `Session` opisywanego dokładniej w dalszej części rozdziału). Po tej operacji obiekt jest związany z zarządcą. Obiekt może stać się trwałym w momencie użycia referencji do niego w innym obiekcie trwałym związanym z zarządcą. Ewentualnie egzemplarz obiektu trwałego może powstać na podstawie informacji z bazy danych uzyskanych w wyniku wykonania zapytania, wyszukania identyfikatora lub przejścia do niego przez graf obiektów z innego obiektu. Innymi słowy, obiekty trwałe zawsze mają związek z obiektem `Session` i są **transakcyjne**.

Obiekty trwałe uczestniczą w transakcjach — ich stan jest synchronizowany z bazą danych na końcu transakcji. W momencie zatwierdzania transakcji stan obiektu znajdujący się w pamięci trafia do bazy danych w wyniku wykonania polecenia `INSERT`, `UPDATE` lub `DELETE`. Procedura ta może wystąpić również w innym momencie. Na przykład Hibernate niejednokrotnie decyduje się na synchronizację z bazą danych przed wykonaniem zapytania. W ten sposób ma pewność, że zapytanie uwzględnii zmiany wprowadzone wcześniej w ramach transakcji.

Obiekt trwały nazywamy **nowym**, gdy zabezpieczył wartość klucza głównego, ale jeszcze nie znalazł się w bazie danych. Obiekt pozostanie „nowy” do momentu synchronizacji.

Oczywiście na końcu transakcji nie występuje aktualizacja bazodanowa wszystkich obiektów trwałych przechowywanych w pamięci. Oprogramowanie ORM musi wykrywać, które z obiektów trwałych zmieniły się w czasie trwania transakcji. Sprawdzenie to nosi nazwę **automatycznego sprawdzania zabrudzenia** (obiekt z modyfikacjami, które nie zostały jeszcze zapisane w bazie danych, traktuje się jako **brudny**). Tego stanu nie widzi aplikacja, więc mówi się o **niewidocznym, transakcyjnym zapisie opóźnionym**. Hibernate stara się jak najbardziej opóźnić propagację zmian do bazy danych, ale ukrywa szczegóły tej działalności przed aplikacją.

Hibernate wykrywa, które atrybuty obiektu uległy zmianie, więc tylko je może dołączać do polecenia aktualizacyjnego `UPDATE`. Pozwala to zwiększyć wydajność, szczególnie w niektórych bazach danych. Najczęściej jednak różnica w wydajności nie jest znaczna. Co więcej, w teorii w niektórych środowiskach może prowadzić do zmniejszenia wydajności. Domyślnie Hibernate dołącza wszystkie kolumny do polecenia `UPDATE` (w ten sposób szablon polecenia powstaje w momencie uruchamiania systemu, a nie w trakcie jego pracy). Jeśli chce się tylko uaktualnić

zmodyfikowane kolumny, można włączyć dynamiczne generowanie SQL, stosując argument `dynamic-update="true"` w odwzorowaniu klasy. Zauważ, że tę funkcję bardzo trudno wprowadzić, ręcznie implementując warstwę trwałości. Semantykę transakcyjną i proces synchronizacji (nazywany **opróżnianiem**) szczegółowo opisuje kolejny rozdział.

Obiekt trwały może stać się ulotnym po wywołaniu metody `delete()` interfejsu zarządcy trwałości powodującej usunięcie z bazy danych wiersza z danymi obiektu.

4.1.3. Obiekt odłączony

Po zakończeniu transakcji obiekty trwałe powiązane z zarządcą trwałości nadal istnieją (jeśli transakcja była udana, ich stan w pamięci został zsynchronizowany ze stanem w bazie danych). W implementacjach ORM z **identycznością o zasięgu procesu** (patrz dalsza część rozdziału) egzemplarze zachowują powiązanie z zarządcą i nadal uważa się je za trwałe.

W Hibernate w momencie wywołania metody `close()` obiektu Session egzemplarze tracą połączenie z zarządcą trwałości. Obiekty te uważa się za odłączone, gdyż nie gwarantuje się synchronizacji ich danych z informacjami w bazie danych, bo nie są od tego momentu zarządzane przez Hibernate. Z drugiej strony obiekty nadal zawierają trwałe dane (które wkrótce mogą się zdezaktualizować). Możliwym (a nawet częstym) jest, że aplikacja przechowuje referencję do obiektu odłączonego poza transakcją (i poza zarządcą). Hibernate dopuszcza ponowne użycie obiektu w nowej transakcji przez jego powiązanie z nowym zarządcą transakcji (po przypisaniu obiekt ponownie traktuje się jako trwały). Ta cecha w znaczącym stopniu wpływa na sposób projektowania aplikacji wielowarstwowych. Możliwość zwracenia obiektów z jednej transakcji do warstwy prezentacji i ponowne użycie ich w innej transakcji stanowi jeden z głównych powodów popularności Hibernate. Przykład użycia opisanego podejścia znajduje się w kolejnym rozdziale, w którym implementujemy długo działające **transakcje aplikacyjne**. Pokażemy również, jak uniknąć antywzorca DTO, wykorzystując obiekty odłączone w rozdziale 8.

Hibernate jawnie udostępnia operację odłączenia — metodę `evict()` obiektu Session. Stosuje się ją jednak głównie w zarządzaniu buforami (ze względów wydajnościowych). Dokonanie jawnego odłączenia **nie** jest operacją typową. Wszystkie obiekty pobrane w trakcie trwania transakcji są niejawnie odłączane w momencie zamknięcia sesji lub w momencie ich serializacji (na przykład w celu przesłania do innego komputera). Hibernate nie musi udostępniać funkcjonalności związanej z sterowaniem odłączania **podgrafów**. Aplikacja steruje głębią pobieranych podgrafów (obiektów aktualnie wczytanych do pamięci), używając języka zapytań i bezpośredniego poruszania się po grafie. W momencie zamknięcia sesji cały podgraf (wszystkie obiekty powiązane z zarządcą trwałości) zostaje odłączony.

Przyjrzyjmy się raz jeszcze poszczególnym stanom, ale tym razem od strony zasięgu identyczności obiektów.

4.1.4. Zasięg identyczności obiektów

Programiści aplikacji sprawdzają identyczność obiektów Javy kodem `a==b`. Jeśli obiekt zmieni stan, czy jego tożsamość w Javie (identyczność) pozostanie ta sama? W aplikacji warstwowej nie musi tak być.

Aby dokładnie zrozumieć ten temat, trzeba pojąć związek między identycznością w Javie (`a==b`) i tożsamością bazodanową (`a.getId().equals(b.getId())`). Czasem oba podejścia dają ten sam wynik, a czasem nie. Warunki, w których identyczność obiektów Javy jest równoważna tożsamości bazodanowej, noszą nazwę **zasięgu identyczności obiektów**.

Istnieją trzy rozwiązania dotyczące zasięgu:

- ◆ Bardzo proste warstwy trwałości bez zasięgu identyczności nie gwarantują, że dwukrotne pobranie tego samego wiersza danych spowoduje zwrócenie do aplikacji dokładnie tego samego obiektu. Wtedy znaczącym problemem staje się modyfikacja przez aplikację dwóch obiektów dotyczących jednego wiersza bazy danych (jak zdecydować, który ze stanów obiektów zapisać trwale w bazie danych?).
- ◆ Warstwa trwałości z transakcyjnym zasięgiem identyczności gwarantuje, że w kontekście pojedynczej aplikacji istnieje tylko jeden obiekt reprezentujący konkretny wiersz bazy danych. Unika się w ten sposób opisanego wcześniej problemu i dopuszcza stosowanie ograniczonego buforowania na poziomie transakcji.
- ◆ **Identyczność o zasięgu procesu** idzie o krok dalej i zapewnia, że w całym procesie (JVM) istnieje tylko co najwyżej jeden obiekt reprezentujący dany wiersz bazy danych.

W typowej aplikacji internetowej lub biznesowej preferuje się identyczność z zasięgiem transakcyjnym. Identyczność o zasięgu procesu oferuje pewne zalety związane z lepszym buforowaniem i modelem programistycznym wielokrotnego użycia tych samych obiektów w wielu transakcjach. Z drugiej jednak strony w wysoce wielowątkowej aplikacji koszt ciągłej synchronizacji dostępu do obiektów trwałych w globalnej mapie identyczności jest zbyt wysoki. Znacznie prostsze i bardziej skalowalne rozwiązanie polega na stosowaniu osobnych zestawów obiektów trwałych dla każdego wątku, by uzyskać identyczność transakcyjną.

Ogólnie możemy powiedzieć, że Hibernate implementuje identyczność z zasięgiem transakcyjnym. W zasadzie jednak zasięg ma ścisły związek z obiektem Session, więc mamy gwarancję identycznych obiektów, jeśli tylko ten sam zarządcą trwałości (Session) będzie uczestniczył w kilku transakcjach. Obiektu Session nie należy utożsamiać z transakcją (bazodanową), bo jest znacznie bardziej elastyczny. Kolejny rozdział omawia różnice i konsekwencje takiego podejścia. Skupmy się ponownie na cyklu życia obiektu trwałego i zasięgu jego identyczności.

Jeśli zażąda się za pomocą jednego obiektu Session dwóch obiektów o tym samym identyfikatorze bazodanowym, uzyska się dwie referencje do tego samego obiektu w pamięci. Poniższy kod obrazuje to zachowanie, stosując kilka operacji `load()` w dwóch sesjach.

```
Session session1 = sessions.openSession();
Transaction tx1 = session1.beginTransaction();

// Załaduj obiekt Category o identyfikatorze 1234.
Object a = session1.load(Category.class, new Long(1234));
Object b = session1.load(Category.class, new Long(1234));

if (a==b) {
    System.out.println("a i b są identyczne.");
}

tx1.commit();
session1.close();

Session session2 = sessions.openSession();
Transaction tx3 = session2.beginTransaction();

Object b2 = session2.load(Category.class, new Long(1234));

if (a!=b2) {
    System.out.println("a i b2 nie są identyczne.");
}

tx2.commit();
session2.close();
```

Obiekty wskazywane przez referencje a i b nie tylko mają tę samą tożsamość bazodanową, ale również tę samą identyczność javową, bo zostały załadowane przez ten sam obiekt Session. Poza tą granicą Hibernate nie gwarantuje identyczności/javowej, więc a i b2 nie są identyczne. Odpowiednie komunikaty przesyłane na konsolę informują o stanie obiektów.

Aby jeszcze bardziej skomplikować rozważania na temat zasięgu identyczności, zastanówmy się, w jaki sposób warstwa trwałości obsługuje referencje do obiektów spoza zasięgu identyczności. Czy Hibernate stosujący warstwę trwałości z identycznością o zasięgu transakcji dopuszcza referencję do obiektu odłączonego (czyli obiektu utrwalonego lub załadowanego w poprzedniej sesji)?

4.1.5. Poza zasięgiem identyczności

Jeśli referencja do obiektu opuszcza strefę gwarantowanej identyczności, nazywamy ją **referencją do obiektu odłączonego**. Dlaczego to szczególne wyróżnienie tego rodzaju referencji jest istotne?

W aplikacjach internetowych najczęściej nie rozkłada się jednej transakcji na kilka działań użytkownika. Użytkownikowi niejednokrotnie bardzo dużo czasu zabiera zastanawianie się nad modyfikacjami. By aplikacja działała szybko i była skalowalna, transakcje bazodanowe należy przeprowadzać sprawnie i od razu zwalniać zasoby bazodanowe. W takim środowisku warto ponownie skorzystać z referencji do obiektu odłączonego. Przykładowo, można wysłać obiekt otrzymany przez jedną jednostkę zadaniową do warstwy prezentacji i uzyskane wyniki po modyfikacjach użytkownika przesyłać do innej jednostki zadaniowej.

W drugiej jednostce najczęściej nie chce się ponownie dołączać całego grafu obiektów. Z powodów wydajnościowych (i innych) lepiej jest, gdy ponowne dołączanie odłączonych obiektów jest selektywne. Hibernate obsługuje selektywne **dołączanie wcześniej odłączonych obiektów**. W ten sposób aplikacja może wydajnie dołączyć podgraf grafu obiektów odłączonych do aktualnej („drugiej”) wersji obiektu Session. Po dołączeniu obiektu do zarządcy trwałości staje się on obiektem trwałym — jego stan zostanie zsynchronizowany z bazą danych na końcu transakcji (z racji automatycznego sprawdzania przez Hibernate zabrudzenia obiektów trwałych).

Ponowne dołączenie może skutkować powstaniem nowych wierszy w bazie danych, jeśli powstała w obiekcie odłączonym referencja do obiektu ulotnego. Na przykład warstwa prezentacji mogła dodać do odłączonego obiektu Item nową ofertę (obiekt Bid). Hibernate wykrywa, że obiekt Bid jest nowy i należy umieścić go w bazie danych. By tego dokonać, musi potrafić odróżnić nowy obiekt ulotny od starego obiektu odłączonego. Obiekty ulotne (w przykładzie obiekt Bid) muszą zostać zapisane; obiekty odłączone (w przykładzie obiekt Item) powinny być ponownie dołączone (i w momencie końca transakcji uaktualnione). Istnieje kilka sposobów wykrywania rodzaju obiektu — jedno z najlepszych sprawdza wartość właściwości identyfikującej. Hibernate sprawdza identyfikator obiektu ulotnego i odłączonego (w momencie ponownego dołączania) i reaguje odpowiednio w zależności od zawartości. Dokładniej opisujemy to zagadnienie w podrozdziale 4.3.4.

Gdy chce się skorzystać z obsługi ponownego dołączania odłączonych obiektów we własnej aplikacji, warto już na etapie projektowania aplikacji dobrze znać zasięg identyczności, czyli zasięg obiektu Session gwarantującego identyczne egzemplarze. Jak tylko wyjdzie się poza opisany zasięg i istnieją odłączone egzemplarze, w grę zaczynają wchodzić kolejne interesujące koncepcje.

Musimy opisać związek między równością obiektów Javy (patrz punkt 3.4.1) i tożsamością bazowaną. Równość to pojęcie identyczności, którym może (a czasem wręcz musi) sterować programista krejący klasy dla obiektów odłączonych. Równość obiektów Javy określa implementacja metod equals() i hashCode() klas trwałych modelu dziedzinowego.

4.1.6. Implementacja equals() i hashCode()

Z metody equals() korzysta kod aplikacji, a co ważniejsze również kolekcje Javy. Przykładowo, kolekcja Set wywołuje metodę equals() dla każdego wstawianego do niej obiektu, by sprawdzić, czy obiekt nie jest duplikatem innego elementu.

Najpierw rozważmy domyślną implementację metody equals() zapewnianą przez klasę java.lang.Object, która sprawdza równość, testując identyczność. Hibernate gwarantuje, że wewnątrz obiektu Session istnieć będzie co najwyżej jeden egzemplarz klasy związany z konkretnym wierszem bazy danych. Wynika stąd, że domyślna implementacja metody equals() jest odpowiednia, gdy nie mieści się egzemplarzy, czyli nigdy nie umieszcza obiektów odłączonych różnych sesji do tego samego zbioru (obiektu Set). W zasadzie ten sam problem pojawia

się również w jednej sesji, jeśli obiekty były serializowane i odserializowane w różnych zasięgach. Jeśli mamy do czynienia z obiektami z różnych sesji, pojawia się możliwość istnienia dwóch nieidentycznych obiektów Item związkanych z tym samym wierszem bazy danych. Można utworzyć złożoną aplikację z równością opartą na identyczności, jeśli tylko zachowa się dyscyplinę z radzeniem sobie z odłączonymi obiektami z różnych sesji (i jednocześnie ostrożnie korzysta się z serializacji). To podejście daje dużą szansę na uniknięcie potrzeby pisania dodatkowego kodu określającego równość obiektów.

Jeśli opisany rodzaj sprawdzania identyczności nie jest odpowiedni, należy przesłaniać metodę equals() w klasach trwałych. Warto pamiętać, że przesłonięcie equals() wiąże się również z przesłonięciem metody hashCode(), by obie metody były spójne (jeśli dwa obiekty są sobie równe, mają te same skróty). Przyjrzymy się kilku sposobom przesłaniania obu metod w klasach trwałych.

Wykorzystanie równości identyfikatorów bazodanowych

Sprytnie rozwiązanie implementuje metodę equals() porównującą jedynie wartość właściwości identyfikatora bazodanowego (najczęściej w postaci sztucznego klucza głównego).

```
public class User {  
    ...  
  
    public boolean equals(Object other) {  
        if (this==other) return true;  
        if (id==null) return false;  
        if (!(other instanceof User)) return false;  
        final User that = (User) other;  
        return this.id.equals(that.getId());  
    }  
  
    public int hashCode() {  
        return id==null ?  
            System.identityHashCode(this) :  
            id.hashCode();  
    }  
}
```

Zauważ, w jaki sposób metoda equals() powraca do identyczności obiektów dla egzemplarzy ulotnych (jeśli id==null) nieposiadających identyfikatora bazodanowego. Okazuje się to logiczne, bo nie mogą one mieć tej samej identyczności związanej z trwałością co inny egzemplarz.

Niestety, to rozwiązanie ma jedną poważną wadę: Hibernate przypisuje wartości identyfikatorom dopiero przy zapisie obiektów do bazy danych. Jeśli aplikacja doda niezapisany obiekt do zbioru Set, jego skrót zmieni się w trakcie przechowywania w zbiorze, co przeczy zasadom określonym dla java.util.Set. Wada ta w zasadzie uniemożliwia zapis kaskadowy (omawiany w dalszej części rozdziału). Z tego względu nie polecamy stosowania równości identyfikatorów bazodanowych.

Porównywanie przez wartość

Lepszy sposób polega na uwzględnianiu wszystkich trwałych właściwości klasy w oderwaniu od identyfikatora bazodanowego, w momencie wykonywania porównywania `equals()`. Właśnie w ten sposób większość osób odbiera znaczenie metody; nadaliśmy temu sposobowi nazwę równości **przez wartość**.

Pojęcie „wszystkie właściwości” nie uwzględnia kolekcji. Stan kolekcji ma związek z inną tabelą, więc uwzględnianie go jest niejednokrotnie błędem. Co ważniejsze, w ten sposób unika się pobierania całego grafu obiektu tylko w celu sprawdzenia równości. W przypadku klasy użytkownika (`User`) nie należy dodać do porównania kolekcji `items` (przedmiotów sprzedanych przez użytkownika). Implementacja metody `equals()` mogłaby wyglądać następująco:

```
public class User {  
    ...  
  
    public boolean equals(Object other) {  
        if (this==other) return true;  
        if (!(other instanceof User)) return false;  
        final User that = (User) other;  
        if (!this.getUsername().equals(that.getUsername()))  
            return false;  
        if (!this.getPassword().equals(that.getPassword()))  
            return false;  
        return true;  
    }  
  
    public int hashCode() {  
        int result = 14;  
        result = 29 * result + getUsername().hashCode();  
        result = 29 * result + getPassword().hashCode();  
        return result;  
    }  
}
```

Niestety, to podejście również ma wady. Oto one:

- ◆ Egzemplarze z różnych sesji przestają być równe, jeśli zmieni się zawartość tylko jednego z nich (na przykład w momencie zmiany hasła przez użytkownika).
- ◆ Egzemplarze z różną tożsamością bazodanową (egzemplarze reprezentujące różne wiersze tabeli bazy danych) system uzna za równe, jeśli nie istnieje żadna wyróżniająca je kombinacja wartości. W przypadku klasy `User` prawdopodobieństwo takiej sytuacji jest zerowe, bo istnieje unikatowa właściwość: `username`.

By zrozumieć zalecane rozwiązanie, należy rozumieć zapis **klucza biznesowego**.

Równość kluczy biznesowych

Klucz biznesowy to właściwość lub pewna kombinacja właściwości, która jest unikatowa dla każdego egzemplarza o tej samej tożsamości bazodanowej. Jest to klucz naturalny, który zostałby wybrany, gdyby nie klucz sztuczny. W odróżnie-

niu od naturalnego klucza głównego nie występuje tu wymóg braku zmian — zmiany mogą się pojawiać, choć warto je ograniczyć do minimum.

Wydaje się, że każda encja powinna zawierać klucz biznesowy, nawet jeśli są to wszystkie właściwości klasy (takie podejście może być odpowiednie dla niektórych klas niezmiennych). Klucz biznesowy to wartość, jaką użytkownik traktuje jako unikatowy identyfikator rekordu. Klucz sztuczny stosuje się, by zwiększyć wygodę i szybkość działania aplikacji oraz bazy danych.

Równość klucza biznesowego oznacza, że metoda `equals()` porównuje jedynie te właściwości, które tworzą klucz biznesowy. To idealne podejście pozwalające uniknąć wymienionych wcześniej problemów. Jedyna wada to wymóg określenia z wyprzedzeniem poprawnego klucza biznesowego. Zadanie to przeprowadza się jednak najczęściej w związku z projektowaniem bazy danych, by wprowadzić sprawdzanie ograniczeń pomagające zapewniające lepszą integralność danych.

W klasie User doskonałym kandydatem na klucz biznesowy jest właściwość `username`: nigdy nie jest pusta, jest unikatowa i rzadko się zmienia (jeśli w ogóle się zmienia).

```
public class User {  
    ...  
  
    public boolean equals(Object other) {  
        if (this==other) return true;  
        if (!(other instanceof User)) return false;  
        final User that = (User) other;  
        return this.username.equals(that.getUsername());  
    }  
  
    public int hashCode() {  
        return username.hashCode();  
    }  
}
```

W niektórych innych klasach klucz biznesowy bywa bardziej złożony, bo składa się z kilku właściwości. Kandydatami na klucze biznesowe klasy Bid mogą być identyfikator przedmiotu wraz z oferowaną kwotą lub identyfikator przedmiotu wraz z datą złożenia oferty. Dobrym kluczem biznesowym dla klasy abstrakcyjnej `BillingDetails` jest właściwość `number` wraz z typem (podklassą) sposobu płatności. Prawie nigdy nie należy przesyłać metody `equals()` w podklasie i stosować do porównań innej właściwości. Nie jest łatwo spełnić w takiej sytuacji wymogu symetryczności i przechodniości równości. Zmieniony klucz biznesowy mógłby nie odpowiadać żadnemu dobrze zdefiniowanemu kandydatowi klucza naturalnego w bazie danych (podklasy mogą przecież znaleźć się w innych tabelach).

Zauważ, że metody `equals()` i `hashCode()` zawsze korzystają z metod dostępowych w celu pobrania właściwości innych obiektów. Jest to ważne, bo obiekt przekazany jako `other` nie musi być rzeczywistym egzemplarzem przechowującym trwałego stan, ale obiektem pośredniczącym. To jedno z miejsc, w których Hibernate nie jest w pełni przezroczysty. Niemniej zawsze zaleca się stosowanie metod dostępowych zamiast bezpośredniego dostępu do właściwości.

Uważaj na zmiany wartości właściwości stanowiących klucz biznesowy. Nie zmieniaj ich, gdy obiekt dziedzinowy znajduje się w zbiorze.

W tym rozdziale wspominaliśmy o zarządcy trwałości. Najwyższy czas dokładniej mu się przyjrzeć i szczegółowo opisać interfejs programistyczny obiektu Session. W kolejnym rozdziale raz jeszcze powrócimy do tematu obiektów odłączonych.

4.2. Zarządcy trwałości

Każde narzędzie trwałości przezroczystej udostępnia interfejs programistyczny zarządcy trwałości, który udostępnia usługi związane z:

- ◆ podstawowymi operacjami CRUD,
- ◆ wykonywaniem zapytań,
- ◆ sterowaniem transakcjami,
- ◆ zarządzaniem buforowaniem na poziomie transakcji.

Zarządcy może składać się z wielu interfejsów (w Hibernate są to interfejsy Session, Query, Criteria i Transaction), ale są one ze sobą ścisłe powiązane wewnętrznie.

Główny interfejs między aplikacją a Hibernate stanowi obiekt Session. To punkt początkowy wszystkich wymienionych operacji. W dalszej części książki zamiennie stosujemy terminy **zarządcą trwałości i sesja**, bo właśnie te stwierdzenia wykorzystuje społeczność użytkowników Hibernate.

Jak rozpocząć stosowanie sesji? Na początku jednostki zadaniowej wątek pobiera obiekt Session z obiektu SessionFactory aplikacji. Aplikacja może stosować wiele obiektów SessionFactory, jeśli używa wielu źródeł danych. Nigdy nie należy tworzyć nowego SessionFactory tylko do obsługi pojedynczego żądania — koszt jego powstania jest ogromny. Z drugiej strony tworzenie obiektów Session jest wyjątkowo tanie, bo obiekt ten nie pobiera obiektu połączenia JDBC (Connection), dopóki nie jest to potrzebne.

Otwarta sesja służy do wczytywania i zapisu obiektów.

4.2.1. Czynienie obiektu trwałym

Pierwszym zadaniem wykonywanym z użyciem obiektem Session jest najczęściej zamiana nowego obiektu ulotnego na obiekt trwały. Do tego celu służy metoda save().

```
User user = new User();
user.getName.setFirstName("Jan");
user.getName.setLastName("Kowalski");

Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

session.save(user);
```

```
tx.commit();
session.close();
```

Najpierw w tradycyjny sposób kreujemy obiekt ulotny user. Oczywiście nic nie stoi na przeszkodzie, by tworzyć go po utworzeniu obiektu Session, bo jeszcze nie występuje między nimi żadne powiązanie. Otwieramy nową sesję, używając obiektu SessionFactory (referencja sessions) i uruchamiany nową transakcję bazodanową.

Wywołanie metody save() czyni obiekt klasy User trwałym i wiąże go z sesją. Na razie Hibernate nie wykonał polecenia INSERT z SQL. Obiekt Session wykonuje polecenia SQL dopiero wtedy, gdy są naprawdę konieczne.

Zmiany w obiekcie trwałym w pewnym momencie należy zsynchronizować z bazą danych. Zadanie to rozpoczyna wywołanie metody commit() obiektu Transaction. Hibernate pobiera połączenie JDBC i wykonuje jedno polecenie INSERT. Na końcu sesja zostaje zamknięta, a połączenie JDBC zwolnione.

Zaleca się (choć nie stanowi to wymogu), by w pełni zainicjalizować obiekt User przed powiązaniem go z obiektem Session. Instrukcja INSERT zawiera wartości przechowywane przez obiekt w momencie wywołania metody save(). Oczywiście można zmienić zawartość obiektu po wywołaniu save(), ale wtedy Hibernate prześle zmiany do bazy danych w poleceniu UPDATE.

Wszystko między session.beginTransaction() i tx.commit() odbywa się w jednej transakcji bazodanowej. Dokładny opis transakcji znajduje się w kolejnym rozdziale. Warto zdawać sobie sprawę z faktu, iż wszystkie operacje bazodanowe znajdujące się wewnątrz transakcji albo są pełnym sukcesem, albo całkowitą porażką. Jeśli choć jedno z poleceń INSERT lub UPDATE wykonywanych przez tx.commit() nie uda się, wszystkie zmiany wprowadzone do obiektów trwałych w czasie transakcji ulegną wycofaniu na poziomie bazy danych. Hibernate nie wycofuje zmian w obiektach trwałych zawartych w pamięci. Ma to sens, gdyż błędą transakcji bazodanowej najczęściej nie można naprawić, więc od razu należy odrzucić całą sesję.

4.2.2. Aktualizacja stanu trwałego obiektu odłączonego

Modyfikacja obiektu user po zamknięciu sesji nie będzie miała wpływu na jego postać w bazie danych. Po zamknięciu sesji egzemplarz user staje się obiektem odłączonym. Można go przypisać do nowego obiektu Session metodą update() lub lock().

Metoda update() wymusza aktualizację stanu obiektu trwałego w bazie danych, zapamiętując potrzebę wykonania polecenia UPDATE. Oto przykład obsługi obiektu odłączonego:

```
user.setPassword("tajne");

Session sessionTwo = sessions.openSession();
Transaction tx = sessionTwo.beginTransaction();

sessionTwo.update(user);
```

```
user.setUsername("janek");

tx.commit();
sessionTwo.close();
```

Nie ma znaczenia, czy zawartość obiektu zmieni się przed czy po wywołaniu update(). Istotny jest sam fakt wywołania metody update() powodujący przypisanie obiektu odłączonego do nowej sesji (i aktualnej transakcji) oraz poinformowanie Hibernate, by potraktował obiekt jako brudny (chyba że włączona została opcja select-before-update w odwzorowaniu klasy trwałej, która powoduje pobranie stanu obiektu z bazy danych i porównanie go ze stanem w pamięci w celu wykrycia różnic).

Wywołanie metody lock() wiąże obiekt z sesją bez wymuszania aktualizacji.

```
Session sessionTwo = sessions.openSession();
Transaction tx = sessionTwo.beginTransaction();

sessionTwo.lock(user, LockMode.NONE);

user.setPassword("tajne");
user.setUsername("janek");

tx.commit();
sessionTwo.close();
```

W tym przypadku ma znaczenie, czy zmiany zostaną wykonane przed czy po powiązaniu obiektu z sesją. Zmiany wykonane przed wywołaniem nie znajdą się w bazie danych. Z tego względu metodę należy stosować tylko wtedy, gdy odłączony obiekt nie uległ zmianie.

Tryby blokad Hibernate opiszymy w kolejnym rozdziale. Stosując LockMode.NONE, informujemy system, by nie przeprowadzał sprawdzenia wersji ani nie zakładał żadnych bazodanowych blokad w momencie dołączania obiektu do sesji. Zastosowanie LockMode.READ lub LockMode.UPDATE spowoduje wykonanie instrukcji SELECT w celu sprawdzenia wersji (i ustawnienia blokady aktualizacyjnej).

4.2.3. Pobranie obiektu trwałego

Obiekt Session służy również do odpytywania bazy danych i pobierania istniejących obiektów trwałych. Hibernate zapewnia w tym obszarze dużą swobodę i elastyczność, co postaramy się zobrazować w tym i kolejnym rozdziale. Interfejs zawiera specjalne metody pomocnicze ułatwiające wykonanie najprostszego zapytania — pobrania obiektu na podstawie identyfikatora. Poniższy kod przedstawia jedną z tych metod o nazwie get().

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

int userId = 1234;
User user = (User) session.get(User.class, new Long(userID));

tx.commit();
session.close();
```

Pobrany obiekt user można przekazać do warstwy prezentacji, by został użyty poza transakcją jako obiekt odłączony (po zamknięciu sesji). Jeśli baza danych nie zawiera wiersza o podanym identyfikatorze, metoda get() zwraca wartość null.

4.2.4. Aktualizacja obiektu trwałego

Dowolny obiekt trwały zwrócony przez get() lub inną metodę wykonywania zapytań jest powiązany z aktualną sesją i kontekstem transakcji. Jeśli zostanie zmieniony, Hibernate zsynchronizuje jego stan z bazą danych. Mechanizm ten nosi nazwę **automatycznego sprawdzania zabrudzenia**. System sam śledzi i zapisaże wszystkie zmiany dokonane w obiekcie w obrębie sesji.

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

int userId = 1234;
User user = (User) session.get(User.class, new Long(userID));

user.setPassword("tajne");

tx.commit();
session.close();
```

Najpierw kod pobiera z bazy danych obiekt o wskazanym identyfikatorze. Następnie go modyfikuje i zmiany propaguje do bazy danych w momencie wywołania metody tx.commit(). Oczywiście zaraz po zamknięciu sesji obiekt należy traktować jako odłączony.

4.2.5. Zmiana obiektu trwałego na ulotny

Obiekt trwały można łatwo zamienić na ulotny, usuwając jego trwałego stan z bazy danych dzięki wywołaniu metody delete().

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

int userId = 1234;
User user = (User) session.get(User.class, new Long(userID));

session.delete(user);

tx.commit();
session.close();
```

Hibernate wykona polecenie DELETE z SQL dopiero w momencie synchronizacji sesji z bazą danych na końcu transakcji.

Po zamknięciu sesji obiekt user staje się typowym obiektem bez trwałości. Zostanie zniszczony przez system odzyskiwania pamięci, jeśli nie odnoszą się do niego żadne referencje. Wtedy całkowicie zniknie zarówno z pamięci, jak i z tabeli bazy danych.

4.2.6. Zmiana obiektu odłączonego na ulotny

Obiekt odłączony zmienia się na ulotny po usunięciu jego stanu trwałego z bazy danych. Nie trzeba w tym celu ponownie dołączać obiektu (metodami `update()` lub `lock()`) do sesji przed jego usunięciem. Wystarczy od razu wywołać metodę `delete()`.

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

session.delete(user);

tx.commit();
session.close();
```

W tym przypadku metoda `delete()` wykonuje dwa zadania: dołącza obiekt do sesji i harmonogramuje jego usunięcie w momencie wykonania metody `tx.commit()`.

Przedstawiliśmy cykl życia obiektu trwałego oraz podstawowe operacje zarządcy trwałości. Łącząc tę wiedzę ze sposobem kreowania odwzorowań klas trwałych opisany w poprzednim rozdziale, można z powodzeniem zacząć tworzyć proste aplikacje stosujące Hibernate. (Jeśli chcesz, możesz przejść do rozdziału 8., by poznać klasę pomocniczą ułatwiającą korzystanie z obiektów `SessionFactory` i `Session`). Zauważ, że w żadnym z przykładów nie przedstawiliśmy kodu obsługi wyjątków. Sposób obsługi bloków `try-catch` jest taki sam, jak w dowolnym innym kodzie Javy, więc napisanie go nie powinno sprawić dużych trudności. Napisz kilka klas trwałych, wykonaj ich odwzorowania i załaduj obiekty w samowystarczalnej aplikacji (nie potrzeba żadnego kontenera serwletów ani serwera aplikacji, wystarczy metoda `main()`). Gdy jednak chce się zapisać powiązane obiekty, czyli ma się do czynienia z bardziej złożoną ich strukturą, wywoływanie metod `save()` i `delete()` dla każdego obiektu byłoby mało wydajnym sposobem pisania aplikacji.

Chcemy wykonać jak najmniej wywołań metod obiektu `Session`. **Trwałość przechodnia** zapewnia bardziej naturalny sposób wymuszania zmian stanów obiektów i sterowania cyklem życia trwałości.

4.3. Trwałość przechodnia w Hibernate

Rzeczywiste aplikacje najczęściej nie korzystają z pojedynczych obiektów, ale z całych grafów powiązanych obiektów. Gdy aplikacja przetwarza graf obiektów trwałych, efektem tych zmian może być graf obiektów składający się z obiektów trwałych, odłączonych i ulotnych. **Trwałość przechodnia** to technika umożliwiająca automatyczną propagację trwałości na podgrafy ulotne i odłączone.

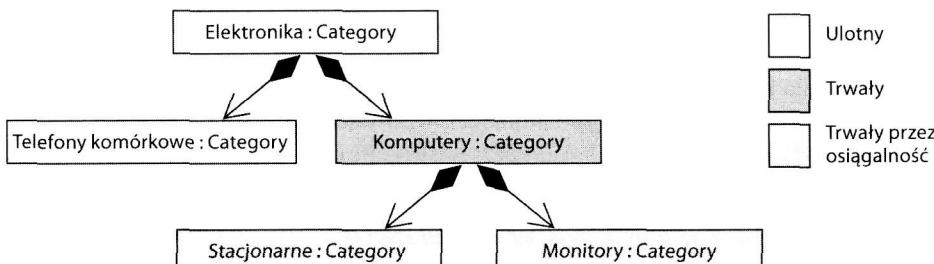
Na przykład, jeśli dodamy nowy obiekt `Category` do aktualnie trwałej hierarchii kategorii, powinien on automatycznie stać się trwały bez potrzeby wywoływania metody `Session.save()`. Nieco inny przykład pojawił się w rozdziale 3., w którym tworzyliśmy odwzorowanie dla związku rodzic-potomek między klasami `Bid` i `Item`. W tamtym rozwiązaniu oferty nie tylko były automatycznie

utrwalane po dodaniu do przedmiotu, ale również automatycznie usuwane z bazy w momencie usuwania przedmiotu aukcji.

Istnieje kilka modeli trwałości przechodniej. Najlepszym z nich jest **trwałość przez osiągalność** omawiana w kolejnym punkcie. Choć pewne podstawowe zasady pozostają te same, Hibernate stosuje własny, bardziej rozwinięty model trwałości przechodniej.

4.3.1. Przechodniość przez osiągalność

Mówiąc, że warstwa trwałości obiektów implementuje trwałość przez osiągalność, jeśli dowolny obiekt staje się trwałym w momencie utworzenia referencji do niego w innym obiekcie, który już jest trwałym. Zachowanie to obrazuje diagram obiektów (nie jest to diagram klas) na rysunku 4.2.



Rysunek 4.2. Trwałość przez osiągalność z głównym obiektem trwałym

W przykładzie obiekt reprezentujący kategorię „Komputery” jest trwałym. Obiekty dotyczące kategorii „Stacjonarne” i „Monitory” również są trwałymi, ponieważ są osiągalne z poziomu obiektu „Komputery”. Obiekty dotyczące kategorii „Elektronika” i „Telefony komórkowe” są ulotne. Zwróć uwagę na założenie, że nawigacja możliwa jest tylko od kategorii głównej do potomnej, więc dostępna jest metoda `computer.getChildCategories()`. Trwałość przez osiągalność jest rekurencyjna. Wszystkie obiekty osiągalne z poziomu obiektu trwałego stają się stałymi albo w momencie uaktywnienia trwałości oryginalnego obiektu lub też tuż przed zapisem stanu pamięci do bazy danych.

Trwałość przez osiągalność zapewnia integralność więzów referencyjnych. Dowolny graf obiektów można w całości odtworzyć, wczytując jedynie jego korzeń. Aplikacja nie musi przejmować się stanem trwałości egzemplarzy, gdy przechodzi przez graf obiektów od jednej asocjacji do drugiej. Bazy danych SQL stosują inne podejście do zapewnienia integralność więzów referencyjnych — korzystają z kluczy obcych i ograniczeń, by wykryć błędy.

W najczystszej odmianie trwałości przez osiągalność baza danych zawiera obiekt **korzenia**, z którego daje się przejść do dowolnego innego obiektu trwałego. W idealnej sytuacji obiekt trwałym, do którego nie da się przejść z korzenia, powinien zostać jak najszybciej usunięty z bazy danych.

Hibernate i inne rozwiązania ORM nie implementują tej postaci. Nie istnieje w bazach danych SQL żadna analogia do obiektu korzenia. Nie jest również

dostępny system oczyszczania trwałych danych wykrywający obiekty niewskazywane przez żadną referencję. Obiektowe systemy bazodanowe mogą implementować mechanizm oczyszczania podobny do tego stosowanego przez maszynę wirtualną Javy. W świecie ORM nie na takiej możliwości. Przeszukiwanie wszystkich tabel w celu wykrycia braków w referencjach nie jest do zaakceptowania ze względów wydajnościowych.

Trwałość przez osiągalność jest najlepszym możliwym pośrodkiem. Pomaga uzyskać trwałość obiektom ulotnym i propagować je do bazy danych bez potrzeby wykonywania wielu wywołań zarządcy trwałości. Niestety (przynajmniej z punktu widzenia baz danych SQL i ORM), nie jest to pełne rozwiązywanie zamiany obiektów trwałych na ulotne i w efekcie usuwaniu ich z bazy danych. Nie można tak po prostu usunąć wszystkich osiągalnych obiektów w momencie usuwania obiektu nadzawanego; inne trwałe obiekty również mogą zawierać referencję do obiektów podrzędnych (encje można współdzielić). Co więcej, nie daje się zagwarantować bezpiecznego usunięcia egzemplarzy, dla których żaden z obiektów trwałych w pamięci nie ma referencji; obiekty w pamięci najczęściej stanowią jedynie niewielki fragment wszystkich obiektów reprezentowanych w bazie danych. Przyjrzymy się bardziej elastycznemu modelowi trwałości przechodniej stosowanemu przez Hibernate.

4.3.2. Trwałość kaskadowa w Hibernate

Model trwałości przechodniej stosowanej w Hibernate używa tej samej podstawowej koncepcji co trwałość przez osiągalność — sprawdza asocjacje, by określić stan trwałości obiektów. Hibernate zapewnia określanie stylu kaskadowego dla każdego odwzorowania asocjacji, co zapewnia bardziej elastyczną i dokładniejszą kontrolę przechodniości wszystkich stanów. Automatycznie odczytuje zadeklarowany styl i wykonuje operacje kaskadowe dla powiązanych obiektów.

Domyślnie Hibernate nie analizuje asocjacji w trakcie poszukiwań obiektów ulotnych i odłączonych, więc zapis, usunięcie lub ponowne dołączenie obiektu Category nie wpływa na jego podkategorie. Stanowi to przeciwieństwo domyślnego zachowania trwałości przez osiągalność. By w Hibernate włączyć przechodniość trwałości, należy zmienić domyślne zachowanie w metadanych odwzorowania.

Poniższe atrybuty modyfikują sposób zachowania Hibernate dla asocjacji:

- ◆ `cascade="none"` — ustawienie domyślne, powoduje ignorowanie asocjacji przez Hibernate.
- ◆ `cascade="save-update"` — informuje Hibernate, by przeszedł przez asocjację w momencie zatwierdzania transakcji, jeśli obiekt został przekazany metodą `save()` lub `update()`, i włączył trwałość wszystkich napotkanych po drodze obiektów ulotnych i odłączonych.
- ◆ `cascade="delete"` — przechodzi przez asocjację i usuwa obiekty trwałe, gdy główny obiekt przekazano do metody `delete()`.
- ◆ `cascade="all"` — powoduje kaskadowe wywoływanie obu powyższych punktów oraz obsługę metod `evict()` i `lock()`.

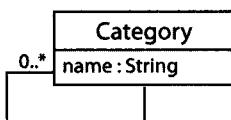
- ◆ `cascade="all-delete-orphan"` — działa podobnie do `cascade="all"`, ale dodatkowo usuwa wszystkie trwałe obiekty encji, które zostały usunięte z asocjacji (na przykład z kolekcji).
- ◆ `cascade="delete-orphan"` — usuwa wszystkie trwałe obiekty encji, które zostały usunięte (przez wyłączenie referencji) z asocjacji (na przykład z kolekcji).

Model stylów kaskadowych ustalanych dla asocjacji jest jednocześnie bardziej elastyczny i mniej bezpieczny od trwałości przez osiągalność. Hibernate nie zapewnia równie mocnych gwarancji co do integralności referencyjnej. Zamiast tego część odpowiedzialności za ich zachowanie kieruje do ograniczeń kluczów obcych nakładanych przez relacyjne bazy danych. Istnieją bardzo konkretne powody takiej decyzji projektowej — aplikacja może wydajnie korzystać z obiektów odłączonych, gdyż na poziomie asocjacji steruje się ponownym dołączaniem obiektów.

Zastanówmy się dłużej nad rozwiązaniem kaskadowym, przedstawiając kilka przykładów odwzorowania asocjacji. Zalecamy przeczytanie następnego podrozdziału w całości, ponieważ kolejne przykłady bazują na poprzednich. Pierwszy przykład jest prosty, umożliwia wygodny sposób zapisu nowych kategorii.

4.3.3. Zarządzanie kategoriami przedmiotów

System administracji aukcjami dopuszcza tworzenie nowych kategorii, zmiany nazw i przenoszenie kategorii w obrębie struktury drzewiastej. Sposób powiązania między kategoriami przedstawia rysunek 4.3.



Rysunek 4.3. Klasa Category z referencją do samej siebie

Dokonajmy odwzorowania klasy i asocjacji.

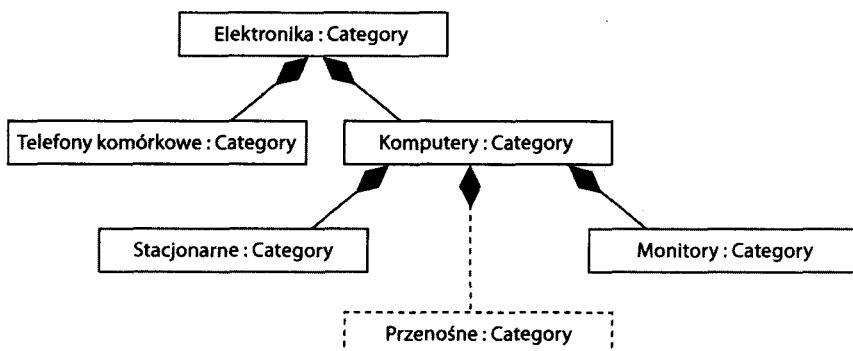
```
<class name="Category" table="CATEGORY">
    ...
    <property name="name" column="CATEGORY_NAME"/>
    <many-to-one
        name="parentCategory"
        class="Category"
        column="PARENT_CATEGORY_ID"
        cascade="none"/>
    <set
        name="childCategories"
        table="CATEGORY"
        cascade="save-update"
        inverse="true">
        <key column="PARENT_CATEGORY_ID"/>
    </set>
</class>
```

```
<one-to-many class="Category"/>
</set>

...
</class>
```

Jest to rekurencyjna, dwukierunkowa asocjacja jeden-do-wielu opisana pokrótko w rozdziale 3. Zakończenie z pojedynczą wartością stanowi element `<many-to-one>`, natomiast zakończenie „wiele” to typ Set oznaczony elementem `<set>`. Obie części dotyczą tej samej kolumny klucza obcego `PARENT_CATEGORY_ID`.

Przypuśćmy, że utworzyliśmy nowy obiekt Category jako potomka kategorii „Komputery” (patrz rysunek 4.4).



Rysunek 4.4. Trwałość przez osiągalność z głównym obiektem trwałym

Istnieje kilka sposobów utworzenia nowego obiektu i zapisania go w bazie danych. Możemy pobrać z bazy danych kategorię „Komputery”, do której mamy zamiar przypisać kategorię „Przenośne”, dodać nową kategorię i zatwierdzić transakcję.

```

Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

Category computer = (Category) session.get(Category.class, computerId);
Category laptops = new Category("Przenośne");

computer.getChildCategories().add(laptops);
laptops.setParentCategory(computer);

tx.commit();
session.close();
```

Obiekt `computer` jest trwały (powiązany z sesją). Asocjacja `childCategories` ma włączony zapis kaskadowy. W ten sposób kod tworzy nową kategorię trwałą w momencie wywołania `tx.commit()`, bo Hibernate wykryje zmianę w liczbie potomków kategorii `computer` i wykona polecenie `INSERT` dodające kategorię.

Wykonajmy przedstawione zadanie raz jeszcze, ale tym razem powiążmy obie kategorie poza jakimkolwiek transakcjami (w aplikacjach rzeczywistych bardzo wygodna okazuje się modyfikacja grafu obiektów na warstwie prezentacji, czyli przed przekazaniem grafu do warstwy trwałości w celu jego zapisu).

```
Category computer = ... // Wczytane w poprzedniej sesji.  
Category laptops = new Category("Przenośne");  
  
computer.getChildCategories().add(laptops);  
laptops.setParentCategory(computer);
```

Odlaczony obiekt computer i inne powiazane z nim obiekty odlaczone sa powiazane z nowym obiektem ulotnym laptop (i na odwrotnie). Czynimy zmiany w grafie obiektow trwałymi, zapisując nowy obiekt w kolejnej sesji Hibernate.

```
Session session = sessions.openSession();  
Transaction tx = session.beginTransaction();  
  
// Utrwalenie nowej kategorii i połączenie jej z kategorią nadzczną.  
session.save(laptops);  
  
tx.commit();  
session.close();
```

Hibernate sprawdzi właściwości identyfikatora bazy danych kategorii nadzędnej dla laptop i poprawnie wykona w bazie danych powiązanie z kategorią „Komputery”. Dodatkowo w momencie wstawiania nowej kategorii poleciem INSERT dołączy wartość identyfikatora rodzica do pola klucza obcego nowego wiersza bazy danych.

Ponieważ dla związku parentCategory zdefiniowano cascade="none", Hibernate pomija wszelkie zmiany dokonane w innych kategoriach hierarchii („Komputery”, „Elektronika” itp.). Nie przenosi kaskadowo wywołania save() do enclii wskazywanych przez tę asocjację. Gdyby włączyć atrybut cascade="save-update" dla odwzorowania <many-to-one> z parentCategory, Hibernate musiałby przejść przez cały graf obiektów znajdujący się w pamięci i zsynchronizować go z zawartością bazy danych. Rozwiążanie to obniżałoby wydajność z powodu wielu bezużytecznych zapytań wysyłanych do bazy danych. W przedstawionej sytuacji nie potrzebujemy (a wręcz nie chcemy) trwałości przechodniej dla asocjacji parent-Category.

Do czego przydają się operacje kaskadowe? Obiekt laptop, co przedstawia wcześniejszy przykład, możemy zapisać bez stosowania odwzorowania kaskadowego. Rozważmy inną sytuację.

```
Category computer = ... // Wczytane w poprzedniej sesji.  
  
Category laptops = new Category("Przenośne");  
Category laptopAccesories = new Category("Akcesoria");  
Category laptopTabletPCs = new Category("Tablety");  
  
laptops.addChildCategory().add(laptopAccesories);  
laptops.addChildCategory().add(laptopTabletPCs);  
  
computer.addChildCategory().add(laptops);
```

(Zwróć uwagę na użycie metody pomocniczej addChildCategory() w celu jednoczesnego ustawnienia obu asocjacji w sposób opisany w rozdziale 3.)

Osobne wymuszanie zapisu każdej z trzech dodanych kategorii byłoby mało wygodne. Na szczęście z racji zastosowania w odwzorowaniu asocjacji childCategories atrybutu cascade="save-update" nie musimy tego czynić. Ten sam kod, który posłużył nam wcześniej do zapisu kategorii „Laptopy”, zapisze w nowej sesji również dwie dodatkowe podkategorie.

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

// Utrwalenie wszystkich trzech nowych kategorii.
session.save(laptops);

tx.commit();
session.close();
```

Zapewne się zastanawiasz, dlaczego styl kaskadowy nosi nazwę "save-update" zamiast samego "save". Założmy, że wszystkie trzy dodane wcześniej kategorie zostały utrwalone i w kolejnym żądaniu (poza sesją i transakcją) dokonujemy kilku zmian w hierarchii kategorii.

```
laptops.setName("Komputery przenośne");
laptopAccessories.setName("Akcesoria i części");
laptopTabletPCs.setName("Komputery tabletowe");

Category laptopBags = new Category("Torby na komputery przenośne");
laptops.AddChildCategory(laptopBags);
```

Dodaliśmy nową kategorię jako potomka dawnej kategorii „Przenośne” oraz zmodyfikowaliśmy nazwy trzech kategorii. Poniższy kod propaguje zmiany do bazy danych.

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

// Aktualizacja trzech starych kategorii i wstawienie nowej.
session.update(laptops);

tx.commit();
session.close();
```

Atrybut cascade="save-update" dla asocjacji childCategories dokładnie odpowiada rzeczywistej sytuacji. Hibernate sam określa, które obiekty muszą zostać ponownie przeniesione do bazy danych. W przedstawionym przykładzie dołączy i zaktualizuje trzy odłączone kategorie (laptops, laptopAccessories i laptopTabletPCs) oraz zapisze nową kategorię podrzędną (laptopBags).

Zauważ, że ostatni kod różni się od dwóch wcześniejszych tylko w jednym wierszu — zmieniła się wywoływana metoda. Ostatni przykład wywołuje metodę update() zamiast save(), bo kategoria laptops została wcześniej utrwalona.

Możemy zmodyfikować wszystkie przykłady, by używały metody saveOrUpdate(). W ten sposób wszystkie trzy fragmenty będą identyczne.

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
```

```
// Niech Hibernate zdecyduje, czy obiekt jest nowy czy odłączony.  
session.saveOrUpdate(laptops);  
  
tx.commit();  
session.close();
```

Metoda `saveOrUpdate()` powoduje propagację stanu obiektu do bazy danych w postaci nowego wiersza, jeśli obiekt jest ulotny, lub w postaci aktualizacji, jeśli obiekt jest odłączony. Innymi słowy, wykonuje dokładnie to samo zadanie dla kategorii `laptops`, które powodował atrybut `cascade="save-update"` dla podkategorii `laptops`.

Ostatnie pytanie: skąd Hibernate wie, które obiekty potomne zostały odłączone, a które są nowymi egzemplarzami?

4.3.4. Rozróżnienie obiektów ulotnych i odłączonych

Ponieważ Hibernate nie przechowuje referencji do odłączonych obiektów, trzeba mu pozwolić rozróżnić obiekt odłączony (na przykład `laptops` z poprzedniego przykładu utworzony przez wcześniejszą sesję) od nowego obiektu ulotnego (na przykład `laptopBags`).

Istnieje wiele rozwiązań. Hibernate zakłada, że obiekt jest nieszanowanym obiektem ulotnym, jeśli:

- ◆ właściwość identyfikującą (o ile istnieje) wynosi `null`,
- ◆ właściwość wersji (o ile istnieje) wynosi `null`,
- ◆ ustawiono w dokumencie odwzorowania klasy właściwość `unsaved-value` i wartość właściwości identyfikującej jest jej równa,
- ◆ ustawiono w dokumencie odwzorowania klasy właściwość `unsaved-value` i wartość właściwości wersji jest jej równa,
- ◆ przekazano obiekt implementujący interfejs `Interceptor`, który zwrócił wartość `Boolean.TRUE` po wywołaniu metody `Interceptor.isUnsaved()` z argumentem będącym testowanym obiektem.

W modelu dziedzinowym jako typ identyfikatora zawsze stosowaliśmy typ `java.lang.Long` przyjmujący domyślnie wartość `null`. Ponieważ stosujemy generowane, sztuczne identyfikatory, rozwiązuje to problem. Nowe obiekty mają wartość identyfikatora ustawioną na `null`, więc Hibernate traktuje je jako nowe i ulotne. Obiekty odłączone mają niepustą wartość identyfikatora, więc system odpowiednio je rozpoznaje.

Gdybyśmy w klasach trwałych użyli typu prostego `long`, dokument odwzorowania wszystkich klas musiałby zawierać dodatkową informację.

```
<class name="Category" table="CATEGORY">  
    <id name="id" unsaved-value="0">  
        <generator class="native"/>  
    ...  
</class>
```

Atrybut `unsaved-value` informuje Hibernate, by traktował obiekty `Category` z wartością identyfikatora równą 0 jako nowe obiekty ulotne. Domyślnie `unsaved-value` wynosi `null`, więc jako typ właściwości identyfikującej wybraliśmy `Long`. W ten sposób nic nie stoi na przeszkodzie, by pominąć atrybut w odwzorowaniach klas aplikacji zarządzającej aukcjami (ten sam typ występuje we wszystkich klasach).

Przypisane identyfikatory niezapisanych obiektów

Przedstawione rozwiązanie działa doskonale dla identyfikatorów sztucznych, ale spowoduje wiele problemów dla kluczy przypisywanych przez aplikację, włączając w to złożone klucze starszych baz danych. Dokładniej tym tematem zajmiemy się w punkcie 8.3.1. Należy unikać w nowych aplikacjach kluczy nadawanych przez aplikację i kluczy złożonych.

Przedstawiliśmy wiedzę pozwalającą zoptymalizować aplikację Hibernate przez redukcję liczby wywołań zarządcy trwałości związanych z zapisem i usuwaniem obiektów. Sprawdź atrybuty `unsaved-value` wszystkich swoich klas i poeksperymentuj z obiektami odłączonymi, by lepiej zrozumieć model trwałości przechodniej stosowany w Hibernate.

Zmienimy teraz perspektywę i zastanowimy się, jak wydajnie pobrać graf obiektów trwałych z bazy danych, czyli jak wczytać dane.

4.4. Pobieranie obiektów

Pobieranie obiektów trwałych z bazy danych to jeden z najbardziej interesujących (i złożonych) elementów współpracy aplikacji z ORM. Hibernate udostępnia następujące sposoby pobierania obiektów z bazy danych.

- ◆ Nawigację po grafie obiektów, zaczynając od załadowanego już obiektu. Kolejne obiekty pobiera się metodami dostępowymi, na przykład `aUser.getAddress().getCity()`. Hibernate automatycznie załaduje (w miarę możliwości z wyprzedzeniem) węzły grafu, gdy obiekt `Session` jest aktywny.
- ◆ Pobranie obiektu na podstawie jego identyfikatora, co stanowi najbardziej wydajny i wygodny sposób uzyskania obiektu, gdy zna się jego unikatowy identyfikator.
- ◆ Zastosowanie języka HQL (*Hibernate Query Language*), który jest obiektowym językiem zapytań.
- ◆ Wykorzystanie interfejsu programistycznego `Criteria` zapewniającego bezpieczeństwo typu i obiektowy sposób określania zapytań bez potrzeby edycji tekstów. Dodatkowo umożliwia generowanie zapytań bazujących na przykładowym obiekcie.
- ◆ Przekazanie standardowych zapytań SQL. Hibernate automatycznie zajmie się odwzorowaniem zbiorów wyników JDBC na graf obiektów trwałych.

W aplikacjach najczęściej stosuje się połączenie kilku wymienionych technik. Każda z metod pobierania może stosować inną metodę sprowadzania danych, czyli strategię definiującą, jaka część grafu obiektów trwałych powinna trafić do pamięci operacyjnej. Dąży się do znalezienia jak najlepszej metody pobierania i sprowadzania danych dla każdej sytuacji przy jednoczesnej minimalizacji liczby zapytań SQL w celu maksymalizacji wydajności.

W niniejszym podrozdziale nie zajmiemy się szczegółowym opisem poszczególnych metod pobierania danych. Skupimy się na podstawowych strategiach sprowadzania danych i sposobach dostrajania plików odwzorowań, by uzyskać dobrą wydajność domyślną dla wszystkich metod dostępu. Przed przystąpieniem do opisu strategii przedstawimy pokrótko metody pobierania. Wspomnimy też o systemie buforowania opisywanym dokładniej w kolejnym rozdziale.

Zaczniemy od najprostszego przypadku, czyli pobierania na podstawie jego identyfikatora bazodanowego (nawigacji po grafie obiektów nie trzeba wyjaśniać z racji jej podobieństwa do typowego programowania aplikacji). Choć we wcześniejszej części rozdziału pojawił się przykład użycia pobierania danych na podstawie identyfikatora, warto poznać dodatkowe informacje na jego temat.

4.4.1. Pobieranie obiektów na podstawie identyfikatora

Poniższy kod Hibernate pobiera z bazy danych obiekt User o konkretnym identyfikatorze:

```
User user = (User) session.get(User.class, userID);
```

Metoda `get()` jest szczególna, gdyż identyfikator jednoznacznie określa co najwyżej jeden egzemplarz klasy. Niejednokrotnie aplikacje stosują identyfikatory jako wygodne wskaźniki na obiekty trwałe. Pobieranie na podstawie identyfikatora wykorzystuje buforowanie — jeśli obiekt został wcześniej pobrany, Hibernate nie komunikuje się z bazą danych.

Hibernate udostępnia również metodę `load()`.

```
User user = (User) session.get(User.class, userID);
```

Metoda `load()` jest starsza. Metoda `get()` pojawiła się w Hibernate na wyraźne życzenie użytkowników. Różnice między nimi są niewielkie.

- ◆ Jeśli `load()` nie odnajdzie obiektu w buforze lub bazie danych, zgłasza wyjątek. Nigdy nie zwraca wartości `null`. Metoda `get()` zwraca `null`, gdy nie odnajdzie obiektu o wskazanym identyfikatorze.
- ◆ Metoda `load()` może zwrócić pośrednika zamiast rzeczywistego obiektu trwałego. Pośrednik stanowi specjalne rozwiązanie rozpoczęjące pobieranie rzeczywistego obiektu, gdy korzysta się z niego po raz pierwszy. Pośredniki omówimy w dalszej części rozdziału. Metoda `get()` nigdy nie zwraca pośrednika.

Wybór między `get()` i `load()` najczęściej nie stanowi problemu. Jeśli wiemy, że określony obiekt trwał istnieje, a jego brak można by uznać za sytuację niepoprawną, warto użyć metody `load()`. Gdy nie mamy pewności, czy obiekt o podanym

identyfikatorze istnieje, lepiej skorzystać z metody `get()` i sprawdzić, czy jej wynik wynosi `null`. Aplikacja może uzyskać poprawną referencję (pośrednika) na obiekt trwały bez odpytywania bazy danych w celu pobrania jego aktualnego stanu. Z tego powodu metoda `load()` nie musi zgłosić wyjątku od razu, gdy nie odnajdzie obiektu w bazie danych lub pamięci podręcznej obiektów — może to zrobić, gdy aplikacja zechce skorzystać z pośrednika.

Uzyskiwanie obiektu na podstawie identyfikatora nie jest tak elastyczne jak dowolne zapytanie.

4.4.2. Wprowadzenie do HQL

HQL to obiektowa odmiana dobrze znanego, relacyjnego języka zapytań SQL. HQL wykazuje duże podobieństwa do ODMG OQL i EJB-QL, ale w odróżniu od OQL daje się łatwo zastosować z relacyjnymi bazami danych. Jest też bardziej elegancki i rozbudowany niż EJB-QL (choć nowsze wersje EJB-QL przypominają HQL). Nauka HQL nie powinna sprawić problemu nikomu, kto zna język SQL.

HQL nie jest językiem modyfikacji danych jak SQL. Służy jedynie pobieraniu danych, a nie ich aktualizacji, wstawianiu i usuwaniu. Synchronizacja stanu obiektu nie należy do zadań programisty, ale zarządcy trwałości.

W większości sytuacji pobiera się jedynie obiekty konkretnej klasy zawierające we właściwościach określone wartości. Poniższe zapytanie pobiera użytkowników o wskazanym imieniu:

```
Query q = session.createQuery("from User u where u.firstname = :fname");
q.setString("fname", "Maks");
List result = q.list();
```

Po przygotowaniu zapytania `q` kod dowiązuje wartość do nazwanego parametru `fname`. Wynik zapytania trafia do listy obiektów `User`.

HQL jest rozbudowany. Choć przez większość czasu nie używa się jego wszystkich funkcji, czasem okazują się one niezbędne. Oto niektóre z jego możliwości:

- ◆ Wprowadzanie ograniczeń do właściwości obiektów powiązanych przez referencje lub z użyciem kolekcji (nawigacja po grafie obiektów za pomocą języka zapytań).
- ◆ Pobranie jedynie właściwości jednej lub wielu encji bez narzutu związanego z wczytaniem całej encji do zasięgu transakcyjnego. Czasem rozwiązanie to nazywa się **zapytaniem raportującym** lub **projekcją**.
- ◆ Sortowanie kolejności zwracanych wyników.
- ◆ Dzielenie wyników na strony.
- ◆ Agregacja z użyciem `group by` i `having` oraz stosowanie funkcji agregujących `sum`, `max`, `min` itp.
- ◆ Wywoływanie funkcji użytkownika języka SQL.
- ◆ Podzapytania (zapytanie zagnieżdżone).

Szczegółowo omówimy wszystkie funkcje w rozdziale 7. wraz z opisem opcjonalnego mechanizmu wykonywania standardowych zapytań SQL.

4.4.3. Zapytania przez określenie kryteriów

Interfejs zapytań przez określenie kryteriów (QBC — *Query By Criteria*) buduje zapytanie na podstawie modyfikowanych w trakcie pracy programu obiektów kryteriów. Podejście to zapewnia dynamiczne określanie ograniczeń bez potrzeby edycji tekstów, a jednocześnie nie traci się elastyczności i siły wyrazu języka HQL. Z drugiej strony zapytania tworzone dzięki kryteriom są często bardziej trudne do odczytania niż w sytuacji, w której zapisano je w HQL.

Pobranie użytkownika na podstawie jego imienia przy użyciu obiektu Criteria nie jest trudne.

```
Criteria criteria = session.createCriteria(User.class);
criteria.add(Expression.like("firstname", "Maks"));
List result = criteria.list();
```

Obiekt Criteria wyboru to tak naprawdę drzewo obiektów Criterion. Klasa Expression zawiera statyczne metody fabryczne zwracające egzemplarze klasy Criterion. Po utworzeniu drzewa zostaje ono przekształcone i przekazane do bazy danych jako zapytanie.

Wielu programistów preferuje QBC, tłumacząc to jego obiektowym interfejsem. Podoba im się, że składnię zapytania można przetworzyć i sprawdzić na etapie komplikacji (zapytania HQL są sprawdzane dopiero w trakcie działania aplikacji).

Ciekawym elementem całego interfejsu kryteriów jest szkielet Criterion. Dopuszcza on rozszerzanie go przez użytkownika, co jest trudne do przeprowadzenia w języku takim jak HQL.

4.4.4. Zapytanie przez przykład

Jako część QBC Hibernate obsługuje zapytania przez przykład (QBE — *Query By Example*). W tym podejściu aplikacja przekazuje egzemplarz klasy, która ma posłużyć do ograniczenia wyników. Egzemplarz ma ustalone tylko niektóre właściwości (na wartości niedomyślne). Zapytanie zwraca wszystkie obiekty o takich samych właściwościach jak wskazane. Nie jest to zbyt elastyczne podejście, ale sprawdza się w niektórych rodzajach aplikacji. Poniższy kod obrazuje użycie QBE:

```
User exampleUser = new User();
exampleUser.setFirstname("Maks");
Criteria criteria = session.createCriteria(User.class);
criteria.add(Example.create(exampleUser));
List result = criteria.list();
```

Typowym zastosowaniem QBE jest ekran wyszukiwania, dzięki któremu użytkownik określa zakres wartości właściwości obiektów poszukiwanych w całym zbiorze wyników. Tego rodzaju działanie trudno w łatwy sposób opisać językiem zapytań — do dynamicznego określania warunków trzeba by zaprzecić modyfikację tekstu.

Zarówno QBC, jak i zapytania przez przykład omawiamy dokładniej w rozdziale 7.

Przedstawiliśmy podstawowe sposoby określania w Hibernate danych do pobrania. Skupimy się teraz na strategiach sprowadzania grafów obiektów. Strategie te definiują, jaka część grafu obiektów (jak duży podgraf) pobierze zapytanie lub operacja odczytu.

4.4.5. Strategie sprowadzania danych

W tradycyjnym dostępnie relacyjnym wszystkie dane potrzebne do obliczeń pobiera się w jednym zapytaniu SQL, stosując złączenia wewnętrzne i zewnętrzne. Niektóre prostsze systemy ORM pobierają dane fragmentami, wykonując wiele zapytań w odpowiedzi na poruszanie się aplikacji po grafie obiektów trwałych. To podejście nie wykorzystuje możliwości drzewiastych złączeniach relacyjnych baz danych i tym samym słabo się skaluje. Jednym z trudniejszych zagadnień ORM — jeśli nie najtrudniejszym — jest zapewnienie wydajnego dostępu do relacyjnych danych przy założeniu, że aplikacja zamierza często korzystać z poruszania się po grafie obiektów.

W aplikacjach, z którymi najczęściej mieliśmy do czynienia (wielu użytkowników, duże rozproszenie, aplikacje internetowe), pobieranie obiektów w wielu fazach dostępu do bazy danych nie jest do zaakceptowania. Dobrze byłoby, aby narzędzia ORM kładły znacznie większy nacisk na literę **R** w skrócie niż obecnie.

Problem efektywnego sprowadzenia do pamięci grafu obiektów (przy minimalnej liczbie dostępów do bazy danych) często rozwiązuje się przez wprowadzenie odpowiednich wskazówek w odwzorowaniach asocjacji stanowiących metadane aplikacji. Problem w tym, że każdy fragment kodu stosujący poszczególne asocjacje wymaga *innej głębi grafu*. To jeszcze nie wszystko. Wydaje się, że tak naprawdę potrzebnych jest wiele szczegółowych strategii sprawdzania danych asocjacji działających określanych na **etapie działania aplikacji**. Hibernate obsługuje oba rozwiązania: określanie domyślnej strategii w plikach odwzorowań i modyfikację jej przez kod działającej aplikacji.

Hibernate zawiera cztery strategie sprowadzania danych do pamięci dla każdej asocjacji. Określa się je w pliku lub w trakcie działania programu:

- ◆ **Sprowadzanie natychmiastowe** — Hibernate natychmiast pobiera powiązany obiekt, używając sekwencyjnych odczytów z bazy danych (lub wyszukania obiektu w buforze).
- ◆ **Sprowadzanie leniwe** — Hibernate pobiera powiązany obiekt lub kolekcję dopiero wtedy, gdy są potrzebne po raz pierwszy. Wynikiem jest nowe zapytanie kierowane do bazy danych (jeśli obiekt nie istnieje w buforze).
- ◆ **Sprowadzanie wyprzedzające** — powiązany obiekt lub kolekcja zostaje sprowadzona z bazy danych razem z obiektem głównym przez zastosowanie złączenia. W przeszłości nie ma potrzeby ponownego wywoływania zapytań do bazy danych.

- ◆ **Sprowadzanie wsadowe** — to podejście pozwala poprawić wydajność sprowadzania leniwego przez pobieranie grupy obiektów lub kolekcji w momencie dostępu do leniwej asocjacji (wsadowe wykonywanie sprowadzania poprawia również wydajność sprowadzania natychmiastowego).

Przyjrzymy się dokładniej każdej ze strategii.

Sprowadzanie natychmiastowe

Sprowadzanie natychmiastowe występuje wtedy, gdy aplikacja pobiera encję z bazy danych, a następnie chwilę później pobiera inną powiązaną encję w kolejnym zapytaniu kierowanym do bazy danych lub bufora. Ta strategia nie zapewnia wysokiej wydajności, chyba że ma się pewność, iż prawie wszystkie obiekty będą pobierane z bufora.

Sprowadzanie leniwe

Gdy klient pobiera encję i związany z nią graf powiązanych obiektów, najczęściej nie trzeba pobierać grafu każdego (niebezpośrednio) powiązanego obiektu. Nie trzeba za jednym zamachem pobierać całej bazy danych. Przykładowo, załadowanie obiektu Category nie powinno powodować załadowania wszystkich obiektów Item znajdujących się w kategorii.

Sprowadzanie leniwe pozwala zdecydować, ile obiektów należy wczytać w pierwszym zapytaniu do bazy danych. Pozostałe asocjacje mogą zostać sprowadzone dopiero wtedy, gdy będą potrzebne. Ten typ sprowadzania to podstawowy element pojęcia trwałości obiektów i pierwszy krok do osiągnięcia akceptowalnej wydajności.

Zalecamy, by na początku wszystkie asocjacje skonfigurować ze sprowadzaniem leniwym (lub wsadowym sprowadzaniem leniwym) w plikach odwzorowań. Strategię tę zawsze można zmienić w razie potrzeb programowo na sprowadzanie wyprzedzające w trakcie działania aplikacji.

Sprowadzanie wyprzedzające (stosujące złączenia)

Sprowadzanie leniwe pomaga zredukować obciążenie bazy danych i często stanoi dobrą strategię domyślną. Z punktu widzenia optymalizacji dostępów przypomina raczej szukanie z zamkniętymi oczami.

Sprowadzanie wyprzedzające jawnie określa, że powiązane obiekty należy załadować wraz z obiektem głównym. Hibernate potrafi zwrócić powiązane obiekty w jednym zapytaniu bazodanowym, stosując złączenie zewnętrzne. Optymalizacja wydajności często wymaga ostrożnego stosowania tej strategii dla wybranych transakcji. Choć można ustawić tę strategię sprowadzania w pliku odwzorowania, w rzeczywistości włącza się ją jedynie okazjonalnie w trakcie działania aplikacji dla konkretnych zapytań HQL.

Sprowadzanie wsadowe

Sprowadzanie wsadowe nie jest tak naprawdę strategią pobierania danych, ale raczej wskazówką mającą na celu zwiększyć wydajność sprowadzania leniwego (i natychmiastowego). W typowym wczytywaniu obiektu lub kolekcji klauzula WHERE zapytania SQL określa identyfikator obiektu lub obiekt zawierający kolekcję. Po włączeniu sprowadzania wsadowego Hibernate poszukuje innych obiektów pośredniczących lub niezainicjalizowanych kolekcji związanych z aktualną sesją i próbuje je wczytać, podając w klauzuli WHERE wiele identyfikatorów.

Nie jesteśmy zwolennikami tego podejścia; sprowadzanie wyprzedzające niemal zawsze będzie szybsze. Sprowadzanie wsadowe to dobre rozwiązanie dla niedoświadczonych użytkowników, którzy chcą osiągnąć dobrą wydajność bez zbyt długiej analizy wykonywanego kodu SQL. Ten typ sprowadzania stosuje wiele mechanizmów EJB2, więc powinieneś być znany wielu osobom.

Zadeklarujmy strategie sprowadzania dla niektórych asocjacji w plikach odwzorowań.

4.4.6. Wybór strategii sprowadzania w odwzorowaniach

Hibernate dopuszcza określanie domyślnych strategii sprowadzania danych asocjacji jako atrybutów metadanych odwzorowań. Nic nie stoi na przeszkodzie, by w trakcie działania programu zmienić domyślną strategię, używając metod zapytań (patrz rozdział 7.). Dodatkowa uwaga: nie wszystkie przedstawiane tu opcje trzeba zrozumieć od razu; zalecamy ogólne przyjrzenie się im teraz i powrócenie do rozdziału, gdy zajdzie potrzeba optymalizacji domyślnych strategii sprowadzania danych w tworzonej aplikacji.

Pewne niedoskonałości formatu zapisu metadanych Hibernate powodują, że odwzorowania kolekcji działają nieco inaczej niż asocjacje jednoelementowe. Osobno zajmiemy się każdym z przypadków. Rozważmy oba końce asocjacji między klasami Bid i Item.

Asocjacja jednoelementowa

Dla asocjacji <many-to-one> lub <one-to-one> sprowadzanie leniwe możliwe jest tylko wtedy, gdy odwzorowanie powiązanej klasy dopuszcza stosowanie pośredników. W klasie Item dopuściliśmy stosowanie pośredników, stosując atrybut lazy="true":

```
<class name="item" lazy="true">
```

Przypomnijmy asocjację od Bid do Item:

```
<many-to-one name="item" class="Item">
```

Gdy z bazy danych pobierzemy obiekt Bid, właściwość asocjacyjna może zawierać egzemplarz podklasy Item wygenerowanej przez Hibernate i kierować wszystkie wywołania metod do innego obiektu Item leniwie pobranego z bazy danych (jest to nieco skrócona definicja pośrednika Hibernate).

Hibernate stosuje dwa różne obiekty, więc nawet asocjacje polimorficzne mogą być pośredniczone. Pobrany obiekt pośrednika w przedstawionym przykładzie zapewne będzie odwzorowana podklasą klasy Item (jeśli istnieją podklasy klasy Item). Jako typ pośrednika można wybrać dowolny interfejs implementowany przez klasę Item. Określamy go atrybutem proxy zamiast atrybutu lazy="true":

```
<class name="Item" proxy="ItemInterface">
```

Zaraz po zadeklarowaniu atrybutu proxy lub lazy w odwzorowaniu klasy Item dowolna asocjacja jednoelementowa jest pośredniczona i sprowadzana leniwie, chyba że zmieni tę domyślną strategię atrybut outer-join.

Atrybut outer-join przyjmuje jedną z trzech wartości:

- ◆ outer-join="auto" — wartość domyślna. Gdy nie poda się atrybutu, Hibernate pobiera powiązany obiekt leniwie przy włączonym pośredniku klasy powiązanej lub używa sprowadzanie wyprzedzającego dla wyłączonego pośrednika (rozwiązań domyślnych).
- ◆ outer-join="true" — Hibernate zawsze pobiera asocjację wyprzedzającą, używając złączenia zewnętrznego (nawet dla włączonego pośrednika). W ten sposób można wybrać różne strategie pobierania asocjacji dla tej samej klasy z pośrednikiem.
- ◆ outer-join="false" — Hibernate nigdy nie pobiera asocjacji z wyprzedzeniem, nawet jeśli pośrednik jest wyłączony. To podejście przydaje się, gdy istnieje duża szansa na istnienie powiązanego obiektu w buforze drugiego poziomu (patrz rozdział 5.). Gdy obiekt nie jest dostępny w buforze, Hibernate pobiera go natychmiast dodatkowym zapytaniem SQL.

Jeśli chcemy włączyć sprawdzanie wyprzedzające dla asocjacji przy włączonym pośredniku, należy użyć poniższego rozwiązania:

```
<many-to-one name="item" class="Item" outer-join="true">
```

W asocjacji one-to-one (omawianej dokładnie w rozdziale 6.) sprowadzanie leniwe jest możliwe koncepcyjnie tylko wtedy, gdy powiązany obiekt zawsze istnieje. Za wskazanie takiej sytuacji odpowiada atrybut constrained="true". Jeśli przedmiot może mieć tylko jedną ofertę, odwzorowanie obiektu Bid może mieć postać:

```
<many-to-one name="item" class="Item" constrained="true">
```

Atrybut constrained warto traktować podobnie do atrybutu not-null odwzorowania <many-to-one>. Informuje to Hibernate, że powiązany obiekt jest wymagany i tym samym jego wartość nie może wynieść null.

By włączyć sprowadzanie wsadowe, w odwzorowaniu klasy Item podajemy atrybut batch-size.

```
<class name="item" lazy="true" batch-size="9">
```

Atrybut ogranicza liczbę elementów pobieranych w jednej operacji wsadowej. Warto wskazać niewielką wartość.

Te same atrybuty (outer-join, batch-size i lazy) pojawią się przy rozpatrywaniu kolekcji, ale zmieni się ich interpretacja.

Kolekcje

W przypadku kolekcji strategie sprowadzania dotyczą nie tylko asocjacji encji, ale również wartości asocjacji (na przykład kolekcję tekstów można pobrać, stosując złączenie zewnętrzne).

Kolekcje, podobnie jak klasy, stosują własne pośredniki nazywane niejednokrotnie **otoczkami kolekcji**. Otoczki te występują zawsze, nawet jeśli wyłączy się sprowadzanie leniwe (Hibernate potrzebuje otoczki, by wykrywać zmiany w kolekcji).

Odwzorowania kolekcji mogą deklarować atrybut lazy, atrybut outer-join, nie używać żadnego z nich lub podawać oba (choć podanie obu nie zapewnia poprawności znaczeniowej). Sensowe opcje są następujące:

- ◆ **Brak obu atrybutów** — jest równoważne outer-join="false" lazy="false". Hibernate pobiera kolekcję z bufora drugiego poziomu lub przez bezpośrednie wywołanie dodatkowego zapytania SQL. Opcja jest domyślna i najbardziej użyteczna, gdy kolekcja ma włączony bufor drugiego poziomu.
- ◆ **outer-join="true"** — Hibernate pobiera asocjację z wyprzedzeniem, stosując złączenie zewnętrzne. W momencie pisania tego tekstu nie było możliwe zadeklarowanie wielu kolekcji tej samej klasy trwałej z atrybutem outer-join="true", bo Hibernate pobiera tylko jedną kolekcję w zapytaniu SQL.
- ◆ **lazy="true"** — Hibernate pobiera kolekcję leniwie, czyli dopiero wtedy, gdy jest naprawdę potrzebna.

Nie zalecamy sprowadzania wyprzedzającego dla kolekcji, więc kolekcję ofert przedmiotu oznaczmy argumentem lazy="true". Warto ją stosować w niemal każdej kolekcji (w zasadzie powinna być opcją domyślną). Zalecamy podawanie jej we wszystkich odwzorowaniach kolekcji.

```
<set name="bids" lazy="true">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
```

Można włączyć sprowadzanie wsadowe dla kolekcji. W tym przypadku rozmiar nie dotyczy liczby ofert w kolekcji, ale liczby pobieranych kolekcji.

```
<set name="bids" lazy="true" batch-size="9">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
```

Odwzorowane informuje, by Hibernate ładował do 9 kolekcji ofert w jednym zadaniu wsadowym w zależności do liczby niezainicjalizowanych kolekcji ofert istniejących w aktualnej sesji. Innymi słowy, jeśli istnieje pięć trwałych egzemplarzy klasy Item w sesji i wszystkie mają niezainicjalizowane kolekcje bids, Hiber-

nate automatycznie załadowuje wszystkie pięć kolekcji w jednym zapytaniu SQL. Jeśli obiektów Item jest 11, pobierze tylko 9 kolekcji. Sprowadzanie wsadowe potrafi znaczco zredukować liczbę zapytań wymaganych do pobrania hierarchii obiektów (na przykład w momencie wczytywania drzewa obiektów Category).

Zajmijmy się przypadkiem szczególnym: asocjacjami wiele-do-wielu omawianymi dokładniej w rozdziale 6. Najczęściej do ich przeprowadzenia używa się tabeli łączającej (nazywanej również tabelą asocjacyjną) przechowującej jedynie klucze dwóch powiązanych tabel w celu uzyskania krotności wiele po obu stronach asocjacji. Tę dodatkową tabelę trzeba uwzględnić przy sprowadzaniu wyprzedzającym. Przyjrzyjmy się prostej asocjacji wiele-do-wielu, która łączy klasy Item i Category:

```
<set name="items" outer-join="true" table="CATEGORY_ITEM">
    <key column="CATEGORY_ID"/>
    <many-to-many column="ITEM_ID" class="Item"/>
</set>
```

W tym przypadku strategia sprowadzania wyprzedzającego odnosi się jedynie do tabeli łączającej CATEGORY_ITEM. Wczytując obiekt Category z tą strategią, Hibernate automatycznie pobierze wszystkie wpisy łączące w CATEGORY_ITEM w jednym zapytaniu, ale nie pobierze obiektów Item!

Hibernate dopuszcza wczytanie wyprzedzające encji zawartych w asocjacji wiele-do-wielu w tym samym zapytaniu. Element <many-to-many> pozwala dostosować to zachowanie do własnych potrzeb.

```
<set name="items" outer-join="true" table="CATEGORY_ITEM">
    <key column="CATEGORY_ID"/>
    <many-to-many column="ITEM_ID" outer-join="true" class="Item"/>
</set>
```

W ten sposób Hibernate pobierze wszystkie obiekty Item dla kategorii w jednym zapytaniu w tym samym czasie, w którym pobierany będzie główny obiekt kategorii. Pamiętaj o zaleceniach dotyczących leniwego wczytywania jako domyślnej strategii sprowadzania danych i o ograniczeniu związanym z pobieraniem w sposób wyprzedzający tylko jednej kolekcji.

Określenie głębi sprowadzania danych

Zajmijmy się globalnym ustawieniem strategii sprowadzania danych: maksymalną głębią. Ustawienie to steruje maksymalną liczbą złączanych zewnętrznie tabel w jednym zapytaniu SQL. Rozważmy łańcuch powiązań: od klasy Category przez Item po Bid. Pierwsza asocjacja jest typu wiele-do-wielu, natomiast druga typu jeden-do-wielu. Obie odpowiadają więc kolekcjom. Jeśli zadeklarujemy obie asocjacje z argumentem outer-join="true" i wczytamy pojedynczy obiekt Category, ile zapytań wykona Hibernate? Czy tylko obiekty Item zostaną wczytane z wyprzedzeniem czy może stanie się tak również z obiektami Bid poszczególnych obiektów Item?

Zapewne spodziewasz się pojedynczego zapytania tworzącego złączenia zewnętrzne tabel CATEGORY, CATEGORY_ITEM, ITEM i BID? Domyślnie nic takiego się nie stanie.

Zachowaniem Hiberante w kwestii złączeń zewnętrznych steruje opcja konfiguracyjna hibernate.max_fetch_depth. Ustawienie jej na 1 (wartość domyślna) spowoduje wykonanie jedynie złączenia między tabelami CATEGORY i CATEGORY_ITEM. Dla wartości 2 Hibernate dodatkowo łączy tabelę przedmiotów akcji. Wartość 3 zapewni złączenie wszystkich czterech tabel w jednym zapytaniu SQL i załadowanie również wszystkich obiektów Bid.

Zalecane wartości głębi złączeń zależą od wydajności złączeń i rozmiaru tabel bazy danych. Aplikację warto testować z niskimi wartościami (poniżej 4), a następnie zwiększać je lub zmniejszać przy optymalizacjach. Hibernate globalne ustawienie złączeń stosuje również dla asocjacji z pojedynczymi elementami (<many-to-one> i <one-to-one>) wykorzystującymi strategie wczytywania wyprzedzającego.

Warto zdawać sobie sprawę z faktu, iż strategie wyprzedzające deklarowane w metadanych odwzorowań mają znaczenie tylko dla pobierania danych na podstawie identyfikatora, z zastosowaniem kryteriów lub ręcznej nawigacji po grafie obiektów. Dowolne zapytanie HQL może określić własną strategię wydobywania danych, ignorując domyślne ustawienia. Kryteria również dopuszczają tymczasowe przysłonięcie strategii (ale nie ich zignorowanie). Tę istotną różnicę dokładniej opiszymy w podrozdziale 7.3.2.

Czasem warto ręcznie zainicjalizować pośrednika lub otoczkę kolekcji. Do tego celu służy proste wywołanie interfejsu programistycznego.

Iinicjalizacja leniwych asocjacji

Pośrednik lub otoczka kolekcji powstaje automatycznie w momencie wywołania jednej z jej metod (wyjątkiem jest metoda pobierająca identyfikator; pośrednik może go zwrócić bez odwoływanego się do rzeczywistego obiektu trwałego). Inicjalizacja może odbyć się tylko wtedy, gdy pośrednik lub otoczka jest aktualnie powiązana z otwartą sesją. Zamknięcie sesji i próba dostępu do niezainicjalizowanego pośrednika spowoduje zgłoszenie błędu.

Z tego powodu czasem warto ręcznie zainicjalizować obiekt przed zamknięciem sesji. Podejście to nie jest tak elastyczne jak pobranie pełnego, wymaganego podgrafu obiektów zapytaniem HQL z zastosowaniem dowolnej strategii wczytywania.

Do inicjalizacji ręcznej służy metoda statyczna `Hibernate.initialize()`.

```
Session session = sessions.openSession();
Transaction tx1 = session.beginTransaction();
```

```
Category cat = (Category) session.get(Category.class, id);
Hibernate.initialize(cat.getItems());
```

```
tx.commit();
session.close();
```

```
Iterator iter = cat.getItems().iterator();
```

Metoda `Hibernate.initialize()` może otrzymać otoczkę kolekcji (jak w przykładzie) lub obiekt pośrednika. W rzadkich przypadkach zachodzi potrzeba sprawdzenia aktualnego stanu właściwości. O stanie informuje metoda `Hibernate.isInitialized()`. Pamiętaj, że `initialize()` nie zostaje wywołane kaskadowo dla powiązanych obiektów.

Alternatywne podejście polega utrzymaniu otwartej sesji tak dugo, by aplikacja zakończyła cały proces przechodzenia po grafie obiektów, a Hibernate automatycznie inicjalizował wszystkie leniwe referencje. Niestety, powoduje to problemy w projekcie aplikacji i demarkacji transakcji. Dokładniej zajmiemy się tym tematem w podrozdziale 8.1. Najlepiej pobrać cały wymagany graf obiektów za jednym zamachem, używając HQL lub kryteriów, a we wszystkich pozostałych przypadkach stosować odpowiednio zoptymalizowane domyślne strategie wczytywania danych.

4.4.7. Optymalizacja pobierania obiektów

Przyjrzyjmy się krokom wykonywanym w celu optymalizacji operacji pobierania danych przez aplikację.

1. Włącz dziennik zapytań SQL dla Hibernate w sposób opisany w rozdziale 2. Warto znać i umieć ocenić same zapytania SQL oraz ich charakterystykę wydajności w kontekście stosowanej bazy danych. Czy złączenie będzie szybsze od dwóch zapytań? Czy wszystkie indeksy są używane poprawnie? Jaki jest poziom trafień bufora zapytań bazy danych? Administrator bazy danych powinien pomóc w ocenie wydajności. Tylko on będzie miał wystarczającą wiedzę na temat bazy danych, by ocenić, który plan wykonania zapytania działa najlepiej.
2. Przejdź przez przypadki użycia aplikacji i zanotuj, ile poleceń SQL (i jakich) wykonuje Hibernate. Przypadek użycia może być pojedynczym ekranem aplikacji internetowej lub serią powiązanych ze sobą ekranów. Krok ten analizuje również, jakie metody trzeba wywołać w każdym przypadku użycia: przejęcie przez graf, pobranie na podstawie identyfikatora, HQL i kryteria. Celem jest takie dobranie domyślnych lub modyfikowanych strategii sprowadzania danych, by zminimalizować liczbę oraz złożoność zapytań dla poszczególnych akcji.
3. Natknąć się można na dwa typowe przypadki:
 - ◆ Jeśli polecenia SQL stosują złączenia o zbyt dużej złożoności lub zbyt powolne, ustaw `outer-join` na wartość `false` dla asocjacji `<many-to-one>` (domyślnie ustawienie to jest włączone). Dodatkowo warto dostroić globalną opcję konfiguracyjną `hibernate.max_fetch_depth`. Pamiętaj, że najlepiej utrzymywać jej wartość w przedziale od 1 do 4.
 - ◆ Jeśli Hibernate wykonuje zbyt dużo poleceń SQL, użyj argumentu `lazy="true"` dla wszystkich odwzorowań kolekcji. Domyślnie Hibernate wykonuje sprowadzenie natychmiastowe dla elementów kolekcji (jeśli

są one encjami, może to powodować dalsze przejścia przez graf). W uzasadnionych przypadkach można włączyć outer-join="true" i wyłączyć wczytywanie leniwe dla wybranych kolekcji. Pamiętaj, że w ten sposób można pobrać tylko jedną kolekcję. Sprowadzanie wsadowe z wartościami w przedziale od 3 do 10 potrafi zoptymalizować pobieranie, jeśli jednostka zadaniowa stosuje kilka „takich samych” kolekcji lub pobiera się drzewo rodziców i potomków.

4. Po zmianie strategii sprowadzania danych ponownie wykonaj przypadek użycia i sprawdź generowany kod SQL. Zapamiętaj wykonane polecenia i zmiany względem poprzedniej sytuacji.
5. Po wykonaniu wszystkich optymalizacji sprawdź wszystko raz jeszcze, by wykryć ewentualne efekty uboczne jednych aktualizacji na inne. Zyskując doświadczenie, pewnych błędów będzie udawało się uniknąć za pierwszym razem.

Przedstawione techniki optymalizacji dają się zastosować nie tylko dla strategii sprowadzania danych. Nic nie stoi na przeszkodzie, by użyć ich do optymalizacji zapytań HQL lub kryteriów, które mogą przesyłać domyślne strategie dla konkretnych przypadków użycia. W rozdziale 7. opisujemy określanie strategii w trakcie działania aplikacji.

W niniejszym podrozdziale zaczęliśmy myśleć o wydajności, w szczególności tej związanej z pobieraniem asocjacji. Oczywiście najszybszym sposobem pobrania obiektu z grafu jest uzyskanie go z bufora pamięciowego. Tym tematem zajmiemy się w następnym rozdziale.

4.5. Podsumowanie

Dynamiczne aspekty niedopasowania obiektowo-relacyjnego są równie ważne jak lepiej poznane i opisane aspekty strukturalne niedopasowania. W rozdziale skupiliśmy się przede wszystkim na cyklu życia obiektów w związku z mechanizmem trwałości. Przedstawiliśmy trzy stany obiektu definiowane przez Hibernate: trwały, ulotny i odlączony. Przejścia między stanami zachodzą po wywołaniu metod obiektu Session lub utworzeniu i usunięciu referencji z grafu obiektów trwałych. Tym ostatnim zagadnieniem zarządzają konfigurowalne style kaskadowe stanowiące model trwałości przechodniej zaimplementowany w Hibernate. Model określa sposoby kaskadowego wykonywania operacji (na przykład zapisu lub usunięcia) dla poszczególnych asocjacji, co jest bardziej elastycznym podejściem niż tradycyjny model **trwałości przez osiągalność**. Naszym celem jest odnalezienie najlepszego stylu kaskadowego dla każdej asocjacji, by w ten sposób zminimalizować liczbę wywołań zarządcy trwałości i zapytań do bazy danych.

Pobieranie obiektów z bazy danych jest również istotne. Można przechodzić przez graf obiektów dziedzinowych, odczytując właściwości i pozwalając Hibernate zająć się całą resztą. Można także wczytywać obiekty na podstawie ich identyfikatora, pisać dowolne zapytania w języku HQL, generować kryteria wyboru

pobieranych danych w sposób obiektowy lub przekazując przykładowy obiekt. Gdy to nie wystarcza, pozostaje jeszcze tworzenie zwykłych zapytań SQL.

Większość metod pobierania obiektów wykorzystuje domyślne strategie sprowadzania danych określone w metadanych odwzorowań (HQL je ignoruje; zapytania kryteriowe mogą je modyfikować). Odpowiednia strategia sprowadzania minimalizuje liczbę zapytań SQL wykonywanych przez obiekty ze strategią leniwą, wyprzedzającą lub wsadową. Optymalizacja aplikacji z Hibernate polega na analizie wynikowych poleceń SQL i odpowiedniej modyfikacji domyślnych strategii.

W kolejnym rozdziale dokładniej przyjrzymy się tematowi **transakcji i buforowania**.

Transakcje, współbieżność i buforowanie

5

W rozdziale:

- ◆ Transakcje i blokady bazodanowe
- ◆ Długo działające transakcje aplikacyjne
- ◆ Bufory Hibernate pierwszego i drugiego poziomu
- ◆ System buforujący w praktyce na przykładzie CaveatEmptor

Po zapoznaniu się z podstawami odwzorowania obiektowo-relacyjnego w Hibernate przyjrzyjmy się jednej z najważniejszych kwestii w projektowaniu aplikacji bazodanowych — **zarządzaniu transakcjami**. W rozdziale zajmiemy się sposobami zarządzania transakcjami przez Hibernate, sposobami obsługi współbieżności i zadaniom buforowania w związku z dwoma poprzednimi zagadnieniami. Prześledźmy działanie przykładowej aplikacji.

Pewne działania aplikacji wymagają wykonywania wielu operacji w tym samym momencie. Przykładowo, zakończenie aukcji w aplikacji CaveatEmptor wymaga wykonania czterech różnych działań:

1. Oznaczenia oferty wygrywającej.
2. Obciążenia sprzedającego kosztami aukcji.
3. Obciążenia wygrywającego kupującego kosztami przedmiotu.
4. Poinformowanie sprzedającego i kupującego o sposobie finalizacji aukcji.

Co się stanie, jeśli nie możemy obciążyć sprzedającego kosztami aukcji z powodu awarii zewnętrznego systemu aukcyjnego? Wymagania biznesowe mogą wskazywać, że albo zostaną poprawnie wykonane wszystkie wymienione działania, albo żadne z nich. Jeśli tak, przedstawiony zbiór zadań nazwiemy **transakcją** lub **jednostką zadaniową**. Gdy nie uda się wykonać choćby jednego z kroków, cała jednostka jest błędna. Transakcje traktuje się jako działania **atomowe** — kilka operacji łączy się razem w jedno niepodzielne zadanie.

Transakcje dopuszczają jednoczesną pracę wielu osób na tych samych danych bez narażania na szwank integralności i poprawności danych; jedna transakcja nie powinna być widoczna ani mieć wpływu na inne wykonywane w tym samym czasie. Do implementacji tego zadania stosuje się kilka różnych strategii nazywanych poziomami **izolacji**.Więcej informacji na ten temat znajduje się w dalszej części rozdziału.

Transakcje wykazują również dwie inne cechy: **spójność** i **trwałość**. Spójność oznacza, że dowolna transakcja korzysta ze spójnego zbioru danych i po swym zakończeniu również pozostawia spójny zbiór danych. Trwałość gwarantuje, że po zakończeniu transakcji wszystkie dokonane w niej zmiany są trwale zapisane, nawet jeśli chwilę później dojdzie na przykład do awarii zasilania. Wszystkie cztery wymienione aspekty transakcji nazywa się w skrócie ACID (*atomicity, consistency, isolation i durability*).

Rozdział rozpoczniemy od omówienia **transakcji bazodanowych** na poziomie systemu, w których to baza danych gwarantuje zachowanie ACID. Przyjrzymy się interfejsom programistycznym JDBC i JTA oraz temu, jak Hibernate będący klientem tych interfejsów radzi sobie ze sterowaniem transakcjami bazodanowymi.

W aplikacji internetowej transakcje bazodanowe muszą trwać wyjątkowo krótko. Powinny zawierać zestaw operacji bazodanowych przemieszany z logiką biznesową. Nie powinny oczekiwać interakcji ze strony użytkownika. Podejście to stosuje inne zasady niż długo działające transakcje aplikacyjne, w których to operacje bazodanowe występują w kilku fazach przemieszanych z interakcjami użytkownika. Istnieje kilka sposobów wprowadzenia transakcji aplikacyjnych

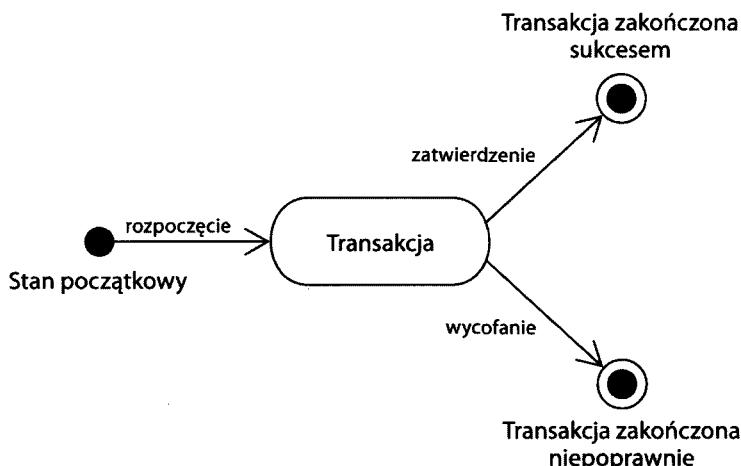
z użyciem Hibernate. Wszystkie zostaną omówione w tym rozdziale. Buforowanie jest znacznie mocniej powiązane z transakcjami, niż mogłoby się początkowo wydawać. W drugiej części rozdziału uzbrojeni w wiedzę na temat transakcji rozpoczęliśmy odsłanianie wyrafinowanej architektury buforującej. Powiemy, które dane są dobrymi kandydatami do zbuforowania i w jaki sposób obsługiwać wspólnie dostęp do bufora. Na końcu włączymy buforowanie w aplikacji CaveatEmptor.

Zacznijmy od podstaw, czyli działania transakcji na najniższym poziomie — w bazie danych.

5.1. Transakcje bazodanowe

Bazy danych implementują jednostkę zadaniową jako **transakcję bazodanową** (nazywaną również **transakcją systemową**).

Transakcje bazodanowe grupują operacje dostępu do danych. Transakcja zawsze kończy się na jeden z dwóch sposobów: jest **zatwierdzana** lub **wycofana**. Dzięki temu zawsze pozostaje operacją atomową. Całą sytuację w sposób graficzny przedstawia rysunek 5.1.



Rysunek 5.1. Stany systemu podczas transakcji

Jeśli transakcja ma zawierać kilka operacji, trzeba określić ramy jednostki zadaniowej. Musimy rozpoczęć transakcję i w pewnym momencie ją zatwierdzić. Gdy wystąpi błąd (czy to na etapie wykonywania operacji, czy też ich zatwierdzania), należy wycofać transakcję, by pozostawić dane w spójny stan. Jest to tak zwana demarkacja transakcji i w zależności od stosowanego interfejsu programistycznego wymaga większej lub mniejszej interwencji ręcznej.

Zapewne niejedna osoba zdobyła już doświadczenie w jednym z dwóch (lub nawet obu) interfejsów programistycznych transakcji dostępnych w Javie: JDBC i JTA.

5.1.1. Transakcje JDBC i JTA

W środowisku niezarządzanym do oznaczania transakcji używamy interfejsu JDBC. Początek transakcji oznacza wywołanie metody `setAutoCommit(false)` dla połączenia JDBC, a jej zakończenie wywołanie metody `commit()`. W dowolnym momencie można wywołać metodę `rollback()`, by wycofać transakcję. Proste, prawda?

Odpowiedzi na pytania **Z jakiego trybu automatycznego zatwierdzania najlepiej korzystać?**

Magiczne ustawienie trybu automatycznego zatwierdzania w połączniach JDBC powoduje niejednokrotnie ogromny zamęt. Gdy baza danych znajduje się w trybie automatycznego zatwierdzania, transakcja bazodanowa jest od razu zatwierdzana po każdym poprawnie wykonanym poleceniu SQL i jednocześnie rozpoczyna się kolejna transakcja. To rozwiązanie jest wygodne, gdy dokonuje się niewielkich zmian w bazie danych, używając pojedynczych, w pełni niezależnych poleceń.

Tryb automatycznego zatwierdzania niemal zawsze nie jest dobrym rozwiązaniem w aplikacji. Aplikacja rzadko stosuje pojedyncze lub nieplanowane polecenia SQL; raczej wykonuje wcześniej zaplanowaną sekwenację powiązanych ze sobą operacji (nigdy nie są to operacje ad hoc). Hibernate automatycznie wyłącza automatycznie zatwierdzanie po uzyskaniu połączenia (od dostawcy połączeń, którym najczęściej jest pula połączeń). Gdy samemu wskazuje się konkretne połączenie w momencie otwierania sesji, wyłączenie automatycznego zatwierdzania leży w gestii programisty aplikacji!

Pamiętaj, że niektóre systemy bazodanowe domyślnie włączają automatyczne zatwierdzanie dla nowych połączeń, a inne tego nie czynią. Warto wyłączyć automatyczne zatwierdzanie w globalnej konfiguracji systemu bazodanowego, by uniknąć potencjalnych problemów. Automatyczne zatwierdzanie należy włączać jedynie dla polecień wykonywanych ad hoc (na przykład w narzędziu do ręcznego pisania zapytań SQL).

W systemie, który przechowuje dane z wielu baz danych, konkretna jednostka zarządzająca może dotyczyć dostępu do więcej niż jednego zbioru danych. W takiej sytuacji osiągnięcie atomowości tylko za pomocą JDBC nie jest możliwe. Potrzeba zarządcy transakcji obsługującego transakcje rozproszone (zatwierdzanie dwufazowe). Komunikacja z zarządcą odbywa się przy użyciu JTA.

W środowisku zarządzanym JTA służy nie tylko do obsługi transakcji rozproszonych, ale również deklaratywnych **transakcji zarządzanych przez kontener (CMT)**. CMT pozwala uniknąć jawnego wywoływać oznaczających granice transakcji w kodzie źródłowym aplikacji. Demarkacją transakcji steruje deskryptor związany z wdrożeniem. Deskryptor opisuje sposób **propagowania** kontekstu transakcji, gdy jeden wątek przechodzi przez kilka różnych EJB.

Nie interesują nas szczegóły bezpośredniej demarkacji JDBC lub JTA. Korzystamy z tych interfejsów pośrednio.

Hibernate komunikuje się z bazą danych za pomocą obiektu `Connection` interfejsu JDBC, więc musi obsługiwać oba sposoby zarządzania transakcjami. W aplikacjach samowystarczalnych (lub internetowych) mamy dostęp jedynie do obsługi

transakcyjnej realizowanej przez JDBC. Stosując serwer aplikacji, mamy do dyspozycji JTA. Ponieważ najlepiej byłoby, gdyby kod Hibernate wyglądał tak samo niezależnie od tego, czy używa się środowiska zarządzanego czy też nie, powstała specjalna warstwa abstrakcji ukrywająca szczegóły działa wspomnianych wcześniej interfejsów. Co więcej, Hibernate można rozszerzyć, dodając na przykład adapter dla usług transakcyjnych CORBA.

Hibernate udostępnia aplikacji zarządzanie transakcji poprzez interfejs Transaction. Nie zmusza jednak do stosowania tego rozwiązania — nic nie stoi na przeszkodzie, by bezpośrednio korzystać z JTA lub JDBC. Nie polecamy tego podejścia i nie opisujemy go w książce.

5.1.2. Interfejs Transaction

Interfejs Transaction udostępnia metody do deklaracji zakresu transakcji bazodanowych. Listing 5.1 przedstawia przykładowe użycie interfejsu.

Listing 5.1. Zastosowanie interfejsu Transaction w Hibernate

```
Session session = sessions.openSession();
transaction tx = null;
try {
    tx = session.beginTransaction();

    concludeAuction();

    tx.commit();
} catch (Exception e) {
    if (tx != null) {
        try {
            tx.rollback();
        } catch (HibernateException he) {
            // zapisz w dzienniku he i zgłoś wyjątek e
        }
    }
    throw e;
} finally {
    try {
        session.close();
    } catch (HibernateException he) {
        throw he;
    }
}
```

Wywołanie session.beginTransaction() oznacza początek transakcji bazodanowej. W środowisku niezarządzanym rozpoczyna transakcję JDBC dla połączenia JDBC. W środowisku zarządzanym uruchamia nową transakcję JTA, jeśli żadna nie istnieje, lub dołącza do istniejącej transakcji JTA. Wszystkim tym zajmuje się Hibernate; programista nie martwi się o szczegóły implementacji.

Wywołanie tx.commit() synchronizuje stan obiektu Session z bazą danych. Hibernate zatwierdza rzeczywistą transakcję tylko wtedy, gdy nową transakcję

rozpoczęto wywołaniem `beginTransaction()` (w obu rodzajach środowisk). Jeśli metoda `beginTransaction()` nie rozpoczęła rzeczywistej transakcji, wywołanie `commit()` jedynie synchronizuje stan obiektu `Session` z bazą danych. Hibernate zakończenie transakcji pozostawia kodowi, który ją rozpoczął. To podejście zgadza się z zachowaniem zdefiniowanym w JTA.

Jeśli metoda `concludeAuction()` zgłosi wyjątek, musimy wymusić wycofanie transakcji wywołaniem `tx.rollback()`. Metoda albo od razu wycofuje transakcję, albo oznacza ją jako transakcję wyłącznie do wycofania (jeśli stosuje się CMT).

Odpowiedzi na pytania

Czy można szybciej wycofać transakcję tylko do odczytu? Jeśli kod transakcji jedynie odczytuje dane bez ich modyfikacji, lepiej wycofać transakcję czy ją zatwierdzić? Co jest szybsze?

Niektórzy programiści odkryli, że wycofywanie w niektórych przypadkach jest szybsze. Ta plotka bardzo szybko rozniosła się po całym świecie. Przetestowaliśmy to podejście dla kilku popularnych systemów bazodanowych i nie stwierdziliśmy żadnych różnic. Nie udało się też dotrzeć do innych przekonujących wyników świadczących o różnicach w szybkości. Nie ma powodów, dla których twórcy baz danych mieliby stosować optymalizację wybiórczo; system zawsze powinien wewnętrznie wybierać najszybszy algorytm zakończenia transakcji. Zalecamy zawsze zatwierdzać transakcje i wycofywanie używać tylko w momencie napotkania błędu.

Niezwykle ważne jest zamknięcie sesji w bloku finalizacji, by mieć pewność, że połączenie JDBC zostanie zwolnione i wróci do puli (za ten krok odpowiada aplikacja, nawet w środowisku zarządzanym).

Uwaga

Kod z listingu 5.1 to standardowe podejście do tworzenia jednostek zadaniowych dla Hibernate. Zawiera kod sprawdzający wyjątki `HibernateException`. Nietrudno zauważyc, że nawet wycofywanie transakcji i zamknięcie sesji może zgłosić wyjątek. Przedstawionego kodu nie warto używać we własnej aplikacji, bo zapewne sposób obsługi wyjątków zostanie wykonany bardziej ogólnie. Przykładowo, można użyć klasy narzędziowej konwertującej wyjątek `HibernateException` na nieweryfikowany wyjątek wykonania oraz ukryć szczegóły wycofania transakcji i zamknięcia sesji. Dokładniej zajmiemy się tym tematem w podrozdziale 8.1.

Warto pamiętać o jednym wymogu: obiekt `Session` należy od razu zamknąć i odrzucić (nie używać ponownie) w momencie zajścia wyjątku. Hibernate nie próbuje wznowiać błędnych transakcji. W praktyce nie stanowi to problemu, bo wyjątków bazodanowych najczęściej nie daje się uratować (na przykład z powodu złamania ograniczeń) i nie istnieje jasno zdefiniowany stan po błędzie. Wdrożona aplikacja nie powinna zgłaszać żadnych wyjątków bazodanowych.

Wspomnieliśmy wcześniej, że wywołanie `commit()` synchronizuje stan sesji z bazą danych. Operację tę nazywa się **opróżnianiem**. Proces odbywa się automatycznie w momencie korzystania z interfejsu `Transaction`.

5.1.3. Opróżnianie sesji

Sesja Hibernate implementuje **przezroczysty zapis opóźniony**. Zmiany dokonane w modelu dziedzinowym w zasięgu sesji nie trafiają od razu do bazy danych. Dzięki temu Hibernate potrafi zgromadzić wiele zmian przy minimalnej liczbie poleceń wysyłanych do bazy danych, co pomaga ograniczyć opóźnienia związane z siecią.

Jeśli właściwość obiektu ulega w tej samej transakcji dwukrotnej zmianie, Hibernate musi wykonać tylko jedno polecenie UPDATE. Inną zaletą zapisu opóźnionego jest możliwość skorzystania z interfejsu wykonywania w JDBC ciągu podobnych operacji INSERT, UPDATE lub DELETE.

Opróżnianie bufora zmian występuje w następujących sytuacjach:

- ◆ zatwierdzania transakcji,
- ◆ czasami przed wykonaniem zapytania,
- ◆ w momencie jawnego wywołania metody `Session.flush()` przez aplikację.

Przesył stanu sesji do bazy danych na końcu transakcji bazodanowej to jedyny sposób zapewnienia trwałości dokonanych zmian — stanowi więc sytuację podstawową. Hibernate nie opróżnia bufora zmian przed każdym zapytaniem. Jeśli jednak zmiany dokonane w obiektach w pamięci mogą mieć wpływ na zwracane wyniki, Hibernate domyślnie dokonuje wcześniej synchronizacji.

Zachowaniem dotyczącym opróżniania można sterować, stosując metodę `session.setFlushMode()` i przekazując jeden z podanych trybów:

- ◆ `FlushMode.AUTO` — wartość domyślna zapewniająca działanie w opisany wcześniej sposób.
- ◆ `FlushMode.COMMIT` — wskazuje, że sesja nie zostanie zapisana przed wykonaniem zapytania (zajdzie jedynie na końcu transakcji przy jej zatwierdzaniu). Pamiętaj, że to ustwienie może doprowadzić do niepoprawności danych, jeśli modyfikowane obiekty korzystają z wyników zapytań wewnętrz transakcji.
- ◆ `FlushMode.NEVER` — powoduje, że jedynie jawnie wywołania metody `flush()` skutkują synchronizacją danych sesji z bazą danych.

Nie zalecamy zmiany na wartości inne niż domyślna. Inne strategie tylko w rzadkich przypadkach zwiększą wydajność. Większość aplikacji nie potrzebuje jawnych wywołań metody `flush()`. Przydają się one tylko w momencie korzystania z wyzwalaczy, mieszania kodu Hibernate z bezpośrednią obsługą JDBC lub rozwiązywania problemów z złe napisanymi sterownikami JDBC. Warto wiedzieć o tej możliwości, ale niekoniecznie z niej korzystać.

Po przedstawieniu ogólnych zagadnień użycia transakcji bazodanowych w połączeniu z interfejsem Hibernate, przejdźmy do tematu wspólnego dostępu do danych.

Sam termin „izolacja transakcji” może sugerować, że temat nie jest obszerny — albo coś jest izolowane, albo nie. Nic bardziej mylnego. Pełna izolacja współbieżnych transakcji bywa wyjątkowo kosztowna w kategoriach skalowalności

aplikacji, więc systemy bazodanowe stosują kilka poziomów izolacji. W większości aplikacji niepełna izolacja nie stanowi problemu. Warto poznać dostępne poziomy, by dobrać najlepszy dla aplikacji stosującej Hibernate i rozumieć integrację systemu ORM z transakcjami bazy danych.

5.1.4. Poziomy izolacji

Bazy danych (i inne systemy transakcyjne) starają się zapewnić izolację transakcji w taki sposób, by przy wielu współbieżnych transakcjach każda z nich sądziła, że jest jedyną transakcją obsługiwana przez system.

Tradycyjne podejście do izolacji wykorzystuje blokady. Transakcja zakłada blokadę na konkretny element danych, tymczasowo uniemożliwiając dostęp do niego dla innych transakcji. Niektóre nowoczesne bazy danych, na przykład Oracle lub PostgreSQL, implementują izolację transakcji, stosując sterowanie współbieżnością przez wersjonowanie (*MVCC — MultiVersion Concurrency Control*) zapewniające na ogół lepszą skalowalność. Omówimy izolacje zakładające model blokujący (choć większość z obserwacji dotyczy również współbieżności przez wersjonowanie).

Cały przedstawiony opis dotyczy transakcji i poziomów izolacji istniejących w systemach bazodanowych. Hibernate nie wprowadza żadnej własnej semantyki i stosuje rozwiązanie zapewniane przez bazę danych. Jeśli uwzględni się lata doświadczeń zdobyte przez twórców systemów bazodanowych w związku z implementacją współbieżności, łatwo dostrzec zalety tego podejścia. Programista aplikacji stosującej Hibernate powinien znać możliwości bazy danych i umieć dostosować poziom izolacji do wymagań aplikacji w kwestii integralności danych.

Problemy związane z izolacją

Przyjrzyjmy się najpierw kilku zjawiskom występującym w trakcie izolacji transakcji. Standard ANSI SQL definiuje poziomy izolacji w oparciu o możliwość wystąpienia następujących zjawisk:

- ◆ **Utraty aktualizacji** — dwie transakcje aktualizują wiersz. Druga wycofuje się, co powoduje utratę obu aktualizacji. Sytuacja występuje w systemach, które nie implementują blokad. Współbieżne transakcje nie są izolowane.
- ◆ **Brudnego odczytu** — jedna transakcja odczytuje zmiany dokonane przez inną, jeszcze nie zatwierdzoną transakcję. Nie jest to bezpieczne, bo zmiany z drugiej transakcji można wycofać.
- ◆ **Niepowtarzalnego odczytu** — transakcja dwukrotnie odczytuje wiersz danych i za każdym razem otrzymuje inny wynik. Przykładowo, inną transakcją zapisała dane do tego wiersza i zatwierdziła transakcję między dwoma odczytami.
- ◆ **Problem utraty wcześniejszego zapisu** — specjalny przypadek niepowtarzalnego odczytu. Wyobraźmy sobie dwie współbieżne transakcje odczytujące wiersz. Pierwsza z nich dokonała w nim zmian i go zapisała.

Następnie ponownie postąpiła druga. Zmiany dokonane przez pierwszą uległy utracie.

- ◆ **Otrzymanie fantomu** — transakcja dwukrotnie wykonuje zapytanie. Drugi zbiór wyników zawiera wiersze niedostępne w pierwszym zbiorze wyników (niekoniecznie musi to być dokładnie to samo zapytanie). Sytuację tę powoduje inna transakcja wstawiająca nowe wiersze między wywołaniem dwóch zapytań.

Po zapoznaniu się z możliwymi sytuacjami niepożądanymi, możemy przejść do określenia poziomów izolacji transakcji i opisania problemów, którym zapobiegają.

Poziomy izolacji

Podstawowe poziomy izolacji definiuje standard ANSI SQL. Nie zawsze wszystkie one są dostępne we wszystkich systemach bazodanowych. JTA definiuje te same poziomy izolacji. Używa się ich do określenia preferowanego poziomu izolacji dla transakcji aplikacji.

- ◆ **Odczyt niezatwierdzony** — dopuszcza brudne odczyty, ale uniemożliwia utratę aktualizacji. Jedna transakcja nie może zmienić danych wiersza, jeśli inna niezatwierdzona transakcja dokonała wcześniej zmian w tym wierszu. Dowolna transakcja może odczytać dowolny wiersz. Ten poziom izolacji łatwo zaimplementować, używając blokad zapisu na wyłączność.
- ◆ **Odczyt zatwierdzony** — nie zapewnia powtarzalnego odczytu, ale przeciwdziała brudnym odczytom. Poziom ten uzyskuje się, stosując chwilowe współdzielone blokady odczytu i blokady zapisu na wyłączność. Transakcje odczytujące nie blokują innych transakcji w kwestii dostępu do danych wiersza. Z drugiej strony zapis danych do wiersza blokuje wszystkim innym transakcjom możliwość jego odczytania.
- ◆ **Odczyt powtarzalny** — rozwiązuje problemy brudnego i niepowtarzalnego odczytu. Mogą jednak wystąpić fantomy. Poziom ten uzyskuje się, stosując chwilowe współdzielone blokady odczytu i blokady zapisu na wyłączność. Transakcje odczytujące blokują transakcje zapisujące (ale nie inne transakcje odczytujące). Transakcje zapisujące blokują wszystkie inne transakcje.
- ◆ **Odczyt szeregowalny** — zapewnia najbardziej restrykcyjny poziom izolacji. Emuluje uszeregowane wykonywanie transakcji, czyli stara się zapewnić taki sam efekt, jak w sytuacji, gdy transakcje wykonuje się jedna po drugiej. Tego rozwiązania nie można zaimplementować przy użyciu wyłącznie blokad. Musi istnieć dodatkowy mechanizm blokujący pokazywanie się nowych wierszy w już działających transakcjach.

Dobrze znać znaczenie wszystkich terminów technicznych. W jaki sposób ma to pomóc w określeniu odpowiedniego poziomu izolacji dla aplikacji?

5.1.5. Wybór poziomu izolacji

Twórcy aplikacji (włączając w to autorów książek) często mają trudności z okresem, który poziom izolacji zastosować dla kreatywnej aplikacji. Zbyt wysoki poziom izolacji zmniejszy wydajność wysoce współbieżącej aplikacji. Zbyt niski poziom izolacji może doprowadzić do subtelnych, trudnych do wykrycia błędów w aplikacji. Co gorsza, błędy te trudno powtórzyć, bo występują tylko przy dużym obciążeniu systemu.

W przedstawionym opisie pojawią się terminy **buforowanie** i **blokady optymistyczne** (dzięki wersjonowaniu). Ich wyjaśnienie znajduje się w dalszej części rozdziału. Można ominąć ten podrozdział i powrócić do niego później, gdy konieczne będzie podjęcie decyzji co do stosowanego w aplikacji poziomu izolacji. Wybór ten w ogromnej mierze zależy od rodzaju aplikacji. Prezentowany opis ma przede wszystkim zalecać pewne rozwiązania, a nie je narzucać.

Hibernate stara się być możliwie przezroczysty w kwestii semantyki transakcyjnej bazy danych, ale buforowanie i blokady optymistyczne wpływają na tę semantykę. Czym kierować się wybierając odpowiedni poziom izolacji dla aplikacji stosującej Hibernate?

Na samym wstępnie należy odrzucić poziom **odczytu niezatwierdzonego**. Niezmierne groźne jest stosowanie niezatwierdzonych zmian z innej transakcji. Wycofanie jednej z błędnych transakcji może wtedy wpłynąć na inne transakcje i doprowadzić do powstania w nich błędów (czyli uzyskania niespójnego stanu bazy danych). Możliwa jest sytuacja, w której dane wycofanej transakcji i tak zostaną zatwierdzone, gdy odczyta je wcześniej inna, poprawnie wykonana transakcja!

Większość aplikacji nie potrzebuje poziomu **odczytu szeregowalnego** (fantomy najczęściej nie stanowią problemu), a poziom ten najtrudniej poddaje się współbieżności. Tylko kilka istniejących aplikacji produkcyjnych stosuje ten poziom izolacji. Inne aplikacje najczęściej stosują blokady pesymistyczne (patrz punkt 5.1.7), które wymuszają szeregowanie wykonywanych transakcji, gdy okazuje się to niezbędne.

Pozostają do wyboru dwa poziomy: odczytu zatwierdzonego i powtarzanego. Rozważmy najpierw ten drugi. Eliminuje on sytuację, w której jedna transakcja nadpisuje zmiany dokonane przez inną współbieżną transakcję (problem utraty wcześniejszego zapisu), jeśli cały dostęp do danych odbywa się w jednej, niepodzielnej transakcji bazodanowej. To ważne zagadnienie, choć istnieją inne metody uniknięcia utraty zapisu.

Załóżmy, że wykorzystujemy wersjonowanie danych. Hibernate potrafi wykonywać to zadanie za programistę w sposób automatyczny. Połączenie (obowiązkowe) pierwszego poziomu buforowania z wersjonowaniem zapewnia większość zalet odczytu powtarzanego. W szczególności zapobiega problemowi utraty wcześniejszego zapisu. Co więcej, bufor pierwszego poziomu daje pewność, że na stan obiektów trwałych nie wpływają inne transakcje. Oznacza to, że poziom izolacji odczytu zatwierdzonego jest do zaakceptowania, jeśli stosuje się wersjonowanie.

Odczyt powtarzalny zwiększa powtarzalność wyników zapytań (tylko w czasie trwania transakcji bazodanowej), ale z powodu możliwości wystąpienia fantomu

nie przynosi wielu korzyści. Pamiętajmy, że niewiele aplikacji internetowych w jednej transakci dwukrotnie odczytuje zawartość tej samej tabeli.

Rozważmy jeszcze drugi (opcjonalny) poziom buforowania w Hibernate. Może zapewnić ten sam poziom izolacji jak transakcja bazodanowa, choć w mniejszym stopniu. Jeśli mocno korzysta się ze strategii współbieżnego bufora dla drugiego poziomu buforowania, który nie oferuje odczytu powtarzanego (patrz strategie odczyt-zapis i nieścisły odczyt-zapis omówione w dalszej części rozdziału), wybór poziomu izolacji staje się prosty: nie można osiągnąć odczytu powtarzanego, więc po co zwiększać obciążenie bazy danych. A co, jeśli nie używa się bufora drugiego poziomu dla krytycznych klas lub stosuje bufor zapewniający pełną izolację w trybie powtarzanego odczytu? Wtedy można użyć odczytu powtarzanego dla transakcji bazodanowych, ale najczęściej nie warto ze względów wydajnościowych.

Wybór poziomu izolacji umożliwia wybór dobrej domyślnej strategii blokowania dla wszystkich transakcji bazodanowych. Jak ustawić poziom izolacji?

5.1.6. Ustawianie poziomu izolacji

Każde połączenie JDBC używa domyślnego poziomu izolacji bazy danych, najczęściej odczytu powtarzanego lub zatwierdzonego. Nic nie stoi na przeszkodzie, by zmieć domyślne ustawienie bazy danych. Równie dobrze zmian można dokonać za pomocą opcji konfiguracyjnej Hibernate.

```
hibernate.connection.isolation = 4
```

Hibernate będzie wtedy ustawał poziom izolacji dla każdego połączenia JDBC uzyskanego z puli połączeń przed rozpoczęciem transakcji. Dostępne wartości są następujące (odpowiadają one stałym z `java.sql.Connection`):

- ◆ 1 — odczyt niezatwierdzony,
- ◆ 2 — odczyt zatwierdzony,
- ◆ 4 — odczyt powtarzalny,
- ◆ 8 — odczyt serializowany.

Hibernate nigdy nie zmienia poziomu izolacji połączeń uzyskanych ze źródła danych zapewnianego przez serwer aplikacji w środowisku zarządzanym. W tej sytuacji domyślny poziom izolacji określa się w konfiguracji serwera aplikacji.

Ustawienie poziomu izolacji jest opcją globalną wpływającą na wszystkie połączenia i transakcje. Czasem zachodzi potrzeba wprowadzenia bardziej restrykcyjnej blokady dla transakcji. Hibernate dopuszcza jawne określenie blokady **pesymistycznej**.

5.1.7. Blokada pesymistyczna

Blokada to mechanizm uniemożliwiający współbieżny dostęp do konkretnego elementu. Gdy transakcja założy blokadę na element, inne transakcje nie mogą go odczytywać i (lub) modyfikować. Blokada może być chwilowa (stosowana tylko

w trakcie odczytu elementu) lub trwać aż do zakończenia transakcji. **Blokada pesymistyczna** to blokada zakładana w momencie odczytu danych, która trwa aż do zakończenia transakcji.

W poziomie odczytu zatwierdzonego (zalecany przez autorów poziom izolacji) baza danych nigdy nie zakłada blokady pesymistycznej, dopóki się jej tego jawnie nie rozkaże. Najczęściej ten typ blokady nie jest najlepszy pod kątem wydajnościowym w wysoce współbieżnej aplikacji. Jednak w pewnych sytuacjach pomaga zapobiec blokadzie wzajemnej na poziomie bazy danych, która zawsze prowadzi do błędnego zakończenia jednej z transakcji. Niektóre bazy danych (na przykład Oracle i PostgreSQL) wykorzystują specjalną składnię, polecenie `SELECT ... FOR UPDATE`, by wskazać potrzebę założenia blokady pesymistycznej. Klasy dialektów baz danych dostarczane z Hibernate informują, czy system bazodanowy obsługuje tę funkcję. Jeśli jej nie obsługuje, Hibernate zawsze wykona zwykłe polecenie `SELECT` bez `FOR UPDATE`.

Klasa `LockMode` z Hibernate pozwala zażądać blokady pesymistycznej dla konkretnego elementu. Co więcej, klasa ta umożliwia wymuszenie na Hibernate pominięcia warstwy buforowania lub prostego sprawdzenia wersji. Korzyści płynące z tych operacji opiszemy dokładniej w części poświęconej buforowaniu i wersjonowaniu.

Przyjrzyjmy się kodowi stosującemu `LockMode`. Jeśli transakcja wygląda następująco:

```
Transaction tx = session.beginTransaction();
Category cat = (Category) session.get(Category.class, catId);
cat.setName("Nowa nazwa");
tx.commit();
```

blokadę pesymistyczną uzyskuje się, modyfikując kod w podany niżej sposób:

```
Transaction tx = session.beginTransaction();
Category cat = (Category) session.get(Category.class, catId,
LockMode.UPGRADE);
cat.setName("Nowa nazwa");
tx.commit();
```

W tym trybie Hibernate wczyta obiekt `Category`, używając zapytania `SELECT ... FOR UPDATE`, by zablokować pobrane wiersze w bazie danych do momentu ich zwolnienia na końcu transakcji.

Hibernate definiuje kilka trybów blokad:

- ◆ `LockMode.NONE` — nie wykonuj zapytania bazodanowego, chyba że obiektu nie ma w buforze.
- ◆ `LockMode.READ` — pomiń oba poziomy buforowania i dokonaj sprawdzenia wersji, by skontrolować, czy dane w pamięci są dokładnie takie same jak w bazie danych.
- ◆ `LockMode.UPGRADE` — pomiń oba poziomy buforowania i dokonaj sprawdzenia wersji (jeśli ma to sens) i uzyskaj blokadę pesymistyczną na poziomie bazy danych (o ile baza danych ją obsługuje).

- ◆ LockMode.UPGRADE_NOWAIT — te same działania co przy UPGRADE, ale Hibernate stosuje zapytanie SELECT ... FOR UPDATE NOWAIT dla bazy danych Oracle. Wyłącza to czekanie na zwolnienie blokad dotyczących współbieżności i natychmiastowe zgłoszenie wyjątku, jeśli blokady nie udała się uzyskać.
- ◆ LockMode.WRITE — uzyskiwany automatycznie, gdy Hibernate zapisał dane do wiersza w aktualnej transakci (jest to tryb wewnętrzny, którego nie można jawnie określić).

Domyślnie metody get() i load() stosują tryb LockMode.NONE. Tryb LockMode.READ przydaje się w metodzie Session.lock() stosowanej z obiektem odłączonym. Oto przykład:

```
Item item = ....;
Bid bid = new Bid();
item.addBid(bid);

...
Transaction tx = session.BeginTransaction();
session.lock(item, LockMode.READ);
tx.commit();
```

Kod przeprowadza sprawdzenie wersji dla odłączonego obiektu Item, by zbadać, czy wiersz bazy danych nie został w międzyczasie uaktualniony przez inną transakcję, przed przystąpieniem do zapisania nowego obiektu Bid w sposób kaskadowy (zakładamy, że asocjacja od Item do Bid ma włączone kaskadowe wykonywanie operacji zapisu).

Jawnie podając inny tryb blokady niż LockMode.NONE, wymusza się na Hibernate pominięcie obu poziomów buforowania i przejście do bazy danych. Wydaje się, że przez większość czasu buforowanie jest ważniejsze od blokad pesymistycznych, więc nie warto ich stosować, dopóki nie są naprawdę potrzebne. Nasza rada: jeśli w projekcie uczestniczy profesjonalny administrator baz danych, niech zadecyduje, które transakcje wymagają blokad pesymistycznych po uruchomieniu w pełni działającej aplikacji. Decyzję podejmuje się, śledząc dokładnie wszystkie zależności między transakcjami, które trudno odgadnąć w czasie projektowania.

Rozważmy inny aspekt współbieżnego dostępu do danych. Większość programistów języka Java dokładnie wie, czym jest transakcja bazodanowa, i niejednokrotnie używa samego słowa **transakcja**. W niniejszej książce tego rodzaju transakcje uważamy za **szczegółowe**, by odróżnić je od transakcji ogólnych. Transakcje ogólne odpowiadają operacjom, które **użytkownik aplikacji** uważa za jedną jednostkę zadaniową. Jaka jest różnica między tymi rodzajami, że trzeba je wyróżniać?

Baza danych izoluje (ogranicza) efekt oddziaływania na siebie współbieżnych transakcji. Każdej transakcji wydaje się, że jest wykonywana jako jedyna i ma pełny dostęp do bazy danych (nawet jeśli tak nie jest). Izolacja jest kosztowna. Baza danych alokuje mnóstwo zasobów dla każdej transakcji w czasie jej trwania. Zasobami tymi są między innymi blokady wierszy odczytywanych lub uaktualnianych przez transakcję, których zadaniem jest uniemożliwić dostęp do tych danych innym transakcjom dopóki nie zakończy się transakcja zakładająca blokadę.

W wysoce współbieżnych systemach blokady mogą utrudnić skalowalność systemu, jeśli trwają dłużej, niż jest to konieczne. Z tego powodu nie należy trzymać otwartej transakcji bazodanowej (a nawet połączenia JDBC) przez zbyt długi czas. Oczywiście wszystko to dotyczy obiektu Transaction z Hibernate, bo tak naprawdę obiekt ten stanowi tylko adapter ukrywający rzeczywisty mechanizm transakcyjny.

Jeśli chce się obsłużyć długo myślącego użytkownika przy zachowaniu zasad ACID dla transakcji, proste transakcje bazodanowe nie wystarczają. Potrzeba nowej koncepcji — długo działających transakcji aplikacyjnych.

5.2. Transakcje aplikacyjne

Procesy biznesowe, które określają jednostkę zadaniową z punktu widzenia użytkownika, niejednokrotnie wymagają kilku interakcji użytkownika. Sytuacja tego typu pojawia się najczęściej, gdy użytkownik podejmuje decyzję co do aktualizacji danych na podstawie ich aktualnej zawartości.

Załóżmy ekstremalną sytuację. Przypuśćmy, że użytkownik wpisuje informacje na kilku kolejnych ekranach (często spotykane kreatory). Dane powiązanych elementów trzeba odczytywać i zapisywać w kilku żądaniach (czyli w kilku transakcjach bazodanowych) aż do momentu, w którym użytkownik kliknie przycisk *Zakończ*. W trakcie procesu wprowadzania dane muszą pozostać spójne, a użytkownik powinien być informowany o wszystkich zmianach w danych dokonanych przez współbieżne transakcje. Tego rodzaju ogólne transakcje nazywamy **transakcjami aplikacyjnymi**, bo wykorzystują szerszą koncepcję jednostki zadaniowej.

Zdefiniujmy ten rodzaj transakcji nieco dokładniej. Większość aplikacji internetowych stosuje następujące podejście do obsługi użytkownika:

1. Pobranie danych i ich wyświetlenie występuje w pierwszej transakcji bazodanowej.
2. Użytkownik przegląda lub edytuje dane poza jakikolwiek transakcją bazodanową.
3. Za utrwalenie modyfikacji odpowiada druga transakcja bazodanowa.

W bardziej złożonych aplikacjach cały proces powtarza się kilkukrotnie, zanim całe zadanie biznesowe ulegnie zakończeniu. Ten sposób działania określa się mianem transakcji aplikacyjnej (**transakcji długo działającej, transakcji użytkownikowej lub transakcji biznesowej**). Preferujemy określenie transakcja aplikacji lub użytkownika, gdyż podkreśla to rozważanie całego zagadnienia od strony użytkownika.

Ponieważ nie można polegać na bazie danych w celu zapewnienia izolacji (a nawet niepodzielności) współbieżnych transakcji aplikacyjnych, izolacja staje się problemem samej aplikacji lub nawet użytkownika.

Omówmy transakcje aplikacyjne na przykładzie.

W aplikacji CaveatEmptor zarówno użytkownik będący autorem komentarza, jak i administrator systemu może otworzyć ekran edycji komentarza w celu edycji

treści komentarza lub jego usunięcia. Założymy, że dwóch administratorów w tym samym momencie otwiera ekran edycji tego samego komentarza. Obaj dokonują zmian i przesyłają wyniki do aplikacji. W tym miejscu mamy trzy sposoby obsługi współbieżnego zapisu danych do bazy danych.

- ◆ Wygrywa ostatnie zatwierdzenie — obie aktualizacje są poprawne, druga nadpisuje pierwszą. System nie wyświetla komunikatu o błędzie.
- ◆ Wygrywa pierwsze zatwierdzenie — pierwsza modyfikacja trafia do bazy danych, natomiast użytkownik drugiej uzyskuje komunikat o błędzie i musi rozpocząć proces biznesowy od nowa, by zapisać swoje zmiany. To rozwiązanie najczęściej nazywa się **blokadą optymistyczną**.
- ◆ System złącza obie aktualizacje — pierwsza modyfikacja jest trwała. W drugiej użytkownik sam podejmuje decyzję, czy chce ją zapisać.

Pierwszy z wymienionych sposobów, czyli wygrana drugiego zatwierdzenia, jest kontrowersyjna: drugi użytkownik nadpisuje zmiany dokonane przez pierwszego użytkownika bez ich przejrzenia, a nawet bez wiedzy, że w ogóle istniały. W przedstawionym przykładzie nie miałoby to prawdopodobnie dużego znaczenia, ale w większości sytuacji podejście to nie jest akceptowalne. W typowych aplikacjach dopuszcza się rozwiązanie drugie lub trzecie. Z naszego punktu widzenia trzeci sposób stanowi tak naprawdę odmianę sposobu drugiego — zamiast informować o błędzie, przedstawiamy oba komunikaty drugiemu użytkownikowi, by je ręcznie scalili. Nie istnieje pojedyncze, najlepsze rozwiązanie. Należy dokładnie sprawdzić wymagania biznesowe, by wybrać najbardziej odpowiedni sposób obsługi.

Pierwszy sposób nie wymaga wykonywania w aplikacji żadnych dodatkowych kroków i będzie realizowany domyślnie. Nie trzeba włączać żadnych opcji w aplikacji lub Hibernate, by go uzyskać. Aplikacja użycie dwóch transakcji bazodanowych: dane komentarza załaduje pierwsza transakcja, a druga transakcja zapisze przesłane zmiany bez sprawdzania, czy wiersz nie został w międzyczasie zmieniony.

Hibernate pomaga w implementacji drugiego i trzeciego rozwiązania, używając wersjonowania zarządzanego i blokad optymistycznych.

5.2.1. Wersjonowanie zarządzane

Wersjonowanie zarządzane korzysta z inkrementowanych numerów wersji lub znaczników czasowych zmienianych na aktualny czas po każdej modyfikacji obiektu. Wersjonowanie zarządzane wymaga dodania nowej właściwości do klasy Comment i odwzorowanie jej elementem <version>. Najpierw przyjrzymy się zmianom w klasie Comment.

```
public class Comment {  
    ...  
    private int version;  
    ...  
    void setVersion(int version) {  
        this.version = version;  
    }  
    int getVersion() {  
        ...  
    }  
}
```

```

        return version;
    }
}

```

Nic nie stoi na przeszkodzie, by metody dostępowe były publiczne. Właściwość odwzorowująca <version> musi znaleźć zaraz za właściwością identyfikującą w pliku odwzorowania klasy Comment.

```

<class name="Comment" table="COMMENTS">
    <id ...>
        <version name="version" column="VERSION"/>
    ...
</class>

```

Numer wersji to zwykły licznik bez żadnego konkretnego znaczenia. Niektórzy preferują stosowanie znaczników czasowych.

```

public class Comment {
    ...
    private Date lastUpdatedDatetime;
    ...
    void setLastUpdatedDatetime(int lastUpdatedDatetime) {
        this.lastUpdatedDatetime = lastUpdatedDatetime;
    }
    int getLastUpdatedDatetime() {
        return lastUpdatedDatetime;
    }
}

<class name="Comment" table="COMMENTS">
    <id ... />
    <timestamp name="lastUpdatedDatetime" column="LAST_UPDATED"/>
    ...
</class>

```

W teorii znacznik czasowy jest mniej bezpieczny, gdyż dwie współbieżne transakcje mogą zostać załadowane i uaktualnione w tej samej milisekundzie. W praktyce sytuacja ta okazuje się mało prawdopodobna. Niemniej i tak zalecamy, by w nowych projektach używać wersji liczbowych zamiast znaczników czasowych.

Nie musisz samemu ustawać wartości właściwości wersji lub znacznika czasowego. Hibernate uczyni to automatycznie w momencie zapisu pierwszego obiektu Comment. Kolejne zapisy będą powodować inkrementację wartości.

Odpowiedzi na pytania

Czy dochodzi do aktualizacji wersji rodzica w momencie modyfikacji potomka? Przypuśćmy, że zmieniamy jedną z ofert kolekcji bids obiektu Item. Czy spowoduje to zwiększenie numeru wersji obiektu Item? Odpowiedź na te i podobne pytania jest prosta: Hibernate zwiększy numerację tylko wtedy, gdy uzna, że obiekt jest brudny. W takim samym stopniu dotyczy to zwykłych właściwości i kolekcji. Przyjrzymy się dokładniej sytuacji związku między Item i Bid: modyfikacja zawartości obiektu Bid nie powoduje zabrudzenia obiektu Item, więc numeracja się nie zwiększy. Gdybyśmy jednak dodali do obiektu Bid nową ofertę, Hibernate zmieni wersję z powodu uaktualnienia obiektu. Oczywiście można klasę Bid uczynić niezmienną, gdyż na ogół nie dopuszcza się zmian oferty.

Przy każdej aktualizacji komentarza Hibernate stosuje kolumnę z numerem wersji w klauzuli WHERE.

```
update COMMENTS set COMMENT_TEXT='Nowy komentarz', VERSION=3  
WHERE COMMENT_ID=123 AND VERSION=2
```

Gdyby inna transakcja aplikacyjna wcześniej uaktualniła ten sam element po jego odczytaniu przez aktualną transakcję, kolumna VERSION nie będzie zawierała wartości 2, więc polecenie SQL nie uaktualni wiersza. Hibernate sprawdza liczbę modyfikacji zwracaną przez sterownik JDBC — jeśli uzyska wartość 0, zgłosi wyjątek `StaleObjectStateException`.

Po wykryciu wyjątku aplikacja może wyświetlić użytkownikowi drugiej transakcji komunikat o błędzie informujący o korzystaniu z niepewnych danych, które zostały zmienione przez inną osobę. Innymi słowy, aplikacja stosuje w tym momencie podejście, w którym **wygrywa pierwsze zatwierdzenie**. Ewentualnie wykrycie wyjątku może prowadzić do wyświetlenia drugiemu użytkownikowi dodatkowego ekranu, w którym scali zmiany dokonane w obu wersjach.

Hibernate znaczco ułatwia stosowanie wersjonowania do implementacji blokad optymistycznych. Czy można razem stosować blokadę optymistyczną i pesymistyczną czy może tylko jedną z nich? Dlaczego tę pierwszą nazywa się **optymistyczną**?

Podejście optymistyczne zakłada, że w zdecydowanej większości sytuacji wszystko przebiega poprawnie — konflikty przy zapisie danych zachodzą rzadko. Zamiast blokować dostęp do danych w sposób pesymistyczny tuż po ich pobraniu (i tym samym wymuszać uszeregowanie wykonywanych działań), optymistyczne sterowanie współbieżnością blokuje zapis i zgłasza błąd dopiero na końcu jednostki zadaniowej.

Obie strategie mają swoje zastosowania. Aplikacje wielużytkownikowe najczęściej domyślnie stosują podejście optymistyczne. Blokady pesymistyczne wykorzystują tylko wtedy, gdy jest to konieczne. Zauważ, że długość blokady pesymistycznej w Hibernate nie może przekroczyć długości jednej transakcji bazodanowej! Innymi słowy, blokady na wyłączność nie można utrzymać między transakcjami bazodanowymi. Nie uważamy tego za coś złego, gdyż prowadziłoby to do przechowywania w pamięci bardzo kosztownej blokady o długości transakcji aplikacyjnej. Każda blokada zmniejsza wydajność, bo dostęp do danych wymaga sprawdzenia istnienia blokady w synchronizowanym zarządcy blokad. Gdy jest to rzeczywiście niezbędne, można samemu utworzyć długoterminową blokadę pesymistyczną, wykorzystując narzędzia dostępne w Hibernate. Choć przykłady znajdują się w witrynie Hibernate, nie zalecamy tego podejścia. Warto dobrze rozważyć wpływ tego rozwiązania na wydajność.

Powróćmy do transakcji aplikacyjnych. Poznaliśmy podstawy wersjonowania zarządzanego i blokad optymistycznych. W poprzednich rozdziałach (i w tym rozdziale) wspominaliśmy, iż sesja Hibernate nie jest równoważna transakcji. Sesja ma elastyczny zasięg. Można ją stosować w różny sposób w transakcjach bazodanowych i aplikacyjnych. Innymi słowy, zmienna jest szczegółowość sesji — może stanowić dowolną jednostkę zadaniową.

5.2.2. Szczegółowość sesji

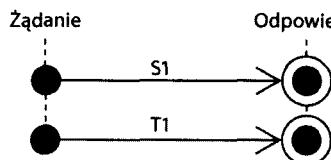
Aby poznać sposoby korzystania z obiektu Session, rozważmy jej związek z transakcją. Wcześniej opisaliśmy dwa podobne pojęcia:

- ◆ zasięg identyczności obiektu (patrz podrozdział 4.1.4),
- ◆ szczegółowość transakcji bazodanowych i aplikacyjnych.

Obiekt Session z Hibernate definiuje zasięg identyczności obiektu. Obiekt Transaction określa zasięg transakcji bazodanowej.

Jaki związek występuje między obiektem Session i transakcjami aplikacyjnymi? Zacznijmy od najprostszego zastosowania sesji.

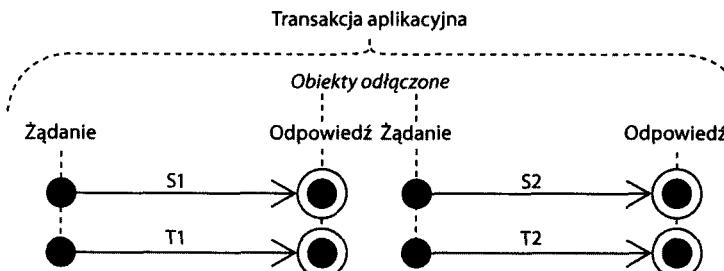
Najczęściej otwieramy nową sesję dla każdego żądania klienta (czyli żądania przeglądarki internetowej) i rozpoczynamy nową transakcję. Po wykonaniu logiki biznesowej zatwierdzamy zmiany w bazie danych i zamkamy sesję przed wysłaniem odpowiedzi do klienta (patrz rysunek 5.2).



Rysunek 5.2. Użycie jednej transakcji na jedną sesję w cyklu żądanie-odpowiedź

Sesja (S1) i transakcja (T1) mają tę samą szczegółowość. Gdy nie korzysta się z transakcji aplikacyjnych, to proste podejście powinno wystarczyć do wykonania aplikacji. Nadajmy temu podejściu nazwę sesja na każde żądanie.

Gdy potrzeba długo działającej transakcji aplikacyjnej, można ją zaimplementować w ten sam sposób, używając obiektów odłączonych i blokad optymistycznych opisanych w poprzednim podrozdziale (patrz rysunek 5.3).

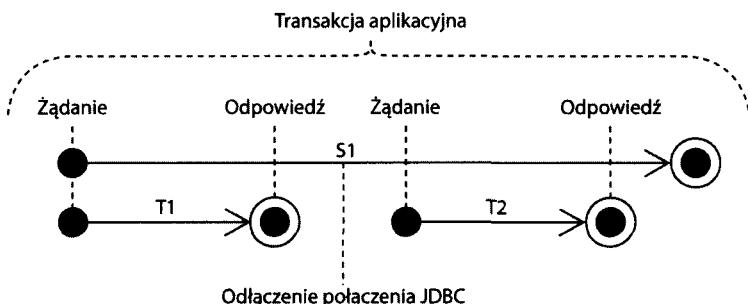


Rysunek 5.3. Implementacja transakcji aplikacyjnej z wieloma sesjami, po jednej na każde żądanie

Przypuśćmy, że transakcja aplikacyjna wymaga dwóch cykli żądanie-odpowiedź, czyli najczęściej dwóch żądań HTTP nadesłanych przez aplikację internetową. Można załadować obiekty w pierwszej sesji, a następnie dołączyć je do drugiej sesji po zmianach dokonanych przez użytkownika. Hibernate automatycznie wykona sprawdzenie wersji. Czas między (S1, T1) i (S2, T2) może być tak długi jak

czas użytkownika na zastanowienie i wpisanie nowych danych. To podejście nazwijmy **sesją na każde żądanie z obiektami odłączonymi**.

Podejście alternatywne polega na użyciu jednego obiektu Session obejmującego wiele żądań, by utworzyć jedną transakcję aplikacyjną. W tym przypadku nie trzeba martwić się o dołączenie odłączonych obiektów, gdyż pozostają one trwałe w kontekście jednej, długoterminowej sesji (patrz rysunek 5.4). Hibernate ponownie odpowiada za przeprowadzanie blokady optymistycznej.



Rysunek 5.4. Implementacja transakcji aplikacyjnej z sesją długoterminową

Obiekt Session serializowalny i tym samym może zostać bez przeszkód zapamiętyany w obiekcie HttpSession serwletu. Oczywiście połączenie JDBC trzeba zamknąć i ponownie uzyskać w kolejnym żądaniu. Do zakończenia połączenia służy metoda `disconnect()`, natomiast do ponownego uzyskania połączenia metoda `reconnect()` obiektu Session. To podejście nosi nazwę sesji na każdą transakcję aplikacji lub sesji długoterminowej.

Najczęściej na samym początku warto zastanowić się nad rozwiązaniem, w którym sesja Hibernate pozostaje otwarta nie dłużej niż pojedyncza transakcja bazodanowa (sesja na każde żądanie). Im dłużej po zakończeniu transakcji sesja pozostaje otwarta, tym większa jest szansa przechowywania w buforze przestarzałych obiektów trwałych (sesja jest obowiązkowym buforem pierwszego poziomu). Nigdy nie należy używać sesji dłużej niż jest to potrzebne do poprawnego zakończenia pojedynczej transakcji aplikacyjnej.

Sposób użycia sesji w transakcjach aplikacyjnych zależy od sposobu zaprojektowania aplikacji. Strategie implementacyjne wraz z przykładami opisujemy w podrozdziale 8.2.

Warto zastanowić się jeszcze nad jedną istotną kwestią. Gdy korzysta się ze starej bazy danych, zapewne nie można umieścić w niej nowej kolumny wersji lub znacznika czasowego dla blokady optymistycznej.

5.2.3. Inne sposoby implementacji blokady optymistycznej

Gdy nie można wprowadzić kolumny wersji lub znacznika czasowego, Hibernate nadal może dokonywać blokady optymistycznej, ale tylko dla obiektów pobranych i zmodyfikowanych w tej samej sesji. Obiekty odłączone muszą korzystać z kolumny wersji lub znacznika czasowego, by zapewnić blokadę optymistyczną.

Alternatywna implementacja blokady optymistycznej porównuje aktualny stan wiersza w bazie danych ze stanem niezmienionych wartości trwałych właściwości z momentu pobrania obiektu (lub ostatniego opróżnienia buforów). Ten tryb blokady włącza atrybut `optimistic-lock="all"` umieszczony w odwzorowaniu klasy.

```
<class name="Comment" table="COMMENT" optimistic-lock="all">
  <id ... />
  ...
</class>
```

W tym rozwiązaniu Hibernate dołączy do klauzuli WHERE wszystkie właściwości.

```
update COMMENTS set COMMENT_TEXT='Nowy tekst'
where COMMENT_ID=123
and COMMENT_TEXT='Stary tekst'
and RATING=5
and ITEM_ID=3
and FROM_USER_ID=45
```

Hibernate może dołączać do klauzuli jedynie zmodyfikowane właściwości (w przykładzie będzie to kolumna `COMMENT_TEXT`), jeśli atrybut będzie miał postać `optimistic-lock="dirty"`.

Nie polecamy tego rozwiązania. Jest wolniejsze, bardziej złożone i mniej pewne od wersjonowania. Poza tym nie działa, jeśli transakcja aplikacyjna korzysta z kilku sesji (a tym samym z obiektów odłączonych).

Ponownie zmieńmy perspektywę i rozważmy kolejny aspekt Hibernate. We wprowadzeniu do rozdziału wspomnieliśmy o ścisłym związku transakcji i buforowania danych. Podstawy transakcji i blokad oraz szczególności sesji stają się niezwykle istotne, gdy rozważa się buforowanie danych na warstwie aplikacji.

5.3. Buforowanie – teoria i praktyka

Podstawą do wskazywania większej wydajności aplikacji z obiektowo-relacyjną warstwą trwałości względem aplikacji stosujących bezpośrednie wywołania JDBC jest możliwość zaawansowanego buforowania danych. Choć wychodzimy z założenia, że aplikacje należy projektować w taki sposób, by działały wydajnie bez stosowania buforowania, pewne ich rodzaje — w szczególności te, które przede wszystkim odczytują dane lub przechowują dużo metadanych — mogą wiele zyskać po dodaniu buforowania.

Omawianie buforowania rozpoczęniemy od informacji podstawowych, między innymi od wyjaśnienia różnych zasięgów buforowania i tożsamości oraz wpływu buforowania na izolację transakcji. Przedstawione informacje dotyczą ogólnych zasad buforowania, nie są związane wyłącznie z Hibernate. Dzięki opisowi łatwiej zrozumiesz, dlaczego system buforowania powstał w taki, a nie inny sposób. Następnie skupimy się na buforowaniu w Hibernate: jego włączaniu, dostrajaniu i zarządzaniu oboma poziomami buforowania. Zalecamy uważną lekturę przedstawionych tu informacji przed włączeniem buforowania we własnej aplikacji.

cji Hibernate. Bez znajomości podstaw ryzykujesz utratę integralności danych i napotkanie trudnych do wykrycia błędów współbieżności.

Buforowanie stara się utrzymać reprezentację aktualnego stanu bazy danych możliwie blisko aplikacji — w pamięci operacyjnej lub na dysku twardym serwera aplikacji. Bufor danych stanowi ich lokalną kopię. Znajduje się między bazą danych i aplikacją. Dzięki niemu można uniknąć zapytania do bazy danych w sytuacji, gdy:

- ◆ aplikacja wyszukuje obiektu na podstawie jego identyfikatora (klucza głównego),
- ◆ warstwa trwałości leniwie pobiera asocjacje.

Możliwe jest też buforowanie wyników zapytań. W rozdziale 7. wskażemy, że w większości sytuacji buforowanie wyników zapytań nie powoduje znaczącego wzrostu wydajności, więc stosuje się je znacznie rzadziej.

Zanim zajmiemy się sposobami działania bufora danych w Hibernate, przyjrzyjmy się różnym metodom przechowywania zbuforowanych danych oraz ich wpływowi na współbieżności i identyczność.

5.3.1. **Strategie i zasięgi buforowania**

Buforowanie to tak istotny element trwałości obiektowo-relacyjnej, że trudno zrozumieć wydajność, skalalność i semantykę transakcyjną implementacji ORM bez poznania stosowanych rodzajów strategii buforowania. Istnieją trzy rodzaje buforów danych:

- ◆ **Zasięg transakcji** — dołączony do aktualnej jednostki zadaniowej, która może być transakcją bazodanową lub aplikacyjną. Jest poprawny i używany tak długo, jak działa jednostka zadaniowa. Każda jednostka zadaniowa stosuje własny bufor.
- ◆ **Zasięg procesu** — współdzielony przez wiele (również współbieżnych) jednostek zadaniowych lub transakcji. Oznacza to, że ten rodzaj buforowania ma duże znaczenie w kwestii izolacji transakcji. Bufor o zasięgu procesu może zapamiętywać całe obiekty lub tylko ich trwały stan.
- ◆ **Zasięg klastra** — współdzielony przez wiele procesów na tym samym komputerze lub na wielu komputerach tworzących klasterek. Wymaga pewnego rodzaju komunikacji międzyprocesowej w celu zachowania spójności. Informacje znajdujące się w buforze należy duplikować we wszystkich węzłach klastra. W wielu (jeśli nie wszystkich) aplikacjach bufor o zasięgu klastra nie ma dużego sensu, gdyż odczyt lub aktualnienie ukazuje się tylko niewiele szybsze niż w przypadku bezpośredniego skorzystania z bazy danych.

Warstwy trwałości mogą wykorzystywać wiele poziomów buforowania. Na przykład **spudłowanie** (czyli nieodnalezienie poszukiwanych danych w buforze) na poziomie transakcji może skutkować próbą wyszukania danych na poziomie procesu. Zapytanie bazodanowe staje się ostatecznością.

Typ buforowania wykorzystywany przez warstwę trwałości wpływa na zasięg identyczności obiektów (związek między identycznością obiektów Javy i tożsamością bazodanową).

Buforowanie i identyczność obiektów

Rozważmy buforowanie o zasięgu transakcji. Wydaje się naturalne, że bufor służy również do zapewnienia zasięgu identyczności obiektów trwałych. Oznacza to wykorzystanie bufora do obsługi identyczności: dwa wyszukania obiektu o tym samym identyfikatorze bazodanowym skutkują zwróceniem tego samego obiektu w danej jednostce zadaniowej. Bufor o zasięgu transakcji idealnie współpracuje z mechanizmem trwałości zapewniającym identyczność obiektów w zasięgu transakcji.

Mechanizmy trwałości z buforowaniem o zasięgu procesu mogą implementować identyczność obiektów o zasięgu procesu. Wtedy identyczność obiektów jest równoważna tożsamości bazodanowej w całym procesie. Dwa wyszukania wykorzystujące ten sam identyfikator bazodanowy w różnych jednostkach zadaniowych otrzymują ten sam obiekt. Ewentualnie obiekty uzyskane z bufora o zasięgu procesu mogą być zwracane *przez wartość*. Bufor zawiera krótki danych, a nie obiekty trwałe. W takiej sytuacji każda jednostka zadaniowa przechowuje własną kopię stanu (krótką) i tworzy własny obiekt trwały. Oznacza to brak identyczności zasięgu bufora i zasięgu identyczności.

Bufor o zasięgu klastra zawsze wymaga komunikacji między procesowej. W rozwiązaniach trwałości wykorzystujących obiekty POJO (między innymi Hibernate) obiekty zawsze przekazuje się zdalnie przez wartość. Innymi słowy, bufor klastrowy nie gwarantuje identyczności obiektów w klastrze. By ją uzyskać, należy korzystać z dwóch wcześniej wymienionych typów buforów.

W typowej aplikacji internetowej lub biznesowej najlepiej, gdy zasięg identyczności obiektów ogranicza się do jednostki zadaniowej, czyli nie ma potrzeby, by dwa różne wątki miały identyczne obiekty. Inne rodzaje aplikacji (aplikacje samowystarczalne lub architektury z „grubym” klientem) mogą preferować identyczność obiektów na poziomie procesu. Ma to duże znaczenie przy ograniczaniu wymagań pamięciowych — zajętość pamięci w identyczności o zasięgu transakcji jest wprost proporcjonalna do liczby współbieżnie wykonywanych jednostek zadaniowych.

Wadą identyczności procesowej jest konieczność synchronizacji dostępu do obiektów trwałych znajdujących się w buforze, co znacząco zwiększa ryzyko blokady wzajemnej.

Buforowanie i współbieżność

Każda implementacja ORM dopuszczająca współdzielenie obiektu trwałego przez wiele jednostek zadaniowych musi wykorzystywać blokady na poziomie obiektu w celu zapewnienia synchronizacji współbieżnego dostępu. Najczęściej implementuje się blokady zapisu i odczytu (przechowane w pamięci) wraz z wykrywaniem blokad wzajemnych. Implementacje takie jak Hibernate, które przechowują osobny zestaw obiektów dla każdej jednostki zadaniowej, w większości sytuacji unikają wspomnianych zagadnień.

Wydaje nam się, że warto za wszelką cenę unikać blokad przechowywanych w pamięci przynajmniej dla aplikacji internetowych i biznesowych, w których niezwykle ważna jest duża skalowalność względem liczby użytkowników. Najczęściej nie zachodzi potrzeba sprawdzania identyczności obiektów wspólnie wątków, bo każdy użytkownik pracuje niezależnie od pozostałych.

To podejście ma wiele zalet, jeśli stosowany system relacyjnej bazy danych implementuje model współbieżności przez wersjonowanie (są to na przykład bazy Oracle i PostgreSQL). Niepożądana okazuje się zmiana znaczenia transakcji lub modelu współbieżności przez bufor trwałości relacyjno-obiektowej.

Raz jeszcze rozważmy dostępne rozwiązania. Bufor o zasięgu transakcji warto stosować, gdy używa się identyczności obiektów na poziomie transakcji. Zapewnia to wysoką wydajność w systemie wieloużytkownikowym. Ten pierwszy poziom buforowania jest obowiązkowy, gdyż zapewnia identyczność obiektów. Można dodatkowo zastosować inny rodzaj buforowania. W niektórych sytuacjach bufor drugiego poziomu o zasięgu procesu (lub klastra) zwracający dane przez wartość okazuje się niezwykle pomocny. W ten sposób powstaje bufor dwupoziomowy. Hibernate korzysta z tego rozwiązania.

Zastanówmy się, jakiego rodzaju dane mogą odnieść korzyści z zastosowania drugiego poziomu buforowania. Innymi słowy, wyjaśnijmy, kiedy warto do obowiązkowego bufora pierwszego poziomu dołączyć opcjonalne buforowanie drugiego poziomu.

Buforowanie i izolacja transakcji

Bufor o zasięgu procesu lub klastra uwiadcznia jednej jednostce zadaniowej dane pobrane przez inną jednostkę. Ma to pewne niemile konsekwencje w kwestii izolacji transakcji.

Po pierwsze, jeśli aplikacja nie ma dostępu do bazy danych na wyłączność, nie należy stosować bufora o zasięgu procesu, chyba że dane zmieniają się bardzo rzadko i mogą zostać bezpiecznie odświeżone po upłynięciu określonego czasu istnienia w buforze. Ten typ danych często występuje w systemach zarządzania treścią, ale rzadko w aplikacjach finansowych.

Zastanówmy się nad dwoma sytuacjami, w których aplikacja może nie mieć dostępu do bazy danych na wyłączność:

- ◆ aplikacje rozproszone (działające w klastrze),
- ◆ współdzielone starsze dane.

Każda aplikacja projektowana z uwzględnieniem dobrej skalowalności musi obsługiwać operacje klastrowe. Bufor o zasięgu procesu nie zapewnia spójności danych przechowywanych w buforach na różnych komputerach wchodzących w skład klastra. W takiej sytuacji warto użyć bufora o zasięgu klastra.

Wiele aplikacji Javy współdzieli dostęp do jednej bazy danych z innymi (najczęściej starszymi) aplikacjami. W takiej sytuacji nie zaleca się stosowania innych buforów poza transakcyjnymi. Bufor nie wie nic o zmianie danych przez starszą aplikację. W zasadzie **istnieje** możliwość implementacji funkcjonalności powodującej odrzucenie przez proces lub klaster niepoprawnych danych, ale nie

powstał żaden standardowy system obsługi tego zadania. Z pewnością funkcja ta nie będzie dostępna w Hibernate. Przy jej implementacji najczęściej jesteś zdany na samego siebie, gdyż jest ona ściśle związana z wykorzystywanym środowiskiem.

Po rozważeniu niewyłącznego dostępu do danych można określić poziom izolacji wymagany przez aplikację. Niezwyczajne ważne okazuje się odkrycie wszystkich wymogów biznesowych w tym względzie, bo nie każda implementacja bufora obsługuje wszystkie poziomy izolacji. Przyjrzymy się rodzajom danych, które mogą skorzystać na wprowadzeniu bufora danych o zasięgu procesu (lub klastra).

Pełny system ORM umożliwia włączenie bufora drugiego poziomu osobno dla każdej klasy. Dobrymi kandydatami do buforowania są dane:

- ◆ rzadko się zmieniające,
- ◆ niekrytyczne (na przykład związane z zarządzaniem treścią),
- ◆ lokalne dla aplikacji (niewspółdzielone z innymi systemami).

Złyimi kandydatami do buforowania dwupoziomowego są dane:

- ◆ często aktualizowane,
- ◆ finansowe,
- ◆ współdzielone ze starszymi aplikacjami.

To nie jedyne współczynniki wpływające na wybór ostatecznego rozwiązania. Wiele aplikacji zawiera klasy o następujących właściwościach:

- ◆ niewielka liczba egzemplarzy,
- ◆ jeden egzemplarz wskazywany przez wiele egzemplarzy innych klas,
- ◆ egzemplarze rzadko (lub w ogóle nie) aktualizowane.

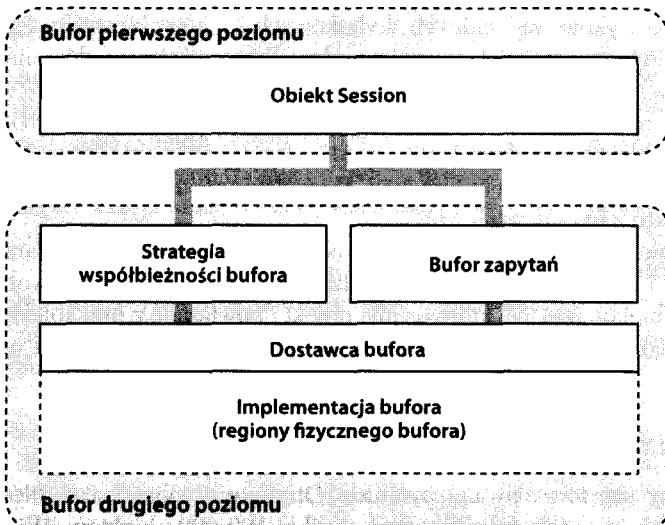
Ten rodzaj danych często nazywa się **danymi referencyjnymi**. Tego rodzaju dane idealnie nadają się do buforowania o zasięgu procesu lub klastra. Aplikacje je stosujące znacznie zwiększą szybkość działania po wprowadzeniu bufora drugiego poziomu. Dane są aktualizowane dopiero po upłynięciu określonego czasu ich przebywania w buforze.

W tym podrozdziale nakreśliliśmy obraz dwupoziomowego systemu buforowania stosującego obowiązkowy bufor o zasięgu transakcji i opcjonalny bufor o zasięgu procesu lub klastra. Okazuje się on w większości zgodny z systemem buforowania Hibernate.

5.3.2. Architektura buforów Hibernate

Jak wspomnieliśmy wcześniej, Hibernate stosuje dwupoziomową architekturę buforującą. Poszczególne elementy systemu przedstawia rysunek 5.5.

Bufor pierwszego poziomu to obiekt Session. Długość życia sesji odpowiada jednej transakcji bazodanowej lub aplikacyjnej (różnicę między nimi wyjaśniliśmy w poprzedniej części rozdziału). Bufor sesji traktuje się jako bufor o zasięgu transakcji. Bufor pierwszego poziomu jest obowiązkowy i nie można go wyłączyć; dodatkowo zapewnia identyczność obiektów wewnętrz transakcji.



Rysunek 5.5. Dwuwarstwowa architektura buforująca w Hibernate

Drugi poziom buforowania w Hibernate jest opcjonalny i może mieć zasięg procesu lub klastra. Stanowi bufor stanu, a nie obiektów trwałych. Strategia współbieżności bufora definiuje szczegóły poziomu izolacji dla konkretnego elementu natomiast dostawca bufora stanowi rzeczywistą, fizyczną implementację bufora. Bufor drugiego poziomu można włączać dla konkretnych klas lub asocjacji.

Dodatkowo Hibernate oferuje bufor wyników zapytań, który integruje się z buforem drugiego poziomu. Element ten jest opcjonalny. Opisujemy go dokładniej w rozdziale 7., ponieważ jego użycie ma ściśły związek z rzeczywistym zapytaniem.

Zaczniemy szczegółowy opis od bufora pierwszego poziomu nazywanego też buforem sesji.

Bufor pierwszego poziomu

Bufor sesji zapewnia, że aplikacja dwukrotnie żądająca tego samego obiektu trwałego w jednej sesji zawsze uzyska ten sam (identyczny) egzemplarz klasy. W niektórych sytuacjach unika się w ten sposób niepotrzebnych zapytań do bazy danych, ale co ważniejsze, zapewnia to, że:

- ◆ Warstwa trwałości nie musi martwić się o przepełnienie bufora z powodu cyklicznych referencji w grafie obiektów.
- ◆ Nigdy nie pojawią się na końcu transakcji konflikty ze sobą reprezentacje tego samego wiersza bazy danych. Za każdym razem istnieje co najwyżej jeden obiekt reprezentujący konkretny wiersz bazy danych. Wszystkie zmiany można bezpiecznie zapisać w bazie danych.
- ◆ Zmiany dokonane w konkretnej jednostce zadaniowej są od razu widoczne dla całego kodu wykonywanego w tej jednostce.

Nie trzeba wykonywać żadnych dodatkowych kroków, by włączyć bufor sesji. Domyślnie jest zawsze włączony i z podanych powyżej powodów nie można go wyłączyć.

Każde przekazanie obiektu do metod `save()`, `update()` lub `saveOrUpdate()` albo pobranie obiektu metodami `load()`, `find()`, `list()`, `iterate()` lub `filter()` powoduje dodanie obiektu do bufora sesji. Wywołanie metody `flush()` synchronizuje stan obiektu z bazą danych.

Gdy nie chcemy przeprowadzać synchronizacji lub gdy przetwarzamy wiele obiektów i chcemy efektywnie zarządzać pamięcią, możemy wywołać metodę `evict()` obiektu `Session`, która usunie obiekt i powiązane z nim kolekcje z bufora pierwszego poziomu. Wspomniana metoda okazuje się przydatna w niejednej sytuacji.

Zarządzanie buforem pierwszego poziomu

Rozważmy często zadawane pytanie: „Otrzymuję wyjątek `OutOfMemoryException` w trakcie próby załadowania i obróbki 100 000 obiektów. W jaki sposób wykonać w Hibernate masową aktualizację?”.

Naszym zdaniem ORM nie nadaje się do wykonywania masowych uaktualnień (lub masowych usunięć) wierszy bazy danych. Gdy w aplikacji pojawia się sytuacja, w której trzeba użyć takiej operacji, prawie zawsze lepiej jest użyć innego podejścia: wywołać procedurę zapamiętaną w bazie danych lub bezpośrednio wykonać polecenie `UPDATE` lub `DELETE` języka SQL. Nie przesyłaj do pamięci operacyjnej wszystkich danych w prostej operacji, którą zdecydowanie wydajniej wykona baza danych. Jeśli aplikacja w większości sytuacji wykonuje operacje masowe, w ogóle nie warto stosować podejścia ORM!

Jeżeli naprawdę chcesz wykonywać operacje masowe w Hibernate, po obsłudze każdego obiektu od razu wywołuj metodę `evict()` (w momencie przechodzenia przez wyniki zapytania), by zapobiec zajęciu całej dostępnej pamięci.

Aby usunąć z bufora sesji wszystkie obiekty, wywołaj metodę `Session.clear()`. Nie zamierzamy przekonywać, że usuwanie obiektów z bufora pierwszego poziomu jest czymś z natury złym, ale w niewielu przypadkach użycia okazuje się dobrą operacją. Czasem użycie projekcji i zapytań raportujących opisanych punkcie 7.4.5 jest lepszym rozwiązaniem.

Zauważ, że usunięcie z bufora, podobnie jak zapis lub zwykłe usunięcie, można stosować dla obiektów powiązanych. Hibernate automatycznie usunie z bufora powiązane obiekty, jeśli atrybut `cascade` odwzorowania zawiera dla danej asocjacji wartość `all` lub `all-delete-orphan`.

Gdy Hibernate nie odnajdzie obiektu w buforze pierwszego poziomu, próbuje uzyskać go z bufora drugiego poziomu, o ile został on włączony dla konkretnej asocjacji.

Bufor drugiego poziomu

Bufor drugiego poziomu ma zasięg procesu lub klastra; wszystkie sesje wspólnie dzielą ten sam dodatkowy bufor. W zasadzie bufor ten ma zasięg równoważny zasięgowi obiektu SessionFactory.

Obiekty trwałe trafiają do bufora drugiego poziomu w postaci **rozłożonej na części**. Przypomina to trochę serializację Javy, choć wykorzystywany algorytm jest znacznie szybszy.

Wewnętrzna implementacja bufora drugiego poziomu nie jest szczególnie interesująca dla programisty aplikacji. Ważniejsze okazuje się poprawne użycie **zasad buforowania** — określenie strategii buforowania i fizycznego dostawcy bufora.

Różne rodzaje danych wymagają różnych zasad buforowania wynikających z: odmiennych współczynników zapisów do odczytów, rozmiarów tabel bazy danych i współdzielenia niektórych tabel z innymi aplikacjami. Bufor drugiego poziomu konfiguruje się z dokładnością do poszczególnych klas lub kolekcji. Nic nie stoi na przeszkodzie, by bufor włączyć dla klas zawierających dane słownikowe i wyłączyć dla klas finansowych. Zasady buforowania określają następujące kwestie:

- ◆ czy bufor drugiego poziomu jest włączony,
- ◆ strategię obsługi współbieżności,
- ◆ zasadę określania daty ważności danych bufora (na podstawie czasu, algorytmu LRU, zmian w pamięci),
- ◆ fizyczny format bufora (pamięć, pliki indeksowane, replikacja w klastrze).

Nie wszystkie klasy mogą skorzystać na buforowaniu, więc niezwykle istotna okazuje się opcja wyłączenia bufora drugiego poziomu. Powtórzmy raz jeszcze: z bufora drugiego poziomu najlepiej skorzystają klasy wielokrotnie odczytywane. Jeśli dane klasy są częściej aktualizowane niż odczytywane, nie warto dla niej włączać bufora, nawet jeśli pozostałe przesłanki wskazują inaczej! Buforowanie drugiego poziomu może okazać się niebezpieczne, jeżeli istnieją inne aplikacje zapisujące do tej samej bazy danych. Innymi słowy, należy bardzo rozważnie podejmować decyzję co do wprowadzenia bufora.

Bufor drugiego poziomu ustawia się w Hibernate w dwóch krokach. Najpierw wskazuje się **strategię współbieżności**. Następnie określa się okres ważności i fizyczne atrybuty bufora, używając **dostawcy bufora**.

Wbudowane strategie współbieżności

Strategia współbieżności to mediator. Odpowiada za zapis i odczyt danych z bufora. Ma duże znaczenie, bo definiuje znaczenie izolacji transakcyjnej dla konkretnego elementu. Należy zdecydować, dla każdej klasy trwałe, jaka strategia powinna być w niej stosowana, jeśli planuje się włączenie bufora drugiego poziomu.

Istnieją cztery wbudowane strategie współbieżności reprezentujące coraz to mniejszy poziom „sztywności” poziomu izolacji:

- ◆ **transactional** — dostępna jedynie w środowisku zarządzanym. Gwarantuje pełną izolację transakcyjną aż do trybu powtarzalnego odczytu (jeśli jest wymagany). Stosuj tę strategię dla głównie odczytywanych danych, jeśli trzeba zapobiec nieaktualnym danym w innych współbieżnych transakcjach, gdyby zdarzyła się aktualizacja.
- ◆ **read-write** — zapewnia poziom izolacji odczytu zatwierdzonego, wykorzystując mechanizm znaczników czasowych. Dostępna jedynie w środowiskach niestosujących klastrów. Stosuj tę strategię dla głównie odczytywanych danych, jeśli trzeba zapobiec nieaktualnym danym w innych współbieżnych transakcjach, gdyby zaszła aktualizacja.
- ◆ **nonstrict-read-write** — nie gwarantuje spójności danych w buforze i bazie danych. Jeśli istnieje duże prawdopodobieństwo współbieżnego dostępu do tej samej encji, ustaw krótki czas ważności bufora. W przeciwnym razie zwiększy się ryzyko odczytu nieaktualnych danych. Tę strategię warto stosować, gdy dane zmieniają się bardzo rzadko (co kilka godzin, dni lub nawet tygodni), a niewielkie prawdopodobieństwo pojawienia się nieaktualnych danych nie stanowi problemu. Hibernate unieważnia zbuforowany element w momencie zapisu obiektu. Jest to jednak operacja asynchroniczna bez żadnych blokad i gwarancji otrzymania przy odczytzie najnowszej wersji danych.
- ◆ **read-only** — strategia współbieżna odpowiednia dla nigdy niezmieniających się danych. Stosuj ją jedynie dla danych słownikowych.

Zmniejszenie sztywności zwiększa wydajność. Szczególnie mocno warto rozważyć wydajność bufora klastrowego z pełnym poziomem izolacji przed wprowadzeniem go do działającego systemu. W wielu sytuacjach lepiej wyłączyć bufor drugiego poziomu dla klasy, jeśli nieaktualne dane nie wchodzą w grę. Warto najpierw sprawdzić wydajność aplikacji bez bufora drugiego poziomu, a dopiero później dla bufora włączonego dla najbardziej odpowiednich klas. Sprawdzanie wydajności po włączeniu bufora dla kolejnych klas pozwala dobrze dostroić aplikację.

Można zaimplementować własną strategię współbieżności, implementując interfejs `net.sf.hibernate.cache.CacheConcurrencyStrategy`, ale jest to zadanie stosunkowo trudne i niezmiernie rzadko przyniesie wzrost wydajności.

Po określaniu strategii współbieżności dla klas kandydujących do buforowania należy wskazać **dostawcę bufora**. Dostawca to dodatkowy moduł fizycznie implementujący system buforujący.

Wybór dostawcy bufora

Hibernate 2.1 wymusza wybór jednego dostawcy bufora dla całej aplikacji. Zawiera wbudowane klasy obsługi następujących dostawców:

- ◆ **EHCache** ma za zadanie oferować prosty bufor o zasięgu procesu w jednej maszynie wirtualnej Javy. Buforowanie może odbywać się w pamięci lub na dysku twardym. Obsługuje opcjonalny bufor wyników zapytań.

- ◆ **OpenSymphony OSCache** to biblioteka obsługująca buforowanie dyskowe i pamięciowe w jednej maszynie wirtualnej. Zawiera bogaty zestaw zasad ważności wpisów i obsługuje bufor wyników zapytań.
- ◆ **SwarmCache** to bufor klastrowy bazujący na JGroups. Używa umiędzniania działającego globalnie dla całego klastra, ale nie obsługuje buforowania wyników zapytań.
- ◆ **JBossCache** to w pełni transakcyjny bufor klastrów replikacyjnych. Także bazuje na bibliotece JGroups. Obsługuje buforowanie zapytań, jeśli zegary komputerów klastra są zsynchronizowane.

Wykonanie adaptera dla innych produktów nie powinno sprawić większych problemów. Wystarczy zaimplementować interfejs `net.sf.hibernate.cache.CacheProvider`.

Nie wszyscy dostawcy bufora mogą być używani ze wszystkimi strategiami współbieżności. Tabela 5.1 przedstawia macierz zgodności pomagającą wybrać odpowiednią kombinację.

Tabela 5.1. Obsługa strategii współbieżności przez różne bufory

Dostawca bufora	read-only	nonstrict-read-write	read-write	transactional
EHCache	X	X	X	
OSCache	X	X	X	
SwarmCache	X	X		
JBossCache	X			X

Ustawienie bufora drugiego poziomu wymaga dwóch kroków:

1. Przejrzienia plików odwzorowań klas trwałych i zdecydowania, które strategie współbieżności byłyby odpowiednie dla klas i ich asocjacji.
2. Włączenia preferowanego dostawcy bufora w globalnej konfiguracji Hibernate i określenia specyficznych dla niego opcji.

Przykładowo, dla bufora OSCache należy określić opcje konfiguracyjne w pliku `oscache.properties`, natomiast dla bufora EHCache trzeba w ścieżce wyszukiwania klas umieścić plik `ehcache.xml`.

Włączmy buforowanie dla klas `Category` i `Item` aplikacji `CaveatEmptor`.

5.3.3. Buforowanie w praktyce

Pamiętaj, że nie musisz jawnie włączać bufora pierwszego poziomu. Zadeklarujmy zasady buforowania i ustawmy dostawców dla bufora drugiego poziomu aplikacji `CaveatEmptor`.

Klasa `Category` ma niewielką liczbę egzemplarzy i nie zmienia się często. Jej obiekty współdzierają wielu użytkowników, więc idealnie nadaje się do buforowania dwupozyciowego. Zaczniemy od dodania do odwzorowania elementu informującego Hibernate o sposobie buforowania obiektów `Category`.

```
<class
    name="Category"
    table="CATEGORY"
    <cache usage="read-write"/>

    <id ...>

</class>
```

Atrybut `usage="read-write"` informuje o potrzebie użycia strategii współbieżności `read-write` dla bufora obiektów `Category`. Hibernate będzie sprawdzał bufor drugiego poziomu przy nawigacji po grafie obiektów (dotyczących `Category`) lub pobieraniu obiektu `Category` na podstawie jego identyfikatora.

Wyбралиśmy strategię `read-write` zamiast `nonstrict-read-write`, gdyż `Category` to klasa o dużej współbieżności związanej z wieloma wykonywanymi jednocześnie transakcjami. Wystarczy dla niej poziom izolacji odczytu zatwierzonego. Wybór `nonstrict-read-write` również nie byłby zły, ponieważ małe nieścisłości w zawartości bazy danych i bufora nie stanowią problemu (hierarchia kategorii nie wpływa na dane finansowe).

Przedstawiony fragment odwzorowania wystarcza, by Hibernate rozpoczęł buforowanie wartości właściwości klasy `Category`, ale nie by pamiętał stan powiązanych encji lub kolekcji. Kolekcje wymagają własnego elementu `<Cache>`. Dla kolekcji `items` ponownie wykorzystamy strategię współbieżności `read-write`.

```
<class
    name="Category"
    table="CATEGORY"
    <cache usage="read-write"/>

    <id ...>

    <set name="items" lazy="true">
        <cache usage="read-write"/>
        <key ...>
    </set>

</class>
```

Bufor zostanie wykorzystany, gdy na przykład wykonamy polecenie `category.getItems().iterate()`.

Aktualnie bufor kolekcji przechowuje jedynie identyfikatory dla powiązanych obiektów `Item`. By same obiekty `Item` również były buforowane, musimy włączyć buforowanie dla klasy `Item`. Tym razem strategia `read-write` również okazuje się najodpowiedniejsza. Użytkownicy nie chcą podejmować decyzji (dodawać obiektów `Bid`) na podstawie potencjalnie nieaktualnych danych. Przejedźmy od razu krok dalej i rozważmy kolekcję obiektów `Bid`. Konkretny obiekt `Bid` w kolekcji jest niezmienny, ale w przedstawionej sytuacji musimy użyć dla kolekcji strategii `read-write`, bo nowe oferty mogą pojawić się w dowolnym momencie (informacja o nich jest niezwykle ważna dla użytkownika systemu).

```
<class
    name="Item"
```

```
table="ITEM"
<cache usage="read-write"/>

<id ...>

<set name="bids" lazy="true">
    <cache usage="read-write"/>
    <key ...>
</set>

</class>
```

Dla niezmiennych obiektów klasy Bid możemy użyć strategii read-only.

```
<class
    name="Bid"
    table="BID"
    <cache usage="read-only"/>

    <id ...>

</class>
```

Zbuforowane dane obiektu Bid są ważne nieskończenie długo, bo ofert się nie aktualizuje. Nie trzeba wskazywać okresu ważności danych w buforze. Obiekty mogą jednak zostać usunięte z bufora, na przykład po przekroczeniu maksymalnej liczby obiektów przechowywanych przez bufor.

Klasa User to przykład klasy, którą można buforować, wykorzystując strategię nonstrict-read-write. Z drugiej strony, czy jest w ogóle sens buforować dane użytkowników?

Po uaktualnieniu odwzorowań ustawmy dostawcę bufora, zasady upływu ważności danych i fizyczne właściwości bufora. Do osobnej konfiguracji klas i kolekcji służą regiony buforowania.

Regiony buforowania

Hibernate przechowuje kolekcje i klasy w osobnych **regionach buforowania**. Region to nazwany bufor — uchwyt pozwalający uzyskać dostęp do kolekcji lub klasy z konkretnej konfiguracji dostawcy bufora i ustawić zasady upływu ważności danych.

W przypadku bufora klas nazwa regionu jest nazwą klasy. Dla kolekcji jest to nazwa klasy połączona z nazwą właściwości. Oznacza to, że obiekty Category znajdują się w regionie org.hibernate.auction.Category, natomiast kolekcja items w regionie org.hibernate.auction.Category.items.

Właściwość konfiguracyjna `Hibernate.hibernate.cache.region_prefix` określa przedrostek nazwy regionu dla konkretnego obiektu SessionFactory. Jeżeli ustawimy przedrostek na node1, klasa Category znajdzie się w regionie node1.org.hibernate.auction.Category. Przedrostek pomaga w organizacji buforów, gdy aplikacja wykorzystuje kilka obiektów SessionFactory.

Po poznaniu regionów buforów ustawmy zasady upływu ważności danych w buforze klasy Category. Najpierw jednak ustalmy dostawcę bufora. Założymy,

że aplikacja aukcyjna korzysta z jednej maszyny wirtualnej i nie musimy stosować implementacji klastrowych (ograniczamy w ten sposób dostępne rozwiązania).

Ustawienie lokalnego dostawcy bufora

Poniższa właściwość informuje Hibernate o dostawcy bufora:

```
hibernate.cache.provider_class=net.sf.ehcache.hibernate.Provider
```

Jako bufor drugiego poziomu wybraliśmy dostawcę EHCache.

Następnie musimy określić zasady upływu ważności danych dla poszczególnych regionów. EHCache stosuje własny plik konfiguracyjny, *ehcache.xml*, który musi znaleźć się w ścieżce wyszukiwania klas. Dystrybucja Hibernate zawiera przykłady plików konfiguracyjnych dla wszystkich wbudowanych dostawców. Zalecamy zapoznanie się z komentarzami zawartymi w tych plikach, by ustawić konkretne opcje dla niektórych klas i domyślne dla pozostałych.

Konfiguracja bufora dla klasy Category w pliku *ehcache.xml* może wyglądać następująco:

```
<cache name="org.hibernate.auction.model.Category"
      maxElementsInMemory="500"
      eternal="true"
      timeToIdleSeconds="0"
      timeToLiveSeconds="0"
      overflowToDisk="false"
/>
```

Istnieje niewielka liczba obiektów Category i są one współdzielone przez wiele współbieżnych transakcji. Z tego powodu wyłączamy upływ ważności, wybierając ograniczenie rozmiaru bufora większe od liczby kategorii i ustawiając atrybut *eternal="true"*. Nie ma potrzeby ograniczania czasu życia danych w buforze, ponieważ klasa Category stosuje strategię read-write i nie ma innych aplikacji zmieniających listę kategorii. Dodatkowo wyłączamy buforowanie na dysku twardym, bo kategorii jest niewiele i umieszczenie ich wszystkich w pamięci nie stanowi problemu.

Obiekty Bid są co prawda niewielkie i niezmienne, ale jest ich dużo. Musimy więc skonfigurować EHCache tak, by rozsądnie gospodarować zajmowaną przez nie w buforze pamięcią. Z tego powodu ustawiamy zarówno maksymalną liczbę przechowywanych obiektów, jak i okres ważności.

```
<cache name="org.hibernate.auction.model.Bid"
      maxElementsInMemory="5000"
      eternal="false"
      timeToIdleSeconds="1800"
      timeToLiveSeconds="100000"
      overflowToDisk="false"
/>
```

Atrybut *timeToIdleSeconds* określa okres ważności w sekundach od momentu ostatniego pobrania obiektu z bufora. Warto dobrze zastanowić się nad wpisywaną wartością, bo nie chcemy, by nieużywane obiekty Bid zajmowały pamięć. Atrybut *timeToLiveSeconds* określa okres ważności w sekundach od momentu umiesz-

czania elementu w buforze. Ponieważ obiekty *Bi&d* się nie zmieniają, nie chcemy, by były usuwane z bufora, gdy aplikacja regularnie z nich korzysta. Z tego powodu ustawiliśmy atrybut na bardzo dużą wartość.

Przedstawiona konfiguracja powoduje usuwanie obiektów *Bi&d* tylko wtedy, gdy nie były używane przez ostatnie 30 minut lub gdy łączna liczba obiektów tego typu przechowywanych w buforze przekroczy 5000 (system usuwa wtedy najdawniej używany obiekt).

W przykładzie wyłączyliśmy buforowanie z użyciem dysku twardego, gdyż założyliśmy wdrożenie aplikacji na tym samym komputerze, na którym znajduje się baza danych. Gdy docelowa architektura jest inna, warto zastanowić się nad włączeniem bufora dyskowego.

Optymalne zasady upływu ważności obiektów silnie zależą od konkretnych danych i rodzaju aplikacji. Trzeba rozważyć wiele zewnętrznych czynników, włączając w to dostępną pamięć operacyjną na serwerze aplikacji, oczekiwane obciążenie bazy danych, opóźnienia przesyłu danych siecią, istnienie zewnętrznych aplikacji itp. Niektórych czynników nie zna się nawet na etapie tworzenia aplikacji, więc niejednokrotnie należy iteracyjnie testować wpływ wydajności poszczególnych rozwiązań na działający system (lub dokonywać symulacji obciążenia).

Przedstawione zagadnienia stają się jeszcze ważniejsze, gdy chce się użyć replikowanego bufora dla kilku komputerów połączonych w jeden klasterek.

Ustawianie bufora replikowanego

EHCache to doskonały dostawca bufora, jeśli aplikacja działa tylko na jednej maszynie wirtualnej. Niestety, aplikacje biznesowe obsługujące jednocześnie kilka tysięcy użytkowników wymagają większej mocy obliczeniowej. W tym przypadku skalowalność rozwiązania waży na jego sukcesie. Aplikacje Hibernate są ze swojej natury skalowalne — to znaczy Hibernate zachowuje się tak samo niezależnie od tego, czy został wdrożony na jednym komputerze czy na kilku. Jedyną funkcją, która należy skonfigurować specjalnie dla operacji klastrowych, jest bufor drugiego poziomu. Zmiany dokonywane z tego powodu w konfiguracji nie są znaczące.

Stosowanie lokalnego bufora drugiego poziomu w systemie klastrowym niekoniecznie jest nieodpowiednie. Niektóre rodzaje danych — szczególnie niezmienne lub odświeżane przez upływ okresu ważności — nie wymagają unieważniania klastrowego i mogą być bezpiecznie buforowane z osobna przez każdy komputer należący do klastra. Wystarczy wtedy uważnie dobrać wartość upływu czasu i włączyć na każdym komputerze klastra lokalny bufor EHCache.

Gdy aplikacja wymaga ścisłej spójności bufora w środowisku klastrowym, trzeba skorzystać z bardziej rozbudowanego dostawcy bufora. Polecamy bufor JBossCache, bo jest w pełni transakcyjnym, przystosowanym do pracy klastrowej systemem buforującym bazującym na bibliotece JGroups. JBossCache zapewnia doskonałą wydajność. Sposób jego komunikacji klastrowej można niemal dowolnie dostosować do potrzeb.

Przejdziemy krok po kroku przez konfigurację JBossCache dla aplikacji CaveatEmptor stosującej klasterek dwuwęzłowy zawierający węzeł A i węzeł B. Niestety, w przykładzie prześledzimy tylko najprostszą sytuację. W rzeczywistości

konfiguracje klastrów są bardzo złożone, a wiele ustawień zależy od konkretnej architektury klastra.

Najpierw musimy sprawdzić, czy wszystkie pliki odwzorowań klas wykorzystują strategię współbieżności read-only lub transactional, bo tylko takie obsługuje JBossCache. Przydatna sztuczka: by uniknąć w przyszłości zastępowania wielu elementów <cache> w plikach odwzorowań, można scentralizować konfigurację strategii współbieżności w pliku *hibernate.cfg.xml*.

```
<hibernate-configuration>
    <session-factory>
        <property .../>
        <mapping .../>

        <class-cache
            class="org.hibernate.auction.model.Item"
            usage="transactional"/>

        <collection-cache
            collection="org.hibernate.auction.model.Item.bids"
            usage="transactional"/>

    </session-factory>
</hibernate-configuration>
```

W przedstawionym przykładzie włączyliśmy buforowanie transakcyjne dla klasy Item i kolekcji bids. Hibernate 2.1 zgłosi konflikt, jeśli poza ustawieniami globalnymi zastosuje się w pliku odwzorowania klasy Item element <cache>. Nie można więc użyć ustawień globalnych do zmiany ustawień w poszczególnych plikach odwzorowań. Zalecamy korzystanie ze scentralizowanego sposobu konfiguracji buforów, szczególnie jeśli nie zna się sposobu wdrożenia aplikacji. Zawsze łatwiej zmienić ustawienia w jednym pliku niż w kilkuset rozsianych w różnych folderach.

Kolejny krok to konfiguracja dostawcy JBossCache. Najpierw włączamy stosowanie tego dostawcy w ustawieniach Hibernate. Gdy ustawienia przechowujemy w pliku *hibernate.xcfg.xml*, wpisujemy poniższy tekst:

```
<property name="cache.provider_class">
    net.sf.hibernate.cache.TreeCacheProvider
</property>
```

JBossCache ma własny plik konfiguracyjny, *treecache.xml*, który musi znaleźć się w ścieżce wyszukiwania klas aplikacji. Najczęściej każdy z węzłów klastra konfiguruje się inaczej, więc trzeba pamiętać, by każdy klaster zawierał w ścieżce wyszukiwania odpowiednią wersję pliku. Przyjrzyjmy się typowemu plikowi konfiguracyjnemu. W klastrze dwuwęzłowym (MyCluster), plik dla klastra A ma następującą postać:

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
    <classpath codebase=".//lib"
               archives="jboss-cache.jar, jgroups.jar"/>
```

```
<!><mbean code="org.jboss.cache.TreeCache"
           name="jboss.cache:service=TreeCache">

    <depends>jboss:service=Naming</depends>
    <depends>jboss:service=TransactionManager</depends>

    <attribute name="ClusterName">MyCluster</attribute>

    <attribute name="CacheMode">REPL_SYNC</attributte>
    <attribute name="SyncReplTimeout">10000</attribute>
    <attribute name="LockAcquisitionTimeout">15000</attribute>
    <attribute name="FetchStateOnStartup">true</attribute>

    <attribute name="EvictionPolicyClass">
        org.jboss.cache.eviction.LRUPolicy
    </attribute>

    <attribute name="EvictionPolicyConfig">
        <config>
            <attribute name="wakeUpIntervalSeconds">5</attribute>
            <!-- Domyślne ustawienia dla wszystkich buforów. -->
            <region name="/_default_">
                <attribute name="maxNodes">5000</attribute>
                <attribute name="timeToIdleSeconds>1000</attribute>
            </region>
            <region name="/org/hibernate/auction/model/Category">
                <attribute name="maxNodes">500</attribute>
                <attribute name="timeToIdleSeconds>5000</attribute>
            </region>
            <region name="/org/hibernate/auction/model/Bid">
                <attribute name="maxNodes">5000</attribute>
                <attribute name="timeToIdleSeconds>1800</attribute>
            </region>
        </config>
    </attribute>

    <attribute name="ClusterConfig">
        <config>
            <UPD bind_addr="192.168.0.1"
                  ip_mcast="true"
                  loopback="false"/>
            <PING timeout="2000"
                  num_initial_members="3"
                  up_thread="false"
                  down_thread="false"/>
            <FD_SOCK/>
            <pbcast.NAKACK gc_lag="50"
                  retransmit_timeout="600,1200,2400,4800"
                  max_xmit_size="8192"
                  up_thread="false" down_thread="false"/>
            <UNICAST timeout="600,1200,2400"
                  window_size="100"
                  min_threshold="10"
                  down_thread="false"/>
        </config>
    </attribute>
```

```

<pbcast.STABLE desired_avg_gossip="20000"
                  up_thread="false"
                  down_thread="false"/>

<FRAG frag_size="8192"
      up_thread="false"
      down_thread="false"/>

<pbcast.GMS join_timeout="5000"
               join_retry_timeout="2000"
               shun="true" print_local_addr="true"/>

<pbcast.STATE_TRANSFER up_thread="true"
                       down_thread="true"/>

</config>
</attribute>

</mbean>
</server>

```

Z pewnością za pierwszym razem przedstawiony plik potrafi przerazić każdego. Na szczęście jego zrozumienie nie jest trudne. Warto zdawać sobie sprawę, że nie jest to tylko plik konfiguracyjny dla JBossCache, ale również: konfiguracja usługi JMX, plik konfiguracyjny dla TreeCache i szczegółowa konfiguracja dla biblioteki komunikacyjnej JGroups.

Pomińmy kilka pierwszych wierszy związanych z wdrożeniem w systemie JBoss (dane tę zostaną użyte tylko w momencie uruchomienia JBossCache wraz z serwerem aplikacji JBoss) i przejdźmy od razu do konfiguracji TreeCache. Ustawienia definiują bufor replikowany używający komunikacji **synchronizowanej**. Oznacza to, że węzeł wysyłający komunikat czeka, aż wszystkie węzły grupy potwierdzą jego otrzymanie. To dobry wybór dla rzeczywiście replikowanego bufora. Asynchroniczna, nieblokująca komunikacja byłaby bardziej odpowiednia, gdyby węzeł B był **węzłem zapasowym**, który natychmiast podejmie działania węzła A, gdy ten przestanie działać, a nie węzłem partnerskim. Węzły zapasowe stosuje się wtedy, gdy głównym zadaniem klastra jest zwiększenie niezawodności zamiast zwiększenia przepustowości. Pozostałe ustawienia są łatwe w interpretacji — określają okres ważności i możliwą liczbę danych w buforach związanych z poszczególnymi klasami.

JBossCache korzysta z modułowych zasad usuwania danych. Wybraliśmy wbudowaną zasadę o nazwie org.jboss.cache.eviction.LRUPolicy, a następnie skonfigurowaliśmy opcje usuwania dla każdego z regionów w podobny sposób, jak miało to miejsce w EHCache.

No końcu przyjrzyjmy się konfiguracji komunikacji międzyklastrowej dla JGroups. Kolejność protokołów jest niezwykle ważna, więc nie należy dowolnie zmieniać ani dodawać żadnych wierszy. Najbardziej interesuje nas protokół UDP. Włączamy dowiązanie do gniazda komunikacyjnego na interfejsie 192.168.0.1 (adres IP węzła A klastra) i uaktywniamy komunikację do wielu odbiorców. Atrybut loopback należy ustawić na wartość true, jeśli węzeł A wykorzystuje system operacyjny Microsoft Windows.

Inne atrybuty są bardziej złożone i ich dokładnego wyjaśnienia warto szukać w dokumentacji JGroup. Dotyczą one przede wszystkim algorytmów odnajdowania nowych węzłów grupy, wykrywania wyłączenia lub błędu węzła i ogólnego zarządzania komunikacją.

Po zmianie strategii współbieżności klas trwałych na transactional (lub read-only) oraz utworzeniu pliku *treecache.xml* dla węzła A możemy przystąpić do uruchomienia aplikacji i sprawdzenia dziennika zdarzeń. Zalecamy włączenie tworzenia dziennika na poziomie DEBUG dla klasy org.jboss.cache. Zauważ wczytanie konfiguracji i rozpoznanie węzła A jako pierwszego węzła klastra. Aby uaktywnić węzeł B, zmień adres IP w konfiguracji i powtórz proces wdrożenia. Zaraz po uruchomieniu aplikacji na drugim komputerze dziennik powinien zarejestrować włączenie drugiego węzła. Od tego momentu aplikacja Hibernate wykorzystuje w pełni transakcyjny bufor klastrowy: każdy element dodany do klastra zostanie poddany replikacji, natomiast uaktualnione elementy będą usuwane.

Warto rozważyć włączenie dodatkowej opcji. Dla dostawców buforów klastrowych warto w konfiguracji Hibetate opcję `hibernate.cache.use_minimal_puts` ustawić na wartość true. Po jej uaktywnieniu Hibernate doda element do bufora dopiero po sprawdzeniu, czy się tam nie znajduje. Ta strategia daje lepszą wydajność, jeśli operacja odczytu z bufora jest znacznie mniej kosztowna od operacji zapisu. Sytuacja tego typu występuje w buforze replikowanym, ale nie w buforze lokalnym, więc domyślnie jest ustawiona na false (by optymalizować buforowanie lokalne). Niezależnie od tego, czy używa się bufora lokalnego czy klastrowego, niejednokrotnie zachodzi potrzeba sterowania nim w sposób programowy, na przykład w celu przeprowadzenia testów lub dostrojenia parametrów.

Sterowanie buforem drugiego poziomu

Hibernate zawiera użyteczne metody pomagające testować i dostrajać bufor. Może się zastanawiasz, jak całkowicie wyłączyć buforowanie drugiego poziomu. Hibernate uaktywni bufor i załaduje dostawców tylko wtedy, gdy w odzworzaniach klas w pliku XML pojawią się deklaracje buforów. Ich usunięcie wyłącza buforowanie. To kolejny powód, by wszystkie ustawienia buforowania przechowywać w głównym pliku konfiguracyjnym *hibernate.cfg.xml*.

Podobnie jak obiekt Session zawiera metody sterujące buforem pierwszego poziomu, tak obiekt SessionFactory zawiera metody sterowania buforem drugiego poziomu.

Wywołanie metody `evict()` usuwa element z bufora. Jako argument podaje się klasę i wartość identyfikatora obiektu.

```
SessionFactory.evict(Category.class, new Long(123));
```

Można również usunąć z bufora wszystkie elementy wskazanej klasy lub kolekcji:

```
SessionFactory.evict("org.hibernate.auction.model.Category");
```

Przedstawiony mechanizm oczyszczania bufora przydaje się niezmiernie rzadko.

5.4. Podsumowanie

Niniejszy rozdział dotyczył sterowania współbieżnością i buforowaniem danych.

Wskazaliśmy, że w jednej jednostce zadaniowej wszystkie operacje muszą się albo powieść, albo cała jednostka nie zostanie wykonana poprawnie (wtedy zmiany wykonane w trwałym należą wycofać). W ten sposób wspomnialiśmy o ogólnych zasadach rządzących transakcjami i zasadzie ACID. Transakcja jest niepodzielna, pozostawia dane w spójnym stanie, działa niezależnie od pozostałych transakcji i zapewnia trwałość zatwierdzonych zmian.

W aplikacjach Hibernate wyróżnić można dwa rodzaje transakcji: krótkie transakcje bazodanowe i długie transakcje aplikacyjne. Najczęściej jako poziom izolacji transakcji bazodanowych wykorzystuje się odczyt zatwierdzony, natomiast implementację transakcji aplikacyjnych zapewnia optymistyczne sterowanie współbieżnością (sprawdzanie wersji lub znaczników czasowych).

Na końcu zajęliśmy się niezwykle istotnym tematem buforowania i jego wykorzystaniem w celu zapewnienia wydajnego działania aplikacji.

Hibernate umożliwia zastosowanie podwójnego systemu buforowania: pierwszy poziom stanowi bufor obiektów (zawarty w obiekcie Session), natomiast drugi poziom buforuje dane. Pierwszy poziom działa zawsze — służy rozwiązywaniu referencji cyklicznych w grafie obiektów i optymalizuje wydajność pojedynczej jednostki zadaniowej. Bufor drugiego poziomu dotyczący procesu lub klastra jest w pełni opcjonalny i najlepiej działa dla klas z obiektami głównie odczytywanymi. Nic nie stoi na przeszkodzie, by włączyć nieulotny bufor drugiego poziomu dla danych słownikowych lub bufor z. w pełni transakcyjnym buforem dla krytycznych danych. Zawsze warto się zastanowić, czy wzrost złożoności rozwiązania przełoży się na większą wydajność. Bufor drugiego poziomu można włączać osobno dla każdej klasy, a nawet dla jej asocjacji i kolekcji. Dobrze zestrojony i dokładnie przetestowany bufor danych z Hibernate uzyskuje wydajność nieosiągalną w większości ręcznie tworzonych warstw dostępu do danych.

Zaawansowane zagadnienia odwzorowań

W rozdziale:

- ◆ System typów Hibernate
- ◆ Własne typy odwzorowań
- ◆ Odwzorowania kolekcji
- ◆ Asocjacje jeden-do-jednego i wiele-do-wielu

W rozdziale 3. omówiliśmy najważniejsze funkcje systemu ORM udostępniane przez Hibernate. Zajęliśmy się wtedy podstawowymi odwzorowaniami klas i właściwości, odwzorowaniem dziedziczenia i komponentu oraz odwzorowaniem asocjacji jeden-do-wielu. W niniejszym rozdziale kontynuujemy poprzedni temat, skupiając się bardziej wyrafinowanych odwzorowaniach kolekcji i asocjacji. W niektórych miejscach ostrzegamy przed zbyt pochopnym użyciem pewnych funkcji. W zasadzie dowolny model dziedzinowy można zaimplementować, używając wyłącznie odwzorowań komponentów i asocjacji jeden-do-wielu (oraz okazjonalnie asocjacji jeden-do-jednego). Do bardziej rozbudowanych odwzorowań warto podchodzić ostrożnie lub nawet ich unikać.

Zanim zaczniemy opis egzotycznych odwzorowań, musimy nieco dokładniej przyjrzeć się systemowi typów Hibernate — w szczególności rozróżnieniu między encjami i typami wartości.

6.1. System typów Hibernate

W punkcie 3.5.1 po raz pierwszy wprowadziliśmy rozróżnienie między encją i typem wartości — podstawowymi elementami systemów ORM w języku Java. Musimy uszczegółowić to zagadnienie, by dobrze zrozumieć stosowane w Hibernate typy odwzorowań, typy wartości i encje.

Encje to ogólne klasy w systemie. Główne dane systemu najczęściej określa się, wyodrębniając elementy biorące udział w wykonywaniu czynności: „użytkownik składa ofertę na przedmiot” to typowe zdanie definiujące trzy encje. Klasy dotyczące typów wartości najczęściej nie pojawiają się w wymaganiach biznesowych — są szczegółowymi klasami reprezentującymi teksty, liczby i kwoty. Zdarzają się jednak sytuacje, w których i klasy typów wartości występują w definicjach — „użytkownik zmienia adres płatności” — niemniej należą one do rzadkości.

Bardziej formalna definicja jest następująca: encja to dowolna klasa, której egzemplarze mają własną tożsamość bazowaną. Typ wartości to klasa, która nie definiuje własnej tożsamości. W praktyce oznacza to, że encje zawierają klucz główny (właściwość identyfikującą), natomiast typy wartości stanowią element encji.

W trakcie pracy systemu w pamięci istnieje graf obiektów zawierający przemieszane obiekty encji i typów wartości. Obiekty encji mogą znajdować się w jednym z trzech stanów: ulotnym, trwałym lub odłączonym. Przedstawionych określeń stanów nie stosuje się dla typów wartości.

Ponieważ tylko encje mają własny cykl życia, metody `save()` i `delete()` interfejsu Session zawsze dotyczą klas encyjnych, a nigdy klas typów wartości. Cykl życia dotyczący trwałości obiektu typu wartości zawsze jest w pełni zależny od cyklu życia zawierającej go encji. Przykładowo, nazwa użytkownika staje się trwała w momencie zapisania obiektu użytkownika; nigdy nie utrwała się jej niezależnie od użytkownika.

W Hibernate typy wartości mogą definiować asocjacje. Nic nie stoi na przeszkodzie, by przejść od obiektu typu wartości do innej encji, ale nigdy nie może zajść sytuacja odwrotna, czyli przejście od innej encji do obiektu typu wartości. Asocjacje zawsze wskazują na encje. Można stąd wysnuć wniosek, że obiekt typu wartości zawsze należy do jednej i tylko jednej encji — nigdy nie jest współdzielony.

Na poziomie bazy danych każda tabela zawiera encje. Hibernate dopuszcza rozwijania, w których przed kodem Javy ukrywa się istnienie dodatkowych encji na poziomie bazy danych. Na przykład asocjacja wiele-do-wielu ukrywa przed aplikacją istnienie tabeli łączącej. Kolekcja tekstów (a dokładniej kolekcja obiektów typów wartości) zachowuje się jak typ wartości z punktu widzenia aplikacji, choć tak naprawdę wykorzystuje własną tabelę. Choć ta funkcjonalność początkowo wydaje się bardzo kusząca, bo upraszcza kod Javy, z czasem przestaliśmy jej ufać. Bardzo często te dodatkowe tabele i tak trzeba udostępnić użytkownikom po rozbudowie wymagań biznesowych. Tabele łączące w asocjacjach wiele-do-wielu niejednokrotnie rozbudowuje się o dodatkowe kolumny wraz z rozwojem aplikacji. Jesteśmy niemal przekonani, by zalecać udostępnianie aplikacji wszystkich tabel bazy danych jako klas encyjnych. Nic nie stoi wtedy na przeszkodzie, by asocjację wiele-do-wielu zaimplementować jako dwie asocjacje jeden-do-wielu do klasy odwzorowującej tabelę łączającą. Decyzję co do konkretnej implementacji pozostawiamy Czytelnikowi, choć do tematu asocjacji wiele-do-wielu wróćmy jeszcze w dalszej części rozdziału.

Klasy encyjne zawsze dotyczą tabeli bazy danych określonej w elementach odwzorowujących <class>, <subclass> lub <joined-subclass>. W jaki sposób odwzorowuje się typy wartości?

Rozważmy odwzorowanie w klasie User dotyczące adresu e-mail.

```
<property  
    name="email"  
    column="EMAIL"  
    type="string"/>
```

Skupmy się na atrybucie type="string". W ORM trzeba odpowiednio połączyć ze sobą typy SQL i typy Javy. Zadaniem tym zajmują się typy odwzorowań Hibernate. Nazwa string stanowi jeden z wbudowanych w Hibernate typów odwzorowań.

Typ string nie jest jedynym wbudowanym w Hibernate. Omawiany ORM zawiera różne typy odwzorowań dla typów podstawowych i niektórych klas JDK.

6.1.1. Wbudowane typy odwzorowań

Wbudowane typy odwzorowań najczęściej mają tę samą nazwę co odwzorowywany typ Javy. Dla niektórych typów Javy istnieje więcej niż jeden typ odwzorowania. Wbudowanych typów nie można stosować do wykonywania dowolnych konwersji, na przykład odwzorowania wartości pola typu VARCHAR na właściwości typu Integer. Do wykonania tego rodzaju zadania należy zdefiniować własny typ wartości, o czym będzie mowa w dalszej części rozdziału.

Zajmiemy się przedstawieniem wbudowanych typów dotyczących typów prostych Javy, daty i czasu, dużych obiektów oraz wskazaniem, jakie typy Javy i SQL obsługują.

Odwzorowanie typów prostych Javy

Podstawowe typy odwzorowań z tabeli 6.1 dotyczą typów prostych Javy (lub ich klasowych otoczek) odwzorowywanych na standardowe typy SQL.

Tabela 6.1. Typy proste

Typ odwzorowania	Typ Javy	Standardowy typ SQL
integer	int lub <code>java.lang.Integer</code>	INTEGER
long	<code>long</code> lub <code>java.lang.Long</code>	BIGINT
short	<code>short</code> lub <code>java.lang.Short</code>	SMALLINT
float	<code>float</code> lub <code>java.lang.Float</code>	FLOAT
double	<code>double</code> lub <code>java.lang.Double</code>	DOUBLE
big_decimal	<code>java.math.BigDecimal</code>	NUMERIC
character	<code>java.lang.String</code>	CHAR(1)
string	<code>java.lang.String</code>	VARCHAR
byte	<code>byte</code> lub <code>java.lang.Byte</code>	TINYINT
boolean	<code>boolean</code> lub <code>java.lang.Boolean</code>	BIT
yes_no	<code>boolean</code> lub <code>java.lang.Boolean</code>	CHAR(1) ('Y' lub 'N')
true_false	<code>boolean</code> lub <code>java.lang.Boolean</code>	CHAR(1) ('T' lub 'F')

Nie wszystkie bazy danych obsługują wszystkie wymienione w tabeli 6.1 typy SQL. Wymienione typy są typami określonymi w standardzie ANSI-SQL. Większość twórców baz danych ignoruje część standardu, bo wprowadzili własne typy przed jego pojawienniem. Sterownik JDBC stosuje ograniczoną abstrakcję typów danych SQL, co umożliwia Hibernate korzystanie ze standardu SQL w momencie wykonywania poleceń języka DML (*Data Manipulation Language*). W przypadku generowania kodu DDL specyficznego dla konkretnej bazy danych sam dokonuje konwersji typu ANSI na typ twórcy systemu bazodanowego, wykorzystując klasy dialektów SQL. Z tego powodu najczęściej nie trzeba w ogóle przejmować się podstawowymi typami danych, gdy to Hibernate ma wygenerować kod tworzący schemat bazy danych.

Typy odwzorowujące datę i czas

Tabela 6.2 wymienia typy Hibernate dotyczące daty, czasu i znaczników czasowych. W modelu dziedzinowym można reprezentować datę i czas, używając klas: `java.util.Date`, `java.util.Calendar` lub podklas `java.util.Date` zdefiniowanych w pakiecie `java.sql`. Wybór klasy zależy od własnych preferencji. Niczego nie narzucamy. Warto jednak pozostać konsekwentnym w swym wyborze.

Tabela 6.2. Typy daty i czasu

Typ odwzorowania	Typ Javy	Standardowy typ SQL
date	java.util.Date lub java.sql.Date	DATE
time	java.util.Date lub java.sql.Time	TIME
timestamp	java.util.Date lub java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

Typy odwzorowań dużych obiektów

Tabela 6.3 przedstawia typy Hibernate związane z obsługą danych binarnych i dużych obiektów. Żadnego z tych typów nie można użyć jako typu dla właściwości identyfikującej.

Tabela 6.3. Typy binarne i dużych obiektów

Typ odwzorowania	Typ Javy	Standardowy typ SQL
binary	byte[]	VARBINARY (lub BLOB)
text	java.lang.String	CLOB
Serializable	dowolna klasa Javy implementująca interfejs java.io.Serializable	VARBINARY (lub BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

Klasy `java.sql.Blob` i `java.sql.Clob` to najwydajniejszy sposób obsługi w Javie dużych obiektów. Niestety, ich egzemplarze można stosować tylko do momentu zakończenia transakcji JDBC. Jeśli klasa trwała definiuje właściwość `java.sql.Blob` lub `java.sql.Clob` (co nie jest dobrym pomysłem), zdecydowanie zmniejsza się pole zastosowań klasy. Przede wszystkim nie można stosować egzemplarzy takiej klasy jako obiektów odłączonych. Co więcej, nie wszystkie sterowniki JDBC obsługują typów `java.sql.Blob` i `java.sql.Clob`. Z tego powodu większy sens ma zastosowanie typów odwzorowań `binary` lub `text`, jeśli pobieranie dużych obiektów do pamięci nie zaszkodzi znacząco wydajności aplikacji.

Najnowsze wskazówki dotyczące korzystania z dużych obiektów w połączeniu z konkretnym systemem bazodanowym znajdują się w witrynie Hibernate.

Typy odwzorowujące klasy JDK

Tabela 6.4 przedstawia kilka typów Javy z klas JDK, które w bazie danych można odwzorowywać jako typ `VARCHAR`.

Element `<property>` nie jest jedynym elementem zawierającym atrybut `type`.

Tabela 6.4. Inne typy dotyczące klas JDK

Typ odwzorowania	Typ Javy	Standardowy typ SQL
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

6.1.2. Zastosowania typów odwzorowań

Wszystkie podstawowe typy odwzorowań Hibernate mogą pojawić się w dowolnym miejscu dokumentu odwzorowania: w opisie zwykłej właściwości, właściwości identyfikującej lub innym elemencie odwzorowania.

Atrybut name pojawia się w elementach <id>, <property>, <version>, <discriminator>, <index> i <element>. Istnieją pewne ograniczenia co do typów, które mogą wystąpić w elementach dotyczących identyfikatorów i deskryptorów.

O przydatności wbudowanych typów odwzorowań świadczyć może poniższe odwzorowanie klasy BillingDetails.

```
<class name="BillingDetails"
      table="BILLING_DETAILS"
      discriminator-value="null">

    <id name="id" type="long" column="BILLING_DETAILS_ID">
        <generator class="native"/>
    </id>

    <discriminator type="character" column="TYPE"/>

    <property name="number" type="string"/>
    ...
</class>
```

Klasę BillingDetails odwzorowujemy jako encję. Właściwości identyfikatora, dyskryminatora i liczby są typami wartości stosującymi wbudowane typy odwzorowań do przeprowadzania konwersji.

Nie zawsze trzeba w dokumencie odwzorowania XML jawnie określić wbudowany typ odwzorowania. Jeżeli odwzorowywana właściwość jest typem Javy java.lang.String, Hibernate skorzysta z interfejsu refleksji i automatycznie zastosuje typ string. Poprzedni dokument odwzorowania można uproszczyć:

```
<class name="BillingDetails"
      table="BILLING_DETAILS"
      discriminator-value="null">

    <id name="id" column="BILLING_DETAILS_ID">
        <generator class="native"/>
    </id>

    <discriminator type="character" column="TYPE"/>
```

```
<property name="number"/>  
...  
</class>
```

Przedstawione zgadywanie typu nie działa najlepiej dla typu Javy `java.util.Date`, ponieważ domyślnie Hibernate stosuje dla klasy `Date` odwzorowanie `timestamp`. Jeśli nie jest potrzebne przechowywanie zarówno daty, jak i czasu, trzeba ręcznie określić typ odwzorowania: `type="date"` lub `type="time"`.

Dla każdego wbudowanego typu odwzorowania zdefiniowano w klasie `net.sf.hibernate.Hibernate` stałą `Stała.Hibernate.STRING` reprezentującą typ odwzorowania `String`. Stałych tych używa się na przykład w dowiązywaniu parametrów zapytania, co zostanie dokładniej omówione w rozdziale 7.

```
session.createQuery("from Item i where i.description like :desc")  
.setParameter("desc", desc, Hibernate.STRING)  
.list();
```

Stałe przydają się również do programowej zmiany metamodelu odwzorowania Hibernate w sposób przedstawiony w rozdziale 3.

Hibernate nie ogranicza się do wbudowanych typów odwzorowań. To właśnie możliwość dowolnego rozszerzania systemu typów odwzorowań czyni Hibernate tak elastycznym.

Tworzenie własnych typów odwzorowań

Języki obiektowe, w tym Java, ułatwiają kreowanie nowych typów — wystarczy utworzyć klasę. Jest to jedna z podstawowych cech obiektowości. Gdyby Hibernate pozwalał wykorzystywać jedynie wbudowane typy odwzorowań w momencie deklarowania klas trwałości, traciłoby się wiele z siły wyrazu Javy. Co więcej, implementacja modelu dziedzinowego byłaby ściśle związana z fizycznym modelem danych, bo konwersja nowych typów byłaby niemożliwa.

Większość rozwiązań ORM dopuszcza określanie przez użytkownika własnych strategii konwersji typów. Mówiąc wtedy o konwerterach. Przykładowo, programista może wprowadzić nowy konwerter, który umożliwi zapamiętanie wartości typu `Integer` w kolumnie typu `VARCHAR`. Hibernate jest znacznie bardziej elastyczny w określaniu konwerterów i używa dla nich nazwy **własne typy odwzorowań**.

Hibernate udostępnia dwa przyjazne dla użytkownika interfejsy do definiowania nowych typów odwzorowań. Interfejsy te redukują nakład pracy związane z kreowaniem własnych typów i chronią typy użytkownika przed ewentualnymi zmianami w rdzeniu systemu ORM. Dzięki temu aktualizacja Hibernate nie pociąga za sobą konieczności wprowadzania zmian w typach odwzorowań. Wiele przykładów użytecznych typów odwzorowań znajduje się w witrynie społeczności użytkowników Hibernate.

Pierwszy z interfejsów programistycznych to `net.sf.hibernate.UserType`. Doskonale nadaje się dla prostych typów odwzorowań i rozwiązywania kilku bardziej złożonych problemów. Przedstawimy jego użycie na przykładzie.

Klasa Bid definiuje właściwość amount; klasa Item definiuje właściwość initialPrice. Obie właściwości są typu walutowego. Do tej pory do reprezentacji tych wartości używaliśmy typu BigDecimal odwzorowanego na typ big_decimal z Hibernate, a dalej na kolumnę typu NUMERIC.

Przypuśćmy, że aplikacja aukcyjna ma obsługiwać wiele walut i w tym celu trzeba zmienić istniejący model dziedzinowy. Zmianę można zaimplementować, dodając nową właściwość do klas Bid i Item — odpowiednio amountCurrency i initialPriceCurrency. Następnie wystarczyłoby odwzorować nowe właściwości na kolumny typu VARCHAR, używając wbudowanego typu odwzorowania currency. Mamy nadzieję, że nigdy nie użyjesz wspomnianej implementacji!

Tworzenie klasy typu użytkownika

Zamiast wspomnianego wcześniej podejścia powinniśmy utworzyć klasę MonetaryAmount zawierającą zarówno informację o kwocie, jak i o walucie. Zauważ, że jest to klasa modelu dziedzinowego — nie jest w żaden sposób powiązana z interfejsem Hibernate.

```
public class MonetaryAmount implements Serializable {
    private final BigDecimal value;
    private final Currency currency;

    public MonetaryAmount(BigDecimal value, Currency currency) {
        this.value = value;
        this.currency = currency;
    }

    public BigDecimal getValue() { return value; }

    public Currency getCurrency() { return currency; }

    public boolean equals(Object o) { ... }
    public int hashCode() { ... }
}
```

Klasa MonetaryAmount jest niezmienna. Warto stosować to podejście w Javie. By poprawnie zamknąć kod klasy, musimy zaimplementować metody equals() i hashCode(). Klasa posłuży do zastąpienia właściwości initialPrice typu BigDecimal w klasie Item. Oczywiście powinniśmy ją zastosować we wszystkich miejscach, w których do zapamiętania ceny użyliśmy wcześniej klasy BigDecimal (na przykład w Bid.amount), a nawet w logice biznesowej (systemie płatności).

Odwzorujmy zmienioną właściwość klasę Item na kolumny bazy danych. Przypuśćmy, że korzystamy ze starej bazy danych zawierającej wszystkie płatności wykonywane w dolarach. Nowa aplikacja obsługuje będzie dowolną liczbę walut (jest to główny cel zmian), ale modyfikacje zawartości bazy danych zajmą administratorom aplikacji trochę czasu. Musimy więc konwertować kwoty na dolary w momencie utrwalania danych obiektu klasy MonetaryAmount i konwertować je z powrotem w momencie wczytywania obiektów.

W tym celu wykonamy klasę MonetaryAmountUserType implementującą interfejs UserType zdefiniowany przez Hibernate. Własny typ odwzorowania przedstawia listing 6.1.

Listing 6.1. Własny typ odwzorowania dla kwot podawanych w dolarach amerykańskich

```
package auction.customtypes;

import ...;

public class MonetaryAmountUserType implements UserType {

    private static final int[] SQL_TYPES = {Types.NUMERIC};

    public int[] sqlTypes() { return SQL_TYPES; } ❶

    public Class returnedClass() { return MonetaryAmount.class; } ❷

    public boolean equals(Object x, Object y) { ❸
        if (x == y) return true;
        if (x == null || y == null) return false;
        return x.equals(y);
    }

    public Object deepCopy(Object value) { return value; } ❹

    public boolean isMutable() { return false; } ❺

    public Object nullSafeGet(ResultSet resultSet, ❻
                             String[] names,
                             Object owner)
        throws HibernateException, SQLException {
        if (resultSet.wasNull()) return null;
        BigDecimal valueInUSD = resultSet.getBigDecimal(names[0]);

        return new MonetaryAmount(valueInUSD, Currency.getInstance("USD"));
    }

    public void nullSafeSet(PreparedStatement statement, ❼
                           Object value,
                           int index)
        throws HibernateException, SQLException {
        if (value == null) {
            statement.setNull(index, Types.NUMERIC);
        } else {
            MonetaryAmount anyCurrency = (MonetaryAmount)value;
            MonetaryAmount amountInUSD =
                MonetaryAmount.convert( anyCurrency,
                                         Currency.getInstance("USD") );
            // Metody convert() nie przedstawiamy w przykładach.
            statement.setBigDecimal(index, amountInUSD.getValue());
        }
    }
}
```

- ❶ Metoda `sqlTypes()` informuje Hibernate, jakiego rodzaju typ kolumny SQL należy zastosować w momencie generowania schematu bazy. Kody typów definiuje klasa `java.sql.Types`. Zauważ, że metoda zwraca tablicę typów. Typ użytkownika może odwzorowywać jedną właściwość na kilka kolumn, choć nasz starszy model danych wykorzystuje tylko jedną kolumnę typu `NUMERIC`.
- ❷ Metoda `returnedClass()` informuje o typie Javy zwracanym przez odwzorowanie tworzone przez użytkownika.
- ❸ Typ odwzorowania odpowiada za wykrywanie zmian w wartościach właściwości. Metoda `equals()` porównuje aktualną wartość właściwości z wartością wcześniejszą, by stwierdzić, czy konieczny jest zapis wartości do bazy danych.
- ❹ Typ odwzorowania częściowo odpowiada za tworzenie wcześniejszej wersji danych. Ponieważ klasa `MonetaryAmount` jest niezmienna, metoda `deepCopy()` zwraca przekazany do niej argument. W przypadku klasy zmiennej należałoby w metodzie wykonać kopię obiektu przekazanego jako argument. System ORM wywołuje tę metodę również w momencie zapisu lub odczytu danych z bufora drugiego poziomu.
- ❺ Hibernate potrafi w pewnym stopniu zoptymalizować typy niezmienne. Metoda `isMutable()` informuje system, czy obiekty danego typu są niezmienne.
- ❻ Metoda `nullSafeGet()` pobiera wartość właściwości z obiektu `ResultSet` połączenia JDBC. W przypadku konwersji walut przydatny może okazać się również dostęp do obiektu właściciela (owner) komponentu. Ponieważ wszystkie kwoty w bazie danych są podane w dolarach, musimy skonwertować wartość zwracaną przez `MonetaryAmount` przed wyświetleniem użytkownikowi danych w odpowiedniej walucie.
- ❼ Metoda `nullSafeSet()` zapisuje wartość właściwości do obiektu `PreparedStatement` połączenia JDBC. Metoda konwertuje kwotę w aktualnej walucie na kwotę w dolarach przed jej zapisem jako wartości typu `BigDecimal`.

Odwzorujmy właściwość `initialPrice` klasy `Item` w następujący sposób:

```
<property name="initialPrice"
          column="INITIAL_PRICE"
          type="auction.customtypes.MonetaryAmountUserType"/>
```

To najprostszy rodzaj przekształcenia dostępny za pomocą interfejsu `UserType`. Możliwe są bardziej wyrafinowane działania. Własny typ odwzorowania może walidować dane, odczytywać je lub zapisywać z katalogu LDAP, a nawet pobierać obiekty trwałe z innego obiektu `Session` dotyczącego zupełnie innej bazy danych. Ograniczeniem jest tylko pomysłowość!

Zamiast przedstawionego powyżej rozwiązania w momencie kreowania nowej bazy lub łatwości aktualizacji istniejącej lepiej byłoby użyć osobnych kolumn dla kwoty i waluty. Nadal moglibyśmy korzystać z interfejsu `UserType`, ale nie pozwa-

lałby on na stosowanie zakresów kwot lub walut w zapytaniach. System zapytań Hibernate (opisywany dokładniej w kolejnym rozdziale) nie wie nic na temat poszczególnych właściwości obiektu MonetaryAmount. Nic nie stoi na przeszkodzie, by pobierać i ustawiać właściwości w kodzie Javy (MonetaryAmount jest zwykłą klasą modelu dziedzinowego). Niestety, ta elastyczność nie istnieje w zapytaniach Hibernate.

Aby własny typ mógł być w pełni wykorzystany w zapytaniach Hibernate, musimy użyć interfejsu CompositeUserType. Interfejs jest bardziej złożony od poprzedniego, ale udostępnia Hibernate bogatszą wiedzę na temat właściwości klasy MonetaryAmount.

Tworzenie klasy z interfejsem `Composite UserType`

Aby zilustrować elastyczność własnych typów odwzorowań, nie zmienimy klasy MonetaryAmount (ani innej klasy trwalej). Modyfikacje dotyczyć będą wyłącznie typu odwzorowania (patrz listing 6.2).

Listing 6.2. Własny typ odwzorowania dla kwot w różnych walutach w nowym schemacie bazy danych

```
package auction.customtypes;

import ...;

public class MonetaryAmountCompositeUserType
    implements CompositeUserType {

    public Class returnedClass() { return MonetaryAmount.class; }

    public boolean equals(Object x, Object y) {
        if (x == y) return true;
        if (x == null || y == null) return false;
        return x.equals(y);
    }

    public Object deepCopy(Object value) {
        return value; // Obiekty klasy MonetaryAmount są niezmienne.
    }

    public boolean isMutable() { return false; }

    public Object nullSafeGet(ResultSet resultSet,
                             String[] names,
                             SessionImplementor session,
                             Object owner)
        throws HibernateException, SQLException {

        if (resultSet.wasNull()) return null;
        BigDecimal value = resultSet.getBigDecimal( names[0] );
        Currency currency =
            Currency.getInstance(resultSet.getString( names[1] )));
        return new MonetaryAmount(value, currency);
    }

    public void nullSafeSet(PreparedStatement statement,
```

```

        Object value,
        int index,
        SessionImplementor session)
        throws HibernateException, SQLException {

    if (value==null) {
        statement.setNull(index, Types.NUMERIC);
        statement.setNull(index+1, Types.VARCHAR);
    } else {
        MonetaryAmount amount = (MonetaryAmount) value;
        String currencyCode =
            amount.getCurrency().getCurrencyCode();
        statement.setBigDecimal( index, amount.getValue() );
        statement.setString( index+1, currencyCode );
    }
}

public String[] getPropertyNames() { ❶
    return new String[] { "value", "currency" };
}

public Type[] getPropertyTypes() { ❷
    return new Type[] { Hibernate.BIG_DECIMAL, Hibernate.CURRENCY };
}

public Object getPropertyValue(Object component, ❸
                               int property)
    throws HibernateException {

    MonetaryAmount MonetaryAmount = (MonetaryAmount) component;
    if (property == 0)
        return MonetaryAmount.getValue();
    else
        return MonetaryAmount.getCurrency();
}

public void setPropertyValue(Object component, ❹
                           int property,
                           Object value) throws HibernateException {
    throw new UnsupportedOperationException("Niezmienna!");
}

public Object assemble(Serializable cached, ❺
                      SessionImplementor session,
                      Object owner)
    throws HibernateException {
    return cached;
}

public Serializable disassemble(Object value, ❻
                                 SessionImplementor session)
    throws HibernateException {
    return (Serializable) value;
}
}

```

- ❶ Klasa CompositeUserType zawiera własne właściwości definiowane metodą `getPropertyNames()`.
- ❷ Każda właściwość ma własny typ definiowany metodą `getPropertyTypes()`.
- ❸ Metoda `getPropertyValue()` zwraca wartość konkretnej właściwości klasy `MonetaryAmount`.
- ❹ Ponieważ klasa `MonetaryAmount` jest niezmienna, nie możemy osobno ustawiać poszczególnych właściwości. Ta metoda jest opcjonalna.
- ❺ Metoda `assemble()` zostaje wywołana w momencie odczytu egzemplarza tego typu z bufora drugiego poziomu.
- ❻ Metoda `disassemble()` zostaje wywołana w momencie zapisu egzemplarza tego typu do bufora drugiego poziomu.

Kolejność właściwości w metodach `getPropertyNames()`, `getPropertyTypes()` i `getPropertiesValues()` musi być taka sama. Właściwość `initialPrice` odwzorowuje dwie kolumny, więc w pliku odwzorowania należy je określić. Pierwsza kolumna dotyczy kwoty, natomiast druga waluty (kolejność określania kolumn musi odpowiadać kolejności zastosowanej w implementacji typu odwzorowania).

```
<property name="initialPrice"
          type="auction.customtypes.MonetaryAmountCompositeUserType">
    <column name="INITIAL_PRICE"/>
    <column name="INITIAL_PRICE_CURRENCY"/>
</property>
```

Teraz w zapytaniach możemy odnosić się do właściwości `amount` i `currency` własnego typu, choć nie pojawiają się one w dokumencie odwzorowania jako niezależne właściwości.

```
from Item i
where i.initialPrice.value > 100.0
  and i.initialPrice.currency = 'PLN'
```

Rozbudowaliśmy pomost między modelem obiektów Javy i schematem bazy danych SQL, używając własnego typu złożonego. W ten sposób obie reprezentacje współpracują elastyczniej.

Jeśli implementacja własnych typów odwzorowań wydaje się trudna, odpręż się — nie trzeba często z niej korzystać. Alternatywą dla klasy `MonetaryAmount` jest odwzorowanie komponentów opisane dokładniej w podrozdziale 3.5.2. Stosowanie własnych typów jest więc dobrowolnym wyborem, a nie koniecznością.

Przyjrzymy się niezwykle ważnemu zastosowaniu własnych typów odwzorowań. Wzorzec projektowy **wyliczenia zapewniającego bezpieczeństwo typów** niemal zawsze pojawia się w aplikacjach biznesowych.

Typy wyliczeniowe

Typ wyliczeniowy w Javie stosuje się bardzo często wtedy, gdy klasa ma stałą i niewielką liczbę niezmiennych egzemplarzy.

Na przykład klasa `Comment` (użytkownicy wystawiają komentarze innym użytkownikom systemu) definiuje właściwość `rating`. W obecnym modelu jest ona

typu int. Bezpieczniejszym ze względu na typy (i znacznie lepszym) sposobem implementacji oceny użytkownika (bo wątpliwe, by aplikacja dopuszczała podawanie dowolnych wartości całkowitych) byłoby utworzenie następującej klasy Rating.

```
package auction;

public class Rating implements Serializable {

    private String name;

    public static final Rating EXCELLENT = new Rating("Wspaniała");
    public static final Rating OK = new Rating("Poprawna");
    public static final Rating LOW = new Rating("Niska");
    private static final Map INSTANCES = new HashMap();

    static {
        INSTANCES.put(EXCELLENT.toString(), EXCELLENT);
        INSTANCES.put(OK.toString(), OK);
        INSTANCES.put(LOW.toString(), LOW);
    }

    private Rating(String name) {
        this.name=name;
    }

    public String toString() {
        return name;
    }

    Object readResolve() {
        return getInstance(name);
    }

    public static Rating getInstance(String name) {
        return (Rating) INSTANCES.get(name);
    }
}
```

Zmodyfikujmy właściwość klasy Comment, by wykorzystywała nowy typ. W bazie danych ocenę warto przechowywać w kolumnie typu VARCHAR. Utworzenie klasy implementującej interfejs UserType dla właściwości oceny nie stanowi dużego wyzwania.

```
package auction.customtypes;

import ...;
public class RatingUserType implements UserType {

    private static final int[] SQL_TYPES = {Types.VARCHAR};
    public int[] sqlTypes() { return SQL_TYPES; }
    public Class returnedClass() { return Rating.class; }
    public boolean equals(Object x, Object y) { return x == y; }
    public Object deepCopy(Object value) { return value; }
    public boolean isMutable() { return false; }
```

```
public Object nullSafeGet(ResultSet resultSet,
                          String[] names,
                          Object owner)
                          throws HibernateException, SQLException {

    String name = resultSet.getString(names[0]);
    return resultSet.wasNull() ? null : Rating.getInstance(name);
}

public void nullSafeSet(PreparedStatement statement,
                       Object value,
                       int index)
                       throws HibernateException, SQLException {

    if (value == null) {
        statement.setNull(index, Types.VARCHAR);
    } else {
        statement.setString(index, value.toString());
    }
}
}
```

Kod jest bardzo podobny do wcześniejszej implementacji interfejsu UserType. Najważniejsze są metody nullSafeGet() i nullSafeSet(), gdyż zawierają logikę sterującą konwersją.

Pewne problemy dotyczące typów wyliczeniowych pojawiają się w momencie tworzenia zapytań. Rozważmy w HQL następujące zapytanie, które pobiera wszystkie komentarze o ocenie „Niska”.

```
Query q = session.createQuery("from Comment c where c.rating = Rating.LOW");
```

Zapytanie nie zadziała poprawnie, ponieważ Hibernate nie wie nic na temat Rating.LOW, więc potraktuje ten fragment dosłownie. By skorzystać z wyliczeń, trzeba dowiązywać parametry i ustawać wartość w osobnym wywoaniu (podejście to zaleca się również z innych względów).

```
Query q = session.createQuery("from Comment c where c.rating = :rating");
q.setParameter("rating", Rating.LOW,
               Hibernate.custom(RatingUserType.class));
```

Ostatni wiersz zawiera przykład użycia metody statycznej Hibernate.custom() w celu konwersji własnego typu odwzorowania na typ Hibernate (Type). W ten sposób system ORM dowiaduje się o sposobie poprawnej interpretacji wartości Rating.LOW.

Jeśli typy wyliczeniowe występują w wielu miejscach aplikacji, warto skorzystać z przedstawionej przykładowej implementacji i wykonać uogólniony typ odwzorowań wyliczeniowych. Od JDK 1.5 dostępny jest nowy sposób określania typów wyliczeniowych, który jednak nie jest obsługiwany przez Hibernate 2.1, trzeba więc stosować własne typy odwzorowań. Pamiętaj, że klasa PersistentEnum w Hibernate 2 została określona jako przestarzała i nie należy jej używać.

Omówiliśmy wszystkie rodzaje odwzorowań Hibernate: wbudowane typy odwzorowań, typy definiowane przez użytkownika i komponenty (rozdział 3.).

Wszystkie traktuje się jako typy wartości, ponieważ odwzorowują typ wartości z obiektu (a nie całą encję) na kolumny bazy danych. Możemy przejść do opisu kolekcji typów wartości.

6.2. Odwzorowywanie kolekcji typów wartości

W rozdziale 3. przedstawiliśmy kolekcje w kontekście związków encji. W tym podrozdziale omówimy kolekcje zawierające egzemplarze typów wartości, włączając w to kolekcje komponentów. Przy okazji zahaczymy o kilka zaawansowanych funkcji odwzorowania kolekcji w Hibernate, które można również zastosować dla kolekcji reprezentujących asocjacje encyjne (patrz dalsza część rozdziału).

6.2.1. Zbiory, pojemniki, listy i odwzorowania

Przypuśćmy, że sprzedawcy mogą dodawać zdjęcia do obiektów Item. Obraz można pobrać tylko poprzez zawierający go przedmiot. Nie musi obsługiwać asocjacji do innych encji w systemie. Przy takich założeniach sensownym wydaje się potraktowanie obrazu jako typu wartości Obiekt Item zawierałby kolekcję obrazów, które Hibernate traktowałby jako składową obiektu bez własnego cyklu życia.

Prześledźmy kilka sposobów implementacji tego zachowania w Hibernate. Na razie założymy, że obrazy są tak naprawdę przechowywane w systemie plików serwera, a baza danych zawiera jedynie informację o nazwie pliku. Nie będziemy zajmować się pobieraniem i umieszczaniem konkretnych obrazów na serwerze.

Użycie zbioru

Najprostsza implementacja to obiekt Set zawierający obiekt String z nazwami plików. Dodajmy właściwość kolekcji do klasy Item.

```
private Set images = new HashSet();
...
public Set getImages() {
    return this.images;
}
public void setImages(Set images) {
    this.images = images;
}
```

W pliku odwzorowania klasy Item musimy dodać poniższe dane:

```
<set name="images" lazy="true" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

Nazwy plików obrazów trafiają do tabeli o nazwie ITEM_IMAGE. Z punktu widzenia bazy danych tabela ta nie stanowi części tabeli ITEM. Hibernate ukrywa to przed programistą aplikacji, tworząc iluzję jednej encji. Element <key> deklaruje klucz

obcy ITEM_ID przechowujący identyfikator encji nadzędnej. Element <element> deklaruje kolekcję jako zbiór typów wartości stosujących odwzorowanie string.

Zbiór nie może zawierać duplikatów, więc klucz główny tabeli ITEM_IMAGE składa się z kolumny wskazanej w <key> (ITEM_ID) i w <element> (FILENAME). Przykład schematu tabel bazy danych dla tej sytuacji przedstawia rysunek 6.1.

ITEM		ITEM_IMAGE	
ITEM_ID	NAME	ITEM_ID	FILENAME
1	Foo	1	fooimage1.jpg
2	Bar	1	fooimage2.jpg
3	Baz	2	barimage1.jpg

Rysunek 6.1. Struktura tabel i przykładowe dane dla kolekcji tekstów

Nie wydaje się poprawnym dwukrotne przypisywanie do jednego przedmiotu tego samego obrazu, ale założmy dopuszczenie takiej sytuacji. Jakie odwzorowanie należy zastosować?

Użycie pojemnika

Nieuporządkowaną kolekcję dopuszczającą duplikację elementów często nazywa się **pojemnikiem**. Co ciekawe, szkielet kolekcji Javy nie definiuje interfejsu dla pojemników. Hibernate dopuszcza użycie obiektów List do symulacji pojemnika. Zgadza się to z najczęstszą reprezentacją pojemnika w aplikacjach. Pamiętaj jednak, że choć obiekt List zakłada pewną kolejność zawartych w nim elementów, Hibernate nie zachowuje tej kolejności w momencie utrwalania zawartości listy traktowanej jako pojemnik. Aby włączyć pojemnik, zmień typ właściwości image z Set na List, wykorzystując konkretną implementację ArrayList (jako typ właściwości można również użyć typu Collection).

Zmiana definicji tabel z poprzedniego punktu dopuszczająca istnienie duplikatów wymaga wprowadzenia innego klucza głównego. Odwzorowanie <idbag> umożliwia dodanie kolumny sztucznego klucza do tabeli reprezentującej kolekcję w podobny sposób, jak dodaje się sztuczne klucze identyfikujące dla klas encyjnych.

```
<idbag name="images" lazy="true" table="ITEM_IMAGE">
    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```

W przedstawionym przykładzie generowany klucz główny trafia do kolumny ITEM_IMAGE_ID. Rysunek 6.2 przedstawia graficzną reprezentację tabel bazy danych.

Dlaczego jako odwzorowania Hibernate użyliśmy <idbag>? Czy istnieje odwzorowanie <bag>? Wkrótce dokładniej przyjrzymy się pojemnikom, ale najpierw zajmiemy się bardziej prawdopodobną sytuacją, czyli zachowaniem kolejności, w jakiej obrazy zostały dodane do obiektu Item. Istnieje wiele sposobów wykonania tego zadania. Jednym z nich jest użycie rzeczywistej listy zamiast pojemnika.

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_IMAGE_ID	ITEM_ID	FILENAME
1	Foo	1	1	foolimage1.jpg
2	Bar	2	1	foolimage1.jpg
3	Baz	3	2	barimage1.jpg

Rysunek 6.2. Struktura tabel stosująca pojemnik ze sztucznym kluczem głównym

Użycie listy

Odwzorowanie `<list>` wymaga dodania do tabeli bazy danych kolumny indeksowej. Określa ona położenie elementu w kolekcji. W ten sposób Hibernate potrafi odtworzyć kolejność elementu kolekcji przy pobieraniu elementów z bazy danych. Listę odwzorowuje się elementem `<list>`.

```
<list name="images" lazy="true" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <index column="POSITION"/>
    <element type="string" column="FILENAME" not-null="true"/>
</list>
```

Klucz główny składa się z kolumn `ITEM_ID` i `POSITION`. Możliwa jest duplikacja elementów (nazw w `FILENAME`), co zgadza się z ogólnym znaczeniem listy w językach programowania. Przedstawiona zmiana nie wymaga modyfikacji klasy `Item`, bo stosuje ten sam typ, co wcześniejszy pojemnik.

Jeśli kolekcja ma postać `[foolimage1.jpg, foolimage1.jpg, foolimage2.jpg]`, kolumna `POSITION` zawiera wartości 0, 1 i 2 (patrz rysunek 6.3).

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_ID	POSITION	FILENAME
1	Foo	1	0	foolimage1.jpg
2	Bar	1	1	foolimage1.jpg
3	Baz	1	2	foolimage2.jpg

Rysunek 6.3. Struktura tabel stosująca listę z kolumną pozycjonującą

Zamiast listy można użyć zwykłej tablicy Javy. Hibernate obsługuje to rozwiązanie. Co więcej, w obu przypadkach szczegóły odwzorowania są identyczne. Nie polecamy stosowania tablic, gdyż nie można ich inicjalizować w sposób leniwy (nie można utworzyć pośrednika dla tablicy na poziomie maszyny wirtualnej).

Teraz przypuśćmy, że poza nazwami plików również użytkownik nadaje obrazom nazwy. Zadanie to w Javie modeluje się, używając obiektu Map, w którym kluczem jest nazwa wpisana przez użytkownika, natomiast wartością nazwa pliku.

Użycie odwzorowania (mapy)

Odwzorowanie dla mapy przypomina odwzorowanie dla listy.

```
<map name="images" lazy="true" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <index column="IMAGE_NAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

Klucz główny składa się z kolumn ITEM_ID i IMAGE_NAME. Kolumna IMAGE_NAME zawiera klucze mapy. Nadal możliwe są duplikaty plików, co przedstawia rysunek 6.4.

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_ID	IMAGE_NAME	FILENAME
1	Foo	1	Obraz Foo	fooimage1.jpg
2	Bar	1	Ten sam obraz Foo	fooimage1.jpg
3	Baz	1	Inny obraz Foo	fooimage2.jpg

Rysunek 6.4. Tabela dla mapy (odwzorowania) używająca tekstów jako indeksów

Przedstawiona mapa nie jest uporządkowana. Co zrobić, by mapa zawsze była porządkowana według nazwy obrazu?

Kolekcje posortowane i uporządkowane

Choć w języku potocznym słowa posortowane i uporządkowane niejednokrotnie znaczą dokładnie to samo, nie jest tak w przypadku trwałych kolekcji w Hibernate. **Kolekcja posortowana** oznacza kolekcję sortowaną w pamięci za pomocą komparatora Javy. **Kolekcja uporządkowana** ma kolejność ustaloną na poziomie bazy danych przy użyciu klauzuli order by.

Włączenie sortowania mapy wymaga jedynie prostej zmiany w dokumencie odwzorowania.

```
<map name="images"
    lazy="true"
    table="ITEM_IMAGE"
    sort="natural">
    <key column="ITEM_ID"/>
    <index column="IMAGE_NAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

Atrybut sort="natural" informuje Hibernate, by stosował obiekt SortedMap i sortował nazwy obrazów zgodnie z metodą compareTo() klasy java.lang.String. Jeżeli chce się uzyskać inny sposób sortowania, na przykład w odwrotnym porządku alfabetycznym, trzeba określić nazwę klasy implementującej interfejs java.util.Comparator w wartości atrybutu sort. Oto przykład:

```
<map name="images"
    lazy="true"
    table="ITEM_IMAGE"
    sort="auction.util.comparator.ReverseStringComparator">
    <key column="ITEM_ID"/>
```

```
<index column="IMAGE_NAME" type="string"/>
<element type="string" column="FILENAME" not-null="true"/>
</map>
```

Zachowanie posortowanej mapy Hibernate odpowiada zachowaniu obiektu `java.util.TreeMap`. Posortowany zbiór (działający podobnie jak `java.util.TreeSet`) odwzorowuje się bardzo podobnie.

```
<set name="images"
      lazy="true"
      table="ITEM_IMAGE"
      sort="natural">
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

Pojemników (nie istnieje `TreeBag`) i list (indeks określa kolejność elementów) nie można sortować.

Ewentualnie można wybrać mapę uporządkowaną, która wykorzystuje możliwości sortowania bazy danych zamiast (prawdopodobnie mniej wydajnego) sortowania w pamięci.

```
<map name="images"
      lazy="true"
      table="ITEM_IMAGE"
      order-by="IMAGE_NAME asc">
    <key column="ITEM_ID"/>
    <index column="IMAGE_NAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

Wyrażenie atrybutu `order-by` stanowi fragment klauzuli `order by` przesyłanej jako zapytanie bazy danych. W przedstawionym przykładzie sortujemy w porządku rosnącym na podstawie kolumny `IMAGE_NAME`. Atrybut może nawet zawierać wywołania funkcji SQL. Oto przykład:

```
<map name="images"
      lazy="true"
      table="ITEM_IMAGE"
      order-by="lower(FILENAME) asc">
    <key column="ITEM_ID"/>
    <index column="IMAGE_NAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

Zauważ, że sortować można względem dowolnej kolumny tabeli kolekcji. Atrybut `order-by` można dodać do zbiorów i pojemników, ale nie do list. Kolejny przykład używa pojemnika.

```
<idbag name="images"
      lazy="true"
      table="ITEM_IMAGE"
      order-by="ITEM_IMAGE_ID desc">
    <collection-id type="long" column="ITEM_IMAGE_ID">
      <generator class="sequence"/>
    </collection-id>
```

```

<key column="ITEM_ID"/>
<element type="string" column="FILENAME" not-null="true"/>
</idbag>

```

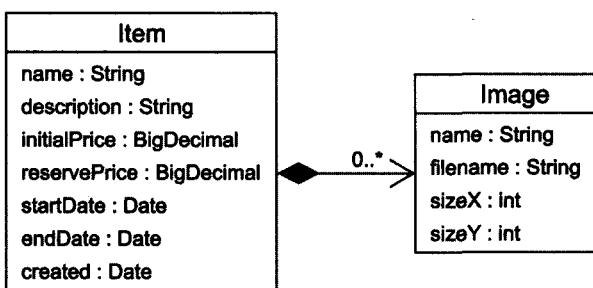
Wewnętrznie do implementacji uporządkowanych zbiorów i map Hibernate wykorzystuje klasy `LinkedListSet` i `LinkedHashMap`, co oznacza możliwość korzystania z tego rozwiązania tylko w JDK 1.4 lub nowszym. Uporządkowane pojemniki można stosować w każdej wersji JDK.

W systemie rzeczywistym zapewne danymi obrazu nie będzie tylko jego nazwa i nazwa pliku. Z pewnością powstanie klasa `Image` przechowującą dodatkowe informacje. Można tę klasę odwzorować jako klasę encyjną. Z drugiej strony wcześniej wykazaliśmy, że nie jest to potrzebne. Zastanówmy się, jak daleko w rozbudowanie danych obrazu możemy się posunąć bez wprowadzania encji (która wymagałaby odwzorowania asocjacji i bardziej złożonego zarządzania cyklem życia).

W rozdziale 3. pokazaliśmy, w jaki sposób Hibernate odwzorowuje zdefiniowane przez nas klasy na komponenty (będące rodzajem typu wartości). Przedstawione tam założenia są prawdziwe, nawet jeśli egzemplarz komponentu zmieni się w kolekcję elementów.

Kolekcje komponentów

Klasa `Image` definiuje właściwości `name`, `filename`, `sizeX` i `sizeY`. Ma tylko jedną asocjację z klasą rodzicielską `Item` (patrz rysunek 6.5).



Rysunek 6.5.
Kolekcja komponentów
`Image` w klasie `Item`

Asocjacja jest tak naprawdę agregacją (czarny romb), więc `Image` stanowi komponent `Item`, natomiast `Item` określa i zarządza cyklem życia obiektu `Image`. Referencji do obrazów nie współdzieli się, więc najlepiej zastosować odwzorowanie komponentowe. Informacja o krotności wskazuje, że dla jednego obiektu `Item` istnieć może wiele (lub zero) obiektów `Image`.

Tworzenie klasy komponentu

Zaimplementujmy klasę `Image`. Będzie to zwykła klasa POJO bez żadnych dodatkowych elementów. W rozdziale 3. wskazaliśmy, że klasa komponentowa nie zawiera właściwości identyfikującej. Musimy jednak zaimplementować metodę `equals()` (i `hashCode()`) porównującą właściwości `name`, `filename`, `sizeX` i `sizeY`, by Hibernate mógł wykryć zmiany w zawartości. W zasadzie `equals()` i `hashCode()`

nie trzeba implementować dla wszystkich klas komponentowych. Niemniej zalecamy ich implementację, bo zawsze lepiej „dmuchać na zimne”.

Klasa Item nie zmienia się — nadal używa interfejsu Set dla obrazów. Oczywiście sama kolekcja nie będzie już przechowywać obiektów String. Dokonajmy odwzorowania kolekcji na kolumny bazy danych.

Odwzorowanie kolekcji

Odwzorowania kolekcji komponentów wykonuje się podobnie jak odwzorowania innych typów wartości. Jedyna różnica polega na zastąpieniu znacznika <element> znacznikiem <composite-element>. Odwzorowanie uporządkowanego zbioru obrazów może mieć następującą postać:

```
<set name="images"
    lazy="true"
    table="ITEM_IMAGE"
    order-by="IMAGE_NAME asc">
<key column="ITEM_ID"/>
<composite-element class="Image">
    <property name="name" column="IMAGE_NAME" not-null="true"/>
    <property name="filename" column="FILENAME" not-null="true"/>
    <property name="sizeX" column="SIZEX" not-null="true"/>
    <property name="sizeY" column="SIZEY" not-null="true"/>
</composite-element>
</set>
```

Ponieważ jest to zbiór, klucz główny zawiera kolumnę klucza obcego oraz wszystkie kolumny elementów: ITEM_ID, IMAGE_NAME, FILENAME, SIZEX i SIZEY. We wszystkich kolumnach pojawił się atrybut not-null="true", bo wszystkie wchodzą w skład klucz głównego (oczywiście jest to wada przedstawionego odwzorowania).

Nawigacja dwukierunkowa

Asocjacja między Item i Image jest jednokierunkowa. Jeśli klasa Image zawierałaby właściwość o nazwie item przechowującą referencję do obiektu Item, w odwzorowaniu należałoby dodać znacznik <parent>.

```
<set name="images"
    lazy="true"
    table="ITEM_IMAGE"
    order-by="IMAGE_NAME asc">
<key column="ITEM_ID"/>
<composite-element class="Image">
    <parent name="item"/>
    <property name="name" column="IMAGE_NAME" not-null="true"/>
    <property name="filename" column="FILENAME" not-null="true"/>
    <property name="sizeX" column="SIZEX" not-null="true"/>
    <property name="sizeY" column="SIZEY" not-null="true"/>
</composite-element>
</set>
```

Realizacja pełnej nawigacji dwukierunkowej nie jest możliwa. Nie można niezależnie pobrać obiektu Image, a następnie przejść do jego obiektu nadziednego.

To istotna kwestia. Można pobrać obiekty Image, stosując odpowiednie zapytanie, ale komponenty oraz inne typy wartości Hibernate pobiera przez wartość. Obiekty Image nie będą więc zawierały referencji do rodzica (referencja będzie równa null). Aby uzyskać przejście w obie strony, trzeba zastosować pełną asocjację rodzinę-potomka w sposób opisany w rozdziale 3.

Deklaracja wszystkich właściwości jako not-null nie wydaje się dobrym pomysłem. Poszukajmy lepszego klucza głównego dla tabeli IMAGE.

Unikanie kolumn niepustych

Skoro zbiór obiektów Image nie jest tym, czego potrzebujemy, przyjrzyjmy się innym kolekcjom. Pojemnik (<idbag>) oferuje sztuczny klucz główny.

```
<idbag name="images"
      lazy="true"
      table="ITEM_IMAGE"
      order-by="IMAGE_NAME asc">
  <collection-id type="long" column="ITEM_IMAGE_ID">
    <generator class="sequence"/>
  </collection-id>
  <key column="ITEM_ID"/>
  <composite-element class="Image">
    <property name="name" column="IMAGE_NAME"/>
    <property name="filename" column="FILENAME" not-null="true"/>
    <property name="sizeX" column="SIZEX"/>
    <property name="sizeY" column="SIZEY"/>
  </composite-element>
</idbag>
```

Tym razem klucz główny składa się tylko z kolumny ITEM_IMAGE_ID. Nie musimy też implementować metod equals() i hashCode() (a przynajmniej nie wymaga tego Hibernate). Co ważniejsze, tylko właściwości wymagające tego od strony logiki biznesowej muszą stosować atrybut not-null="true". Rysunek 6.6 wskazuje, że w ogóle nie trzeba wypełniać niektórych kolumn tabeli obrazów.

ITEM_IMAGE

ITEM_IMAGE_ID	ITEM_ID	IMAGE_NAME	FILENAME
1	1	Obraz Foo 1	fooimage1.jpg
2	1	Obraz Foo 1	fooimage1.jpg
3	2	Obraz Bar 1	barimage1.jpg

Rysunek 6.6. Kolekcja komponentów Image wykorzystująca pojemnik ze sztucznymi kluczami głównymi

Zauważ, że przedstawione odwzorowanie pojemnikowe nie różni się znacząco od standardowego odwzorowania związku rodzinę-potomka. Tabele są identyczne. Nawet kod Javy wygląda podobnie. Wybór rozwiązania zależy od własnych preferencji. Pamiętaj, że związek rodzinę-potomka zapewnia współdzielenie referencji z innymi encjami i obsługuje nawigację dwukierunkową.

Możemy nawet usunąć z klasy Image właściwość name i ponownie jako klucz mapy używać nazwy pliku.

```
<map name="images"
    lazy="true"
    table="ITEM_IMAGE"
    order-by="IMAGE_NAME asc">
<key column="ITEM_ID"/>
<index type="string" column="IMAGE_NAME"/>
<composite-element class="Image">
    <property name="filename" column="FILENAME" not-null="true"/>
    <property name="sizeX" column="SIZEX"/>
    <property name="sizeY" column="SIZEY"/>
</composite-element>
</map>
```

Podobnie jak w jednym z wcześniejszych rozwiązań, klucz główny składa się z kolumn ITEM_ID i IMAGE_NAME.

Klasa elementu złożonego, na przykład Image, nie jest ograniczona do stosowania prostych typów właściwości, jak filename. Może zawierać inne komponenty odwzorowywane deklaracją `<nested-composite-element>` lub nawet asocjacje `<many-to-one>` do encji. Nie może jednak zawierać kolekcji. Element złożony z asocjacją wiele-do-jednego bywa użyteczny, więc do tego pomysłu wróćmy jeszcze dalszej części rozdziału.

Ostatecznie zakończyliśmy temat typów wartości. Przejdziemy do technik odwzorowania asocjacji encyjnych. Prosty związek rodzic-potomek z rozdziału 3. to tylko jeden z wielu rodzajów asocjacji dostępnych w Hibernate. Większość z nich bardzo rzadko pojawia się w rzeczywistych aplikacjach.

6.3. Odwzorowanie asocjacji encyjnych

Gdy używamy słowa **asocjacja**, zawsze myślimy o związku między encjami. W rozdziale 3. przedstawiliśmy jednokierunkową asocjację jeden-do-wielu, następnie zmieniliśmy ją w asocjację dwukierunkową, a na końcu przekształciliśmy w związek rodzic-potomek (asocjacje jeden-do-wielu i wiele-do-jednego).

Asocjacja jeden-do-wielu to najważniejszy rodzaj asocjacji. W zasadzie posuniemy się nawet do zalecenia omijania szerokim łukiem innych, bardziej egzotycznych rodzajów asocjacji, jeśli można je zastąpić prostą dwukierunkową asocjacją jeden-do-wielu i wiele-do-jednego. Asocjację wiele-do-wielu zawsze można przedstawić jako połączenie dwóch asocjacji wiele-do-jednego związanych z jedną klasą pośredniczącą. Co więcej, model ten znacznie łatwiej rozbudować, więc w przedstawianej aplikacji unikamy asocjacji wiele-do-wielu.

Po tych ostrzeżeniach przejdźmy od razu do przeglądu rodzajów odwzorowań asocjacji dostępnych w Hibernate. Zaczniemy od asocjacji jeden-do-jednego.

6.3.1. Asocjacja jeden-do-jednego

W rozdziale 3. stwierdziliśmy, że związek między klasami User i Address (użytkownik stosuje zarówno adres domowy, jak i adres obciążania płatnością) najlepiej reprezentować odwzorowaniem <component>. Jest to najprostszy sposób realizacji związku jeden-do-jednego, ponieważ cykl życia jednego obiektu jest ściśle związany z czasem życia drugiego. Poza tym ten rodzaj asocjacji to kompozycję.

Co jednak należałoby zrobić, gdybyśmy jednak zdecydowali się na osobną tabelę dla adresów i zamodelowali klasę Address jako encję? Powstałaby rzeczywiście asocjacja jeden-do-jednego. Odwzorowanie nowej wersji klasy Address miałoby poniższą postać:

```
<class name="Address" table="ADDRESS">
    <id name="id" column="ADDRESS_ID">
        <generator class="native"/>
    </id>
    <property name="street"/>
    <property name="city"/>
    <property name="zipcode"/>
</class>
```

Teraz klasa potrzebuje właściwości identyfikującej; nie jest już klasą komponentową. W Hibernate istnieją dwa sposoby reprezentacji asocjacji jeden-do-jednego dla klasy Address. Pierwszy z nich dodaje kolumnę klucza obcego do tabeli USER.

Asocjacja stosująca klucz obcy

Najprostszy sposób reprezentacji asocjacji między klasą User i adresami polega na wprowadzeniu odwzorowania <many-to-one> z uaktywnieniem unikalności klucza obcego. Nie jest to intuicyjne, bo **wiele** (many) nie pasuje do opisu żadnego z końców asocjacji jeden-do-jednego! Z punktu wiedzenia Hibernate nie istnieje duża różnica między tymi dwoma rodzajami asocjacji. Dodajmy kolumnę klucza obcego o nazwie BILLING_ADDRESS_ID do tabeli USER i odwzorujmy ją w następujący sposób:

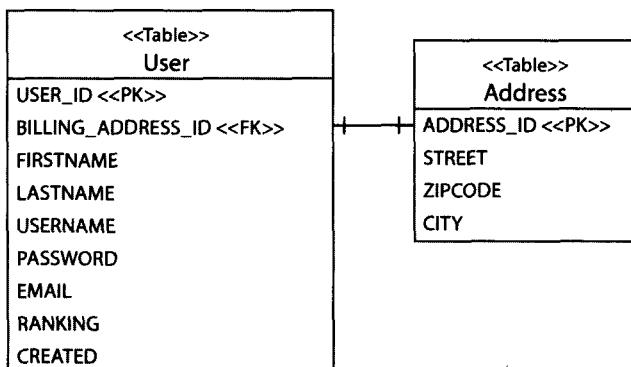
```
<many-to-one name="billingAddress"
    class="Address"
    column="BILLING_ADDRESS_ID"
    cascade="save-update"/>
```

Zauważ wybranie stylu kaskadowego save-update. Oznacza to, że obiekt Address przejdzie w tryb trwałości po dodaniu go do trwałego obiektu User. Prawdopodobnie styl kaskadowy all również miałby duży sens, bo usunięcie użytkownika powinno wiązać się z usunięciem adresu (pamiętaj, że teraz adres ma własny cykl trwałego życia).

Schemat bazy danych nadal będzie dopuszczał duplikację wartości kolumny BILLING_ADDRESS_ID w tabeli USER, więc dwóch użytkowników będzie mogło skorzystać z referencji do tego samego adresu. By rzeczywiście włączyć pełną asocjację jeden-do-jednego, dodamy atrybut unique="true" do elementu <many-to-one>. W ten sposób w modelu relacyjnym będzie mógł istnieć tylko jeden użytkownik wskazanego adresu.

```
<many-to-one name="billingAddress"
    class="Address"
    column="BILLING_ADDRESS_ID"
    cascade="all"
    unique="true"/>
```

Zmiana dodaje ograniczenie związane z unikatowością do kolumny BILLING_ADDRESS_ID kodu DDL generowanego przez Hibernate — powstaje struktura tabel przedstawiona na rysunku 6.7.



Rysunek 6.7.
Asocjacja jeden-do-jednego z dodatkową kolumną klucza obcego

Co zrobić, jeśli chcemy, by przez asocjację dało się przejść z obiektu Address do obiektu User za pomocą kodu Javy? W rozdziale 3. przedstawiliśmy tworzenie dwukierunkowej kolekcji jeden-do-wielu. Z drugiej strony założyliśmy, że każdy adres należy tylko do jednego użytkownika, więc wcześniejsze rozwiązanie nie wydaje się słusze. Nie chcemy mieć kolekcji użytkowników w klasie Address. Dodajmy do klasy Address właściwość o nazwie user (typu User) i odwzorujmy ją w następujący sposób:

```
<one-to-one name="user"
    class="User"
    property-ref="billingAddress" />
```

Odwzorowanie informuje Hibernate, że asocjacja user klasy Address stanowi przeciwieństwo asocjacji billingAddress klasy User.

W kodzie asocjację między dwoma obiektami tworzymy następująco:

```
Address address = new Address();
address.setStreet("Parkowa 12");
address.setCity("Ruda Śląska");
address.setZipcode("41-700");
```

```
Transaction tx = session.beginTransaction();
User user = (User) session.get(User.class, userId);
address.setUser(user);
user.setBillingAddress(address);
tx.commit();
```

Na zakończenie musimy odwzorować właściwość homeAddress klasy User. Nie jest to trudne: dodajemy kolejny element `<many-to-one>` w odwzorowaniu klasy User,

informując, że powinien dotyczyć nowej kolumny klucza obcego o nazwie HOME_ADDRESS_ID.

```
<many-to-one name="homeAddress"
  class="Address"
  column="HOME_ADDRESS_ID"
  cascade="save-update"
  unique="true"/>
```

Tabela USER definiuje dwa klucze obce dotyczące odnoszące się do klucza głównego tabeli ADDRESS: HOME_ADDRESS_ID i BILLING_ADDRESS_ID.

Niestety, nie możemy obu asocjacji uczynić dwukierunkowymi, bo nie wiemy, czy dany adres jest adresem domowym czy adresem płatności. Dokładniej, nie możemy wskazać dwóch nazw właściwości w atrrybutie property-ref odwzorowania właściwości user. Moglibyśmy zamienić klasę Address na klasę abstrakcyjną z podklasami HomeAddress i BillingAddress i odwzorować asocjacje na podklasy. Wtedy wszystko działałoby poprawnie, ale to rozwiązanie jest zbyt złożone dla przedstawianej sytuacji.

Zalecamy unikanie definiowania więcej niż jednej asocjacji jeden-do-jednego między dwiema klasami. Jeśli trzeba, pozostaw asocjację jednokierunkową. Jeśli zawsze istnieje tylko jeden egzemplarz Address na każdy egzemplarz User, istnieje alternatywne rozwiązanie do właśnie przedstawionego. Zamiast definiować kolumnę klucza obcego w tabeli USER, używa się asocjacji klucza głównego.

Asocjacja klucza głównego

Dwie tabele powiązane asocjacją klucza głównego współdzielą ten sam klucz główny. Klucz główny jednej tabeli jest jednocześnie kluczem głównym drugiej tabeli. Podstawową trudność w implementacji tego rozwiązania stanowi zapewnienie identyczności klucza w momencie zapisu obiektów. Zanim jednak zajmiemy się tym problemem, odwzorujmy asocjację klucza głównego.

W asocjacji klucza głównego **oba** końce asocjacji odwzorowuje się deklaracją <one-to-one>. Oznacza to, że nie możemy jednocześnie odwzorować obu adresów — dostępny jest tylko jeden. Każdy wiersz tabeli USER ma odpowiadający mu wiersz tabeli ADDRESS. Dwa adresy wymagałyby dodatkowej tabeli, więc przedstawiany styl odwzorowania nie byłby odpowiedni. Właściwość dotycząca jednego adresu nazwijmy address i dodajmy jej odwzorowanie do metadanych klasy User:

```
<one-to-one name="address"
  class="Address"
  cascade="save-update"/>
```

Podobny krok wykonajmy dla klasy Address:

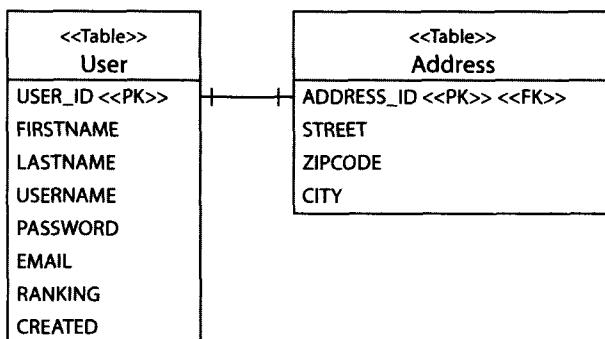
```
<one-to-one name="user"
  class="User"
  constrained="true"/>
```

Najciekawszym elementem jest użycie atrybutu constrained="true". Informuje on Hibernate, że istnieje ograniczenie klucza obcego dla klucza głównego ADDRESS, które odnosi się do klucza głównego tabeli USER.

Teraz musimy zapewnić, by po raz pierwszy zapisywane egzemplarze klasy Address uzyskiwały tę samą wartość identyfikatora co obiekt User. W tym celu stosujemy specjalną strategię generowania klucza w Hibernate o nazwie foreign.

```
<class name="Address" table="ADDRESS">
    <id name="id" column="ADDRESS_ID">
        <generator class="foreign">
            <param name="property">user</param>
        </generator>
    </id>
    ...
    <one-to-one name="user"
        class="User"
        constrained="true"/>
</class>
```

Element `<param>` nazwany property wewnętrz generatora foreign umożliwia nazwanie asocjacji jeden-do-jednego klasy Address — w przedstawionym przykładzie jest to asocjacja user. Generator foreign sprawdza powiązany obiekt (klasy User) i wykorzystuje jego identyfikator jako nowy identyfikator obiektu Address. Strukturę tabel przedstawia rysunek 6.8.



Rysunek 6.8.
Tabele asocjacji jeden-do-jednego z współdzieloną wartością klucza głównego

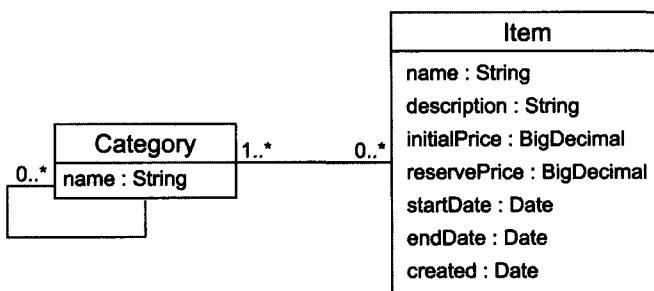
Kod tworzący asocjacje nie zmienia się dla przedstawionego przykładu — jest taki sam jak użyty wcześniej dla odwzorowania wykorzystującego styl wiele-do-jednego.

Do omówienia pozostał już tylko jeden rodzaj asocjacji: wiele-do-wielu.

6.3.2. Asocjacje wiele-do-wielu

Rysunek 6.9 przedstawia asocjację między klasami Category i Item jako asocjację wiele-do-wielu.

W rzeczywistych systemach asocjacja wiele-do-wielu pojawia się rzadko. Z doświadczenia wiemy, że poza powiązaniem dwóch wierszy z różnych tabel niejednokrotnie potrzebne są pewne dodatkowe informacje (na przykład czas



Rysunek 6.9. Asocjacja wiele-do-wielu między klasami Category i Item

i data umieszczania przedmiotu w kategorii) — w tej sytuacji stosuje się pośrednią klasę asocjacyjną. W Hibernate klasę tę odwzorowuje się jako encję i w obu łączonych klasach tworzy się asocjacje jeden-do-wielu. Być może bardziej przyjazne okazuje się użycie klasy elementu złożonego. Techniką tą zajmiemy się w dalszej części książki.

W tym miejscu wykonamy jednak prawdziwą asocjację wiele-dow-wielu. Zaczniemy od przykładu jednokierunkowego.

Jednokierunkowa asocjacja wiele-do-wielu

Jeżeli aplikacja potrzebuje jedynie komunikacji jednokierunkowej, odwzorowanie jest proste. Jednokierunkowe asocjacje wiele-do-wielu nie są trudniejsze od opisywanych wcześniej kolekcji typów wartości. Jeśli obiekt Category zawiera zbiór obiektów Item, używamy następującego odwzorowania:

```

<set name="items"
      table="CATEGORY_ITEM"
      lazy="true"
      cascade="save-update">
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</set>

```

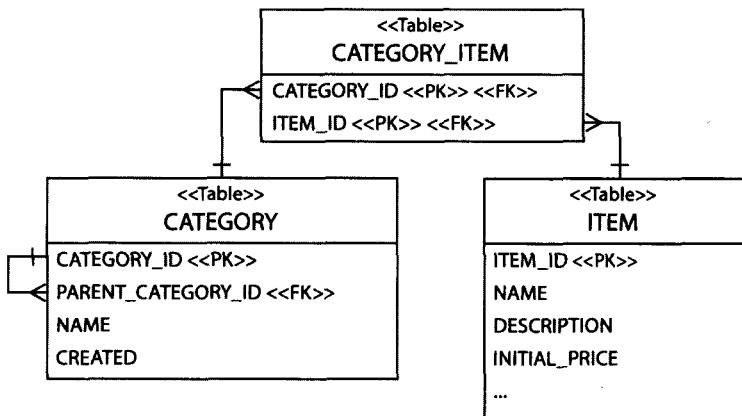
Podobnie jak kolekcja typów wartości asocjacja wiele-do-wielu stosuje własną tabelę nazywaną tabelą łączącą lub asocjacyjną. W przedstawionym przykładzie zawiera ona dwie kolumny: klucze obce z wartościami kluczy głównych tabel CATEGORY i ITEM. Klucz główny nowej tabeli składa się z obu kolumn. Pełną strukturę tabeli przedstawia rysunek 6.10.

Możemy również zastosować pojemnik z wyróżnioną kolumną klucza głównego.

```

<idbag name="items"
        table="CATEGORY_ITEM"
        lazy="true"
        cascade="save-update">
    <collection-id type="long" column="CATEGORY_ITEM_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</idbag>

```



Rysunek 6.10. Asocjacja wiele-do-wielu między encjami odwzorowana na tabelę łączącą

W odwzorowaniu <idbag> klucz główny jest tworzony sztucznie w kolumnie CATEGORY_ITEM_ID. Umożliwia to dwukrotne dodanie tego samego obiektu Item do jednej kategorii (nie wydaje się to szczególnie przydatne).

Co więcej, nic nie stoi na przeszkodzie, by skorzystać z kolekcji indeksowanej (mapy lub listy). Poniższy przykład używa listy.

```

<list name="items"
      table="CATEGORY_ITEM"
      lazy="true"
      cascade="save-update">
    <key column="CATEGORY_ID"/>
    <index column="DISPLAY_POSITION"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</list>

```

Klucz główny składa się z kolumn CATEGORY_ID i DISPLAY_POSITION. Dzięki temu odwzorowaniu każdy obiekt Item zna swoje miejsce w kategorii.

Tworzenie asocjacji obiektów również nie sprawia trudności.

```

Transaction tx = session.beginTransaction();
Category cat = (Category) session.get(Category.class, categoryId);
Item item = (Item) session.get(Item.class, itemId);

cat.getItems().add(item);

tx.commit();

```

Dwukierunkowe asocjacje wiele-do-wielu są nieznacznie trudniejsze do wykonania.

Dwukierunkowe asocjacje wiele-do-wielu

Gdy odwzorowaliśmy dwukierunkową asocjację jeden-do-wielu w podrozdziale 3.7, wyjaśniliśmy, dlaczego jeden z końców asocjacji musi zawierać atrybut inverse="true". Zalecamy ponowne przeczytanie tamtego wyjaśnienia.

Ta sama zasada obowiązuje dla dwukierunkowych asocjacji wiele-do-wielu: każdy wiersz tabeli łączącej zawiera dwa elementy kolekcji, po jednym na każdy koniec asocjacji. W pamięci asocjacja między obiektami Item i Category jest reprezentowana przez egzemplarz klasy Item w kolekcji items obiektu Category i egzemplarz klasy Category w kolekcji categories obiektu Item.

Zanim przejdziemy do odwzorowania przypadku dwukierunkowego, poinformujmy o potrzebie zmiany kodu łączącego obiekty.

```
cat.getItems.add(item);
item.getCategories().add(category);
```

Jak zwykle asocjacja dwukierunkowa (niezależnie od jej krotkości) wymaga ustalenia obu jej końców.

W trakcie deklaracji dwukierunkowej asocjacji wiele-do-wielu należy wskazać, która strona asocjacji ma wpływać na aktualizację tabeli łączącej, używając atrybutu inverse="true". Należy samemu wybrać najbardziej odpowiednią końcówkę.

Przypomnijmy odwzorowanie kolekcji items z poprzedniego podrozdziału:

```
<class name="Category" table="CATEGORY">
    ...
    <set name="items"
        table="CATEGORY_ITEM"
        lazy="true"
        cascade="save-update">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </set>
</class>
```

To samo odwzorowanie można wykorzystać ponownie po stronie Category asocjacji dwukierunkowej. Odwzorowanie po stronie Item ma następującą postać:

```
<class name="Item" table="ITEM">
    ...
    <set name="categories"
        table="CATEGORY_ITEM"
        lazy="true"
        inverse="true"
        cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </set>
</class>
```

Zauważ użycie atrybutu inverse="true". Informuje on Hibernate, by ignorował zmiany dokonane w kolekcji categories i używał tylko drugiej strony asocjacji (kolekcji items) jako reprezentacji, którą należy synchronizować ze stanem bazy danych.

Nie bez powodu dla obu końców kolekcji użyliśmy atrybutu cascade="save-update". Wersje cascade="all", cascade="delete" i cascade="all-delete-orphan" nie mają sensu dla asocjacji wiele-do-wielu, bo egzemplarz z potencjalnie wieloma rodzicami nie powinien być usuwany tuż po usunięciu jednego z rodziców.

Jakie kolekcje mogą stosować dwukierunkowe asocjacje wiele-do-wielu? Czy należy stosować ten sam typ kolekcji na obu końcach? Przykładowo, sensownym wydaje się zastosowanie listy na końcu bez atrybutu `inverse="true"` (lub z jawnie ustawianą wartością `false`) i pojemnika na końcu z tym atrybutem.

Możemy użyć dowolnego z wcześniej przedstawionych odwzorowań dla jednokierunkowej asocjacji wiele-do-wielu po stronie nieinwersyjnej asocjacji dwukierunkowej. Nic nie stoi na przeszkodzie, by użyć znaczników `<set>`, `<idbag>`, `<list>` i `<map>` w tej samej postaci co w wersji jednokierunkowej.

Po stronie inwersyjnej można użyć `<set>` i przedstawionego poniżej `<bag>`.

```
<class name="Item" table="ITEM">
    ...
    <bag name="categories"
        table="CATEGORY_ITEM"
        lazy="true"
        inverse="true" cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Item" column="CATEGORY_ID"/>
    </bag>
</class>
```

To pierwszy przypadek przedstawienia deklaracji `<bag>` — przypomina deklarację `<idbag>`, ale nie wprowadza kolumny ze sztucznym kluczem. Umożliwia użycie obiektu `List` (ale w znaczeniu pojemnika) zamiast obiektu `Set` w klasie trwałej. Zaleca się to podejście, jeśli nieinwersyjna strona odwzorowania asocjacji wiele-do-wielu używa mapy, listy lub pojemnika (wszystkie te kolekcje dopuszczają duplikację). Pamiętaj, że pojemnik nie zachowuje kolejności elementów, mimo że korzysta z obiektu `List`.

Pozostałych odwzorowań nie należy stosować dla inwersyjnego końca asocjacji. Kolekcje indeksowe (listy i mapy) nie mają tu zastosowania, bo Hibernate nie inicjalizuje i nie przechowuje kolumny indeksowej dla `inverse="true"`. Dotyczy to także wszystkich innych odwzorowań asocjacji stosujących kolekcje — kolekcji indeksowanej (a nawet tablicy) nie można stosować w połączeniu z atrybutem `inverse="true"`.

Przyjrzelimy się asocjacji wiele-do-wielu. W tracie jej opisu wspomnieliśmy o alternatywnie — odwzorowaniu elementu złożonego. Przeanalizujmy ją dokładniej.

Zastosowanie kolekcji komponentów dla asocjacji wiele-do-wielu

Przypuśćmy, że musimy poza dodaniem obiektu `Item` do `Category` zapamiętać pewne dodatkowe informacje. Niech będzie to data i czas dodania oraz nazwa użytkownika wprowadzającego zmianę. Potrzebujemy klasy Javy do reprezentacji tych informacji.

```
public class CategorizedItem {
    private String username;
    private Date dateAdded;
    private Item item;
```

```

private Category category;
...
}

```

Pomineliśmy przedstawienie metod dostępowych oraz metod `equals()` i `hashCode()` wymaganych do poprawnego działania klasy komponentowej.

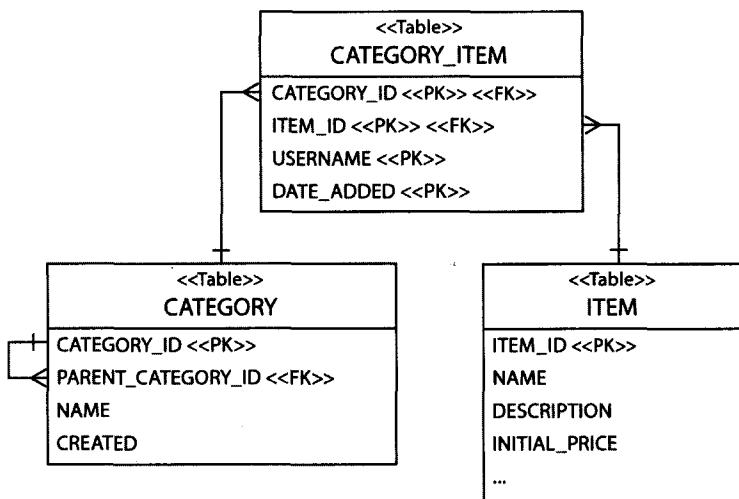
Kolekcję `items` z klasy `Category` odwzorowujemy następująco:

```

<set name="items" lazy="true" table="CATEGORY_ITEMS">
    <key column="CATEGORY_ID"/>
    <composite-element class="CategorizedItem">
        <parent name="category"/>
        <many-to-one name="item"
            class="Item"
            column="ITEM_ID"
            not-null="true"/>
        <property name="username" column="USERNAME" not-null="true"/>
        <property name="dateAdded" column="DATE_ADDED" not-null="true"/>
    </composite-element>
</set>

```

Używamy elementu `<many-to-one>`, by zadeklarować asocjację do `Item`. Używamy również odwzorowania `<property>`, by zadeklarować dodatkowe informacje przechowywane w egzemplarzach klasy. Tabela łącząca składa się z czterech kolumn: `CATEGORY_ID`, `ITEM_ID`, `USERNAME` i `DATE_ADDED`. Kolumny właściwości `CategorizedItem` nigdy nie powinny zawierać wartości `null` — w przeciwnym razie nie będziemy w stanie odróżnić wierszy tabeli łączącej (wszystkie kolumny wchodzą w skład klucza głównego). Strukturę tabeli przedstawia rysunek 6.11.



Rysunek 6.11. Asocjacja wiele-do-wielu stosująca komponent

W zasadzie zamiast zapamiętywać wartość `username`, równie dobrze moglibyśmy przechowywać referencję do obiektu `User`. Wtedy uzyskujemy odwzorowanie asocjacji potrójnej.

```

<set name="items" lazy="true" table="CATEGORY_ITEMS">
    <key column="CATEGORY_ID"/>
    <composite-element class="CategorizedItem">
        <parent name="category"/>
        <many-to-one name="item"
            class="Item"
            column="ITEM_ID"
            not-null="true"/>
        <many-to-one name="user"
            class="User"
            column="USER_ID"
            not-null="true"/>
        <property name="dateAdded" column="DATE_ADDED" not-null="true"/>
    </composite-element>
</set>

```

To wyjątkowo egzotyczny stwór! Gdy tylko pojawi się tego rodzaju zapis asocjacji, warto się zastanowić, czy jednak nie lepiej odwzorować CategorizedItem jako klasę encyjną z dwiema asocjacjami jeden-do-wielu. Co gorsza, nie można zmienić przedstawionego odwzorowania na dwukierunkowe: komponent (czyli CategorizedItem) z definicji nie może mieć współdzielonych referencji. Nie uda się więc przejść od Item do CategorizedItem.

W poprzednim podrozdziale wspomnialiśmy o niektórych ograniczeniach odwzorowania wiele-do-wielu. Jedno z nich, niemożność stosowania indeksowanych kolekcji po inwersywnej stronie asocjacji, dotyczy również asocjacji jeden-do-wielu, jeśli są dwukierunkowe. Raz jeszcze przyjrzymy się asocjacjom jeden-do-wielu i wiele-do-jednego, by odświeżyć pamięć i uściślić pewne kwestie opisane w rozdziale 3.

Asocjacje jeden-do-wielu

W rozdziale 3. przedstawiliśmy większość informacji na temat asocjacji jeden-do-wielu. Wykonaliśmy odwzorowanie typowego związku rodzic-potomek między dwiema trwałymi klasami encyjnymi (Item i Bid). Asocjacja była dwukierunkowa — używała odwzorowań <one-to-many> i <many-to-one>. Końcówka „wiele” była w Javie zaimplementowana jako obiekt Set: kolekcja bids w klasie Item. Jeszcze raz zajmijmy się tym odwzorowaniem, by prześledzić kilka przypadków specjalnych.

Pojemnik o znaczeniu zbioru

Jeśli rzeczywiście zachodzi potrzeba wprowadzenia obiektu List dla potomków klasy rodzicielskiej, można zastąpić zbiór odwzorowaniem <bag>. W przykładzie najpierw musimy zastąpić typ kolekcji bids z klasy trwałej Item obiektem List. Odwzorowanie asocjacji między Item i Bid w zasadzie pozostaje bez zmian.

```

<class
    name="Bid"
    table="BID">
    ...
    <many-to-one

```

```
        name="item"
        column="ITEM_ID"
        class="Item"
        not-null="true"/>
    </class>

<class
    name="Item"
    table="ITEM">
    ...
    <bag
        name="bids"
        inverse="true"
        cascade="all-delete-orphan">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </bag>
</class>
```

Zmieniliśmy jedynie nazwę `<set>` na `<bag>`. Warto pamiętać, że zmiana nie jest użyteczna: struktura tabel nadal wymusza brak duplikatów, więc tak naprawdę odwzorowanie `<bag>` ciągle ma takie samo znaczenie jak zbiór. Niektórzy preferują listy dla asocjacji zamiast innych struktur danych, inni nie są tak oporni w kwestii różnorodności. Zalecamy wykorzystywanie odwzorowania `<set>` dla typowych związków rodzic-potomek.

Oczywistym (ale jednocześnie niepoprawnym) rozwiązaniem byłoby użycie rzeczywistego odwzorowania `<list>` dla bids z dodatkową kolumną przechowującą pozycję elementów. Przypomnij sobie wspomniane w tym rozdziale ograniczenie Hibernate — nie można stosować indeksowanych kolekcji po inwersyjnej stronie asocjacji. Strona z atrybutem `inverse="true"` nie jest w ogóle rozważana przez system w momencie zapisu stanu obiektu, więc informacja o indeksie zostanie bezpowrotnie utracona.

Jeśli związek rodzic-potomek będzie jednokierunkowy (nawigacja tylko od rodzica do dziecka), można zastosować kolekcję indeksowaną (gdyż koniec „wiele” nie będzie inwersyjny). Dobre zastosowania jednokierunkowych asocjacji jeden-do-wielu w praktyce zdarzają się rzadko. Nie pojawia się taka sytuacja w przykładowym systemie aukcyjnym. W rozdziale 3. rozpoczęliśmy co prawda od jednokierunkowego odwzorowania Item i Bid, ale szybko przeszliśmy na asocjację dwukierunkową.

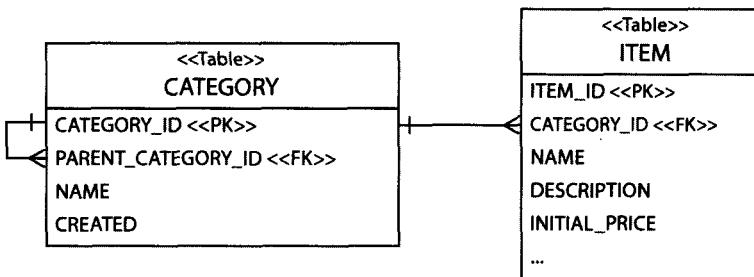
Zastosujmy inny przykład implementacji jednokierunkowej asocjacji jeden-do-wielu z kolumną indeksową.

Odwzorowanie jednokierunkowe

Na potrzeby tego podpunktu założymy, że asocjację między Category i Item należy zamienić na asocjację jeden-do-wielu (przedmiot może należeć do co najwyżej jednej kategorii) i dodatkowo obiekt Item nie ma zawierać referencji do swojej kategorii. W kodzie Javy zamodelujemy związek jako kolekcję nazwaną items w klasie Category. W zasadzie nie musimy nic zmieniać, jeśli kolekcja nie jest indeksowa. Gdy items jest obiektem Set, odwzorowanie ma następującą postać:

```
<set name="items" lazy="true">
  <key column="CATEGORY_ID"/>
  <one-to-many class="Item"/>
</set>
```

Pamiętaj, że asocjacja jeden-do-wielu nie musi podawać nazwy tabeli. Hibernate automatycznie kojarzy nazwę kolumny z odwzorowania kolekcji (CATEGORY_ID) z tabelą ITEM. Strukturę tabeli przedstawia rysunek 6.12.



Rysunek 6.12. Standardowa asocjacja jeden-do-wielu z kolumną klucza obcego

Druga strona asocjacji, klasa Item, nie zawiera referencji do obiektu Category. Obecnie nic nie stoi na przeszkodzie, by użyć kolekcji indeksowanej w klasie Category. Odwzorowanie po zmianie właściwości items na typ List miałoby poniższą postać:

```
<list name="items" lazy="true">
  <key>
    <column name="CATEGORY_ID" not-null="false"/>
  </key>
  <index column="DISPLAY_POSITION"/>
  <one-to-many class="Item"/>
</list>
```

Zwróć uwagę na nową kolumnę **DISPLAY_POSITION** tabeli ITEM, która przechowuje pozycję obiektu Item w kolekcji.

Warto w tym momencie zastanowić się pewną kwestią, która, jak wiemy z doświadczenia, potrafi wpuścić w maliny wielu początkujących użytkowników. W jednokierunkowej asocjacji jeden-do-wielu kolumna klucza głównego CATEGORY_ID tabeli ITEM musi dopuszczać wartości null. Obiekt Item można zapisać bez jakiegokolwiek wiedzy na temat obiektu Category — jest to przecież niezależna encja! Jest to spójny model i odwzorowanie. Warto się nad nim dwa razy zastanowić, gdy ma się do czynienia z niepustym kluczem obcym lub związkiem rodzic-potomek. Najbardziej poprawna okazuje się asocjacja dwukierunkowa z kolekcją wykorzystującą obiekt Set.

Choć znamy już wszystkie techniki odwzorowywania asocjacji dla zwykłych encji, musimy się jeszcze zastanowić nad przebiegiem asocjacji między różnymi poziomami hierarchii dziedziczenia. Przyjrzyjmy się obsłudze asocjacji encji polymorficznych w Hibernate.

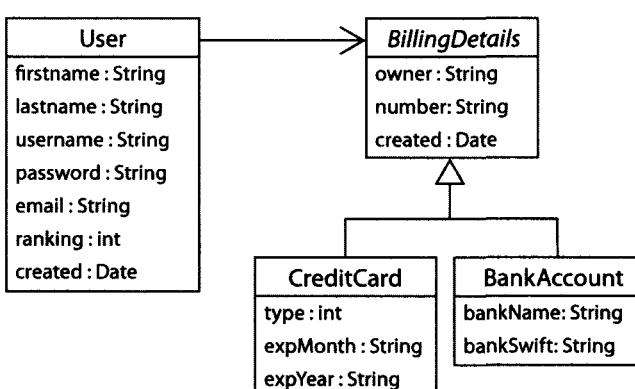
6.4. Odwzorowanie asocjacji polimorficznych

Polimorfizm to jedna z najistotniejszych cech języków obiektowych takich jak Java. Hibernate jako pełne rozwiązanie ORM obsługuje asocjacje i zapytania polimorficzne. Zdziwiające, że zaszliśmy tak daleko bez wspomniania o polimorfizmie. Zaskakujące, że w zasadzie niewiele można w tym temacie powiedzieć — korzystanie z polimorfizmu w Hibernate jest tak proste, że nie trzeba spędzać wiele czasu na jego opisywaniu.

Zacznijmy od rozważenia asocjacji wiele-do-jednego, która wykorzystuje klasę mogącą posiadać podklasy. W takim przypadku Hibernate gwarantuje możliwość tworzenia połączeń do dowolnego egzemplarza podklasy w taki sam sposób jak do egzemplarza klasy bazowej.

6.4.1. Polimorficzna asocjacja wiele-do-jednego

Asocjacja polimorficzna to asocjacja, która może dotyczyć egzemplarzy podklas klasy jawnie określonej w metadanych odwzorowania. Przypuśćmy, że dla jednego użytkownika nie może istnieć kilka klas `BillingDetails`, a tylko jedna (patrz rysunek 6.13).



Rysunek 6.13.
Użytkownik korzysta tylko z jednego obiektu sposobu płatności

Odwzorowujemy asocjację na abstrakcyjną klasę `BillingDetails` w następujący sposób:

```

<many-to-one name="billingDetails"
    class="BillingDetails"
    column="BILLING_DETAILS_ID"
    cascade="save-update"/>
  
```

Ponieważ klasa `BillingDetails` jest abstrakcyjna, w trakcie pracy aplikacji asocjacja będzie wskazywać na jedną z podklas: `CreditCard` lub `BankAccount`.

Wszystkie odwzorowania asocjacji opisane w niniejszym rozdziale obsługują polimorfizm. Nie trzeba wykonywać żadnych dodatkowych kroków, by w Hibernate uaktywnić asocjacje polimorficzne. Wystarczy podać nazwę dowolnej odwzorowanej klasy trwałej w odwzorowaniu asocjacji (lub też pozwolić Hibernate

ROZDZIAŁ 6.**Zaawansowane zagadnienia odwzorowań**

uzyskać tę informację na podstawie refleksji). Jeśli klasa deklaruje elementy <sub-class> lub <joined-subclass>, asocjacja będzie typu polimorficznego.

Poniższy kod przedstawia utworzenie asocjacji do egzemplarza podklasy CreditCard.

```
CreditCard cc = new CreditCard();
cc.setNumber(ccNumber);
cc.setType(ccType);
cc.setExpiryDate(ccExpiryDate);

Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, uid);
user.setBillingDetails(cc);

tx.commit();
session.close();
```

Gdy w drugiej transakcji przejdziemy przez asocjację, Hibernate automatycznie pobierze egzemplarz klasy CreditCard.

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, uid);
// Wywołuje metodę pay() rzeczywiście podklasy.
user.getBillingDetails().pay(paymentAmount);

tx.commit();
session.close();
```

Należy jednak uważać na pewną kwestię: jeśli BillingDetails zostało odwzorowane z atrybutem lazy="true", Hibernate zastosuje dla asocjacji pośrednika. W takiej sytuacji nie uda się rzutować referencji na konkretną klasę CreditCard. Nawet operator instanceof zachowa się dziwnie.

```
User user = (User) session.get(User.class, uid);
BillingDetails bd = user.getBillingDetails();
System.out.println( bd instanceof CreditCard ); // wyświetli "false"
CreditCard cc = (CreditCard) bd; // ClassCastException!
```

Rzutowanie zgłasza wyjątek, ponieważ bd jest egzemplarzem pośrednika. Metoda wywołana dla pośrednika trafia do identycznej metody leniwie tworzonego egzemplarza klasy CreditCard. By przeprowadzić bezpieczne rzutowanie, należy użyć metody Session.load().

```
User user = (User) session.get(User.class, uid);
BillingDetails bd = user.getBillingDetails();
// Pobiera egzemplarz klasy bez korzystania z bazy danych.
CreditCard cc = (CreditCard) session.load( CreditCard.class, bd.getId() );
expiryDate = cc.getExpiryDate();
```

Po wywołaniu metody load() bd i cc dotyczą dwóch różnych egzemplarzy pośrednika korzystających z tego samego egzemplarza klasy CreditCard.

Przedstawionej sytuacji można uniknąć, wyłączając leniwe wczytywanie. Poniższy przykład używa zapytania dokonującego wyprzedzającego pobrania wszystkich powiązanych obiektów.

```
User user = (User) session.createCriteria(User.class)
    .add( Expression.eq("id", uid) )
    .setFetchMode("billingDetails", FetchMode.EAGER)
    .uniqueResult();
// Referencja billingDetails z User zostaje pobrana z wyprzedzeniem.
CreditCard cc = (CreditCard) user.getBillingDetails();
expiryDate = cc.getExpiryDate();
```

Rzeczywiście obiektowy kod nie powinien stosować zbyt wielu operacji instanceof lub rzutowań. Gdy pojawią się problemy z pośrednikami, warto się zastanowić, czy nie wynika to z błędu w projekcie (może istnieje bardziej polimorficzne podejście?).

Hibernate w ten sam sposób obsługuje asocjacje jeden-do-jednego. Co dzieje się ze stroną „wiele” asocjacji?

6.4.2. Kolekcje polimorficzne

Dokonajmy zamiany poprzedniego przykładu na jego oryginalną postać dostępną w aplikacji CaveatEmptor. Jeśli użytkownik stosuje wiele obiektów BillingDetails, używamy dwukierunkowej asocjacji jeden-do-wielu. Odwzorowanie BillingDetails ma postać:

```
<many-to-one name="user"
  class="User"
  column="USER_ID"/>
```

W odwzorowaniu klasy User pojawia się fragment:

```
<set name="billingDetails"
  lazy="true"
  cascade="save-update"
  inverse="true">
  <key column="USER_ID"/>
  <one-to-many class="BillingDetails"/>
</set>
```

Dodanie klasy CreditCard nie sprawia problemu:

```
CreditCard cc = new CreditCard();
cc.setNumber(ccNumber);
cc.setType(ccType);
cc.setExpiryDate(ccExpiryDate);

Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, uid);
// Wywołanie metody pomocniczej ustawiającej oba końce.
user.addBillingDetails(cc);
```

```
tx.commit();
session.close();
```

Metoda `addBillingDetails()` wywołuje metody `getBillingDetails().add(cc)` i `cc.setUser(this)`.

Możemy przejść przez kolekcję i polimorficznie obsłużyć egzemplarze klas `CreditCard` i `BankAccount` (z drugiej strony musimy uważać, by dwukrotnie nie naliczyć użytkownikowi płatności).

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, uid);
Iterator iter = user.getBillingDetails().iterator();
while ( iter.hasNext() ) {
    BillingDetails bd = (BillingDetails) iter.next();
    // Wywołuje CreditCard.pay() lub BankAccount.pay().
    bd.pay(ccPaymentAmount);
}
tx.commit();
session.close();
```

We wcześniejszych przykładach zakładaliśmy, iż `BillingDetails` jest klasą odwzorowywaną przez konkretny dokument odwzorowania Hibernate i strategia dziedziczenia to jedna tabela na hierarchię lub jedna tabela na podklasę. Nie omówiliśmy jeszcze strategii jedna tabela na klasę konkretną, w której to `BillingDetails` nie jest jawnie wymieniana w pliku odwzorowania (pojawia się tylko w definicji podklas Javy).

6.4.3. Asocjacje polimorficzne i jedna tabela na klasę konkretną

W punkcie 3.6.1 zdefiniowaliśmy strategię odwzorowania z jedną tabelą na klasę konkretną i wskazaliśmy jej trudności z reprezentacją asocjacji polimorficznych, gdyż nie można odwzorować związku klucza obcego do tabeli abstrakcyjnej klasy bazowej. W tej strategii nie istnieje tabela abstrakcyjnej klasy bazowej. Istnieją tylko tabele konkretnych klas.

Załóżmy, że chcemy przedstawić polimorficzną asocjację wiele-do-jednego z klasą `User` do `BillingDetails`, gdy hierarchia klasy `BillingDetails` jest odwzorowywana za pomocą strategii jednej tabeli na każdą klasę konkretną. Istnieje tabela `CREDIT_CARD` i `BANK_ACCOUNT`, ale nie tabela `BILLING_DETAILS`. W tabeli `USER` potrzebujemy dwóch informacji, by jednoznacznie powiązać użytkownika z obiektem `CreditCard` lub `BankAccount`:

- ◆ nazwy tabeli, w której znajduje się powiązany egzemplarz,
- ◆ identyfikatora powiązanego egzemplarza.

Tabela `USER` wymaga dodania kolumny `BILLING_DETAILS_TYPE` poza istniejącą już kolumną `BILLING_DETAILS_ID`. Do odwzorowania tej asocjacji stosujemy element `<any>`.

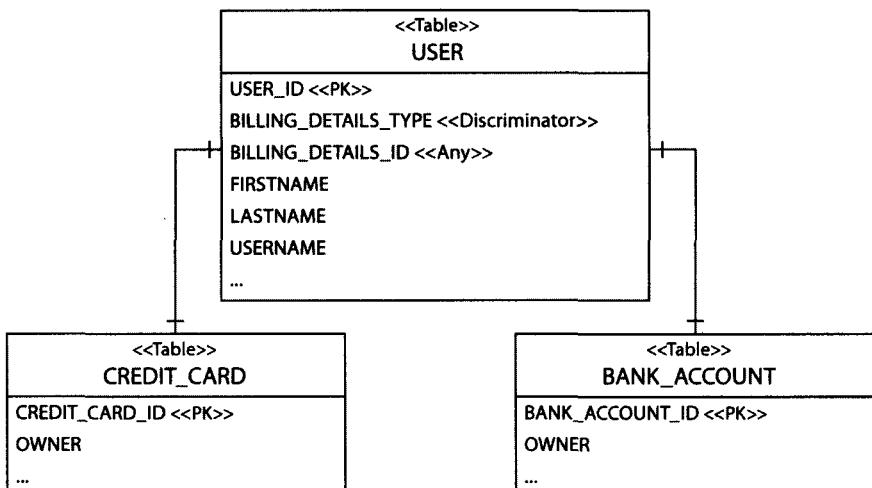
```
<any name="billingDetails"
    meta-type="string"
    id-type="long"
    cascade="save-update">
<meta-value value="CREDIT_CARD" class="CreditCard"/>
<meta-value value="BANK_ACCOUNT" class="BankAccount"/>
<column name="BILLING_DETAILS_TYPE"/>
<column name="BILLING_DETAILS_ID"/>
</any>
```

Atrybut meta-type określa typ Hibernate dla kolumny BILLING_DETAILS_TYPE. Atrybut id-type określa typ dla kolumny BILLING_DETAILS_ID (klasy CreditCard i BankAccount muszą stosować ten sam typ identyfikatora). Kolejność podawania kolumn ma znaczenie: najpierw musi wystąpić typ, a dopiero później identyfikator.

Elementy <meta-value> informują Hibernate, w jaki sposób ma interpretować wartość kolumny BILLING_DETAILS_TYPE. Nie trzeba podawać pełnych nazw tabel — dyskryminatorem typu może być dowolna wartość. Poniższy przykład koduje typ przy użyciu tylko dwóch znaków.

```
<any name="billingDetails"
    meta-type="string"
    id-type="long"
    cascade="save-update">
<meta-value value="CC" class="CreditCard"/>
<meta-value value="CA" class="BankAccount"/>
<column name="BILLING_DETAILS_TYPE"/>
<column name="BILLING_DETAILS_ID"/>
</any>
```

Przykład takiej struktury tabel przedstawia rysunek 6.14.



Rysunek 6.14. Zastosowanie kolumny dyskryminatora i asocjacji any

W tym momencie pojawia się pierwszy problem związany ze stosowanym rodzajem asocjacji — nie możemy dodać ograniczenia klucza obcego do kolumny

BILLING_DETAILS_ID, gdyż zawarta tam wartość dotyczyć będzie raz klucza głównego tabeli BANK_ACCOUNT, a raz klucza tabeli CREDIT_CARD. Musimy w inny sposób zapewnić integralność danych (na przykład dodając wyzwalacz).

Duże trudności sprawia napisanie odpowiednich złączeń SQL dla przedstawionej asocjacji. W szczególności, mechanizm zapytań Hibernate nie obsługuje tego rodzaju asocjacji. Nie jest też ona pobierana złączeniem zewnętrznym. Zalecamy unikanie asocjacji `<any>`, chyba że okazuje się to niezbędne.

Nietrudno zauważyc, że dla strategii dziedziczenia jednej tabeli na klasę konkretną polimorfizm asocjacji okazuje się znacznie bardziej skomplikowany w realizacji niż w pozostałych sytuacjach. Nie stosuj tej strategii odwzorowania, gdy używasz asocjacji polimorficznych. W pozostałych strategiach dziedziczenia asocjacje polimorficzne okazują się znacznie proste. Nawet nie trzeba specjalnie o nich myśleć.

6.5. Podsumowanie

Niniejszy rozdział uzupełnił wiedzę dotyczącą odwzorowania stosowanego w ORM i technik zapobiegania niedopasowaniu strukturalnemu. Możesz teraz bez przeszkód odwzorować wszystkie encje i asocjacje modelu dziedzinowego aplikacji CaveatEmptor.

System typów Hibernate wyróżnia **encje** i **typy wartości**. Encja ma własny cykl życia trwałości i tożsamość bazodanową. Typ wartości w pełni zależy od zawiązującej go encji.

Hibernate definiuje bogaty zestaw wbudowanych typów odwzorowań. Gdy predefiniowane typy nie wystarczają, można wprowadzić własne typy lub odwzorowania komponentów, a nawet zaimplementować dowolną konwersję typu Javy na typ SQL.

Właściwości będące kolekcjami traktuje się jako typ wartości. Kolekcja nie ma własnej tożsamości i należy do jednej przechowującej ją encji. Przedstawiliśmy różne sposoby odwzorowywania kolekcji, włączając w to kolekcje typów wartości i kolekcje encji.

Hibernate obsługuje asocjacje jeden-do-jednego, jeden-do-wielu i wiele-do-wielu między encjami. Zalecamy unikać ostatniego z wymienionych rodzajów asocjacji. Asocjacje Hibernate są polimorficzne. Opisaliśmy dodatkowo różne aspekty tworzenia związków dwukierunkowych.

Wydajne pobieranie obiektów

W rozdziale:

- ◆ Cechy zapytania Hibernate
- ◆ Język HQL, kryteria i standardowe zapytania SQL
- ◆ Zapytania złożone, raportujące i dynamiczne
- ◆ Optymalizacja sprowadzania danych i zapytań w trakcie działania aplikacji

Zapytania to najbardziej interesująca część pisania dobrego kodu dostępu do danych. Złożone zapytanie wymaga długiego czasu obróbki, aż doprowadzone zostanie do najlepszej postaci. Jego wpływ na wydajność aplikacji niejednokrotnie jest ogromny. Z drugiej strony, im większe ma się doświadczenie w pisaniu zapytań, tym łatwiej się je tworzy. To, co początkowo wydaje się trudne, niejednokrotnie staje się proste po poznaniu niektórych bardziej zaawansowanych cech Hibernate.

Gdy wcześniej przez wiele lat tworzyłeś zapytania SQL, możesz martwić się zmniejszeniem elastyczności i ekspresyjności tych działań w ORM. W Hibernate tak nie jest.

Rozbudowany język zapytań w Hibernate pozwala wyrazić niemalże dokładnie to samo, co niejednokrotnie trzeba było wyrazić w SQL, ale w sposób obiektowy — używając klas i ich właściwości. Oczywiście pewne zadania nie mają sensu w zapytaniach obiektowych. Na przykład zapytania Hibernate nie obsługują podpowiedzi dostępnych w niektórych bazach danych. W tych rzadkich przypadkach nic nie stoi na przeszkodzie, by w Hibernate skorzystać z zapytań SQL specyficznych dla wykorzystywanego rodzaju bazy danych.

W rozdziale 4. wspomnieliśmy, że istnieją trzy sposoby określania zapytań w Hibernate. Pierwszy sposób to zapytania HQL:

```
session.createQuery("from Category c where c.name like 'Laptop%'");
```

Interfejs programistyczny kryteriów dopuszcza tworzenie **zapytań przez kryteria (QBC)** lub **zapytań przez przykład (QBE)**.

```
session.createCriteria(Category.class)
    .add(Expression.like("name", "Laptop%"));
```

Ostatni ze sposobów to bezpośrednie zapytania SQL z automatycznym odwzorowaniem zbiorów wyników na obiekty:

```
session.createSQLQuery("select {c.*} from CATEGORY {c} where NAME like
    'Laptop%', 
    "C",
    Category.class);
```

Niniejszy rozdział opisuje wszystkie trzy sposoby. Warto traktować niniejszy rozdział jako podręcznik — niektóre podrozdziały są pisane zwięźle, ale zawierają wiele przykładowych zapytań dla różnych sytuacji. Niejednokrotnie pomijamy optymalizacje mogące mieć zastosowanie w aplikacji CaveatEmptor, by zwiększyć czytelność kodu. Zamiast wykorzystywać wygodniejszą klasę MonetaryAmount, używamy kwot umieszczanych w klasie BigDecimal.

Najpierw zajmijmy się przedstawieniem sposobu wykonywania zapytań. Nie trzeba na razie skupiać się na samych zapytaniach — ich znaczenie wyjaśniamy w dalszej części rozdziału.

7.1. Wykonywanie zapytań

Interfejsy Query i Criteria definiują kilka metod sterowania wykonywaniem zapytania. Dodatkowo interfejs Query ma metody dowiezywania konkretnych wartości do parametrów zapytania. Aby w aplikacji wykonać zapytanie, należy pobrać jeden z egzemplarzy interfejsu, wykorzystując obiekt Session.

7.1.1. Interfejsy zapytań

Aby utworzyć nowy egzemplarz Query, wywołaj metodę `createQuery()` lub `createSQLQuery()`. Pierwsza z metod dotyczy zapytań HQL.

```
Query hqlQuery = session.createQuery("from User");
```

Metoda `createSQLQuery()` pozwala tworzyć zapytania SQL z zastosowaniem specyficznych rozwiązań wykorzystywanego systemu bazodanowego.

```
Query sqlQuery = session.createSQLQuery("select {u.*} from USERS {u}", "u", User.class);
```

W obu przypadkach Hibernate zwraca nowo utworzony obiekt Query, który może służyć do ustalenia konkretnego trybu wykonania zapytania oraz właściwego wykonania zapytania.

Aby pobrać egzemplarz Criteria, wywołaj metodę `createCriteria()`, przekazując klasę obiektów, które mają zostać zwrócone. Jest to tak zwana **encja korzeniowa** zapytania opartego o kryteria. Dla klasy User wywołanie ma postać:

```
Criteria crit = session.createCriteria(User.class);
```

Egzemplarza Criteria używa się w ten sam sposób co egzemplarza Query. Dodatkowo może on służyć do konstrukcji obiektowej reprezentacji zapytania przez dodawanie egzemplarzy `Criterion` i dołączanie asocjacji do innych obiektów Criteria. Szczegóły pojawią się w dalszej części rozdziału. Na razie kontynuujmy sposoby wykonywania zapytań.

Stronicowanie wyników

Stronicowanie to bardzo popularna technika zwracana wyników. Użytkownicy uzyskują wyniki wyszukiwania (na przykład konkretnych przedmiotów) podzielone na strony zawierające za jednym razem ograniczoną liczbę wierszy danych (na przykład 10). Użytkownik ręcznie przechodzi między kolejnymi i poprzednimi stronami. Oba interfejsy, Query i Criteria, obsługują podział wyników na strony.

```
Query query = session.createQuery("from User u order by u.name asc");
query.setFirstResult(0);
query.setMaxResults(10);
```

Wywołanie metody `setMaxResults(10)` powoduje ograniczenie zbioru wyników do pierwszych 10 obiektów pobranych z bazy danych. W przykładowym zapytaniu przez kryteria pobierana strona rozpoczyna się w środku zbioru wyników.

```
Criteria crit = session.createCriteria(User.class);
crit.addOrder(Order.asc("name"));
crit.setFirstResult(40);
crit.setMaxResults(20);
List results = crit.list();
```

Zapytanie pobiera 20 obiektów, zaczynając od czterdziestego. W języku SQL nie istnieje standardowy sposób określania podziału wyników na strony. Na szczęście Hibernate „wie”, jak wydajnie wykonać to zadanie dla konkretnego typu bazy danych.

Można wykorzystać **łańcuchowe wykonywanie metod** (dla metod zwracających obiekty zamiast typu void) zarówno dla interfejsu Query, jak i Criteria. Poprzednie dwa przykłady można zapisać następująco:

```
List results = session.createQuery("from User u order by u.name asc")
    .setFirstResult(0)
    .setMaxResults(10)
    .list();

List results = session.createCriteria(User.class)
    .addOrder(Order.asc("name"))
    .setFirstResult(40)
    .setMaxResults(20)
    .list();
```

Łańcuchowe wywoływanie metod zmniejsza długość kodu. Obsługuje je wiele interfejsów programistycznych Hibernate.

Pobieranie listy wyników i iteracja przez nie

Metoda `list()` wykonuje zapytanie i zwraca wyniki jako listę.

```
List results = session.createQuery("from User").list();
```

W pewnych zapytaniach mamy pewność, że wynikiem będzie tylko jeden obiekt — na przykład pobieramy tylko ofertę o najwyższej kwocie. Ze zbioru wyników można ją odczytać, używając metody `result.get(0)` i ustawiając wcześniej metodą `setMaxResults(1)` liczbę wyników na 1. Zapytanie można wykonać metodą `uniqueResult()`, bo wiemy, że istnieje co najwyżej jeden wynik.

```
Bid maxBid = (Bid) session.createQuery("from Bid b order by b.amount desc")
    .setMaxResults(1)
    .uniqueResult();

Bid bid = (Bid) session.createCriteria(Bid.class)
    .add(Expression.eq("id", id))
    .uniqueResult();
```

Gdyby zapytanie zwróciło więcej niż jeden wynik, Hibernate zgłosi wyjątek.

Interfejsy Query i Session zapewniają również metodę `iterate()`, która zwraca ten sam zbiór wyników co metoda `list()` (albo `find()`), ale korzysta z innej strategii sprowadzania danych. Metoda `iterate()` w pierwszym zapytaniu SQL pobiera jedynie wartości kluczy głównych (identyfikatorów). Następnie stara się odnaleźć stan pobieranych obiektów w buforze przez wykonaniem ponownego zapytania

pobierającego pozostałe wartości właściwości. Technika ta pozwala zoptymalizować pobieranie danych w niektórych szczególnych sytuacjach. Zostały one wskazane w podrozdziale 7.6.

Odpowiedzi na pytania Czy metoda `Session.find()` jest szybsza od metody `Query.list()`? Interfejs `Session` zawiera skróty do kilku najczęściej wykonywanych metod. Zamiast tworzyć obiekt `Query`, można od razu wywołać metodę `Session.find("from User")`. Wynik jest dokładnie ten sam jak w metodzie `Query.list()`. Obie metody działają równie szybko. To samo dotyczy metody `iterate()`. Samemu można wybrać preferowane rozwiązanie. Możliwe, że w przyszłych wersjach Hibernate skrótowe metody interfejsu `Session` zostaną oznaczone jako wycofane, by umożliwić ich usunięcie. Zalecamy stosowanie interfejsu `Query`.

Interfejs `Query` zawiera metody pozwalające dowiązywać wartości do parametrów zapytania.

7.1.2. Dowiązywanie parametrów

Oto przykład kodu, którego nigdy nie należy pisać:

```
String queryString = "from Item i where i.description like '"  
    + searchString + "'";  
List result = session.createQuery(queryString).list();
```

Jednym z powodów, dla których nigdy nie należy pisać tego rodzaju kodu, jest możliwość wprowadzenia przez złośliwego użytkownika w tekście wyszukiwania (obiekt `searchString`) tekstu typu:

```
'foo' and callSomeStoredProcedure() and 'bar' = 'bar'
```

Nietrudno zauważyć, że proste wyszukanie nie jest już takie proste — dodatkowo wywołuje w bazie danych procedurę zapamiętaną! Znaki apostrofów nie zostaną odpowiednio zamienione, więc baza danych potraktuje wywołanie procedury jako element zapytania. Tworzenie zapytań w przedstawiony powyżej sposób otwiera poważną lukę w bezpieczeństwie aplikacji, gdyż dopuszcza wykonanie w bazie danych dowolnego kodu. Użytkownicy nawet całkowicie przypadkowo mogą zaszkodzić aplikacji, umieszczając w wyszukiwanym tekście apostrof. Nigdy nie przekazuj do bazy danych niesprawdzonych wartości uzyskanych od użytkownika! Wystarczy użyć prostego mechanizmu, by zapobiec wspomnianej sytuacji.

Sterownik JDBC zawiera funkcje zapewniające bezpieczny sposób dowiązywania wartości do parametrów SQL. Wie dokładnie, które znaki specjalne trzeba zmienić, więc luka w bezpieczeństwie w ogóle się nie pojawi. W przypadku `searchString` sterownik JDBC automatycznie zniesie szczególne znaczenie apostrofów w tekście — nie będą traktowane jako znaki specjalne.

Co więcej, stosując parametry, dajemy bazie danych możliwość lepszego buforowania zapytań, co niejednokrotnie znacząco zwiększa wydajność.

Istnieją dwa sposoby dowiązywania parametrów: parametry pozycyjne i parametry nazwane. Hibernate obsługuje zarówno znane z JDBC parametry pozycyjne

(oznaczane jako ? w kodzie zapytania), jak i parametry nazywane (poprzedzane znakiem dwukropka).

Użycie parametrów nazwanych

Używając parametrów nazwanych, zmieńmy wcześniejsze zapytanie w następujący sposób:

```
String queryString = "from Item i where i.description like :searchString";
```

Dwukropki przed nazwą parametru wskazują, że jest to parametr nazwany. Następnie trzeba wykorzystać interfejs Query do dowiązania wartości do parametru searchString.

```
List result = session.createQuery(queryString)
    .setString("searchString", searchString)
    .list();
```

Ponieważ searchString jest wartością tekstową przekazywaną przez użytkownika, używamy metody setString() interfejsu Query, by powiązać ją z parametrem searchString. Kod stał się czytelniejszy, znacznie bezpieczniejszy, a samo zapytanie można wykorzystać wielokrotnie, jeśli tylko zmienia się zawartość dowiązanego parametru.

Często stosuje się wiele parametrów.

```
String queryString = "from Item i "
    + "where i.description like :searchString"
    + "and item.date > :minDate";
List result = session.createQuery(queryString)
    .setString("searchString", searchString)
    .setDate("minDate", minDate)
    .list();
```

Użycie parametrów pozycyjnych

Nic nie stoi na przeszkodzie, by skorzystać z parametrów pozycyjnych:

```
String queryString = "from Item i "
    + "where i.description like ?"
    + "and item.date > ?";
List result = session.createQuery(queryString)
    .setString(0, searchString)
    .setDate(1, minDate)
    .list();
```

Niestety, to podejście jest mniej czytelne od poprzedniego. Co gorsza, łatwiej uszkodzić kod z takim zapisem, na przykład przy zmianie treści zapytania. Wystarczy poniższa modyfikacja, by wszystko przestało działać:

```
String queryString = "from Item i "
    + "and item.date > ?"
    + "where i.description like ?";
```

Każda zmiana położenia dowiązanych parametrów wymusza zmianę kodu dowiązującego. Prowadzi to do kodu bardzo czułego na zmiany. Zalecamy w nowym kodzie unikać parametrów pozycyjnych.

Warto wskazać, że parametr nazwany może w zapytaniu pojawić się kilku-krotnie bez potrzeby jego wielokrotnego dowiązywania.

```
String userString = "from User u where u.username like :searchString"  
+ " or u.email like :searchString";  
List result = session.createQuery(userString)  
.setString("searchString", searchString)  
.list();
```

Dowiązywanie dowolnych argumentów

W przykładach użyliśmy metod `setString()` i `setDate()` do dowiązania argumentów. Interfejs `Query` zawiera wiele innych metod dowiązujących argumenty dotyczące wbudowanych typów Hibernate: od `setInteger()` przez `setTimestamp()` po `setLocale()`.

Szczególnie użyteczna okazuje się metoda `setEntity()`, która dowiązuje trwałąencję.

```
session.createQuery("from Item item where item.seller = :seller")  
.setEntity("seller", seller)  
.list();
```

Istnieje metoda ogólna dowiązująca argument dowolnego typu Hibernate.

```
String queryString = "from Item item where item.seller = :seller and "  
+ "item.description like :desc";  
  
session.createQuery(queryString)  
.setParameter("seller", seller, Hibernate.entity(User.class))  
.setParameter("desc", description, Hibernate.STRING)  
.list();
```

Przedstawiony sposób działa również dla typów zdefiniowanych przez użytkownika, na przykład dla `MonetaryAmount`.

```
Query q = session.createQuery("from Bid bid where bid.amount > :amount");  
q.setParameter("amount", givenAmount,  
Hibernate.Custom(MonetaryAmountUserType.class));  
List result = q.list();
```

W pewnych sytuacjach Hibernate potrafi odgadnąć typ na podstawie klasy parametru wartości. W takiej sytuacji nie trzeba jawnie określać typu Hibernate.

```
String queryString = "from Item item where item.seller = :seller and "  
+ "item.description like :desc";  
  
session.createQuery(queryString)  
.setParameter("seller", seller)  
.setParameter("desc", description)  
.list();
```

Zauważ, że automatyczne rozpoznawanie działa również dla takich klas jak seller. Powyższe podejście można stosować dla typów Integer, String i Boolean, ale nie dla typów takich jak Date, które można dowiązywać do typów Hibernate timestamp, date i time. W takiej sytuacji konieczne okazuje się użycie odpowiedniej dla typu metody dowiązania lub zastosowanie trzeciego argumentu w metodzie setParameter().

Jeżeli mamy do czynienia z JavaBean zawierającym parametry seller i description, możemy użyć metody setProperties(), by dowiązać parametry. W takiej sytuacji wartości przekazuje się wewnątrz obiektu odpowiedniego typu.

```
Item item = new Item();
item.setSeller(seller);
item.setDescription(description);

String queryString = "from Item item "
+ "where item.seller=:seller and "
+ "item.description like :desc";

session.createQuery(queryString).setProperties(item).list();
```

Metoda setProperties() stara się dopasować nazwy parametrów nazwanych do nazw właściwości obiektu JavaBeans, by w ten sposób poznać typ Hibernate i dowiązać wartość. W praktyce podejście to okazuje się mniej przydatne niż się początkowo wydaje, gdyż nie można automatycznie odgadnąć niektórych typów (na przykład dat).

Metody dowiązania parametrów interfejsu Query są bezpieczne pod kątem wartości null, więc poniższy kod działa bez zgłaszania błędów:

```
session.createQuery("from User as u where u.username = :name")
.setString("name",null)
.list();
```

Niestety, najczęściej wynik działania przedstawionego przykładu będzie inny od oczekiwanej. Wynikowy kod SQL będzie zawierał porównanie typu username = null, które w terminologii SQL zawsze będzie równe null. By uzyskać poprawny efekt, należy zastosować operator is null.

```
session.createQuery("from User as u where u.email is null").list();
```

Do tej pory wszystkie przykłady zawierały osadzone w kodzie zapytania HQL. To podejście jest poprawne dla krótkich zapytań, ale gdy tworzy się złożone zapytania rozkładające się na wiele wierszy, umieszczanie ich bezpośrednio w kodzie Java staje się mało wygodne.

7.1.3. Zapytania nazwane

Umieszczanie wielowierszowych zapytań HQL bezpośrednio w kodzie Javy nie jest dobrym pomysłem. Hibernate dopuszcza umieszczanie treści zapytań w metadanych odwzorowań — umieszczane tam zapytania określa się mianem zapytań nazwanych. Nic nie stoi na przeszkodzie, by wraz z danymi odwzorowania w jed-

nym pliku XML umieścić również zapytania wykorzystywane przez trwałą klasę. W celu pobrania zapytania do aplikacji, podaje się jego nazwę.

Metoda `getNamedQuery()` pobiera egzemplarz `Query` dla nazwanego zapytania.

```
session.getNamedQuery("findItemsByDescription")
    .setString("description", description)
    .list();
```

W niniejszym przykładzie wykonujemy nazwane zapytanie `findItemsByDescription`, ale wcześniej dowiązujemy do niego nazwany parametr. Zapytanie nazwane znajduje się w metadanych odwzorowania w elemencie `<query>`.

```
<query name="findItemsByDescription"><![CDATA[
    from Item item where item.description like :description
]]></query>
```

Nazwane zapytania nie muszą być zapytaniami HQL; równie dobrze mogą to być zwykłe zapytania SQL — co ważne, kod Javy nie będzie widział między nimi żadnej różnicy.

```
<sql-query name="findItemsByDescription"><![CDATA[
    select {i.*} from ITEM {i} where DESCRIPTION like :description
]]>
<return alias="i" class="Item"/>
</sql-query>
```

Warto to tym pamiętać, jeśli zechce się w przyszłości optymalizować zapytania HQL, konwertując je do zapytań SQL. Przedstawione podejście przydaje się również wtedy, gdy trzeba przenieść starszą aplikację do Hibernate, a zapytania SQL uzyskuje się ze starszego kodu stosującego JDBC. Dzięki zapytaniom nazwanym można łatwo przenieść te zapytania do plików odwzorowań.

Tworzeniem zapytań SQL zajmiemy się jeszcze w dalszej części rozdziału. Powróćmy na razie do języka HQL i kryteriów.

7.2. Proste zapytania dotyczące obiektów

By poznać składnię i semantykę HQL, zacznijmy od prostych zapytań. Choć będziemy prezentować alternatywny zapis przy użyciu kryteriów, pamiętaj, że zalecanym sposobem określania złożonych zapytań jest język HQL. Znając zapytanie HQL, stosunkowo łatwo przetworzyć je na zestaw kryteriów — droga w drugą stronę jest znacznie bardziej skomplikowana.

Uwaga

Do testowania zapytań Hibernate ad hoc można skorzystać z dwóch narzędzi: aplikacji Java wykorzystującej Swing o nazwie **Hibern8IDE** i modułu Eclipse o nazwie **Hibernator**. Oba narzędzia pozwalają wybierać dokumenty odwzorowania Hibernate, łączyć się z bazą danych i przeglądać wyniki wykonywanych interaktywnie zapytań. Hibern8IDE zapewnia nawet prototypowanie zapytań przez kryteria, wykorzystując JavaBean Shell. Łączy do obu narzędzi znajdują się na witrynie Hibernate.

7.2.1. Najprostsze zapytanie

Najprostsze zapytanie pobiera wszystkie egzemplarze konkretnej klasy trwałej. W HQL ma ono postać:

```
from Bid
```

W zapytaniu przez kryteria ma postać:

```
session.createCriteria(Bid.class);
```

Oba zapytania generują następujący kod SQL:

```
select B.BID_ID, B.AMOUNT, B.ITEM_ID, b.CREATED from BID B
```

Nawet w tym prostym przykładzie widać, że język HQL jest znacznie krótszy w zapisie od kodu SQL.

7.2.2. Zastosowanie aliasów

Najczęściej, gdy w HQL tworzy się zapytanie dla klasy, trzeba podać alias dla tej klasy pozwalający odnieść się do niej w dalszej części zapytania.

```
from Bid as bid
```

Słowo kluczowe as jest opcjonalne, więc poniższy zapis okazuje się równoważny wcześniejszemu:

```
from Bid bid
```

Alias warto traktować w podobny sposób jak deklarację zmiennej tymczasowej w poniższym kodzie języka Java:

```
for (Iterator i = allQueriedBids.iterator(); i.hasNext(); ) {
    Bid bid = (Bid) i.next();
    ...
}
```

Zastosowaliśmy alias `bid` dla egzemplarzy klasy `Bid`, by na jego podstawie móc odnosić się do wartości właściwości tych egzemplarzy w dalszej części kodu. Aby przypominać sobie o podobieństwie, zalecamy wykorzystywanie tej samej konwencji nazewnictwa dla aliasów co dla zmiennych tymczasowych (najczęściej zapis-Wielbłędowy). W książce, by kod był krótki i czytelny, najczęściej zamiast pełnych nazw elementów stosujemy jednoliterowy skróty.

Uwaga

Słów kluczowych HQL nigdy nie piszemy wielkimi literami. Podobnie postępujemy ze słowami kluczowymi SQL. Wielkie litery wyglądają brzydko, a chyba wszystkie obecnie stosowane terminale dobrze radzą sobie z wielkimi i małymi literami. HQL nie zwraca uwagi na wielkość liter, więc nic nie stoi na przeszkodzie, by napisać `FROM Bid as bid`.

Zapytania przez kryteria wykorzystują aliasy niejawne. Encja korzeniowa kryterium zawsze otrzymuje alias `this`. Zajmiemy się tym tematem nieco dokładniej,

gdy opiszemy tworzenie złączeń kryteriów. Gdy stosuje się kryteria, najczęściej w ogóle nie trzeba myśleć o aliasach.

7.2.3. Zapytania polimorficzne

Wskazaliśmy, że HQL jest obiektowym językiem zapytań, więc powinien obsługiwać **zapytania polimorficzne**, czyli zapytania obejmujące swym zasięgiem egzemplarze klasy bazowej i wszystkich jej podklas. Przedstawiliśmy na tyle dużo elementów języka HQL, by wprowadzić polimorfizm. Rozważmy zapytanie:

```
from BillingDetails
```

Zapytanie zwraca obiekty typu `BillingDetails` (klasa abstrakcyjna). W przedstawionym przykładzie klasy konkretne będą podklasami `BillingDetails`: `CreditCard` i `BankAccount`. Jeśli chcemy pobrać egzemplarze konkretnej podklasy, musimy napisać:

```
from CreditCard
```

Klasa wymieniona w klauzuli `from` nie musi być odwzorowaną klasą trwałą — może być dowolną klasą. Poniższe zapytanie zwróci wszystkie obiekty trwałe:

```
from java.lang.Object
```

Nic nie stoi na przeszkodzie, by zamiast klasy wskazać interfejs. Kolejny przykład zwraca wszystkie trwałe obiekty serializowalne.

```
from java.io.Serializable
```

Zapytania przez kryterium również obsługują polimorfizm.

```
session.createCriteria(BillingDetails.class).list();
```

Zapytanie zwraca egzemplarze `BillingDetails` i wszystkich jej podklas. Następny fragment kodu zwraca wszystkie obiekty trwałe.

```
session.createCriteria(java.lang.Object.class).list();
```

Polimorfizm dotyczy nie tylko klas jawnie wymienionych w klauzuli `from`, ale również asocjacji polimorficznych, o których wkrótce powiemy coś więcej.

Po przyjrzeniu się klauzuli `from` przejdźmy do innych elementów języka HQL.

7.2.4. Ograniczenia

Najczęściej nie zamierzamy pobierać wszystkich egzemplarzy klasy. Musimy w jakiś sposób przekazać informację o właściwościach interesujących nas obiektów. Mówimy wtedy o **ograniczeniach**. Klauzula `where` służy do definiowania ograniczeń w języku SQL i HQL. Wyrażenia mogą mieć dowolną złożoność. Zaczniijmy od czegoś prostego:

```
from User u where u.email = 'foo@hibernate.org'
```

Zauważ wskazanie ograniczenia dla konkretnej właściwości, email, klasy User w sposób zorientowany obiektowo. Podobnie jak w Javie `u.email` nie można skrócić do samego email.

W zapytaniach przez kryterium należy utworzyć obiekt Criterion wyrażający ograniczenie. Klasa Expression zawiera metody fabryczne do tworzenia obiektów Criterion. Utwórzmy wcześniejsze zapytanie, stosując kryteria, i od razu je wykonajmy.

```
Criterion emailEq = Expression.eq("email", "foo@hibernate.org");
Criteria crit = session.createCriteria(User.class);
crit.add(emailEq);
User user = (User) crit.uniqueResult();
```

Utworzony obiekt Criterion zawiera proste wyrażenie (Expression) sprawdzające równość elementów. Dodajemy obiekt do kryterium. Metoda uniqueResult() wykonuje zapytanie i jako wynik zwraca dokładnie jeden obiekt.

Oczywiście powyższy kod można zapisać w skróconej formie, stosując łańcuchowanie metod.

```
User user = (User) session.createCriteria(User.class)
    .add(Expression.eq("email", "foo@hibernate.org"))
    .uniqueResult();
```

Nowym elementem Javy wprowadzonym w JDK 1.5 jest import statyczny. Można w ten sposób skrócić zapis niektórych fragmentów kodu. Dodając wcześniej wiersz:

```
static import net.sf.hibernate.Expression.*;
```

możemy jeszcze bardziej skrócić kod zapytania przez kryterium:

```
User user = (User) session.createCriteria(User.class)
    .add(eq("email", "foo@hibernate.org"))
    .uniqueResult();
```

Kod SQL wygenerowany we wszystkich trzech zapytaniach jest taki sam i ma postać:

```
select U.USER_ID, U.FIRSTNAME, U.LASTNAME, U.USERNAME, U.EMAIL
from USER U
where U.EMAIL = 'foo@hibernate.org'
```

Język HQL obsługuje wszystkie inne typowe operatory porównujące.

7.2.5. Operatory porównania

Ograniczenia wyraża się w logice trójstanowej. Klauzula where jest wyrażeniem logicznym wyliczanym na wartości true, false lub null dla każdej krotki obiektów. Klauzula zawiera wyrażenia logiczne porównujące właściwości obiektów z właściwościami innych obiektów lub stałymi podanymi w zapytaniu HQL.

Odpowiedzi na pytania	Czym jest logika trójstanowa? Wiersz zostaje dołączony do zbioru wyników tylko wtedy, gdy klauzula where zwróci wartość true. W Javie obiekt-NierownyNull==null zwraca false, natomiast null==null zwraca true. W SQL zarówno KOLUMN_A_NIEROWNA_NULL=null, jak i null=null zwracają
------------------------------	---

null zamiast true. Z tego powodu SQL wymaga osobnego operatora, IS NULL, do testowania wartości null. Logika trójstanowa zapewnia obsługę wyrażeń, które mogą być użyte dla kolumn o wartości null. Istnieje (niekoniecznie potrzebne) rozszerzenie SQL tworzące logikę binarną w modelu relacyjnym, by uzyskać efekt podobny do tego z języka Java.

HQL obsługuje te same podstawowe operatory co SQL: =, <>, <, >, >=, <=, between, not between, in i not in. Oto kilka przykładów:

```
from Bid bid where bid.amount between 1 and 10  
from Bid bid where bid.amount > 100  
from User u where u.email in ("foo@hibernate.org", "bar@hibernate.org")
```

W zapytaniach przez kryteria wszystkie wymienione operatory są dostępne w klasie Expression.

```
session.createCriteria(Bid.class)  
    .add(Expression.between("amount", new BigDecimal(1), new BigDecimal(10)))  
    .list();  
  
session.createCriteria(Bid.class)  
    .add(Expression.gt("amount", new BigDecimal(100)))  
    .list();  
  
String[] emails = { "foo@hibernate.org", "bar@hibernate.org" };  
session.createCriteria(User.class)  
    .add(Expression.in("email", emails))  
    .list();
```

Ponieważ relacyjna baza danych wykorzystuje logikę trójstanową, testowanie wartości null wymaga dodatkowych zabiegów. Pamiętaj, że w bazie danych null = null nie zwraca wartości true, ale null. W zasadzie wszystkie porównania używające operatora null zwracają wartość null. Zarówno HQL, jak i kryteria, pozwalają użyć w ograniczeniach operatora is null.

```
from User u where u.email is null
```

Zapytanie zwraca wszystkich użytkowników bez adresu e-mail. Podobną znaczeniowo składnię udostępnia interfejs Criteria.

```
session.createCriteria(User.class)  
    .add(Expression.isNull("email"))  
    .list();
```

Możemy zechcieć zwrócić wszystkich użytkowników posiadających adres e-mail:

```
from User u where u.email is not null
```

```
session.createCriteria(User.class)  
    .add(Expression.isNotNull("email"))  
    .list();
```

Klauzula where z HQL obsługuje wyrażenia arytmetyczne, których nie można wprowadzić w kryteriach.

```
from Bid bid where (bid.amount / 0.71) - 100.0 > 0.0
```

Wyszukiwanie bazujące na tekście wymaga nieco innego podejścia, szczególnie jeśli chce się sprawdzać jedynie fragmenty tekstu w wyrażeniu ograniczającym.

7.2.6. Dopasowywanie tekstów

Operator `like` umożliwia wyszukiwanie z użyciem znaków wieloznacznych. Podobnie jak w SQL są to znaki `%`, `_` i `^`.

```
from User u where u.firstname like "G%"
```

Wyrażenie ogranicza wyniki do użytkowników mających imię rozpoczynające się od wielkiej litery G. Można także zanegować operator lub wyszukiwać fragmentu nieznajdującego się na początku tekstu.

```
from User u where u.firstname not like "%Foo Bar%"
```

W zapytaniach przez kryteria wyszukiwanie może albo stosować te same znaki wieloznaczne, albo używać klasy `MatchMode`. Klasa `MatchMode` stanowi element interfejsu `Criteria`. Służy do tworzenia wyrażeń dopasowujących fragmenty tekstu bez potrzeby edycji tych fragmentów w celu dostosowania do zasad wprowadzania wieloznaczności. Dwa poniższe rozwiązania są sobie równoznaczne:

```
session.createCriteria(User.class)
    .add(Expression.like("firstname", "G%"))
    .list();

session.createCriteria(User.class)
    .add(Expression.like("firstname", "G", MatchMode.START))
    .list();
```

Dostępne w `MatchMode` stałe to: `START`, `END`, `ANYWHERE` i `EXACT`.

Wyjątkowo użyteczną zaletą HQL jest możliwość wywoływania w klauzuli `where` dowolnej funkcji SQL. Jeśli baza danych obsługuje funkcje definiowane przez użytkownika (większość dopuszcza ich tworzenie), nic nie stoi na przeszkodzie, by z nich skorzystać — zarówno w dobrych, jak i złych celach. Na razie rozważmy użyteczność standardowych funkcji SQL o nazwach `upper()` i `lower()`. Dzięki nim można wprowadzić wyszukiwanie nieuwzględniające wielkości liter.

```
from User u where lower(u.email) = 'foo@hibernate.org'
```

Interfejs `Criteria` z Hibernate 2.1 nie dopuszcza wywoływania funkcji SQL. Umożliwia jednak wyszukiwanie bez rozróżniania wielkości liter.

```
session.createCriteria(User.class)
    .add(Expression.eq("email", "foo@hibernate.org").ignoreCase())
    .list();
```

Niestety, HQL nie wprowadza własnego, standardowego operatora łączenia fragmentów tekstu; stosuje operator wykorzystywany przez używany system bazodanowy. Większość baz danych dopuszcza poniższą składnię:

```
from User user
    where (user.firstname || ' ' || user.lastname) like 'G% K%'
```

Do niektórych bardziej egzotycznych elementów klauzuli where z HQL powrócimy jeszcze w tym rozdziale. Na razie wyrażenia ograniczające zawierały tylko jeden element. Połączmy kilka elementów, używając operatorów logicznych.

7.2.7. Operatory logiczne

Operatory logiczne (i nawiasy grupujące) zapewniają tworzenie złożonych wyrażeń typu:

```
from User user
    where user.firstname like 'G%' and user.lastname like 'K%'

from User user
    where (user.firstname like 'G%' and user.lastname like 'K%')
        or user.email in ("foo@hibernate.org", "bar@hibernate.org")
```

Gdy do obiektu Criteria dodaje się kilka obiektów Criterion, są traktowane jako koniunkcja (odpowiadająca operatorowi and).

```
session.createCriteria(User.class)
    .add(Expression.like("firstname", "G%"))
    .add(Expression.like("lastname", "K%"))
```

Istnieją dwa sposoby zastosowania alternatywy (operatora or). Pierwszy wymaga użycia Expression.or() wraz z Expression.and().

```
Criteria crit = session.createCriteria(User.class)
    .add(
        Expression.or(
            Expression.and(
                Expression.like("firstname", "G%"),
                Expression.like("lastname", "K%")
            ),
            Expression.in("email", emails)
        )
    );
```

Drugie podejście wymaga użycia metod Expression.disjunction() i Expression.conjunction().

```
Criteria crit = session.createCriteria(User.class)
    .add(Expression.disjunction()
        .add(Expression.conjunction()
            .add(Expression.like("firstname", "G%"))
            .add(Expression.like("lastname", "K%"))
        )
        .add(Expression.in("email", emails))
    );
```

Wydaje nam się, że oba rozwiązania są naprawdę brzydkie, nawet pomimo spełdzenia kilku minut na ich odpowiednim formatowaniu. Statyczny import z JDK 1.5 mógłby poprawić czytelność. Niemniej, jeśli nie tworzy się zapytania w trakcie pracy aplikacji, warto korzystać z bardziej przejrzystych zapytań HQL. Złożone zapytania przez kryteria mają sens tylko wtedy, gdy tworzy się je programowo: na

przykład w sytuacji, gdy istnieje ekran wyszukiwania ze złożonymi opcjami — wtedy klasa CriteriaBuilder zamieniająca ograniczenia na obiekty Criteria byłaby przydatna.

7.2.8. Kolejność wyników zapytań

Wszystkie języki zapytań dopuszczają określenie kolejności (sortowanie) wyników zapytań. HQL podobnie jak SQL stosuje w tym celu klauzulę `order by`.

Poniższe zapytanie zwraca wszystkich użytkowników posortowanych alfabetycznie względem nich nazw.

```
from User u order by u.username
```

Do określania sposobu sortowania (malejąco lub rosnąco) służą odpowiednio słowa `desc` i `asc`.

```
from User u order by u.username desc
```

Można sortować jednocześnie po kilku właściwościach.

```
from User u order by u.lastname asc, u.firstname asc
```

Zapytania przez kryteria również obsługują sortowanie wyników.

```
List results = session.createCriteria(User.class)
    .addOrder(Order.asc("lastname"))
    .addOrder(Order.asc("firstname"))
    .list();
```

Do tej pory zajmowaliśmy się jedynie prostymi zapytaniami HQL i kryteriami. Poznaliśmy klauzulę `from` i aliasy klas. Łączyliśmy różne ograniczenia operatorami logicznymi. Cały czas jednak operowaliśmy w obrębie jednej klasy trwałe — w klauzuli `from` zawsze istniała nazwa tylko jednej klasy. Nie zajęliśmy się jeszcze tematem **złączania asocjacji** w trakcie działania aplikacji.

7.3. Złączanie asocjacji

Złączenie scala dwie (lub więcej) relacji. Możemy na przykład złączyć dane zawarte w tabelach ITEM i BID (patrz rysunek 7.1). Rysunek nie przedstawia wszystkich kolumn tabel, stąd linie przerywane na ich końcach.

Większość osób, gdy słyszy słowo „złączenie” w kontekście baz danych SQL, myśli o złączeniu wewnętrznym. Jest to tylko jeden z możliwych rodzajów złączeń — najprostszy do zrozumienia. Rozważmy polecenie SQL i wyniki z rysunku 7.2. Przedstawione polecenie zapisano w stylu złączeń określonym w standardzie ANSI.

Gdy złączymy tabele ITEM i BID za pomocą wspólnego atrybutu z kolumny `ITEM_ID`, uzyskamy wszystkie przedmioty i ich oferty w jednej dużej tabeli. Zauważ, że wyniki zawierają tylko te przedmioty, dla których złożono choć jedną ofertę. Jeśli

ITEM

ITEM_ID	NAME	INITIAL_PRICE
1	Foo	2.00
2	Bar	50.00
3	Baz	1.00

BID

BID_ID	ITEM_ID	AMOUNT
1	1	10.00
2	1	20.00
3	2	55.50

Rysunek 7.1. Tabele ITEM i BID to oczywiście kandydaci do operacji złączenia

```
from ITEM I inner join BID B on I.ITEM_ID = B.ITEM_ID
```

ITEM_ID	NAME	INITIAL_PRICE	BID_ID	ITEM_ID	AMOUNT
1	Foo	2.00	1	1	10.00
1	Foo	2.00	2	1	20.00
2	Bar	50.00	3	2	55.50

Rysunek 7.2. Tabela wyników dla złączenia wewnętrznego dwóch tabel według standardu ANSI SQL

chcemy pobrać wszystkie przedmioty oraz ich oferty (i przy braku ofert dla przedmiotu zastosować wartości null), musimy użyć złączenia zewnętrznego (w tym przypadku **lewego**) przedstawionego na rysunku 7.3.

```
from ITEM I left outer join BID B on I.ITEM_ID = B.ITEM_ID
```

ITEM_ID	NAME	INITIAL_PRICE	BID_ID	ITEM_ID	AMOUNT
1	Foo	2.00	1	1	10.00
1	Foo	2.00	2	1	20.00
2	Bar	50.00	3	2	55.50
3	Baz	1.00	null	null	null

Rysunek 7.3. Wynik lewego złączenia zewnętrznego dwóch tabel według standardu ANSI SQL

Złączenie tabel działa w następujący sposób. Najpierw powstaje iloczyn kartezjański obu tabel tworzony na podstawie wszystkich możliwych kombinacji wierszy ITEM z wierszami BID. Następnie wyniki są filtrowane przy użyciu warunku **złączenia**. Bazy danych w rzeczywistości stosują bardziej wyrafinowane algorytmy złączeń, by nie musieć tworzyć w pamięci ogromnego ilocznego kartezjańskiego i na jego podstawie filtrować wszystkich wierszy. Warunek złączenia to zwykle wyrażenie — jeśli jego wartość wyniesie true, złączony wiersz należy umieścić w wynikach. W przypadku lewego złączenia zewnętrznego każdy wiersz tabeli ITEM (lewej), który nigdy nie dopasował się do żadnego warunku złączenia, również trafia do wyników. W takiej sytuacji wszystkie kolumny z tabeli BID otrzymują wartość null (prawie złączenie zewnętrzne pobrałoby wszystkie oferty, nawet jeśli nie dotyczą one żadnego przedmiotu — oczywiście w przedstawianym przykładzie takie złączenie nie ma sensu).

W SQL warunek złączenia najczęściej podaje się jawnie (niestety, nie jest możliwe użycie nazwy ograniczenia klucza obcego, by w ten sposób ułatwić określanie sposobu złączenia). Warunek złączenia określa się albo w klauzuli `on` (styl ANSI), albo też w klauzuli `where` (tak zwany **styl theta**): `where I.ITEM_ID = B.ITEM_ID`.

7.3.1. Złączenia w Hibernate

W Hibernate nie zawsze trzeba jawnie określać warunku złączenia. Wystarczy podać nazwę odwzorowanej asocjacji klas Javy. Na przykład klasa `Item` zawiera asocjację o nazwie `bids` zawierającą klasy `Bid`. Jeśli nazwiemy asocjację w zapytaniu, Hibernate będzie dysponował wystarczającą wiedzą, by poprawnie określić wyrażenie złączenia tabel. W ten sposób kod zapytania jest znacznie krótszy od odpowiadającego mu zapytania SQL.

HQL stosuje cztery sposoby wyrażania złączeń (zewnętrznych i wewnętrznych):

- ◆ typowe złączenie w klauzuli `from`,
- ◆ złączenie sprowadzające w klauzuli `from`,
- ◆ złączenie w stylu **theta** w klauzuli `where`,
- ◆ niejawne złączenie wynikające z asocjacji.

W dalszej części rozdziału przedstawimy, w jaki sposób tworzyć złączenia między klasami, które nie mają zdefiniowanej asocjacji (złączenie w stylu theta), oraz w jaki sposób pisać niejawne złączenia wynikające z asocjacji w klauzulach `where` lub `select` (lub `w group by`, `order by` i `having`). Najczęściej najkrótszą postać złączenia zapewnia złączenie w klauzuli `from` (typowe i sprowadzające), więc zajmiemy się nim w pierwszej kolejności. Pamiętaj, że semantyka złączeń HQL jest podobna do operacji złączeń z SQL, ale nie identyczna.

Hibernate rozróżnia nie rodzaj złączenia, ale jego powód. Przypuśćmy, że pobierany obiekt `Item`. Istnieją dwa powody, dla których możemy zechcieć dołączyć do wyników również obiekty `Bid`.

Mogliśmy chcieć ograniczyć zwracane obiekty `Item` na podstawie prostego kryterium opartego na danych znajdujących się we właściwościach `Bid`. Przykładowo chcemy pobrać wszystkie przedmioty, których przynajmniej jedna oferta wynosi co najmniej 100 dolarów. Wymaga to użycia złączenia wewnętrznego.

Z drugiej strony możemy być zainteresowani tylko obiektami `Item` i nie chcieć od razu pobierać wszystkich związanych z nimi ofert (kolekcja `bids` nie powinna zostać zainicjalizowana). Hibernate najpierw pobierze przedmioty. Oferty załaduje leniwie, jeśli program wykona metodę `item.getBids().iterator()`.

Trzecie podejście to załadowanie w złączeniu zewnętrznym wszystkich obiektów `Bid` razem ze wszystkimi obiektami `Item` w tym samym zapytaniu. Mówimy wtedy o **sprowadzaniu wyprzedzającym**. Pamiętaj, że lepiej jest domyślnie odwzorować wszystkie asocjacje jako leniwe, by sprowadzanie wyprzedzające było przeprowadzane dopiero po jawnej zmianie strategii sprowadzania.

Najpierw zajmijmy się ostatnią z wymienionych sytuacji.

7.3.2. Pobieranie asocjacji

W języku HQL sprowadzenie wyprzedzające danych asocjacji przez złączenie zewnętrzne wymusza się słowem kluczowym `fetch` po klauzuli `from`.

```
from Item item
left join fetch item.bids
where item.description like '%gc%'
```

Zapytanie w jednej instrukcji zwraca wszystkie przedmioty, które w opisie zawierają ciąg znaków gc, oraz wszystkie związane z nimi oferty. W ten sposób wszystkie obiekty `Item` uzyskane dzięki zapytaniu mają od razu zainicjalizowane kolekcje `bids`. Taki sposób zapisu klauzuli `from` nazywamy **złączeniem sprowadzającym**. Celem jego wprowadzenia jest zwiększenie wydajności — używamy tej składni, gdy wiemy, że w trakcie przetwarzania obiektów `Item` będziemy również korzystać z kolekcji `bids`.

To samo zadanie można wykonać za pomocą kryteriów.

```
session.createCriteria(Item.class)
.setFetchMode("bids", FetchMode.EAGER)
.add(Expression.like("description", "gc", MatchMode.ANYWHERE))
.list();
```

Oba zapytania spowodują utworzenie następującego polecenia SQL:

```
select I.DESCRIPTION, I.CREATED, I.SUCCESSFUL_BID, B.BID_ID,
B.AMOUNT, B.ITEM_ID, B.CREATED
from ITEM I
left outer join BID B on I.ITEM_ID = B.ITEM_ID
where I.DESCRIPTION like '%gc%'
```

Stosując tę samą składnię, można również z wyprzedzeniem pobrać asocjacje wiele-do-jednego i jeden-do-jednego.

```
from Bid bid
left join fetch bid.item
left join fetch bid.bidder
where bid.amount > 100

session.createCriteria(Bid.class)
.setFetchMode("item", FetchMode.EAGER)
.setFetchMode("bidder", FetchMode.EAGER)
.add(Expression.gt("amount", new BigDecimal(100)))
.list();
```

Oba zapytania spowodują utworzenie następującego polecenia SQL:

```
select I.DESCRIPTION, I.CREATED, I.SUCCESSFUL_BID, B.BID_ID,
B.AMOUNT, B.ITEM_ID, B.CREATED,
UL.USERNAME, U.PASSWORD, U.FIRSTNAME, U.LASTNAME
from BID B
left outer join ITEM I on I.ITEM_ID = B.ITEM_ID
left outer join USER U on U.USER_ID = B.BIDDER_ID
where B.AMOUNT > 1000
```

Słowo kluczowe `left` jest w języku HQL opcjonalne, więc powyższe przykłady mogą wykorzystywać tylko `join fetch`. Choć stosowanie przedstawionego rozwiązania wydaje się proste, warto pamiętać o kilku kwestiach.

- ◆ **HQL zawsze ignoruje ustawienia sprowadzającego (złączenie zewnętrzne) z dokumentu odwzorowania.** Jeśli odwzorowanie zawiera dla asocjacji atrybut `outer-join="true"`, dowolne zapytanie HQL zachowuje się tak, jakby go w ogóle nie było. By włączyć pobieranie wyprzedzające, trzeba jawnie wskazać je w zapytaniu. By było ciekawiej, zapytania przez kryteria nie ignorują ustawienia zawartego w odwzorowaniu! Wystąpienie atrybutu `outer-join="true"` spowoduje pobranie przez kryteria również danych asocjacji — podobnie jak czynią to metody `Session.get()` i `Session.load()` pobierające dane na podstawie identyfikatora. By w kryteriach jawnie wyłączyć sprowadzanie wyprzedzające, użyj metody `setFetchMode("bids", FetchMode.LAZY)`. HQL jest możliwie elastyczny, bo pozwala dowolnie zmienić strategię sprowadzania danych w trakcie działania aplikacji.
- ◆ **Hibernate 2.1 pozwala z wyprzedzeniem pobrać tylko jedną kolekcję.** To rozsądne ograniczenie, ponieważ pobieranie w jednym zapytaniu kilku kolekcji skutkowałoby wynikiem będącym iloczynem kartezjańskim. W kolejnych wersjach ograniczenie może zostanie zniesione, ale przed jego użyciem zalecamy mocno się zastanowić nad liczbą uzyskiwanych w ten sposób wyników. Liczba danych przesyłanych z bazy danych do aplikacji łatwo może osiągnąć wiele megabajtów, ale większość informacji od razu zostanie odrzucona (Hibernate spłaszcza tabelaryczny zbiór wyników, by wykonać graf obiektów). Można pobierać dowolną liczbę asocjacji wiele-do-jednego i jeden-do-jednego.
- ◆ **Po pobraniu kolekcji Hibernate nie zwraca różnych wierszy w zbiorze wyników.** Przykładowo, pojedynczy obiekt `Item` może kilkukrotnie pojawić się na liście wyników, jeśli wykonywało się złączenie zewnętrzne z ofertami. Warto wtedy samemu zapewnić unikalność wyników, stosując na przykład kod `distinctResults = new HashSet(resultList)`. Zbiór nie dopuszcza duplikacji zawartych w nim elementów.

Oto w jaki sposób Hibernate implementuje tak zwaną **strategię sprowadzania asocjacji w trakcie działania aplikacji**. Stanowi ona niezwykle ważny element zapewniający osiągnięcie wysokiej wydajności systemu ORM. Przejdzmy do pozostałych operacji złączeń.

7.3.3. Aliasy i złączenia

Opisaliśmy znaczenie klauzuli `where` w ograniczaniu wyników. Najczęściej zwarcane wyniki trzeba ograniczyć na podstawie wartości z różnych powiązanych ze sobą klas (złączanych tabel). Jeśli chcemy wykonać to zadanie, używając złączenia w klauzuli `from`, musimy klasie przypisać alias.

```
from Item item
join item.bids bid
  where item.description like '%gc%'
    and bid.amount > 100
```

Zapytanie klasie Item przypisuje alias item, natomiast klasie Bid alias bid. W ten sposób w klauzuli where możemy wpisać ograniczenia dotyczące obu aliasów.

Wynikowy kod SQL ma postać:

```
select I.DESCRIPTION, I.CREATED, I.SUCCESSFUL_BID, B.BID_ID,
B.AMOUNT, B.ITEM_ID, B.CREATED
from ITEM I
left outer join BID B on I.ITEM_ID = B.ITEM_ID
where I.DESCRIPTION like '%gc%'
and B.AMOUNT > 100
```

Zapytanie zwraca wszystkie kombinacje obiektów Bid i Item, ale w odróżnieniu od złączenia sprowadzającego kolekcja bids z klasą Item nie zostaje automatycznie zainicjalizowana przez zapytanie! Co oznacza więc wspomniana wcześniej **kombinacja?** Parę uporządkowaną (bid, item). W wynikach zapytania Hibernate reprezentuje parę uporządkowaną jako tablicę. Prześledźmy pełny przykład wraz z kodem i wynikami zapytania.

```
Query q = session.createQuery("from Item item join item.bids bid");
Iterator pairs = q.list().iterator();
while (pairs.hasNext()) {
    Object[] pair = (Object[]) pairs.next();
    Item item = (Item) pair[0];
    Bid bid = (Bid) pair[1];
}
```

Zamiast obiektu List z elementami Items zapytanie zwraca obiekt List z tablicą Object[]. W tablicy pod indeksem 0 znajduje się obiekt Item, natomiast pod indeksem 1 obiekt Bid. Konkretny obiekt Item może wystąpić kilka razy, po jednym na każdy związany z nim obiekt Bid.

Różni się to znaczco od przypadku, w którym korzystamy ze złączenia sprowadzającego dane z wyprzedzeniem. Zapytanie to zwracało obiekt List z elementami Item zawierającymi zainicjalizowaną kolekcję bids.

Jeśli w zbiorze wyników nie potrzebujemy obiektów Bid, możemy do zapytania HQL wprowadzić klauzulę select. Jest opcjonalna (w języku SQL jest wymagana), więc używamy jej tylko wtedy, gdy nie jesteśmy zadowoleni z domyślnie zwracanego zbioru wyników. By pobrać jedynie wybrane obiekty, w klauzuli umieszczamy ich aliasy.

```
select item
from Item item
join item.bids bid
  where item.description like '%gc%'
    and bid.amount > 100
```

Teraz wynikowy kod SQL ma postać:

```
select I.DESCRIPTION, I.CREATED, I.SUCCESSFUL_BID
from ITEM I
```

```
inner join BID B on I.ITEM_ID = B.ITEM_ID
where I.DESCRIPTION like '%gc%'
and B.AMOUNT > 100
```

Zbiór wyników zawiera tylko elementy Item. Co więcej, ponieważ jest to **złączenie wewnętrzne**, zwrócone zostaną tylko obiekty Item zawierające choć jeden obiekt Bid.

```
Query q = session.createQuery("select i from Item i join i.bids b");
Iterator items = q.list().iterator();
while (items.hasNext()) {
    Item item = (Item) items.next();
}
```

W HQL aliasy wykorzystuje się w ten sam sposób, niezależnie od tego, czy są to złączane asocjacje czy bezpośrednie odwołania. Po określeniu aliasu w klauzuli `from` można go stosować w klauzuli `where` i opcjonalnej klauzuli `select`. Klauzula `select` z HQL jest wysoce elastyczna. Zajmiemy się nią dokładniej w dalszej części rozdziału.

Istnieją dwa sposoby określenia złączenia w interfejsie Criteria — podobnie jak istnieją dwa zastosowania aliasów w ograniczeniach. Pierwszym jest metoda `createCriteria()` interfejsu Criteria. Tak, oznacza to zagnieżdżone wywoływanie `createCriteria()`:

```
Criteria itemCriteria = session.createCriteria(Item.class);
itemCriteria.add(Expression.like("description", "gc", MatchMode.ANYWHERE));
Criteria bidCriteria = itemCriteria.createCriteria("bids");
bidCriteria.add(Expression.gt("amount", new BigDecimal("100")));
List results = itemCriteria.list();
```

Nic nie stoi na przeszkodzie, by i w tej sytuacji zastosować łańcuchowe wywołanie metod.

```
List results = session.createCriteria(Item.class)
    .add(Expression.like("description", "gc", MatchMode.ANYWHERE))
    .createCriteria("bids")
    .add(Expression.gt("amount", new BigDecimal("100")))
    .list();
```

Utworzenie obiektu Criteria dla bids z klasy Item powoduje powstanie złączenia wewnętrznego tabel zawierających obiekty klasy. Zauważ, że `list()` można wywołać dla dowolnego egzemplarza Criteria bez zmiany wyników zapytania.

Drugi sposób określania zapytania przez kryteria polega na wskazaniu aliasu złączanej klasy.

```
List results = session.createCriteria(Item.class)
    .createAlias("bids", "bid")
    .add(Expression.like("description", "gc", MatchMode.ANYWHERE))
    .add(Expression.gt("bid.amount", new BigDecimal("100")))
    .list();
```

To podejście nie stosuje drugiego egzemplarza Criteria, więc właściwości złączanego elementu trzeba poprzedzić aliasem określonym metodą `createAlias()`. Właściwości głównej encji (Item) nie muszą zawierać przedrostka z aliasem.

Ewentualnie można dla nich użyć przedrostka `this`. Poniższy kod jest równoważny wcześniejszemu:

```
List results = session.createCriteria(Item.class)
    .createAlias("bids", "bid")
    .add(Expression.like("this.description", "gc", MatchMode.ANYWHERE))
    .add(Expression.gt("bid.amount", new BigDecimal("100")))
    .list();
```

Domyślnie zapytanie przez kryteria zwraca w wynikach tylko główną encję — w przedstawionym przykładzie obiekty `Item`. Podsumujmy to w kolejnym przykładzie.

```
Iterator items = session.createCriteria(Item.class)
    .createAlias("bids", "bid")
    .add(Expression.like("this.description", "gc", MatchMode.ANYWHERE))
    .add(Expression.gt("bid.amount", new BigDecimal("100")))
    .list().iterator();

while (items.hasNext()) {
    Item item = (Item) items.next();
    // Wykonaj odpowiednie działania.
}
```

Pamiętaj, że kolekcja `bids` obiektu `Item` nie jest zainicjalizowana. Ograniczeniem zapytań przez kryteria jest niemożność połączenia metody `createAlias()` z trybem sprowadzania wyprzedzającego, na przykład nie można użyć wywołania `setFetchMode("bids", FetchMode.EAGER)`.

Jeśli chcemy zwrócić zarówno dopasowane obiekty `Item`, jak i `Bid`, musimy poprosić Hibernate, by zwrócić wyniki w postaci obiektu Map.

```
Iterator itemBidMaps = session.createCriteria(Item.class)
    .createAlias("bids", "bid")
    .add(Expression.like("this.description", "gc", MatchMode.ANYWHERE))
    .add(Expression.gt("bid.amount", new BigDecimal("100")))
    .returnMaps()
    .list().iterator();

while (itemBidMaps.hasNext()) {
    Map map = (Map) itemBidMaps.next();
    Item item = (Item) map.get("this");
    Bid bid = (Bid) map.get("bid");
    // Wykonaj odpowiednie działania.
}
```

To istotna różnica w domyślnym zachowaniu HQL i zapytań przez kryteria. Domyślnie HQL zwraca wszystkie encje, jeśli jawnie ich nie ograniczymy.

Czasem można skorzystać z mniej rozbudowanego sposobu wyrażania złączeń. Hibernate dopuszcza niejawne złączenia dla asocjacji.

7.3.4. Złączenia niejawne

Do tej pory w HQL stosowaliśmy proste nazwy właściwości poprzedzone aliasami klas: bid.amount i item.description. HQL obsługuje rozbudowane ścieżki do właściwości z dwóch powodów:

- ◆ by tworzyć ograniczenia dla komponentów,
- ◆ by określać niejawne złączenia dla asocjacji.

Przykład wykorzystujący pierwszy powód jest prosty:

```
from User u where u.address.city = 'Bangkok'
```

Części odwzorowanego komponentu Address określamy za pomocą notacji kropkowej. Podobny sposób wskazywania właściwości dopuszcza też interfejs Criteria.

```
session.createCriteria(User.class)
    .add(Expression.eq("address.city", "Bangkok"));
```

Drugi powód stosowania ścieżek dostępny jest tylko w HQL. Oto przykład dla niego:

```
from Bid bid where bid.item.description like '%gc%'
```

Wynikiem jest niejawne złączenie dla asocjacji wiele-do-jednego między Bid i Item. Złączenia niejawne zawsze są wzbudzane przez asocjacje wiele-do-jednego lub jeden-do-jednego, ale nie przez asocjacje z wartością będącą kolekcją (nie można napisać item.bids.amount).

Nic nie stoi na przeszkodzie, by podać dłuższą ścieżkę do właściwości. Gdyby asocjacja od Item do Category była typu wiele-do-jednego (zamiast wiele-do-wielu), moglibyśmy napisać:

```
from Bid bid where bid.item.category.name like 'Laptop%'
```

Nie zalecamy zbytniego rozbudowywania ścieżek, które mogą powodować dodatkowe złączenia. Złączenia mają znaczący wpływ na optymalizację aplikacji, więc zawsze warto wiedzieć, ile ich będzie w konkretnym zapytaniu. Rozważmy następujące zapytanie (ponownie przyjmując asocjację typu wiele-do-jednego od Item do Category):

```
from Bid bid
    where bid.item.category.name like 'Laptop%'
        and bid.item.successfulBid.amount > 100
```

Ile złączeń w języku SQL potrzeba, by wykonać to zapytanie? Nawet jeśli podasz poprawną odpowiedź, na pewno zajmie Ci to kilka sekund. Odpowiedź brzmi: trzy. Wygenerowany kod SQL ma postać typu:

```
select ...
from BID B
inner join ITEM I on B.ITEM_ID = I.ITEM_ID
inner join CATEGORY C on I.CATEGORY_ID = C.CATEGORY_ID
inner join BID SB on I.SUCCESSFUL_BID_ID = SB.BID_ID
where C.NAME like 'Laptop%'
and SB.AMOUNT > 100
```

Większą czytelność przekazu osiągniemy, stosując poniższą wersję zapytania:

```
from Bid bid
join bid.item item
  where item.category.name like 'Laptop%'
    and item.successfulBid.amount > 100
```

Możemy nawet jeszcze bardziej opisowo wskazać wyniki do zwrócenia:

```
from Bid as bid
join bid.item as item
join item.category as cat
join item.successfulBid as winningBid
  where cat.name like 'Laptop%'
    and winningBid.amount > 100
```

Kontynuujmy rozważania, tworząc złączenia z dowolnymi atrybutami przy użyciu stylu theta.

7.3.5. Złączenia w stylu theta

Iloczyn kartezjański umożliwia pobranie wszystkich kombinacji egzemplarzy dwóch lub większej liczby klas. Następujące zapytanie zwraca wszystkie pary uporządkowane obiektów User i Category:

```
from User, Category
```

Oczywiście powyższe zapytanie ma niewielki sens praktyczny. Stosuje się je tylko w połączeniu ze złączeniami w stylu theta.

W tradycyjnym SQL złączenie theta to iloczyn kartezjański oraz warunek złączenia zawarty w klauzuli where, który ogranicza liczbę zwracanych wyników po przeprowadzeniu iloczynu.

W HQL składnia znana ze złączenia theta przydaje się, gdy warunek złączenia nie jest związkiem z kluczem obcym odwzorowanym na asocjację do innej klasy. Przypuśćmy, że w rekordach dziennika przechowujemy nazwę użytkownika z klasy User, zamiast stosować asocjację od LogRecord do User. Klasy nic o sobie nie wieją, gdyż nie są ze sobą powiązane. Następujące zapytanie znajduje wszystkich użytkowników wraz z ich danymi z dziennika dzięki użyciu złączeniu theta.

```
from User user, LogRecord log where user.username = log.username
```

Warunkiem złączenia okazuje się właściwość username istniejąca w obu klasach. Jeśli wpisy w obu tabelach zawierają tę samą nazwę użytkownika, zostają złączone (złączenie wewnętrzne) i dołączone do wyników. Wyniki zapytania zawierają pary uporządkowane:

```
Iterator i = session.createQuery(
  "from User user, LogRecord log where user.username = log.username")
.list().iterator();

while (i.hasNext()) {
  Object[] pair = (Object[]) i.next();
```

```
User user = (User) pair[0];
LogRecord log = (LogRecord) pair[1];
}
```

Sposób zwracania wyników zmienimy, dodając klauzulę select.

Sytuacje wymagające użycia złączenia theta nie zdarzają się często. Pamiętaj, że interfejs Criteria nie umożliwia określania iloczynu kartezańskiego ani złączeń theta. Hibernate 2.1 nie dopuszcza złączenia zewnętrznego dwóch tabel, jeśli nie została dla nich wskazane odwzorowanie z asocjacją.

7.3.6. Porównywanie identyfikatorów

Bardzo często zdarza się, że zapytanie porównuje wartość klucza głównego lub obcego do parametru lub wartości innego klucza. Gdy przeniesiemy to zagadnienie na poziom obiektowy, okaże się, że tak naprawdę porównujemy referencje do obiektów. HQL obsługuje następujące zapytanie:

```
from Item i, User u
where i.seller = u and u.username = 'Jacek'
```

W zapytaniu `i.seller` odnosi się do klucza obcego z tabeli ITEM (kolumna SELLER_ID) dotyczącego tabeli USER. Z drugiej strony `u` dotyczy klucza głównego tabeli USER (kolumna USER_ID). Zapytanie stosuje złączenie theta i jest równoważne referowanemu stylowi zapisu:

```
from Item i join i.seller u
where u.username = 'Jacek'
```

Z drugiej strony poniższego złączenia theta nie można wyrazić złączeniem w klauzuli from.

```
from Item i, Bid b
where i.seller = b.bidder
```

W takiej sytuacji `i.seller` i `b.bidder` są kluczami obcymi dotyczącymi tabeli USER. Zauważ, że przedstawione zapytanie jest niezwykle ważne w analizowanej aplikacji, bo pozwala sprawdzić, czy użytkownicy nie podbijają cen, próbując kupić własny przedmiot.

Często należy porównać wartość klucza obcego z parametrem zapytania — rozważmy przykład znajdujący wszystkie obiekty Comment konkretnego użytkownika.

```
User user = ...
Query q = session.createQuery("from Comment c where c.fromUser = :user");
q.setEntity("user", user);
List result = q.list();
```

Czasem będziemy chcieli wyrazić tego rodzaju zapytania w kategoriach wartości identyfikujących, a nie referencji do obiektów. Dostęp do wartości identyfikującej uzyskuje się na podstawie nazwy właściwości identyfikującej (jeśli istnieje) lub specjalnej właściwości o nazwie `id`. Każda encyjna klasa trwała zawiera tę spe-

cjalną właściwość, nawet jeśli nie implementuje się jej jawnie w klasie (patrz podrozdział 3.4.2).

Poniższe zapytania są równoważne wcześniejszym podanym:

```
from Item i, User u
  where i.seller.id = u.id and u.username = 'Jacek'
from Item i, Bid b
  where i.seller.id = b.bidder.id
```

Nic nie stoi na przeszkodzie, by wartość identyfikatora wykorzystać jako parametr zapytania.

```
Long userId = ...
Query q = session.createQuery("from Comment c where c.fromUser.id = :id");
q.setLong("id", userId);
List result = q.list();
```

Powinieneś sobie zdawać sprawę z ogromnej różnicy dwóch poniższych zapytań:

```
from Bid b where b.item.id = 1
from Bid b where b.item.description like '%gc'
```

Drugie zapytanie stosuje niejawne złączenie tabel. Pierwsze nie korzysta z żadnego złączenia.

Omówiliśmy większość elementów zapytań Hibernate związań z pobieraniem obiektów dla logiki biznesowej. W kolejnym podrozdziale skupimy się na zapytaniach związanych przede wszystkim z analizą i raportowaniem.

7.4. Tworzenie zapytań raportujących

Zapytania raportujące wykorzystują możliwości systemów baz danych SQL związane z wydajnym grupowaniem i agregacją danych.

Są ze swojej natury bardziej relacyjne i nie zawsze zwracają encje. Na przykład zamiast pobierać encje Item, które są transakcyjne (i automatycznie sprawdzane pod kątem zabrudzenia), typowe zapytanie raportujące może pobrać tylko nazwy przedmiotów i ich ceny wywoławcze. Jeśli to jedyna informacja, jakiej potrzebujemy (możliwe, że nawet w formie zagregowanej — na przykład pobieramy najwyższą cenę wywoławczą w kategorii), by wykonać ekran raportu, nie potrzebujemy encji transakcyjnych, co pozwala uniknąć niewielkiego narzutu związanego z automatycznym sprawdzaniem zabrudzenia i buforowaniem w obiekcie Session.

W tym podrozdziale pominiemy zapytania przez kryteria, bo w Hibernate 2.1 nie jest możliwe tworzenie w nich zapytań raportujących.

Ponownie zajmijmy się strukturą zapytania HQL.

Jedyną wymaganą klauzulą dla zapytania HQL jest klauzula `from`. Wszystkie pozostałe są opcjonalne. Pełna struktura zapytania HQL może mieć postać przedstawioną poniżej:

```
[select ...] from ... [where ...]
[group by ... [having ...]] [order by ...]
```

W poprzednim podrozdziale omówiliśmy klauzule from, where i order by. Dodatkowo użyliśmy klauzuli select do wskazania, które encje należy zwrócić z zapytania wykorzystującego złączenie.

W zapytaniach raportujących klauzula select służy do projekcji natomiast klauzule group by i having do agregacji.

7.4.1. Projekcja

Klauzula select dokonuje projekcji, czyli określa, jakie obiekty lub właściwości mają znaleźć się w wynikach zapytania. Przedstawione wcześniej zapytanie zwraca uporządkowane pary obiektów Item i Bid:

```
from Item item join item.bids bid where bid.amount > 100
```

Jeśli jednostka zadaniowa wymaga tylko obiektów Item, możemy powyższe zapytanie uzupełnić elementem select:

```
select item from Item item join item.bids bid where bid.amount > 100
```

Jeżeli jedynie wyświetlamy użytkownikowi listę przedmiotów, do jej zbudowania wystarcza tylko niektóre właściwości.

```
select item.id, item.description, bid.amount  
from Item item join item.bids bid where bid.amount > 100
```

Przedstawione zapytanie zwróci każdy wiersz wyników jako tablicę Object[] o długości 3. To zapytanie raportujące — wszystkie uzyskane obiekty nie są encjami Hibernate i nie są transakcyjne. Używa się ich tylko do odczytu zawartości bazy danych. Oto przykład:

```
Iterator i = session.createQuery("select item.id, item.description,  
    bid.amount " +  
    "from Item item join item.bids bid where bid.amount > 100")  
    .list()  
    .iterator();  
  
while (i.hasNext()) {  
    Object[] row = (Object[]) i.next();  
  
    Long id = (Long) row[0];  
    String description = (String) row[1];  
    BigDecimal amount = (BigDecimal) row[2];  
  
    // ... wyświetl wartości, tworząc raport  
}
```

Dynamiczne tworzenie obiektów

Ponieważ poprzedni przykład wymagał pisania sporej ilości kodu i nie był zbyt obiektowy (korzystanie z danych tabeli umieszczonych w tablicach), możemy zdefiniować klasę reprezentującą każdy wiersz wyników i użyć w zapytaniu konstrukcji select new.

```
select new ItemRow( item.id, item.description, bid.amount )
       from Item item join item.bids bid where bid.amount > 100
```

Zakładając istnienie odpowiedniego konstruktora dla klasy ItemRow (klasę musimy napisać sami), zapytanie zwróci nowe, ulotne egzemplarze klasy ItemRow. Pełny kod przedstawia kolejny przykład:

```
Iterator i = session.createQuery(
    "select new ItemRow( item.id, item.description, bid.amount ) " +
    "from Item item join item.bids bid where bid.amount > 100")
    .list()
    .iterator();

while (i.hasNext()) {
    ItemRow row = (ItemRow) i.next();
    // ... wyświetl wartości, tworząc raport
}
```

Własna klasa ItemRow nie musi być trwała — nie trzeba jej odwzorowywać na tabele bazy danych ani w żaden inny sposób informować Hibernate o jej istnieniu. Klasa ItemRow staje się tak naprawdę klasą przesyłu danych dającą większą elastyczność przy generowaniu raportu.

Pobieranie różnych wyników

Zastosowanie klauzuli select nie gwarantuje uzyskania unikatowego zbioru wynikowych wierszy. Na przykład opisy z obiektów Item nie mają włączonej unikalności, więc poniższe zapytanie może zwrócić ten sam opis kilkukrotnie:

```
select item.description from Item item
```

W takiej sytuacji trudno sobie wyobrazić, że pojawienie się w wynikach dwóch identycznych opisów ma jakikolwiek sens, więc gdy istnieje prawdopodobieństwo zaistnienia duplikatów, warto dodać słowo distinct. Oto przykład:

```
select distinct item.description from Item item
```

Dzięki tej operacji w wynikach pojawią się tylko unikatowe wiersze opisów.

Wywołanie funkcji SQL

W niektórych dialektach języka SQL obsługiwanych przez Hibernate można wywoływać w klauzuli select funkcje specyficzne dla konkretnej bazy danych (pamiętaj, że jest to dopuszczalne również w klauzuli where). Następujące zapytanie poza właściwością obiektu Item pobiera również datę i czas obowiązujące na serwerze bazodanowym (składnia Oracle):

```
select item.startDate, sysdate from Item item
```

W klauzuli select niekoniecznie jesteśmy ograniczeni tylko do funkcji specyficznych dla danego dostawcy. Nic nie stoi na przeszkodzie, by skorzystać z ustanowionych funkcji SQL:

```
select item.startDate, item.endDate, upper(item.name) from Item item
```

Zapytanie zwróci tablicę Object[] z datą początku i końca aukcji oraz nazwą przedmiotu aukcji zapisaną wielkimi literami.

W szczególności można wywoływać **funkcje agregujące SQL**.

7.4.2. Agregacja

Hibernate rozpoznaje następujące funkcje agregujące: count(), min(), max(), sum() i avg().

Zapytanie zliczające obiekty Item ma postać:

```
select count(*) from Item
```

Wynikiem jest wartość typu Integer:

```
Integer count = (Integer) session.createQuery("select count(*) from Item")
    .uniqueResult();
```

Zauważ zastosowanie znaku gwiazdki, który ma to samo znaczenie co w języku SQL.

Tym razem zliczymy tylko te przedmioty, które mają ustawioną wartość successfulBid:

```
select count(item.successfulBid) from Item item
```

Kolejne zapytanie wylicza łączną kwotę wszystkich najlepszych ofert.

```
select sum(item.successfulBid.amount) from Item item
```

Zapytanie zwraca wartość typu BigDecimal. Zauważ zastosowanie niejawnego złączenia w klauzuli select — przechodzimy dzięki asocjacji successfulBid od obiektu Item do Bid.

Kolejne zapytanie zwraca maksymalną i minimalną kwotę oferty dla konkretnego przedmiotu:

```
select min(bid.amount), max(bid.amount)
    from Bid bid where bid.item.id = 1
```

Wynikiem jest para uporządkowana obiektów BidDecimal (dwa obiekty BidDecimal w tablicy Object[]).

Specjalna funkcja count(distinct) ignoruje powtórzenia wartości:

```
select count(distinct item.description) from Item item
```

Wywołanie funkcji agregującej w klauzuli select bez zastosowania klauzuli group by powoduje zwinięcie całego zbioru wyników do jednego wiersza zawierającego zagregowaną wartość. Oznacza to, że klauzula select bez klauzuli group by stosująca choćby jedną funkcję agregującą może zawierać tylko inne funkcje agregujące.

By uzyskać bardziej złożone dane statystyczne, trzeba skorzystać z grupowania.

7.4.3. Grupowanie

Podobnie jak w języku SQL dowolna właściwość (lub alias), która wystąpi w HQL poza funkcją agregującą w klauzuli select, musi się również pojawić w klauzuli group by.

Rozważmy zapytanie zliczające użytkowników o takich samych nazwiskach:

```
select u.lastname, count(u) from User u  
group by u.lastname
```

Hibernate dla tego przypadku wygeneruje następujący kod SQL:

```
select U.LAST_NAME, count(U.USER_ID)  
from USER U  
group by U.LAST_NAME
```

W przykładzie u.lastname nie znajduje się wewnętrz funkcji agregującej, ale określa sposób grupowania. W HQL nie musimy wskazywać zliczanej właściwości. Hibernate wygeneruje kod SQL automatycznie używający klucza głównego, jeśli użyjemy aliasu określonego w klauzuli from.

Następne zapytanie znajduje średnią wartość oferowanej kwoty dla poszczególnych przedmiotów:

```
select bid.item.id, avg(bid.amount) from Bid bid  
group by bid.item.id
```

Hibernate po wykonaniu zapytania zwróci parę uporządkowaną identyfikatorów obiektów Item oraz średnią kwotę ofert. Zauważ użycie specjalnej właściwości id do określenia identyfikatora klasy trwałej niezależnie od rzeczywistej nazwy właściwości.

Kolejne zapytanie zlicza oferty i określa średnią kwotę ofert dla niesprzedanego przedmiotu:

```
select bid.item.id, count(bid), avg(bid.amount)  
from Bid bid  
where bid.item.successfulBid is null  
group by bid.item.id
```

Zapytanie wykorzystuje niejawne złączenie dzięki asocjacji. Aby złączenie wskazać jawnie w klauzuli from (ale bez wcześniejszego sprowadzania), możemy użyć następującego wyrażenia:

```
select bidItem.id, count(bid), avg(bid.amount)  
from Bid bid  
join bid.item bidItem  
where bidItem.successfulBid is null  
group by bidItem.id
```

By zainicjalizować kolekcję bids obiektów Item, skorzystamy ze złączenia sprowadzającego i odniesiemy się do asocjacji od drugiej strony.

```
select item.id, count(bid), avg(bid.amount)  
from Item item  
fetch join item.bids bid
```

```
where item.successfulBid is null
group by item.id
```

Czasem zachodzi potrzeba dodatkowego ograniczenia wyników przez wybór tylko niektórych wartości z grupy.

7.4.4. Ograniczanie grup klauzulą having

Klauzula where służy do nakładania ograniczeń na konkretne wiersze. Klauzula having nakłada ograniczenia na grupy.

Następne zapytanie zlicza użytkowników, których nazwisko zaczyna się na literę A:

```
select user.lastname, count(user)
  from User u
 group by user.lastname
    having user.lastname like 'A%'
```

W klauzulach select i having obowiązuje ta sama zasada: jedynie grupowane właściwości mogą pojawić się poza funkcją agregującą. Kolejne zapytanie zlicza liczbę ofert niesprzedanego przedmiotu, ale tylko dla przedmiotów mających co najmniej 10 ofert.

```
select item.id, count(bid), avg(bid.amount)
  from Item item
    join item.bids bid
   where item.successfulBid is null
 group by item.id
    having count(bid) > 10
```

Większość zapytań raportujących używa klauzuli select do wyboru listy rzutowanych lub agregowanych właściwości. Zauważ, że gdy klauzula zawiera więcej niż jedną właściwość lub alias, Hibernate zwraca wynik w postaci krotki: każdy wiersz wyników reprezentuje egzemplarz Object[]. Krotki nie są wygodnie i nie zapewniają bezpieczeństwa typów, więc warto stosować wspomniany wcześniej konstruktor select new. Tworzone dynamicznie obiekty można stosować w połączeniu z agregacją i grupowaniem.

Po zdefiniowaniu metody ItemBidSummary z konstruktorem przyjmującym obiekty Long, String i Integer możemy użyć zapytania w postaci:

```
select new ItemBidSummary(bid.item.id, count(bid), avg(bid.amount))
  from Bid bid
 where bid.item.successfulBid is null
 group by bid.item.id
```

W wyniku wykonania zapytania poszczególne elementy wiersza znajdą się we właściwościach obiektu ItemBidSummary i będą zawierały identyfikator obiektu Item, liczbę ofert dla przedmiotu i średnią kwotę oferty. To podejście jest bezpieczne pod kątem typów. Co więcej, klasę ItemBidSummary łatwo rozbudować, by poza transferem danych odpowiadała również za formatowanie zwróconych wartości.

7.4.5. Poprawa wydajności zapytań raportujących

Zapytania raportujące mogą mieć niemały wpływ na wydajność aplikacji, więc dokładniej zajmijmy się tym tematem.

Jedyną sytuacją, w której zanotowaliśmy znaczący narzut kodu Hibernate w porównaniu ze zwykłymi wywołaniami JDBC (co ciekawe dla nierealistycznego zbioru testowego), było użycie zapytań tylko do odczytu dla lokalnej bazy danych. Baza danych potrafi w całości zbuforować wyniki zapytania i odpowiedzieć bardzo szybko, jeśli testy prowadzi się dla niewielkiego zbioru danych: wtedy zwykły SQL i sterownik JDBC okaże się rozwiązaniem najszybszym.

Z drugiej strony, niezależnie od ilości danych Hibernate musi dodawać wynikowe obiekty zapytania do bufora obiektu Session (i być może do bufora drugiego poziomu), zarządzać unikatowością itp. Zapytania raportujące mogą pominąć narzut związany z buforem sesyjnym. Narzut wprowadzany przez zapytanie raportujące Hibernate w porównaniu z bezpośrednim wywołaniem SQL dla JDBC najczęściej nie jest mierzalny, nawet w tak nierealistycznym przypadku jak wczytanie miliona obiektów z lokalnej bazy danych bez opóźnień powodowanych przez sieć.

Zapytania raportujące w HQL dzięki projekcji pozwalają określić właściwości, którymi jesteśmy zainteresowani. W tego rodzaju zapytaniach nie pobieramy całych encji, a jedynie wybrane ich właściwości lub wartości uzyskane z agregacji.

```
select user.lastname, count(user), from User user  
group by user.lastname
```

Zapytanie nie zwraca trwałej encji, więc Hibernate nie dodaje obiektu transakcyjnego do bufora sesyjnego. Oznacza to również brak obiektu do sprawdzania pod kątem zmian.

Zapytania raportujące szybciej zwalniają zaalokowaną pamięć, ponieważ obiekty nie są przechowywane w obiekcie Session aż do zamknięcia sesji — zostaną usunięte przez mechanizm czyszczenia pamięci zaraz po usunięciu referencji do nich przez aplikację, czyli najczęściej tuż po utworzeniu raportu.

Wszystkie te kwestie najczęściej mają drugorzędne znaczenie, więc nie warto od razu przepisywać całej aplikacji, by stosowała zapytania raportujące tylko do odczytu zamiast transakcyjnych, buforowanych i monitorowanych obiektów. Zapytania raportujące wymagają więcej kodu i są mniej obiektowe. Są też mniej wydajne, jeśli rozważy się narzut wprowadzany przez zdальną komunikację z serwerem bazy danych. Przed zastosowaniem tej optymalizacji warto zaczekać do sytuacji, w której tradycyjne podejście okaże się całkowicie nieefektywne.

Powróćmy do typowych zapytań zwracających encje, bo zostało jeszcze dużo elementów Hibernate do odkrycia.

7.5. Techniki tworzenia zaawansowanych zapytań

Z opisywanych tu technik w Hibernate korzysta się raczej rzadko, ale warto o nich wiedzieć. W niniejszym podrozdziale omówimy programowe tworzenie kryteriów dla przykładowych obiektów. O tym sposobie wspominaliśmy we wcześniejszej części rozdziału.

Filtrowanie kolekcji również stanowi poręczną technikę — do filtrowania można zaprzeczą bazę danych, zamiast usuwać niechciane obiekty z pamięci. Na podzapytaniach i zapytaniach w języku SQL zakończymy opis technik stosowania zapytań w Hibernate.

7.5.1. Zapytania dynamiczne

Często zapytania tworzy się programowo, włączając poszczególne kryteria w zależności od wyboru dokonanego przez użytkownika. Przykładowo, administrator systemu chce poszukać użytkowników z dowolną kombinacją imienia i nazwiska oraz pobrać wyniki uporządkowane według nazwy użytkownika. W HQL do wykonania tego zadania trzeba zająć się modyfikacją tekstu:

```
public List findUsers(String firstname, String lastname)
    throws HibernateException {

    StringBuffer queryString = new StringBuffer();
    boolean conditionFound = false;

    if (firstname != null) {
        queryString.append("lower(u.firstname) like :firstname ");
        conditionFound = true;
    }

    if (lastname != null) {
        if (conditionFound) queryString.append("and ");
        queryString.append("lower(u.lastname) like :lastname ");
        conditionFound = true;
    }

    String fromClause = conditionFound ? "from User u where " : "from User
    u ";

    queryString.insert(0, fromClause).append("order by u.username");

    Query query = getSession().createQuery(queryString.toString());

    if (firstname != null)
        query.setString("firstname", '%' + firstname.toLowerCase() + '%');

    if (lastname != null)
        query.setString("lastname", '%' + lastname.toLowerCase() + '%');

    return query.list();
}
```

Kod jest wyjątkowo nieprzyjazny i nieczytelny, więc spróbujmy innego podejścia. Interfejs Criteria wydaje się znaczaco poprawiać sytuację.

```
public List findUsers(String firstname, String lastname)
    throws HibernateException {

    Criteria crit = getSession().createCriteria(User.class);
```

```
if (firstname != null) {
    crit.add(Expression.ilike("firstname", firstname,
        MatchMode.ANYWHERE));
}
if (lastname != null) {
    crit.add(Expression.ilike("lastname", lastname, MatchMode.ANYWHERE));
}

crit.addOrder(Order.asc("username"));

return crit.list();
}
```

Kod jest znacznie krótszy. Metoda `ilike()` dokonuje porównania bez uwzględniania wielkości liter. Bez wątpienia to rozwiązanie jest lepsze. Jeśli jednak ekran wyszukiwania zawiera wiele opcjonalnych kryteriów, istnieje jeszcze lepsze podejście.

Zauważ, że dodanie nowego kryterium wyszukiwania zmienia liczbę parametrów metody `findUsers()`. Ponieważ wszystkie wyszukiwane właściwości należą do klasy `User`, dlaczego nie użyć jej jako wzorca?

To podejście stosuje QBE. Zapytanie powstaje na podstawie przekazanego obiektu wyszukiwanej klasy, w którym niektóre właściwości zmieniono z ich wartości domyślnych, i zwraca wszystkie klasy trwałe zawierające właściwości ustalone we wskazany sposób. Hibernate implementuje QBE jako część interfejsu Criteria.

```
public List findUsers(User u) throws HibernateException {

    Example exampleUser =
        Example.create(u).ignoreCase().enableLike(MatchMode.ANYWHERE);

    return getSession().createCriteria(User.class).add(exampleUser).list();
}
```

Metoda `create()` zwraca nowy egzemplarz `Example` dla podanego egzemplarza klasy `User`. Metoda `ignoreCase()` wyłącza dla przykładowego zapytania uwzględnianie wielkości liter dla wszystkich tekstowych właściwości. Wywołanie `enableLike()` wskazuje, że dla właściwości tekstowych należy użyć operatora `like`. Argument określa tryb porównywania.

Ponownie uprościliśmy kod. Co ciekawe, obiekt `Example` jest zwykłym kryterium, więc nic nie stoi na przeszkodzie, by połączyć go ze zwykłym zapytaniem przez kryteria.

Prześledźmy przykład takiego podejścia, ograniczając wyniki wyszukiwania tylko do użytkowników z niesprzedaonymi przedmiotami. Do zapytania dołączymy kryterium, które ograniczy wyniki na podstawie zawartości kolekcji `items` zawierającej obiekty `Item`.

```
public List findUsers(User u) throws HibernateException {

    Example exampleUser =
        Example.create(u).ignoreCase().enableLike(MatchMode.ANYWHERE);
```

```

        return getSession().createCriteria(User.class).add(exampleUser)
            .createCriteria("items").add(Expression.isNull("successfulBid"))
            .list();
    }
}

```

Możemy nawet połączyć właściwości User z właściwościami Item w tym samym wyszukiwaniu:

```

public List findUsers(User u, Item i) throws HibernateException {

    Example exampleUser =
        Example.create(u).ignoreCase().enableLike(MatchMode.ANYWHERE);

    Example exampleItem =
        Example.create(i).ignoreCase().enableLike(MatchMode.ANYWHERE);

    return getSession().createCriteria(User.class).add(exampleUser)
        .createCriteria("items").add(exampleItem)
        .list();
}

```

W tym momencie zastanów się przez chwilę, ile ręcznie tworzonego kodu SQL i JDBC wymagałaby implementacja tak złożonego systemu wyszukiwania. Nie będziemy tu przytaczać przykładu, bo zająłby kilka stron.

7.5.2. Filtry kolekcji

Najczęściej chcemy wykonać zapytanie dla wszystkich elementów wskazanej kolekcji. Przykładowo dysponujemy obiektem Item i chcemy pobrać wszystkie złożone dla niego oferty ułożone względem czasu ich złożenia. Znamy dobry sposób wykonania tego rodzaju zapytania:

```

List results = session.createQuery("from Bid b where b.item = :item "
    "order by b.amount asc")
    .setEntity("item", item)
    .list();

```

Zapytanie działa doskonale, ponieważ asocjacja między ofertami i przedmiotami jest dwukierunkowa, a każdy obiekt Bid zna obiekt Item, do którego przynależy. Wyobraźmy sobie jednak asocjację jednokierunkową — obiekt Item zawiera kolekcję obiektów Bid, ale nie istnieje asocjacja odwrotna od Bid do Item.

W takiej sytuacji spróbujmy następującego zapytania:

```

String query = "select bid from Item item join item.bids bid "
    "+ "where item = :item order by bid.amount asc";

List results = session.createQuery(query)
    .setEntity("item", item)
    .list();

```

Zapytanie nie jest wydajne, bo stosuje niepotrzebne złączenie. Lepszym, bardziej eleganckim podejściem okazuje się filtr kolekcji — specjalne zapytanie stosowane dla trwałej kolekcji (lub tablicy). Najczęściej służy do dodatkowego ograniczenia wyników.

niczenia lub ułożenia wyników. Używamy go na już załadowanym obiekcie Item i jego kolekcji bids:

```
List results = session.createFilter(item.getBids(), "order by this.amount asc").list();
```

Filtr powoduje zwrócenie identycznych wyników, jak w pierwszym zapytaniu. Również działa podobnie. Filtr kolekcji stosuje niejawną klauzulę from i niejawny warunek where. Alias this dotyczy niejawnie elementów kolekcji bids.

Filtre kolekcji **nie są** wykonywane w pamięci. Kolekcja ofert nie musi być zainicjalizowana w momencie wywołania filtru. Nawet po jego użyciu pozostanie niezainicjalizowana. Filtrów nie można stosować dla kolekcji ulotnych lub wyników zapytań. Są dostępne jedynie dla trwałych kolekcji obiektów powiązanych z sesją Hibernate.

Jedyną wymaganą klauzulą w zapytaniu HQL jest from. Ponieważ filtr kolekcji ma niejawną klauzulę from, podany filtr jest całkowicie poprawny:

```
List results = session.createFilter(item.getBids(), "").list();
```

Nie jedną osobę (włączając w to również twórców tego elementu) zdziwiło, że ten trywialny filtr potrafi być użyteczny! Nic nie stoi na przeszkodzie, by dokonać podziału elementów kolekcji na strony.

```
List results = session.createFilter(item.getBids(), "")  
    .setFirstResult(50)  
    .setMaxResults(100)  
    .list();
```

Najczęściej jednak dla dzielonych na strony elementów stosuje się klauzulę order by.

Nawet jeśli nie potrzebujemy klauzuli from w filtrze kolekcji, można ją dodać. Filtr kolekcji nie musi zwracać filtrowanych elementów kolekcji. Następujące zapytanie zwraca dowolny obiekt Category z tą samą nazwą co kategoria podanej kolekcji.

```
String filterString = "select other from Category other where this.name =  
other.name";  
List results = session.createFilter(cat.getChildCategories(),  
filterString).list();
```

Kolejne zapytanie zwraca kolekcję obiektów User reprezentujących użytkowników, którzy złożyli ofertę na dany przedmiot:

```
List results = session.createFilter(item.getBids(), "select  
this.bidder").list();
```

Następne zapytanie zwraca wszystkie oferty tych użytkowników (także te dotyczące innych przedmiotów):

```
List results = session.createFilter(item.getBids(), "select  
elements(this.bidder.bids")  
.list();
```

Zapytanie wykorzystuje specjalną funkcję HQL o nazwie `elements()` (omawiamy ją dokładniej w dalszej części rozdziału), by wybrać wszystkie elementy kolekcji.

Najważniejszym powodem istnienia filtrów kolekcji jest zapewnienie aplikacjom możliwości pobrania niektórych elementów kolekcji bez inicjalizacji całej jej zawartości. W przypadku bardzo dużych kolekcji istotna staje się wydajność. Następujące zapytanie pobiera wszystkie oferty złożone przez użytkownika w ostatnim tygodniu.

```
List results = session.createFilter(user.getBids(), "where this.created
> :oneWeekAgo")
.setTimestamp("oneWeekAgo", oneWeekAgo)
.list();
```

Także to zapytanie nie inicjalizuje kolekcji `bids` obiektu `User`.

7.5.3. Podzapytania

Podzapytania to istotny bardzo użyteczny element języka SQL. Podzapytanie to zapytanie SQL osadzone we wnętrzu innego zapytania — najczęściej w klauzulach `select`, `from` lub `where`.

Język HQL obsługuje podzapytania w klauzuli `where`. Trudno wymyślić wiele użytecznych zastosowań podzapytań w klauzuli `from`, ale z drugiej strony podzapytania klauzuli `select` mogą być przydatne (może pamiętaś z rozdziału 3., że odwzorowanie właściwości pochodnej jest tak naprawdę podzapytaniem w `select`). Nie wszystkie systemy bazodanowe obsługiwane przez Hibernate mają wbudowaną obsługę podzapytań. Na przykład baza danych MySQL dopiero od wersji 4.1 obsługuje podzapytania. Osiagniecie maksymalnej przenośności aplikacji wymaga rezygnacji z tego elementu.

Wyniki podzapytania mogą zawierać pojedynczy wiersz lub wiele wyników. Najczęściej podzapytania zwracające jeden wiersz dokonują agregacji. Następujące podzapytanie zwraca łączną liczbę przedmiotów sprzedanych przez użytkownika. Zewnętrzne zapytanie zwraca wszystkich użytkowników, którym udało się sprzedać ponad 10 przedmiotów.

```
from User u where 10 < (
    select count(i) from u.items i where i.successfulBid is not null
)
```

Jest to tak zwane **zapytanie skorelowane** — korzysta z aliasu `u` zapytania zewnętrznego. Poniżej znajduje się przykład **zapytania nieskoreowanego**:

```
from Bid bid where bid.amount + 1 >= (
    select max(b.amount) from Bid b
)
```

Podzapytanie z przykładu zwraca maksymalną oferowaną kwotę w całym systemie. Zapytanie główne zwraca wszystkie oferty, które tylko o dolara różniły się od maksimum.

Zauważ, że w obu przypadkach podzapytanie znalazło się w nawiasach. Nie można ich opuścić.

Podzapytania nieskorelowane są nieszkodliwe. Nie ma powodów, dla których należałoby ich unikać. Zawsze też można je zapisać jako dwa zapytania, bo od siebie nie zależą. W przypadku podzapytań skorelowanych warto pomyśleć o ich wydajności. W rozbudowanej bazie danych koszt prostego podzapytania skorelowanego jest podobny do kosztu złączenia. Niestety, tego rodzaju podzapytań nie można zapisać jako kilku następujących po sobie i w pełni niezależnych zapytań.

Jeśli podzapytanie zwraca wiele wierszy, najczęściej stosuje się je wraz z **kwantyfikatorami**. Język ANSI SQL (i HQL) definiuje następujące kwantyfikatory:

- ◆ any,
- ◆ all,
- ◆ some (synonim any),
- ◆ in (synonim = any).

Poniższe zapytanie zwraca przedmioty, w których wszystkie oferty są mniejsze od 100:

```
from Item item where 100 > all (select b.amount from item.bids b)
```

Kolejne zapytanie zwraca wszystkie przedmioty z ofertami większymi od 100:

```
from Item item where 100 < any (select b.amount from item.bids b)
```

Następne zapytanie zwraca przedmioty z ofertami równymi dokładnie 100:

```
from Item item where 100 = some ( select b.amount from item.bids b )
```

To samo wykonuje następujące zapytanie:

```
from Item item where 100 in ( select b.amount from item.bids b )
```

HQL obsługuje specjalną skróconą składnię dla podzapytań, które korzystają z elementów lub indeksów kolekcji. Poniższe zapytanie używa specjalnej funkcji HQL o nazwie `elements()`.

```
List list = session.createQuery("from Category c where :item in  
elements(c.items)")  
.setEntity("item", item)  
.list();
```

Zapytanie zwraca wszystkie kategorie, do których należy przedmiot, i jest równoważne następującemu HQL, w którym podzapytanie występuje bardziej jawnie:

```
List list = session.createQuery("from Category c where :item in  
(from c.items)")  
.setEntity("item", item)  
.list();
```

Poza funkcją `elements()` HQL zapewnia również funkcje `indices()`, `maxelement()`, `minelement()`, `maxindex()`, `minindex()` oraz `size()`. Wszystkie one są w jakiś sposób powiązane z podzapytaniami wykonywanymi dla przekazanej kolekcji. Więcej informacji na ich temat znajduje się w dokumentacji Hibernate. Tutaj ich nie przedstawiamy, bo są rzadko stosowane.

Podzapytania to złożone zagadnienie. Warto zadać sobie pytanie, czy warto ich często używać, bo niejednokrotnie zapytania z podzapytaniami udaje się zamienić na pojedyncze zapytania ze złączeniami i agregacją. Czasem jednak okazują się niezbędne.

Mamy nadzieję, że przekonując pokazaliśmy elastyczność i łatwość użycia zapytań Hibernate. Język HQL oferuje niemalże wszystkie elementy dostępne w ANSI SQL. Niestety, czasem zdarzają się sytuacje, w których trzeba napisać zapytanie w SQL, szczególnie gdy chce się skorzystać ze specyficznych elementów oferowanych przez używany system bazodanowy i mocno wykraczających poza standard ANSI.

7.5.4. Zapytania SQL

Wydaje nam się, że istnieją dwa dobre przykłady użycia w Hibernate zapytań SQL: HQL nie oferuje mechanizmu podpowiedzi SQL i nie obsługuje zapytań hierarchicznych (na przykład klauzuli CONNECT BY z Oracle). Oczywiście nie sądzimy, że są to jedyne przykłady przydatności zwykłego SQL.

W tych rzadkich przypadkach nic nie stoi na przeszkodzie, by bezpośrednio skorzystać z interfejsu JDBC. Niestety, będzie się to wiązało ze żmudnym przekształcaniem obiektu ResultSet z JDBC na graf obiektów. Można tego uniknąć, stosując wbudowaną w Hibernate obsługę zapytań SQL.

Najpierw jednak trzeba nauczyć się pewnej sztuczki. Wyniki zapytania SQL mogą zwrócić stan wielu encji w jednym wierszu danych, a nawet stan wielu egzemplarzy tej samej encji. Musimy więc pomóc w rozróżnieniu poszczególnych encji. Hibernate używa specjalnego schematu nazewnictwa dla kolumn wyników, by poprawnie odwzorować zawarte w kolumnach dane na właściwości konkretnych egzemplarzy. W miarę możliwości chcemy uniknąć uwidaczniania schematu nazewnictwa użytkownika. Do określania nowych nazw kolumn warto skorzystać z aliasów kolumn i symboli zastępczych.

Następujące zapytanie SQL przedstawia, w jaki sposób mogą wyglądać symbole zastępcze (umieszczone w nawiasach klamrowych):

```
String sql = "select u.USER_ID as {uzer.id}, u.FIRSTNAME as {uzer.firstname}, "
+ "u.LASTNAME as {uzer.lastname} from USERS u";
```

Każdy symbol zastępczy zawiera nazwę właściwości w stylu HQL. Wywołując zapytanie, musimy również dostarczyć klasę encyjną, której dotyczy nazwa uzer w symbolach. W ten sposób Hibernate dowiaduje się o typie zwracanym przez zapytanie.

```
List results = session.createSQLQuery(sql, "uzer", User.class).list();
```

Jeśli klasa User jest odwzorowana na tabelę USERS, podawanie wszystkich kolumn i ich odwzorowań byłoby żmudne. Istnieje wygodny skrót:

```
List results = session.createSQLQuery("select {uzer.*} from USERS uzer",
"uzer", User.class).list();
```

Symbol zastępczy {user.*} zostaje automatycznie zastąpiony listą odwzorowanych nazw kolumn i ich aliasami, by pobrać wszystkie właściwości klasy User. Nazwa z symbolu musi być taka sama jak nazwa aliasu tabeli w zapytaniu SQL (w przykładzie jest to user).

Poniższe rozwiązanie działa, ale go nie polecamy:

```
List results = session.createSQLQuery("select {users.*} from users".
    "users", User.class).list();
```

W tym przykładzie nie ma jawnego określenia aliasu. Niejawny alias ma tę samą nazwę co nazwa tabeli (zwróć uwagę na zapis jej małymi literami).

Zapytanie SQL może zwrócić krotkę z encjami (Hibernate reprezentuje ją jako tablicę Object[]):

```
List tuples = session.createSQLQuery("select {u.*}, {b.*} from USERS u inner
join BID b"
    + " where u.USER_ID = b.BIDDER_ID",
    new String[] { "u", "b" },
    new Class[] {User.class, Bid.class} )
.list();
```

Nazwane zapytanie SQL można umieścić poza kodem aplikacji w pliku odwzorowania Hibernate. Wtedy do określenia zwracanego typu służy element <return>.

```
<sql-query name="findUsersAndBids"><![CDATA[
    select {u.*}, {b.*} from USERS u inner join BID b
    where u.USER_ID = b.BIDDER_ID
]]>
<return alias="u" class="User"/>
<return alias="b" class="Bid"/>
</sql-query>
```

Ponieważ zapytania SQL są ściśle związane z rzeczywistymi tabelami i kolumnami (a nie ich odwzorowaniami), zalecamy umieszczanie ich definicji w dokumencie odwzorowania zamiast osadzania ich w kodzie Javy.

W pewnych szczególnych sytuacjach, gdy potrzeba jeszcze większej swobody sterowania wykonaniem zapytania lub uruchomienia procedury zapamiętaną, Hibernate może zwrócić referencję do obiektu połączenia JDBC. Wywołanie metody session.connection() zwraca aktualnie aktywne w danej sesji połączenie JDBC. Zamknięcie połączenia nie należy do obowiązków programisty aplikacji — wykonuje on własne zapytanie SQL, dalej używa sesji i na końcu zamkna obiekt Session. To samo dotyczy transakcji — nie wolno ich zatwierdzać ani wycofywać (chyba że samemu zajmuje się zarządzaniem połączaniami bazodanowymi dla Hibernate bez pomocy puli połączeń i kontenera źródła danych).

Odpowiedzi na pytania

W jaki sposób w Hibernate uruchomić procedurę zapamiętaną? Hibernate 2.x nie obsługuje procedur zapamiętywanych w sposób bezpośredni. Należy pobrać połączenie JDBC i samemu wykonać procedurę. Hibernate 3 zawiera bezpośrednią obsługę procedur zapamiętywanych, która dopuszcza odwzorowanie operacji CUD dla encji na procedury zapamiętane i bezpośrednie wywoływanie procedur zapamiętywanych z poziomu interfejsu Hibernate.

W trakcie tworzenia lub testowania zapytań w aplikacji trudno nie natknąć się na jeden z problemów wydajnościowych systemów ORM. Na szczęście wiemy, jak im zaradzić (lub przynajmniej ograniczyć skutki). Cały proces nosi nazwę **optymalizacji pobierania obiektów**. Prześledźmy rozwiązania typowych problemów.

7.6. Optymalizacja pobierania obiektów

Uzyskanie odpowiedniej wydajności aplikacji powinno na pierwszym miejscu polegać na możliwie najlepszym ustawieniu opcji związanych ze strategią sprowadzania i użyciem pośredników (patrz rozdział 4.). Pamiętaj, że włączenie bufora drugiego poziomu uważamy za ostatni krok optymalizacji.

Złączenia sprowadzające, część strategii sprowadzania w trakcie działania aplikacji wprowadzona w tym rozdziale, wymaga dodatkowej uwagi. Niestety, pewnych kwestii nie można rozwiązać zmianami ustawień — można ich tylko unikać, o ile to możliwe.

7.6.1. Rozwiązywanie problemu n+1 pobrania danych

Największym zabójcą wydajności w aplikacjach z obiektami trwałymi i bazami danych SQL jest **problem n+1 pobrania danych**. W trakcie boju o poprawę wydajności aplikacji stosującej Hibernate to pierwszy element, któremu warto się dobrze przyjrzeć.

Zaleca się, by niemal dla wszystkich asocjacji włączać leniwy inicjalizację. Najczęściej oznacza to ustawienie wszystkich kolekcji na `lazy="true"` i taką zmianę niektórych asocjacji jeden-do-jednego i wiele-do-jednego, by domyślnie nie wykonywać złączeń zewnętrznych. To jedyny sposób, by uniknąć pobierania wszystkich obiektów z bazy danych w każdej transakcji. Niestety, powoduje to również powstanie problemu n+1 pobrania. Najłatwiej przedstawić ten problem, analizując proste zapytanie, które pobiera wszystkie obiekty `Item` konkretnego użytkownika:

```
Iterator items = session.createCriteria(Item.class)
    .add(Expression.eq("item.seller", user))
    .list()
    .iterator();
```

Zapytanie zwraca listę obiektów `Item`, w których kolekcja `bids` nie jest zainicjalizowana. Przypuśćmy, że teraz zamierzamy odnaleźć maksymalną ofertę dla każdego przedmiotu. Oto jeden ze sposobów wykonania tego zadania:

```
List maxAmounts = new ArrayList();
while (items.hasNext()) {
    Item item = (Item) items.next();
    BigDecimal maxAmount = new BigDecimal("0");
    for (Iterator b = item.getBids().iterator(); b.hasNext(); ) {
        Bid bid = (Bid) b.next();
        if ( bid.getAmount().compareTo(maxAmount) == 1 )
            maxAmount = bid.getAmount();
```

```
        }
        maxAmounts.add(new maxAmount(item.getId(), maxAmount));
    }
```

To rozwiązanie ma poważny problem (nie wspominając już o tym, iż lepiej wykonać je funkcją agregującą w bazie danych). Za każdym razem, gdy pobieramy kolekcję ofert kolejnego obiektu Item, Hibernate musi ją pobierać z bazy danych. Jeśli pierwsze zapytanie zwróciło 20 przedmiotów, cała transakcja wymaga wykonania jednego początkowego select i 20 dodatkowych pobierających kolekcje. Opóźnienia w pobieraniu mogą nie być do zaakceptowania, jeśli system pobiera dane ze odległej bazy danych. Przedstawiona operacja jest daleka od wydajnego rozwiązania, więc z pewnością nikt by jej nie użył. Problem $n+1$ pobrań danych najczęściej ukrywa się w bardziej złożonej logice aplikacji i niełatwo go wychwycić.

Pierwsza próba poradzenia sobie z problemem może polegać na wyłączeniu **sprawdzania wsadowego**. Zmiana odwzorowania kolekcji bids może przynieść następujący efekt:

```
<set name="bids" lazy="true" inverse="true" batch-size="10">
```

Po wyłączeniu jednokrotnego sprawdzania wielu kolekcji Hibernate pobiera 10 kolejnych kolekcji wraz z pierwszą. Redukuje to problem $n+1$ pobrań do problemu $n/10+1$ pobrań. W wielu aplikacjach pozwoli to osiągnąć odpowiednio małe opóźnienie. Z drugiej strony oznacza, że w innych transakcjach kolekcje pobierane są niepotrzebnie. Jak się okazuje, to jeszcze nie najlepszy sposób zmniejszania liczby zapytań do bazy danych.

Znacznie, znacznie lepsze rozwiązanie polega na zastosowaniu w HQL agregacji, by to baza danych wyliczyła maksymalne oferty. W ten sposób unikamy problemu.

```
String query = "select MaxAmount(item.id, max(bid.amount))"
    + " from Item item join item.bids bid"
    + " where item.seller = :user group by item.id";
```

```
List maxAmounts = session.createQuery(query).setEntity("user", user).list();
```

Niestety, nie zawsze w ten sposób uda się obejść problem. Przypuśćmy, że chcemy wykonać bardziej złożone obliczenia dla ofert niż tylko wyznaczenie maksimum i preferujemy wykonanie ich w aplikacji Javy.

Mogimy włączyć sprawdzanie wyprzedzające na poziomie dokumentu odwzorowania:

```
<set name="bids" inverse="true" outer-join="true">
```

Atrybut outer-join jest dostępny dla kolekcji i innych asocjacji. Wymusza na Hibernate wczytanie asocjacji z wyprzedzeniem, używając złączenia zewnętrznego.

Przypomnijmy raz jeszcze, że zapytania HQL ignorują atrybut outer-join. Z drugiej strony stosują go zapytania przez kryteria.

Z punktu widzenia przedstawianej transakcji problem został rozwiązany. Mogimy wczytać wszystkie oferty w jednym zapytaniu razem z przedmiotami. Niestety, każda inna transakcja pobierająca obiekty za pomocą metod get(), load()

lub zapytań przez kryteria także pobierze oferty, choć w ogóle nie musi ich potrzebować. Pobieranie niepotrzebnych danych obciąża zarówno aplikację, jak i serwery baz danych, niejednokrotnie zmniejszając współpracę całego systemu przez nakładanie zbyt wielu blokad na poziomie bazy danych.

Z tego względu uważamy, iż włączanie sprawdzania wyprzedzającego na poziomie pliku odwzorowania niemal zawsze jest złym pomysłem. Atrybut outer-join z Hibernate to coś na kształt konia trojańskiego niskiej wydajności dla odwzorowań kolekcji (na szczęście domyślnie jest wyłączony). Włączenie atrybutu ma sens, ale tylko w niektórych asocjacjach <many-to-one> lub <one-to-one> (domyślnie Hibernate stosuje dla nich tryb auto, o którym więcej mówimy w podrozdziale 4.4.6). W asocjacjach bywa „zabójczy” dla wydajności.

Zalecamy skorzystać z rozwiązania oferowanego przez Hibernate już w trakcie działania aplikacji, czyli deklaracji strategii sprowadzania asocjacji w kodzie Javy.

```

List results = session.createCriteria(Item.class)
    .add(Expression.eq("item.seller", user))
    .setFetchMode("bids", FetchMode.EAGER)
    .list();

// Czyni wyniki unikatowymi.
Iterator items = new HashSet(results).iterator();

List maxAmounts = new ArrayList();
for ( ;items.hasNext(); ) {
    Item item = (Item) items.next();
    BigDecimal maxAmount = new BigDecimal("0");
    for ( Iterator b = item.getBids().iterator(); b.hasNext(); ) {
        Bid bid = (Bid) b.next();
        if ( bid.getAmount().compareTo(maxAmount) == 1 )
            maxAmount = bid.getAmount();
    }
    maxAmounts.add(new maxAmount(item.getId(), maxAmount));
}

```

Sprowadzanie wsadowe i wyprzedzające zostało wyłączone na poziomie plików odwzorowań, więc domyślnie kolekcja jest leniwa. Sprowadzanie wyprzedzające włączamy jawnie metodą setFetchMode() tylko w wymagających tego sytuacjach. Wcześniej w rozdziale wskazaliśmy, że jest to równoznaczne użyciu fetch jako klauzuli from zapytania HQL.

Przedstawiony kod ma pewną wadę: lista zwróconych wyników z zapytaniami przez kryterium nie musi być różna. Zapytanie pobierające kolekcję za pomocą złączenia zewnętrznego z pewnością będzie zawierało duplikaty przedmiotów. To na aplikację spada odpowiedzialność za zapewnienie unikalności, jeśli jest wymagana. W przykładowym kodzie duplikaty z wyników usuwa obiekt HashSet którego używamy do przejścia przez wyniki.

Znaleźliśmy ogólne rozwiązanie problemu n+1 pobrań danych. Zamierzamy pobierać tylko główne obiekty w pierwszym wywołaniu, a następnie sprowadzić dane asocjacji w trakcie przechodzenia przez graf obiektów, stosujemy procedurę dwuetapową:

1. Pobieramy wszystkie potrzebne dane w pierwszym zapytaniu, podając które dokładnie asocjacje będą wykorzystywane w jednostce zadaniowej.
2. Przechodzimy przez graf sprowadzonych wcześniej obiektów, co jest równie szybkie jak przechodzenie przez obiekty wykonane od podstaw w aplikacji.

To jedyne prawdziwe rozwiązanie problemu niedopasowania świata obiektowego, w którym dane uzyskuje się przez nawigację, i świata relacyjnego, gdzie dane uzyskuje się przez złączenia.

Istnieje jeszcze jedno rozwiązanie problemu $n+1$ pobrań. Dla niektórych klas lub kolekcji z niewielką liczbą egzemplarzy można wszystkie dane przechowywać w buforze drugiego poziomu i w ten sposób unikać zapytań bazodanowych. Jeżeli tylko ma to sens, warto z tej propozycji korzystać (w przypadku kolekcji bids klasy Item bufor drugiego poziomu nie wchodzi w grę, bo danych byłoby za dużo).

Problem $n+1$ pobrań może pojawić się w momencie stosowania metody `list()` obiektu `Query` do pobierania wyników. Jak wspomnieliśmy, problem często ukrywa się w bardziej złożonej logice. Zalecamy stosować się do porad wskazanych w podrozdziale 4.4.7, by odnaleźć takie sytuacje. Generowanie zbyt wielu pobrań możliwe jest również w metodach `find()` (metoda skrócona interfejsu `Session`), `load()` i `get()`.

Istnieje trzecia metoda interfejsu, której jeszcze nie opisywaliśmy. Jej opis okazuje się niezwykle ważny, bo to właśnie ona odpowiada za $n+1$ pobrania danych!

7.6.2. Zapytania `iterate()`

Metoda `iterate()` interfejsów `Session` i `Query` zachowuje się inaczej niż metody `find()` i `list()`. Powstała przede wszystkim dlatego, by możliwe stało się wszystkich zalet bufora drugiego poziomu.

Rozważmy następujący kod:

```
Query categoryByName = session.createQuery("from Category c where c.name like :name");
categoryByName.setString("name", categoryNamePattern);
List categories = categoryByName.list();
```

Zapytanie skutkuje wykonaniem instrukcji `select` z SQL ze wszystkimi kolumnami tabeli `CATEGORY` podanymi w klauzuli `select`.

```
select CATEGORY_ID, NAME, PARENT_ID from CATEGORY where NAME like ?
```

Jeśli spodziewamy się, że kategorie są już zbuforowane w sesji lub buforze drugiego poziomu, potrzebujemy tylko identyfikatorów (kluczy w buforze). Ograniczy to znaczco ilość danych przesyłanych z bazy danych. Następujące polecenie SQL będzie nieco bardziej wydajne:

```
select CATEGORY_ID from CATEGORY where NAME like ?
```

Do jego utworzenia wykorzystamy metodę `iterate()`:

```
Query categoryByName = session.createQuery("from Category c where c.name like
    :name");
categoryByName.setString("name", categoryNamePattern);
List categories = categoryByName.list();
```

Początkowe zapytanie pobiera jedynie klucze główne kategorii. Następnie przechodzimy przez wyniki, a Hibernate poszukuje obiektów Category w buforze sesji i drugiego poziomu. Jeśli danych nie odnajdzie, wykonuje dodatkowe polecenie `select`, by je pobrać na podstawie znanego klucza głównego.

W większości sytuacji optymalizacja okazuje się niewielka. Znacznie ważniejsze jest ograniczenie pobieranych wierszy niż liczby pobieranych kolumn. Jeśli jednak obiekty zawierają długie teksty, rozwiązanie to zminimalizuje ilość danych przesyłanych siecią i tym samym zmniejszy opóźnienia.

Przyjrzymy się jeszcze jednej optymalizacji, która jednak nie jest dostępna w każdej sytuacji. Do tej pory mówiliśmy (w rozdziale 5.) jedynie o buforowaniu wyników na podstawie identyfikatora (włączając w to niejawne wyszukiania, na przykład przy wczytywaniu asocjacji leniwych). Możliwe jest również buforowanie wyników zapytań Hibernate.

7.6.3. Buforowanie zapytań

Dla aplikacji, które przeprowadzają wiele zapytań i niewiele wstawień, usunięć i aktualizacji, buforowanie zapytań może mieć duży wpływ na wydajność. Z drugiej strony, gdy aplikacja często zapisuje dane, bufor zapytań nie zostanie efektywnie wykorzystany. Hibernate automatycznie usuwa wpis z bufora zapytań w momencie wykonywania dowolnego wstawienia, aktualizacji lub zapisu dla dowolnej tabeli zawartej w zapytaniu.

Podobnie jak nie ma sensu buforować wszystkich klas i kolekcji, tak nie wszystkie zapytania skorzystają z buforowania. Jeśli ekran wyszukiwania zawiera wiele różnorodnych kryteriów ograniczających, istnieje małe prawdopodobieństwo, że ktoś inny wybierze te same kryteria. Lepiej więc nie włączać buforowania dla tego rodzaju zapytania.

Bufor zapytań nie zapamiętuje encji zwracanych przez zbiór wyników, a jedynie wartości identyfikatorów. Z drugiej strony, Hibernate zapamięta wszystkie dane zapytania stosującego projekcję. Na przykład zapytanie "select u, b.created from User u, Bid b where b.bidder = u" spowoduje zbuforowanie identyfikatorów użytkowników i obiektów dat złożenia ofert. Za rzeczywiste zbuforowanie encji odpowiada bufor drugiego poziomu (i oczywiście bufor sesji). Jeśli aplikacja przedstawione zapytanie wykona ponownie, Hibernate będzie posiadał daty złożenia ofert, natomiast obiekty użytkowników poszuka w obu buforach na podstawie zapamiętanego ich identyfikatora. Bufor zapytań przypomina w działaniu strategię wyszukiwania metody `iterate()`.

Buforowanie zapytań włącza się w globalnych właściwościach Hibernate.

```
hibernate.cache.use_query_cache true
```

Przedstawione ustawienie nie wystarcza, by system zbuforował wyniki zapytania. Domyślnie zapytania ignorują bufor. By włączyć buforowanie konkretnego zapytania (czyli zapewnić umieszczenie wyników w buforze i pobieranie ich z niego przy drugim użyciu), należy użyć interfejsu Query.

```
Query categoryByName = session.createQuery("from Category c where c.name = :name");
categoryByName.setString("name", categoryName);
categoryByName.setCacheable(true);
```

Niestety, nawet ten zabieg nie zapewnia pełnej szczegółowości. Różne zapytania wymagać mogą różnego okresu ważności (zasad unieważniania). Dla każdego zapytania można określić osobny, nazwany **region** bufora.

```
Query userByName = session.createQuery("from User u where u.name = :uname");
userByName.setString("uname", userName);
userByName.setCacheable(true);
userByName.setCacheRegion("UserQueries");
```

Zasady unieważnienia określa się na podstawie nazw regionów. Po włączeniu buforowania zapytań dostępne są regiony:

- ◆ domyślny region buforowanych zapytań:

```
net.sf.hibernate.cache.QueryCache,
```

- ◆ każdy nazwany region,
- ◆ **bufor znaczników czasowych**, net.sf.hibernate.UpdateTimestampsCache, który stanowi specjalny region przechowujący znaczniki czasowe najnowszych aktualnień każdej tabeli.

Hibernate używa bufora znaczników czasowych do sprawdzenia, czy zapamiętane wyniki zapytania nie są przestarzałe. Sprawdza, czy w buforze tym data ostatniej aktualizacji, wstawienia lub usunięcia jest nowsza od daty utworzenia bufora zapytania. Jeśli tak, bufor zostaje zniszczony, a system przeprowadza standardowe zapytanie. By nie uzyskać trudnych do wykrycia błędów, warto tak dobrać czas unieważniania bufora znaczników czasowych, by zawarte tam dane nie wygasły szybciej niż zapytanie. Można pójść na łatwiznę i w ogóle wyłączyć unieważnianie bufora znaczników ze względu na czas istnienia.

Kilką ostatnich słów na temat optymalizacji. Pamiętaj, że zagadnienia typu $n+1$ pobrań danych potrafią spowolnić aplikację w niesamowity sposób. Staraj się unikać tego problemu, dobierając odpowiednią strategię sprowadzania. Przed włączeniem buforowania czegokolwiek warto sprawdzić, czy na pewno sposób pobierania jest poprawny.

Z naszego punktu widzenia bufor drugiego poziomu to ważny element ORM, ale nie powinien stanowić pierwszej linii obrony przy niewystarczającej wydajności. Błędów w projekcie zapytań lub przesadnej złożoności modelu obiektowego na pewno nie ukryje włączenie buforowania „wszystkiego, co się da”. Jeśli aplikacja działa z odpowiednią wydajnością dopiero po kilku godzinach, a nawet dniach (co oznacza wypełnienie bufora), warto poszukać własnych błędów projektowych, niewydajnych zapytań lub problemu $n+1$ pobrań.

7.7. Podsumowanie

Nie sądzimy, że po przeczytaniu tego rozdziału tylko jeden raz będziesz pamiętał wszystkie sztuczki i wiedział, jak stosować zapytania HQL i kryteria. Rozdział ten powinien stanowić dobry „trwały magazyn wiedzy” w codziennej pracy. Zapraszamy do czytania fragmentów rozdziału, gdy tylko będzie to konieczne.

Przykładowy kod przedstawiony w rozdziale dotyczy trzech podstawowych technik tworzenia zapytań w Hibernate: HQL, zapytania przez kryteria wraz z zapytaniami przez przykład i bezpośrednie wykonywanie polecień SQL.

Język HQL wydaje się najbardziej użyteczny. Napisane w nim zapytania łatwo zrozumieć. Poza tym stosuje nazwy klas trwałych i właściwości zamiast nazw tabel i kolumn. Dopuszcza ograniczanie zbioru wyników i projekcje, a nawet pozwala pobrać wszystkie obiekty implementujące wskazany interfejs. Można stosować funkcje SQL i operatory logiczne, ale zawsze odbywa się to na poziomie nazw właściwości i klas. Nazwane parametry umożliwiają dowiązywanie argumentów zapytań w sposób bezpieczny pod kątem typów, jak i samej aplikacji. Hibernate implementuje zapytania raportujące — wielu innym systemom ORM brakuje tego elementu.

Większość wymienionych zalet obejmuje również zapytania przez kryteria. Zamiast tworzyć ciągi znaków opisujące dane do pobrania, używa się wywołań metod i kreowania obiektów. Tak zwane obiekty przykładowe łączy się z kryteriami, by na przykład pobierać „wszystkie przedmioty wyglądające jak przykłady”.

Zawsze najważniejsze okazuje się wydajne pobieranie powiązanych obiektów — jest to równoznaczne jak najdokładniejszemu wskazaniu fragmentu grafu obiektów, z którego zamierza się korzystać, by w miarę możliwości został pobrany w jednym zapytaniu. Hibernate stosuje leniową, wyprzedzającą i wsadową strategię sprowadzania danych określana w plikach odwzorowań lub dynamicznie w trakcie działania aplikacji. Złączenia dla asocjacji i iteracja przez wyniki pozwala w wielu sytuacjach zapobiec problemowi **n+1** poobrań danych. Największy spadek wydajności aplikacje notują przy częstych zapytaniach dotyczących niewielkich porcji danych — warto więc je zminimalizować. Z drugiej strony nie należy przesadzać, zbyt wiele zbędnych danych pobieranych w jednym zapytaniu również zmniejsza wydajność.

Najlepszy sposób tworzenia zapytań i idealna strategia sprowadzania obiektów silnie zależy od kreowanej aplikacji. Przykłady przedstawione w tym rozdziale powinny dobrze przygotować do określania najlepszych metod postępowania we własnych programach.

Tworzenie aplikacji stosujących Hibernate



W rozdziale:

- ◆ Projektowanie aplikacji Hibernate wykorzystującej serwlety
- ◆ Integracja z EJB w środowisku zarządzanym
- ◆ Tworzenie implementacji używającej transakcji aplikacyjnych
- ◆ Obsługa starszych danych i tworzenie dzienników audytowych

Hibernate zaprojektowano w taki sposób, by mógł być wykorzystywany w dowolnej możliwej do przewidzenia architekturze programistycznej (oczywiście przy założeniu, że aplikacja powstaje w języku Java). Nic nie stoi na przeszkodzie, by działał wewnętrz systemu serwletów — czyli dostępnych szkieletach aplikacji internetowych: Struts, WebWork lub Tapestry — wewnętrz kontenera EJB, klienta Swing, kontenera lekkiego lub nawet serwera JMX, na przykład JBoss.

Każde z tych środowisk wymaga infrastruktury integrującej Hibernate z dostępnymi sposobami zarządzania żądaniami, transakcjami i zasobami bazodanowymi. Rzeczywiście Hibernate zawiera komponenty ułatwiające integrację w typowych środowiskach, włączając w to obsługę JTA, źródeł danych dowiązanych dzięki JNDI, JMX, JCA i zarządców transakcji wszystkich popularnych serwerów aplikacyjnych. Co więcej, niektóre szkielety tworzenia aplikacji, na przykład Spring lub Keel, zawierają wbudowaną obsługę Hibernate. Inne zawierają moduły dodatkowe zapewniające taką obsługę, na przykład Tapestry, Apache Avalon i PicoContainer. Serwer aplikacji JBoss zawiera specjalną obsługę archiwizacyjną dla Hibernate i integrację Hibernate jako komponentu zarządzanego zgodnie z JMX.

Pomimo tych wielu opcji — lub też **właśnie z ich powodu** — trudno się zdecydować, w jaki sposób zintegrować Hibernate z konkretną architekturą bazującą na języku Java. Bez wątpienia trzeba napisać kod związany z infrastrukturą, by wprowadzić własny projekt aplikacji. W rozdziale omówimy kilka typowych architektur i wskażemy, jak w każdej z nich zintegrować Hibernate. Nie będziemy jednak zajmować się integracją z konkretnymi szkieletami aplikacji. Nie przewidujemy, że którakolwiek z aplikacji będzie idealnie pasowała do przedstawianych sytuacji. Nie zakładamy również, że przedstawiany kod będzie działał poprawnie w innej aplikacji. Staramy się wskazać kilka typowych wzorców postępowania, choć konkretna ich postać często zależy od docelowej aplikacji. Z tego powodu wszystkie przykłady są napisane w czystym kodzie Javy stosującym jedynie serwlety i interfejs EJB — nie korzystamy z żadnych zewnętrznych szkieletów aplikacji.

W pierwszej części rozdziału omówimy aplikacje warstwowe i wpływ tego sposobu programowania na kod związany z trwałością danych. W drugiej części powrócimy do interesującego tematu — transakcji aplikacyjnych (patrz rozdział 5.) — by przedstawić interesujące przykłady implementacji tego rodzaju transakcji za pomocą Hibernate. W trzeciej części zajmiemy się specjalnymi typami danych (szczególnie starszymi) i pokażemy, w jaki sposób zapewnić ich integrację z Hibernate. Utworzymy też obiekt Interceptor dla Hibernate, który przydaje się w wielu sytuacjach specjalnych.

8.1. Projektowanie aplikacji warstwowych

W rozdziale 1. podkreślaliśmy ogromne znaczenie poprawnego projektowania aplikacji warstwowych. Warstwy pozwalają osiągnąć separację zadań, czyniąc kod bardziej czytelnym przez grupowanie podobnych w działaniu fragmentów. Z drugiej strony warstwy niosą ze sobą pewien koszt — każda kolejna warstwa

zwiększa ilość kodu potrzebą do implementacji nawet prostej funkcji. Im więcej kodu, tym trudniej w przyszłości zmienić funkcjonalność.

Nie zamierzamy formułować żadnych wniosków na temat poprawnej liczby warstw (a już na pewno nie określić zadań każdej z nich), ponieważ „najlepszy” projekt różni się od aplikacji do aplikacji. Poza tym pełny opis zagadnień projektowych wykracza poza ramy niniejszej książki. Wydaje nam się jednak, że zbytnia przesada z kreowaniem rozwiązań architektonicznych to choroba społeczności użytkowników Javy, a nazbyt ambitne podejście do architektur wielowarstwowych odcisnęło piętno na zwiększeniu kosztu wytwarzania oprogramowania i odczuwanej złożoności J2EE. Z drugiej strony, sądzimy, że osobna warstwa trwałości danych wydaje się sensownym rozwiązaniem dla większości aplikacji, a mieszanie logiki biznesowej i prezentacyjnej z kodem trwałości nigdy nie wychodzi nikomu na dobrze.

W niniejszym podrozdziale przedstawimy, w jaki sposób oddzielić kod dotyczący Hibernate od warstw biznesowej i prezentacyjnej: najpierw w środowisku serwletów, a następnie w środowisku EJB. Potrzebujemy prostego przypadku użycia z aplikacji CaveatEmptor, by przedstawić całe zagadnienie.

Gdy użytkownik składa ofertę na przedmiot, aplikacja CaveatEmptor musi wykonać w jednym żądaniu następujące kroki:

1. Sprawdzić, czy kwota podana przez użytkownika jest większa od maksymalnej obecnie oferty dla tego przedmiotu.
2. Sprawdzić, czy aukcja nie została zakończona.
3. Utworzyć nową ofertę dla przedmiotu.

Gdy którykolwiek z kroku nie uda się zrealizować, użytkownik powinien dowiedzieć się o powodzie niezrealizowania żądania. Jeżeli oba sprawdzenia uda się przejść poprawnie, użytkownik powinien dowiedzieć się o złożeniu oferty. Sprawdzenia to reguły biznesowe. Gdy wystąpi błąd połączenia z bazą danych, należy poinformować użytkownika o chwilowej niedostępności systemu (element infrastrukturalny).

Prześledźmy sposób implementacji całego zagadnienia w systemie serwletowym, na przykład serwerze Tomcat.

8.1.1. Użycie Hibernate w systemie serwletowym

Najpierw musimy się dowiedzieć, w jaki sposób uzyskać w aplikacji egzemplarz obiektu Session. Utwórzmy prostą klasę pomocniczą (lub narzędziową), która obsłuży konfigurację i inicjalizację obiektu SessionFactory (patrz rozdział 2.) i tym samym zapewni łatwy dostęp do nowych sesji. Pełny kod tej klasy przedstawia listing 8.1.

Listing 8.1. Prosta klasa narzędziowa dla Hibernate

```
public class HibernateUtil {  
    private static final SessionFactory sessionFactory;  
}
```

```

static { ❶
    try {
        Configuration cfg = new Configuration();
        sessionFactory = cfg.configure().buildSessionFactory(); ❷
    } catch (Throwable ex) {
        ex.printStackTrace(System.out); ❸
        throw new ExceptionInInitializerError(ex);
    }
}

public static Session getSession() throws HibernateException { ❹
    return sessionFactory.openSession();
}
}

```

- ❶ Obiekt SessionFactory trafia do zmiennej statycznej i finalnej. Wszystkie wątki mogą współdzielić tę jedną wartość, ponieważ obiekt SessionFactory jest bezpieczny pod kątem wątków.
- ❷ Obiekt SessionFactory powstaje bloku inicjalizacji statycznej. Blok zostaje wykonany tuż po załadowaniu klasy przez maszynę wirtualną.
- ❸ Proces tworzenia obiektu SessionFactory na podstawie obiektu Configuration został przedstawiony we wcześniejszych rozdziałach.
- ❹ Wyłapujemy i opakowujemy wyjątki Throwable, ponieważ chcemy również wychwycić wyjątki NoClassDefFoundError i inne należące do podklasy Error, a nie tylko wyjątki Exception i RuntimeException. Zawsze warto rejestrować wyjątki — są sytuacje, w których wyjątek inicjalizacji statycznej rzeczywiście wystąpi. Oczywiście należy użyć własnego systemu dziennika zamiast System.out.
- ❺ Klasa narzędziowa zawiera tylko jedną metodę publiczną, metodę fabryki dla nowych sesji. Moglibyśmy również wprowadzić metodę getSessionFactory(), ale przedstawiona wersja zaoszczędza wiersz kodu przy każdym tworzeniu obiektu Session.

Ta bardzo prosta implementacja przechowuje obiekt SessionFactory w zmiennej statycznej. Można nawet przechowywać referencję do SessionFactory w Servlet-Context lub innym rejestrze obejmującym zasięgiem całą aplikację.

Zauważ, że projekt jest w pełni bezpieczny pod kątem klastrów. Obiekt SessionFactory nie ma stanu (nie zawiera żadnej informacji na temat działających transakcji). Wyjątkiem jest sytuacja, w której włączono bufor drugiego poziomu. Niemniej nawet wtedy to na dostawcę bufora spada obowiązek zachowania spójności bufora w całym klastrze. Nic więc nie stoi na przeszkodzie, by tworzyć wiele obiektów SessionFactory (w rzeczywistości ich liczbę warto ograniczyć do minimum, bo zajmują dużą zasobów; koszt ich inicjalizacji również jest znaczny).

Po rozwiązaniu problemu, w którym miejscu umieścić obiekt SessionFactory, możemy przejść do kolejnych kroków implementacji przypadku użycia. Większość aplikacji Java wykorzystuje pewien rodzaj wzorca MVC (*Model-View-Controller*, czyli model, widok i kontroler) dla aplikacji internetowych. Nawet aplikacje bazu-

jące wyłącznie na serwletach stosują wzorzec, używając szablonów JSP lub Velocity do implementacji widoku, by w ten sposób oddzielić sposób prezentacji od logiki biznesowej. Napiszmy serwlet kontrolera.

Prosty serwlet akcji

Stosując podejście MVC, napiszemy kod implementujący przypadek użycia „umieść ofertę” w metodzie execute() akcji nazwanej PlaceBidAction (patrz listing 8.2). Zakładamy użycie szkieletu aplikacji internetowej, więc nie przedstawiamy sposobu odczytu parametrów żądania i sposobu przejścia do kolejnej strony. Przedstawiony kod można nawet umieścić w metodzie doPost() zwykłego serwletu (nie uważamy tej pierwszej implementacji za najlepszą — będziemy ją stale udoskonalać).

Listing 8.2. Implementacja prostego przypadku użycia w jednej metodzie execute()

```
public void execute() {  
  
    Long itemId = ...; // Pobierz wartość z żądania.  
    Long userId = ...; // Pobierz wartość z żądania.  
    BigDecimal bidAmount = ...; // Pobierz wartość z żądania.  
  
    try {  
        Session session = HibernateUtil.getSession(); ❶  
        Transactiontx = session.beginTransaction();  
        try {  
  
            // Wczytaj obiekt Item, którego dotyczy oferta. ❷  
            Item item = (Item) session.load(Item.class, itemId,  
                LockMode.UPGRADE);  
  
            // Sprawdź, czy aukcja nadal trwa. ❸  
            if (item.getEndDate().before(new Date())) {  
                ... // Przejdz do strony błędu.  
            }  
  
            // Sprawdź kwotę najwyższej oferty. ❹  
            Query q = session.createQuery("select max(b.amount)" +  
                " from Bid b where b.item =  
                :item");  
            q.setEntity("item", item);  
            BigDecimal maxBidAmount = (BigDecimal) q.uniqueResult();  
            if (maxBidAmount.compareTo(bidAmount) > 0) {  
                ... // Przejdz do strony błędu.  
            }  
  
            // Dodaj nowy obiekt Bid do obiektu Item. ❺  
            User bidder = (User) session.load(User.class, userId);  
            Bid newBid = new Bid(bidAmount, item, bidder);  
            item.addBid(newBid);  
        }  
    }  
}
```

```

... // Umieść nowy obiekt w zasięgu kolejnej strony.

tx.commit(); ❶

... // Przejdź do strony poprawnego złożenia oferty.

} catch (HibernateException ex) { ❷
    if (tx != null) tx.rollback();
    throw ex;
} finally { ❸
    session.close();
}
} catch (HibernateException ex) {
    ... // Zgłoś wyjątek specyficzny dla aplikacji.
} ❹
}

```

- ❶ Najpierw pobieramy nowy obiekt Session z klasy narzędziowej i uruchamiamy transakcję bazodanową.
- ❷ Wczytujemy obiekt Item z bazy danych na podstawie wartości identyfikatora i zakładamy blokadę pesymistyczną (zabezpiecza to przed wprowadzaniem w tym samym czasie dwóch nowych ofert dla jednego przedmiotu).
- ❸ Jeśli data zakończenia aukcji jest wcześniejsza od aktualnej, przechodzimy do strony błędu. Najczęściej dla tego rodzaju błędu potrzeba bardziej wyrafinowanej obsługi, na przykład pełnego komunikatu z wyjaśnieniem przyczyn.
- ❹ Zapytaniem HQL sprawdzamy, jaka jest najwyższa oferta dla aktualnego przedmiotu w bazie danych. Jeśli nowa oferta jest mniejsza od już istniejącej, przechodzimy do strony błędu.
- ❺ Jeśli wszystkie sprawdzenia przeszliśmy pomyślnie, umieszczamy nową ofertę w obiekcie Item. Nie musimy wykonywać ręcznego zapisu — obiekt oferty zostanie zapisany automatycznie z racji trwałości przechodniej (stosowanej kaskadowo dla wszystkich obiektów Bid z obiektu Item).
- ❻ Zatwierdzenie transakcji bazodanowej powoduje zapisanie aktualnego stanu sesji do bazy danych.
- ❼ Jeśli dowolna metoda wewnętrznego bloku try-catch zgłosi wyjątek, musimy wycofać transakcję i ponownie zgłosić wyjątek.
- ❽ Niezależnie od sytuacji sesję musimy zamknąć, by zwolnić zasoby bazy danych.
- ❾ Zewnętrzny blok try-catch odpowiada za wyjątki zgłasiane przez Session.close() i Transaction.rollback() oraz za ponowne zgłoszenia wyjątków wewnętrznych.

Pierwszą wadą przedstawionego rozwiązania jest mało czytelny zlepek tworzony przez cały kod obsługujący sesje, transakcje i wyjątki. Ponieważ najczęściej sposób obsługi tych elementów będzie podobny we wszystkich akcjach, warto

go scentralizować. Jedno z rozwiązań może polegać na umieszczeniu go w metodzie execute() abstrakcyjnej klasy bazowej dla wszystkich aukcji.

Przy dostępie do nowej oferty na stronie potwierdzającej złożenie oferty pojawi się problem **leniwejinicjalizacji**. Gdy dojdziemy do strony JSP, sesja Hibernate już dawno jest zamknięta, więc nie możemy skorzystać z leniwej asocjacji. Zastanów się nad tą kwestią — wydaje nam się, że początkujący użytkownicy Hibernate nie od razuauważają wskazany problem.

Doskonałym rozwiązańem obu wskazanych problemów jest wzorzec sesji lokalnowątkowej.

Sesja lokalnowątkowej

Sesja lokalnowątkowa to pojedynczy egzemplarz sesji powiązany z konkretnym żądaniem. Implementuje kontekst trwałości w podobny sposób jak JTA implementuje kontekst transakcji. Dowolne komponenty wywołane w tym samym żądaniu współdzielą tę samą sesję i kontekst trwałości.

Szczegółne użyteczne okazuje się dołączenie JSP do kontekstu trwałości. JSP pobiera dane z modelu dziedzinowego, nawigując po grafie obiektów zaczynającym się od obiektu trwałego z zasięgu sesji lub żądania — przykładem niech będzie chociażby obiekt nowej oferty umieszczony w zasięgu żądania. Niestety, graf obiektów może zawierać niezainicjalizowane asocjacje (pośredników i kolekcje), które należy odwiedzić (i zainicjalizować) w momencie tworzenia docelowego widoku.

W przedstawianym przykładzie strona JSP być może wyświetli wszystkie przedmioty, które oferujący nową kwotę ma na sprzedaż, wywołując metodę newBid.getBidder().getItems().iterator() (nie jest to może najważniejsza informacja dla składającego ofertę, ale nic nie stoi na przeszkodzie, by nie mogła się pojawić). Ponieważ asocjacja items jest leniwa, z pewnością nie będzie jeszcze zainicjalizowana.

Ponieważ pod koniec metody execute() akcji zamknęliśmy sesję Hibernate, system zgłosi wyjątek LazyInitializationException w momencie dostępu do asocjacji — połączenie bazodanowe nie jest dostępne i graf obiektów został odłączony, więc Hibernate nie potrafi pobrać kolekcji. Można tak zapisać akcję, by zapewnić pełną inicjalizację wszystkich leniwych asocjacji przed przejściem do tworzenia widoku (zajmiemy się tym tematem nieco później). Lepiej jednak byłoby pozostawić sesję otwartą aż do czasu wykonania widoku.

Odpowiedzi na pytania	Dlaczego Hibernate nie może tworzyć nowego połączenia (lub sesji), gdy musi leniwie zainicjalizować asocjacje? Po pierwsze, sądzimy, że znacznie lepiej od razu zainicjalizować wszystkie wymagane obiekty dla konkretnego przypadku użycia, stosując sprowadzanie wyprzedzające (wtedy najczęściej unika się problemu $n+1$ pobrań). Co więcej, otwieranie nowych połączeń bazodanowych (a tym samym tymczasowych transakcji bazodanowych!) w sposób niejawny i przezroczysty dla programisty naraża aplikację na problemy z izolacją transakcji. Kiedy należy zamknąć sesję i zakończyć tymczasową transakcję? Po zafiklowaniu każdej leniwej asocjacji? Stojmy na stanowisku, że ramy obowiązywania transakcji
------------------------------	--

powinny być jasno określone przez programistę aplikacji. Aby włączyć leniwe sprowadzanie dla odłączonego obiektu, można użyć metody lock() nowej sesji.

Wzorzec sesji lokalnowątkowej zapewnia uzyskanie pojedynczego obiektu Session na żądanie, czyli jednego egzemplarza sesji dla widoku i wielu potencjalnych metod execute(). Java zawiera klasę ThreadLocal do implementacji zmiennych o zasięgu wątku. Wzorzec sesji lokalnowątkowej łączy obiekt ThreadLocal z elementem przechwytyującym (interceptorem) lub filtrem servletu, który zamknie sesję na końcu żądania po przygotowaniu odpowiedzi, ale jeszcze przed jej wysłaniem do klienta.

Najpierw rozbudujmy klasę pomocniczą HibernateUtil. Zamiast otwierać nową sesję w momencie wywołania getSession(), zwracamy obiekt Session przechowywany w zmiennej LocalThread. W ten sposób zawsze uzyskujemy sesję związaną z aktualnym wątkiem. Klasa HibernateUtil to doskonałe miejsce do implementacji ogólnego kodu wyjątków, więc dodamy jeszcze kilka statycznych metod pomocniczych otaczających wyjątki. Pełny kod poprawionej klasy HibernateUtil zawiera listing 8.3.

Listing 8.3. Ulepszona wersja klasy HibernateUtil stosująca zmienne lokalnowątkowe

```
public class HibernateUtil {

    private static SessionFactory sessionFactory;
    private static final ThreadLocal threadSession = ❶
        new ThreadLocal();
    private static final ThreadLocal threadTransaction = ❷
        new ThreadLocal();

    static {
        // Inicjalizacja obiektu SessionFactory.
    }

    public static Session getSession() { ❸
        Session s = (Session) threadSession.get();
        // Otwórz nową sesję, jeśli wątek jeszcze żadnej nie posiada.
        try {
            if (s == null) {
                s = sessionFactory.openSession();
                threadSession.set(s);
            }
        } catch (HibernateException ex) {
            throw new InfrastructureException(ex);
        }
        return s;
    }

    public static void closeSession() { ❹
        try {
            Session s = (Session) threadSession.get();
            threadSession.set(null);
            if (s != null && s.isOpen())

```

```
s.close();
} catch (HibernateException ex) {
    throw new InfrastructureException(ex);
}
}

public static void beginTransaction() { ❶
    Transaction tx = (Transaction) threadTransaction.get();
    try {
        if (tx == null) {
            tx = getSession().beginTransaction();
            threadTransaction.set(tx);
        }
    } catch (HibernateException ex) {
        throw new InfrastructureException(ex);
    }
}

public static void commitTransaction() { ❷
    Transaction tx = (Transaction) threadTransaction.get();
    try {
        if (tx != null && !tx.wasCommitted()
            && !tx.wasRolledBack())
            tx.commit();
        threadTransaction.set(null);
    } catch (HibernateException ex) {
        rollbackTransaction();
        throw new InfrastructureException(ex);
    }
}

public static void rollbackTransaction() { ❸
    Transaction tx = (Transaction) threadTransaction.get();
    try {
        threadTransaction.set(null);
        if (tx != null && !tx.wasCommitted() && !tx.wasRolledBack())
            tx.rollback();
    } catch (HibernateException ex) {
        throw new InfrastructureException(ex);
    } finally {
        closeSession();
    }
}
```

-
- ❶ Obiekt Session aktualnego wątku znajduje się w zmiennej ThreadLocal.
 - ❷ Używamy jednej transakcji bazodanowej dla wszystkich operacji, więc stosujemy dodatkową zmienną ThreadLocal dla obiektu Transaction. W ten sposób zarówno sesja, jak i transakcja zostają powiązane z aktualnym wątkiem. Działania kilku akcji wewnątrz wątku uczestniczą w ten samej transakcji bazodanowej.
 - ❸ Metoda getSession() została rozbudowana o zmienną lokalnowątkową. Dodatkowo otaczamy weryfikowalny wyjątek HibernateException

wyjątkiem nieweryfikowalnym InfrastructureException (część aplikacji CaveatEmptor).

- ❶ Wyjątki zgłasiane przez Session.close() również otoczyliśmy statyczną metodą pomocniczą.
- ❷ Kod powodujący uruchomienie nowej transakcji przypomina kod z metody getSession().
- ❸ Jeśli zatwierdzenie transakcji bazodanowej zawiedzie, natychmiast ją wycofujemy. Nic nie robimy, jeśli transakcja została już zatwierdzona lub wycofana.
- ❹ Po wycofaniu transakcji zamykamy sesję.

Przedstawiona klasa narzędziowa jest znacznie bardziej użyteczna niż pierwsza. Zapewnia sesje i transakcje bazodanowe wspólne dla całego lokalnego wątku. Dodatkowo zamienia wszystkie wyjątki na wyjątki czasu wykonania zdefiniowane przez tworzoną aplikację (lub jej szkielet). Znacząco upraszcza to obsługę wyjątków w kodzie aplikacji. Wzorzec sesji lokalnowątkowej daje elastyczność współdzielenia jednej sesji przez wszystkie akcje i stronę JSP konkretnego wątku. To samo dotyczy transakcji bazodanowej. Cały wątek wykorzystuje jedną transakcję bazodanową — metody beginTransaction() i commitTransaction() wywołuje tylko wtedy, gdy jest to wymagane ze względu na logikę biznesową.

Wywołanie metody getSession() po raz pierwszy dla danego wątku tworzy nowy obiekt Session. Zajmijmy się teraz drugą częścią wzorca sesji lokalnowątkowych, czyli zamknięciu sesji po wykonaniu widoku zamiast na końcu metody execute().

Ten drugi element zaimplementujemy, używając filtra serwletu. Możliwe są i inne rozwiązania, na przykład szkielet aplikacji WebWork2 oferuje dołączane klasy przechwytyjącej. Filtr serwletu ma za zadanie zamknąć obiekt Session przed wysłaniem danych do klienta (ale po wykonaniu wszystkich akcji i zrenderowaniu widoku). Odpowiada również za zatwierdzenie aktywnych transakcji bazodanowych. Listing 8.4 przedstawia metodę doFilter() serwletu.

Listing 8.4. Metoda doFilter() zamyka sesję Hibernate

```
public void doFilter(ServletRequest request, ServletResponse response,
                      FilterChain chain) throws IOException, ServletException
{
    try {
        chain.doFilter(request, response);
        HibernateUtil.commitTransaction();
    } finally {
        HibernateUtil.closeSession();
    }
}
```

Nie uruchamiamy transakcji bazodanowej ani nie otwieramy sesji, dopóki nie zażąda jej aplikacja. Kolejne akcje i widok wykorzystują sesję i transakcję otwartą za pierwszym razem. Po wykonaniu wszystkich akcji (serwletów) i zrenderowaniu widoku zatwierdzamy oczekujące transakcje bazodanowe. Na końcu, niezależnie od tego, co się stało, zamykamy sesję, by zwolnić zasoby.

Powróćmy do metody execute() i uprościmy ją.

```
public void execute() {  
    // Pobierz wartość z żądania.  
  
    try {  
        HibernateUtil.beginTransaction();  
        Session session = HibernateUtil.getSession();  
  
        // Wczytaj obiekt Item, którego dotyczy oferta.  
        // Sprawdź, czy aukcja nadal trwa.  
        // Sprawdź kwotę najwyższej oferty.  
        // Dodaj nowy obiekt Bid do obiektu Item.  
        // Umieść nowy obiekt w zasięgu kolejnej strony.  
        // Przejdz do strony poprawnego złożenia oferty.  
  
    } catch (HibernateException ex) {  
        throw new InfrastructureException(ex);  
    } catch (Exception ex) {  
        // Zgłoś wyjątek specyficzny dla aplikacji.  
    }  
}
```

Zredukowaliśmy kod obsługi błędów do jednego bloku try-catch. Możemy bezpiecznie ponownie zgłaszać wyjątki weryfikowalne typu `HibernateException` jako nieweryfikowalne wyjątki czasu wykonania. Do rzeczywistej obsługi błędów wykorzystujemy własną hierarchię wyjątków (lub hierarchię szkieletu aplikacji).

Niestety, wzorzec sesji lokalnowątkowej nie jest idealny. Zmiany dokonane w obiektach sesji zostają przesłane do bazy danych w nieznanych nam momencach. Mamy pewność, że znajdą się w bazie danych tylko wtedy, gdy transakcja zostanie zatwierdzona. Z drugiej strony zatwierdzenie transakcji następuje dopiero po zrenderowaniu widoku. Problemem staje się rozmiar bufora serwera serwletów — jeśli zawartość widoku przekroczy rozmiar bufora, zapewne część jego zawartości zostanie wysłana do klienta z wyprzedzeniem. Co więcej, taka sytuacja może wystąpić wielokrotnie w trakcie renderowania widoku (pierwsze wysłanie powoduje również przesłanie nagłówków HTTP). Jeżeli instrukcje SQL wykonywane przy zatwierdzaniu transakcji spowodują zgłoszenie próby złamania ograniczeń bazodanowych, użytkownik chwilę wcześniej mógł otrzymać stronę o poprawnym złożeniu oferty! Nie możemy zmienić kodu statusu (na przykład na 500 Internal Server Error), ponieważ klient otrzymał wcześniej inny kod (na przykład 200 OK).

Istnieje kilka rozwiązań zapobiegających powstaniu tego rzadkiego wyjątku: można zmienić rozmiar bufora serwera serwletów lub zapisać sesję `Hibernate` przed przejściem do renderowania widoku (dodając metodę `flushSession()` do klasy `HibernateUtil`). Niektóre szkielety aplikacji internetowych nie od razu

wypełniają bufor odpowiedzi renderowaną zawartością — stosują własny obiekt OutputStream i wysyłają odpowiedź na wyjście tylko po jej utworzeniu w całości. Przedstawiony problem dotyczy więc w zasadzie tylko sytuacji, w których sami programujemy całe serwlety.

Aukcja stała się znacznie czytelniejsza. Niestety, nadal zawiera w sobie trzy różne zadania: przepływ stron, dostęp do trwałego magazynu danych i logikę biznesową. Klauzula catch z HibernateException również wydaje się być nie na miejscu. Zajmijmy się ostatnim z wymienionych zadań, bo wydaje się najważniejsze.

Tworzenie „inteligentnych” modeli dziedzinowych

Podstawowy pomysł wzorca MVC polega na przejrzystej separacji logiki sterującej (w przedstawianej aplikacji byłaby to logika przechodzenia do kolejnych stron), definicji widoku i logiki biznesowej. Aktualnie aukcja zawiera trochę logiki biznesowej (czyli kodu przydatnego nawet wtedy, gdyby aplikacja miała uzyskać całkowicie inny interfejs użytkownika) i trochę modelu dziedzinowego („głupich” obiektów przechowujących dane). Klasy trwałe definiują stan, ale nie zachowanie.

Przenieśmy logikę biznesową do modelu dziedzinowego. Zwiększy to długość kodu, ale w dłuższej perspektywie zapewni możliwość jego wielokrotnego wykorzystania. Jest też bardziej obiektywne, co otwiera nowe perspektywy rozszerzania logiki biznesowej (na przykład wprowadzenie wzorca strategii dla różnych strategii zgłaszania ofert). Dodajmy nową metodę placeBid() do klasy Item.

```
public Bid placeBid(User bidder, BigDecimal bidAmount)
    throws BusinessException {

    // Czy akcja nadal trwa?
    if (this.getEndDate().before(new Date()) )
        throw new BusinessException("Aukcja już się skończyła.");
    }

    // Utwórz nowy obiekt Bid.
    Bid newBid = new Bid(bidAmount, this, bidder);

    // Dodaj nową ofertę do obiektu Item.
    this.addBid(newBid);
    return newBid;
}
```

Kod wymusza stosowanie zasad biznesowych ograniczających dowolność danych w obiektach biznesowych, ale nie wykonuje żadnego kodu dostępu do danych. Celem jest zawarcie w klasach modelu dziedzinowego logiki biznesowej bez odwoływanego się kodu zapewniającego trwałość danych. Zauważ, że przedstawiona metoda nie zawiera kodu sprawdzającego najwyższą z aktualnych ofert. Pamiętaj: klasa nie wie nic o trwałości, gdyż możemy potrzebować jej obiektów także poza kontekstem trwałości (na przykład na warstwie prezentacji). Możemy nawet zaimplementować sprawdzanie najwyższej oferty w metodzie placeBid(), przechodząc przez kolekcję bids klasy Item i odnajdując największą kwotę. Ponieważ takie podejście nie byłoby równie wydajne jak zapytanie HQL, sprawdzenie zaimplementujemy w innym miejscu. Kod akcji uprościł się do postaci:

```
public void execute() {  
    // Pobierz wartość z żądania.  
  
    try {  
        HibernateUtil.beginTransaction();  
        Session session = HibernateUtil.getSession();  
  
        // Wczytaj obiekt Item, którego dotyczy oferta.  
        Item item = (Item) session.load(Item.class, itemId);  
  
        // Sprawdź kwotę najwyższej oferty.  
        Query q = session.createQuery("select max(b.amount)" +  
            " from Bid b where b.item = :item");  
        q.setEntity("item", item);  
        BigDecimal maxBidAmount = (BigDecimal) q.uniqueResult();  
        if (maxBidAmount.compareTo(bidAmount) > 0) {  
            throw new BusinessException("Podana kwota jest za niska.");  
        }  
        // Dodaj nowy obiekt Bid do obiektu Item.  
        User bidder = (User) session.load(User.class, userId);  
        Bid newBid = item.placeBid(bidder, bidAmount);  
  
        // Umieść nowy obiekt w zasięgu kolejnej strony.  
        // Przejdź do strony poprawnego złożenia oferty.  
  
    } catch (HibernateException ex) {  
        throw new InfrastructureException(ex);  
    } catch (BusinessException ex) {  
        // Wykonaj kod specyficzny dla wyjątku.  
    } catch (Exception ex) {  
        // Zgłoś wyjątek specyficzny dla aplikacji.  
    }  
}
```

Logika biznesowa umieszczania oferty jest niemal w całości zawarta w metodzie placeBid() i logice sterującej akcją. Możemy nawet określić inny sposób przechodzenia do kolejnych stron, wyłapując i przekazując dalej odpowiednie wyjątki. Wzorzec MVC niewiele mówi na temat umieszczenia w tym wszystkim trwałości. Jesteśmy pewni, że kod Hibernate nie powinien znaleźć się w akcji, ale na osobnej warstwie trwałości. Umieścimy kod w klasie DAO i utworzymy fasadę dla operacji trwałości.

Obiekty dostępu do danych

Mieszanie kodu dostępu do danych i logiki sterującej łamie zasadę rozdzielania zadań. W zasadzie wszystkie aplikacje (poza tymi najprostszymi) powinny ukrywać wywołania interfejsu Hibernate przed logiką biznesową za fasadą. Mniejsze aplikacje mogą używać jednego obiektu PersistenceManager, inne mogą preferować projektowanie nakierowane na polecenia. My preferujemy wzorzec DAO.

Wzorzec projektowy DAO pojawił się w dokumentach Java BluePrints firmy Sun. Został nawet wykorzystany w niesławnej aplikacji demonstracyjnej Java Petstore. DAO definiuje interfejs dla trwałych operacji (CRUD i wyszukiwania)

powiązany z konkretną trwałą encją. Zaleca grupowanie kodu dotyczącego trwałości tej encji.

Utwórzmy klasę ItemDAO, która zatrzymie w sobie cały kod odpowiadający za trwałość obiektów Item. Na razie będzie zawierała jedynie metody getItemById() oraz getMaximumBidAmount(). Pełną implementację klasy DAO przedstawia listing 8.5.

Listing 8.5. Prosta klasa DAO wykonująca operacje na trwałym magazynie dla obiektów Item

```
public class ItemDAO {

    public ItemDAO() {
        HibernateUtil.beginTransaction();
    }

    public Item getItemById(Long itemId) {
        Session session = HibernateUtil.getSession();
        Item item = null;
        try {
            item = (Item) session.load(Item.class, itemId);
        } catch (HibernateException ex) {
            throw new InfrastructureException(ex);
        }
        return item;
    }

    public BigDecimal getMaxBidAmount(Long itemId) {
        Session session = HibernateUtil.getSession();
        BigDecimal maxBidAmount = null;
        try {
            String query = "select max(b.amount) from Bid b where b.item.id = :itemId";
            Query q = session.createQuery(query);
            q.setLong("itemId", itemId.longValue());
            maxBidAmount = (BigDecimal) q.uniqueResult();
        } catch (HibernateException ex) {
            throw new InfrastructureException(ex);
        }
        return maxBidAmount;
    }
}
```

W momencie utworzenia obiektu ItemDAO albo tworzymy nową transakcję bazowaną, albo dodajemy do już istniejącej w aktualnym wątku. To, czy metoda getMaximumBidAmount() znajdzie się w klasie ItemDAO czy BidDAO, zależy od sposobu postrzegania aukcji przez programistę. Ponieważ stosujemy identyfikator obiektu Item, wydaje się, że lepiej umieścić ją w ItemDAO. Ponieważ obiekt DAO zajmuje się obsługą wyjątków HibernateException i ich zamianą na wyjątki InfrastructureException, wreszcie udało się przenieść cały kod obsługi wyjątków Hibernate poza akcję.

Potrzebujemy jeszcze klasy UserDAO, która na razie zawiera tylko jedną metodę: getUserById().

```
public class UserDAO {  
  
    public UserDAO() {  
        HibernateUtil.beginTransaction();  
    }  
  
    public User getUserById(Long userId) {  
        Session session = HibernateUtil.getSession();  
        User user = null;  
        try {  
            user = (User) session.load(User.class, userId);  
        } catch (HibernateException ex) {  
            throw new InfrastructureException(ex);  
        }  
        return user;  
    }  
}
```

W tym momencie warto dostrzec kolejną zaletę wzorca sesji lokalnowątkowej. Wszystkie obiekty DAO współdzielą tę samą sesję Hibernate (i transakcję bazodanową), choć nie muszą przekazywać jej jawnie jako argument tworzonego obiektu DAO. To ważna zaleta, która staje się coraz istotniejsza wraz z rozbudową aplikacji i powstawaniem złożonych warstw.

Po dodaniu nowych klas DAO możemy jeszcze bardziej uprościć kod akcji. Oto jego kolejna wersja:

```
public void execute() {  
    // Pobierz wartość z żądania.  
  
    try {  
        ItemDAO itemDAO = new ItemDAO();  
        UserDAO userDAO = new UserDAO();  
  
        if (itemDAO.getMaxBidAmount(itemId).compareTo(bidAmount) > 0)  
            throw new BusinessException("Podana kwota jest za niska.");  
  
        Item item = itemDAO.getItemById(itemId);  
        Bid newBid = item.placeBid(userDAO.getUserById(userId), .bidAmount);  
  
        // Umieść nowy obiekt w zasięgu kolejnej strony.  
        // Przejdź do strony poprawnego złożenia oferty.  
  
    } catch (BusinessException ex) {  
        // Przejdź do strony błędu.  
    } catch (Exception ex) {  
        // Zgłoś wyjątek specyficzny dla aplikacji.  
    }  
}
```

Zauważ jak samodokumentujący stał się kod w porównaniu z pierwszą wersją. Osoba w ogóle nieznająca Hibernate zrozumie go od razu po przeczytaniu nazw metod nawet bez dodatkowych komentarzy.

Jesteśmy niemal zadowoleni z przedstawionej implementacji przypadku użycia. Wszystkie metody są krótkie, czytelne i możliwe do wielokrotnego wykorzystania. Nieprzyjemny kod dotyczący wyjątków i transakcji znalazł się poza główną infrastrukturą. Pozostały pewne drobne sprawy: jeden fragment logiki biznesowej nadal występuje w implementacji akcji — sprawdzenie maksymalnej kwoty oferty. Kod zgłaszający wyjątek `BusinessException` powinien trafić do modelu dziedzinowego.

Pojawia się istotne pytanie: jeśli przeniesiemy omawiany fragment kodu do metody `placeBid()` klasy `Item`, implementacja modelu dziedzinowego będzie zależna od interfejsu trwałości (klasy DAO). Warto tego uniknąć, bo prowadzi to do trudności w wykonaniu testów jednostkowych obiektów dziedzinowych i logiki biznesowej (trwałość „przecieka” do modelu dziedzinowego). Czy naprawdę nie mamy wyboru i musimy pozostawić fragment logiki biznesowej w logice sterującej?

Rozwiązaniem okazuje się refaktoryzacja metody `placeBid()`, dodanie dwóch nowych metod klasy `ItemDAO` i zmodyfikowanie kodu sterującego w sposób następujący:

```
BigDecimal currentMaxAmount = itemDAO.getMaxBidAmount(itemId);
BigDecimal currentMinAmount = itemDAO.getMinBidAmount(itemId);
Item item = itemDAO.getItemById(itemId);
User user = userDao.getUserById(userId);
newBid = item.placeBid(user, newAmount, currentMaxAmount, currentMinAmount);
```

Zmianie uległo kilka elementów. Logika biznesowa i wyjątki trafiły do metody `placeBid()`. Metodę wywołujemy z nowymi argumentami: aktualnym maksimum i minimalną kwotą oferty. Wartości pobieramy nowymi metodami klasy `ItemDAO`. W akcji serwletu pozostały w zasadzie tylko wywołania dotyczące trwałej warstwy i wywołania rozpoczęjące logikę biznesową. Logika biznesowa znalazła się w modelu dziedzinowym i można ją dowolnie wykorzystać — w żaden sposób nie zależy od warstwy trwałości interfejsu DAO. Podobne problemy zapewne pojawią się w tworzonych aplikacjach, więc warto przygotować się na modyfikacje kodu zwiększające separację warstw.

Powróćmy do opisu wzorca DAO. W zasadzie trudno tu nawet mówić o wzorcu — istnieje wiele sposobów implementacji podstawowego pomysłu. Niektórzy programiści łączą szkielet DAO z abstrakcyjnym wzorcem fabryki, co umożliwia przełączanie się między różnymi mechanizmami trwałości w trakcie pracy aplikacji. Najczęściej potrzebę tę motywuje się chęcią uniezależnienia się od konkretnego dostawcy systemu bazy danych SQL. Ponieważ Hibernate dosyć dobrze (choć nie idealnie) uniezależnia kod Javy od dialekta SQL konkretnego twórcy bazy danych, lepiej nie utrudniać sobie życia dodatkowymi abstrakcjami.

W następnym podrozdziale przyjrzymy się, w jaki sposób przekształcić przedstawiony kod, by działał w kontenerze EJB. Chcemy w jak najmniejszym stopniu zmienić obecną implementację. Przez wszystkie poprzednie rozdziały wskazywaliśmy zalety POJO i trwałości przezroczystej jako doskonałe metody zwiększającej przenośność kodu między różnymi platformami — wyglądałybyśmy wyjątkowo głupio, gdyby teraz się okazało, że dla kontenera EJB musimy wszystko zmienić.

8.1.2. Stosowanie Hibernate w kontenerze EJB

Z naszego punktu widzenia najważniejsza różnica między aplikacją bazującą na serwletach a aplikacją, gdzie logikę biznesową i dostęp do danych wykonuje kontener EJB, polega na fizycznej separacji warstw. Jeśli kontener EJB działa jako osobny proces niezależny od serwera serwletów, należy dążyć do minimalizacji żądań przesyłanych przez warstwę serwletu do warstwy EJB. Każde żądanie międzyprocesowe wprowadza opóźnienie, zwiększąc tym samym czas odpowiedzi aplikacji i redukując współbieżność (bo potrzeba albo dłuższych transakcji, albo większej ich liczby).

Z tego względu wszystkie dostępy do danych powiązane z pojedynczym żądaniem użytkownika powinny mieć miejsce w jednym żądaniu przesyłanym do warstwy EJB. Nie możemy więc zastosować wcześniejszego rozwiązania z leniwym pobieraniem danych przez warstwę widoku z obiektów dziedzinowych. Warstwa biznesowa (EJB) musi zatroszczyć się o sprowadzenie wszystkich danych potrzebnych do poprawnego zrenderowania widoku.

W rzeczywistych systemach stosujących ziarenka encyjne można zauważyci tego rodzaju podejście. Wzorzec fasady sesyjnej pozwala tym systemom grupować całą aktywność związaną z żądaniem konkretnego użytkownika w jedno żądanie kierowane do warstwy EJB. Kwestionowany wzorzec **DTO (Data Transfer Object)** umożliwia spakowanie w jedną całość wszystkich danych potrzebnych widokowi. DTO to klasa przechowująca stan konkretnej encji — traktuj ją jako JavaBean lub POJO bez metod biznesowych. Środowisko ziarenek encyjnych potrzebuje DTO, bo ziarenka encyjne nie dają się serializować i tym samym przekazywać między warstwami. W naszym przypadku klasy POJO można serializować, więc naturalnie odrzucamy wzorzec DTO.

Przemyślenia na temat DTO

Pomysł, by w aplikacji bazującej na EJB warstwa internetowa nie mogła bezpośrednio komunikować się z modelem dziedzinowym, głęboko zakorzenił się w praktykach J2EE i myśleniu programistów. Wątpimy, iż pomysł ten nagle zniknie, szczególnie że istnieją pewne przesłanki przemawiające na jego korzyść. Nie należy jednak mylić tych argumentów z rzeczywistym powodem, dla którego DTO stało się tak powszechnie akceptowanym rozwiązaniem.

Wzorzec DTO pojawił się, gdy społeczność użytkowników J2EE zauważała, że mechanizm szczegółowego dostępu zdalnego do ziarenek encyjnych działa powoli i nie zapewnia odpowiedniej skalowalności. Co gorsza, samych ziarenek encyjnych nie można serializować, więc do przenoszenia stanów obiektów biznesowych między warstwami potrzeba innych klas.

Obecnie istnieją dwa bliźniacze wyjaśnienia dotyczące stosowania DTO: DTO implementuje **uzewnętrznianie** danych między warstwami i DTO wymusza **separację** warstwy internetowej od warstwy logiki biznesowej. Do nas przemawia tylko drugie wyjaśnienie, choć zalety separacji stają pod znakiem zapytania, jeśli uwzględnimy się ich koszt. Nie zalecamy całkowitego unikania DTO (w innych zagadnieniach jesteśmy bardziej restrykcyjni). Zamiast tego wymienimy argumenty

przemawiające za i przeciw stosowaniu wzorca DTO w aplikacjach używających Hibernate. Zalecimy uważne ich rozważenie przy podejmowaniu decyzji o wprowadzeniu DTO we własnej aplikacji.

Prawdą jest, że DTO usuwa bezpośrednią zależność widoku od modelu dziedzinowego. Jeśli projekt uwzględnia w sposób niezależny zadania programisty i projektanta witryny, może to mieć pewne zalety. W szczególności, DTO daje szansę na spłaszczenie modelu dziedzinowego przez usunięcie asocjacji i przekształcenie danych do formatu najbardziej przydatnego dla widoku. Z drugiej strony, z doświadczenia wiemy, że wszystkie warstwy aplikacji są ściśle związane z modelem dziedzinowym (i to niezależnie od tego, czy używa się czy też nie klas DTO). Nie traktujemy tego jako czegoś złego, choć w miarę możliwości warto zmniejszyć zależności.

Pierwszą oznaką, że z DTO jest coś nie tak, jest to, że w przeciwnieństwie do ich nazwy, nie są tak naprawdę obiekty. DTO definiuje stan bez żadnego zachowania. Element o takiej charakterystyce automatycznie staje się podejrzany w kontekście podejścia obiektowego. Co gorsza, stan definiowany przez DTO często odpowiada stanowi określanemu przez obiekty biznesowe modelu dziedzinowego — ta tak głośno wskazywana separacja uzyskiwania dzięki wzorcowi DTO nagle okazuje się zwykłą **duplikacją**.

Wzorzec DTO powoduje dwa nieprzyjemne efekty wskazane w książce autora Fowlera [1999]: efekt **zmiany na zasadzie rozprysku**, w której to niewielka zmiana wymagań systemowych pociąga za sobą modyfikacje w wielu klasach; efekt **równoległej hierarchii klas**, w której dwie różne hierarchie klas zawierają podobne klasy w stosunku jeden do jednego. W tym przypadku równoległą hierarchię klas łatwo zauważyc — system stosujący wzorzec DTO ma klasy Item i ItemDTO, User i UserDTO itd. Zmiana na zasadzie rozprysku da o sobie znać, gdy dodamy nową właściwość do klasy Item. Musimy zmienić nie tylko widok i klasę Item, ale również klasę ItemDTO i kod składający ItemDTO na podstawie właściwości klasy Item (ten ostatni fragment kodu bywa szczególnie uciążliwy do pielęgnowania).

DTO nie zawsze jest złe. Kod wskazany przed chwilą jako szczególnie trudny do pielęgnowania — tzw. **składacz** — ma pewną wartość nawet w kontekście Hibernate. Składacz DTO to wygodny sposób zapewnienia, by wszystkie potrzebne dane zostały w całości sprowadzone z bazy danych przed przekazaniem obiektów do warstwy internetowej. Jeżeli aplikacja z Hibernate zbyt często zgłasza wyjątki LazyInitializationException na warstwie internetowej, daj szansę wzorcowi DTO, który wymusza dyscyplinę w pobieraniu wszystkich niezbędnych danych przez kopowanie ich z obiektów biznesowych (sami nie musimy dyscyplinować się w ten sposób, ale zależy to od doświadczenia w obsłudze Hibernate).

Niejednokrotnie DTO służy do transferu danych między luźno powiązanymi aplikacjami (nasza dyskusja skupia się na transferze danych między różnymi warstwami tej samej aplikacji). JMS i SOAP wydają się lepiej realizować tego rodzaju przesyły.

W aplikacji CaveatEmptor nie użyjemy DTO. Zamiast niego fasada sesyjna warstwy EJB wróci obiekty biznesowe modelu dziedzinowego do warstwy internetowej.

Wzorzec fasady sesyjnej

Wzorzec fasady sesyjnej stosowany obecnie przez większość aplikacji J2EE jest dobrze znany większości programistów języka Java [Marinescu 2002]. Fasada sesyjna to ziarenko sesyjne EJB, które działa jako zewnętrzny interfejs dla biznesowego komponentu programistycznego. Użycie ziarenek sesyjnych daje szansę skorzystania z zalet deklaratywnych transakcji i bezpieczeństwa EJB, by zapewnić na tyle ogólne usługi, aby uniknąć wielu szczegółowych wywołań międzyprocesowych. Nie będziemy szczegółowo opisywać tego wzorca projektowego, bo jest dobrze znany i mało kontrowersyjny. Zamiast tego przedstawimy przykład modyfikacji wcześniejszej akcji tak, by używała fasady sesyjnej.

Kod z poprzedniego podrozdziału wymaga dwóch znaczących zmian. Po pierwsze, zmienimy klasę `HibernateUtil`, by obiekt `SessionFactory` znajdował się w rejestrze JNDI zamiast w zmiennej statycznej. Zmiana ta nie ma jakichś głębszych podstaw. Chcemy jedynie dostosować się do sposobu obsługi innych obiektów tego typu (na przykład `UserTransaction` z JTA) w środowisku EJB. Zmieńmy statyczny kod inicjalizujący klasy `HibernateUtil` i usuńmy statyczną zmienną `sessionFactory`.

```
static {
    try {
        new Configuration().configure().buildSessionFactory();
        // Obiekt SessionFactory znajduje się w JNDI, patrz plik hibernate.cfg.xml.
    } catch (Throwable ex) {
        ex.printStackTrace(System.out);
        throw new ExceptionInInitializerError(ex);
    }
}

public static SessionFactory getSessionFactory() {
    SessionFactory sessions = null;
    try {
        Context ctx = new InitialContext();
        String jndiName = "java:hibernate/HibernateFactory";
        sessions = (SessionFactory)ctx.lookup(jndiName);
    } catch (NamingException ex) {
        throw new InfrastructureException(ex);
    }
    return sessions;
}
```

Teraz za każdym razem, gdy musimy pobrać obiekt `SessionFactory` (na przykład w metodzie `getSession()`), musimy korzystać z metody pomocniczej `getSessionFactory()`. Dodatkowo w sposób opisany w punkcie 2.4.2 trzeba skonfigurować Hibernate, by po wywoaniu metody `buildSessionFactory()` umieszczał obiekt `SessionFactory` w JNDI.

W kolejnym kroku przenosimy część kodu z akcji serwletu do metody bidForItem() nowego ziarenka sesyjnego CaveatEmptorFacade. Ta zmiana uwidacznia ograniczenia specyfikacji EJB. W implementacji serwletowej mogliśmy wykonać całą obsługę transakcji i wyjątków w filtrze serwletów i klasie narzędziowej. Filtr serwletów to implementacja wzorca elementu przechwytyującego zgodna ze specyfikacją serwletów. To zadziwiające, że specyfikacja EJB nie określa standardowego sposobu implementacji przechwytywania dla wywołań metod EJB. Pewne kontenery, na przykład JBoss lub WebLogic, stosują własne wersje interfejsu przechwytyującego. Zalecamy ich stosowanie, jeśli przenośność kodu nie jest najważniejsza. W książce chcemy przedstawić kod działający we wszystkich serwerach aplikacji, musimy więc przenieść cały niewygodny kod obsługi transakcji i wyjątków do metody bidForItem(). (Co ciekawe, w kolejnym podrozdziale użyjemy wzorca polecenia z EJB i ponownie przeniesiemy cały kod w inne miejsce!)

Zdalny interfejs fasady sesyjnej nie jest szczególnie trudny:

```
public interface CaveatEmptorFacade extends javax.ejb.EJBObject {
    public Bid bidForItem(Long userId, Long itemId, BigDecimal bidAmount)
        throws RemoteException;
}
```

Klasa implementująca ziarenko ma postać:

```
public class CaveatEmptorFacadeBean implements javax.ejb.SessionBean {
    public void setSessionContext(SessionContext sessionContext)
        throws EJBException, RemoteException {}
    public void ejbRemove() throws EJBException, RemoteException {}
    public void ejbActivate() throws EJBException, RemoteException {}
    public void ejbPassivate() throws EJBException, RemoteException {}

    public Bid bidForItem(Long userId, Long itemId, BigDecimal bidAmount)
        throws RemoteException {

        Bid newBid = null;
        try {
            ItemDAO itemDAO = new ItemDAO();
            UserDAO userDAO = new UserDAO();

            BigDecimal currentMaxAmount = itemDAO.getMaxBidAmount(itemId);
            BigDecimal currentMinAmount = itemDAO.getMinBidAmount(itemId);
            Item item = itemDAO.getItemById(itemId);
            User user = userDAO.getUserById(userId);

            newBid = item.placeBid(user, newAmount, currentMaxAmount,
                currentMinAmount);

            HibernateUtil.commitTransaction();
        } finally {
            HibernateUtil.closeSession();
        }
    }

    return newBid;
}
```

Pamiętaj, że wywołanie `HibernateUtil.commitTransaction()` niekoniecznie musi zatwierdzić transakcję bazodanową — Hibernate w niewidoczny sposób obsługuje przypadek, w którym działa w kontenerze EJB z JTA, więc tak naprawdę transakcja bazodanowa zostanie zatwierdzona dopiero wtedy, gdy zatwierdzi ją kontener. Wywołanie powoduje jednak przesłanie wszystkich zmian w sesji do bazy danych.

Błąd jednej z reguł biznesowych powoduje zgłoszenie wyjątku `BusinessException` do klienta ziarenka sesyjnego. Błąd części dotyczącej infrastruktury spowoduje zgłoszenie wyjątku `InfrastructureException`. Oba rodzaje wyjątków zostaną otoczone wyjątkiem `EJBException`, który to przed wysłaniem do klienta zostanie otoczony raz jeszcze, tym razem wyjątkiem `RemoteException` (oba otoczenia obsługują kontener EJB). Zadaniem akcji (na warstwie internetowej) jest odpowiednia interpretacja otoczonych wyjątków i wyświetlenie użytkownikowi odpowiedniego komunikatu. Nowy kod akcji ma postać:

```
public void execute() {  
  
    // Pobierz wartości z żądania.  
    try {  
        Context ctx = new InitialContext();  
        String jndiName = "java:comp/ejb/CaveatEmptorFacade";  
        CaveatEmptorFacade ejbFacade = (CaveatEmptorFacade)  
            ctx.lookup(jndiName);  
  
        Bid newBid = ejbFacade.bidForItem(iserId, itemId, bidAmount);  
  
        // Umieść nowy obiekt Bid w zasięgu następnej strony.  
  
        // Przejdź do strony przyjęcia oferty.  
    } catch (RemoteException ex) {  
  
        // Pobierz wyjątek EJBException zawierający wyjątek czasu wykonania  
        // dotyczący infrastruktury lub reguł biznesowych.  
    }  
}
```

Po przedstawieniu przykładu porzucamy wzorzec fasady sesyjnej, by przedstawić rozwiązanie bazujące na wzorcu polecenia. Podejście to okazuje się elastyczne i w pewnych sytuacjach lepsze od fasady sesyjnej.

Wzorzec polecenia EJB

Wzorzec polecenia EJB zastępuje metody fasady sesyjnej — na przykład `bidForItem()` — klasami poleceń, czyli w przedstawianym przykładzie klasą `BidForItemCommand`. Metodę `execute()` polecenia wywołuje bezstanowe ziarenko sesyjne nazywane **uchwytom polecenia**. Uchwyt polecenia pozwala skorzystać z zalet transakcji i bezpieczeństwa kontenera. Dodatkowo implementuje ogólny sposób obsługi wyjątków (może nawet zapewniać cały szkielet przechytywania). Samo polecenie zawiera jednostkę logiki aplikacji, parametry wejściowe i wyjściowe. Egzemplarz powstaje w akcji klienta, następnie trafia do uchwytu polecenia, zostaje

wykonany w kontekście warstwy EJB i na końcu wraca do klienta z wynikiem operacji.

Wzorzec polecenia redukuje ilość kodu przez ogólny sposób obsługi niektórych zadań oraz przez ograniczenie niektórych fragmentów wymaganych przez specyfikację EJB. Polecenia to proste klasy POJO. Łatwo je napisać i wielokrotnie wykorzystywać. Co ważne, nic nie stoi na przeszkodzie, by działały poza kontenerem EJB (podobnie jak klasy POJO modelu dziedzinowego). Jedyne wymaganie polega na implementacji następującego interfejsu:

```
public interface Command extends Serializable {
    public void execute() throws CommandException;
}
```

Zauważ, że polecenie musi obsługiwać serializację, by mogło zostać przekazane między warstwami. Interfejs definiuje umowę między poleceniem a jego uchwytem. Zdalny interfejs uchwytu polecenia również jest prosty:

```
public interface CommandHandler extends javax.ejb.EJBObject {
    public Command executeCommand(Command command)
        throws RemoteException, CommandException;
}
```

Ogólna implementacja uchwytu polecenia to tak naprawdę bezstanowe ziarenko sesyjne EJB:

```
public class CommandHandlerBean implements SessionBean, CommandHandler{

    public void setSessionContext(SessionContext sessionContext)
        throws EJBException, RemoteException {}
    public void ejbRemove()
        throws EJBException, RemoteException {}
    public void ejbActivate()
        throws EJBException, RemoteException {}
    public void ejbPassivate()
        throws EJBException, RemoteException {}

    public Command executeCommand(Command command)
        throws RemoteException, CommandException {

        try {
            command.execute();
        } catch (CommandException ex) {
            HibernateUtil.rollbackTransaction();
            throw ex;
        }
        return command;
    }
}
```

Przedstawiony kod jest wyjątkowo ogólny (nie musimy nawet implementować metod ziarenka sesyjnego). Przechwytuje dowolny wyjątek zgłoszony przez polecenie i ustawia aktualną transakcję kontenera na stan wycofania (pamiętaj, że Hibernate przezroczyci obsługującą JTA). Jeżeli wyjątek nie wystąpi, metoda execute() zwraca polecenie wraz z ewentualnymi parametrami wyjściowymi wy-

maganymi do poprawnego renderingu widoku. Możemy nawet wymusić wycofanie transakcji przez kontener, jeśli nie wyłapiemy wyjątku aplikacji (CommandException). Implementacja akcji serwletu klienckiego wywołującego uchwyt polecenia okazuje się bardzo podobna do wersji wcześniejszych:

```
public void execute() {  
  
    // Pobierz wartości z żądania.  
  
    BidForItemCommand bidFormItem = new BidForItemCommand(userId, itemId,  
        bidAmount);  
  
    try {  
        Context ctx = new InitialContext();  
        String jndiName = "java:comp/ejb/CaveatEmptorCommandHandler";  
        CommandHandler handler = (CommandHandler) ctx.lookup(jndiName);  
  
        bidFormItem = (BidForItemCommand) handler.executeCommand(bidFormItem);  
  
        // Umieść nowy obiekt Bid w zasięgu następnej strony.  
        // bidFormItem.getNewBid();  
  
        // Przejdź do strony przyjęcia oferty.  
  
    } catch (CommandException ex) {  
        // Udpakuj i przekaz do strony błędu.  
        ex.getCause();  
    } catch (Exception ex) {  
        // Zgłoś wyjątek specyficzny dla aplikacji.  
    }  
}
```

Najpierw tworzymy nowy obiekt BidForItemCommand i przekazujemy do niego parametry wejściowe uzyskane wcześniej z żądania HTTP. Następnie po wyszukaniu ziarenka sesyjnego będącego uchwytem, wykonujemy polecenie. Nowo utworzony obiekt Bid znajduje się w jednym z parametrów wyjściowych polecenia. Pobieramy go metodą dostępową getNewBid(). Polecenie to zwykły JavaBean z dodatkową metodą execute() wykonywaną przez uchwyt polecenia.

```
public class BidForItemCommand implements Command {  
  
    private Long userId;  
    private Long itemId;  
    private BigDecimal bidAmount;  
  
    private Bid newBid;  
  
    public BidForItemCommand(Long userId, Long itemId, BigDecimal bidAmount)  
    {  
        this.userId = userId;  
        this.itemId = itemId;  
        this.bidAmount = bidAmount;  
    }  
  
    public Bid getNewBid() {
```

```

        return newBid;
    }

    public void execute() throws CommandException {
        try {
            ItemDAO itemDAO = new ItemDAO();
            UserDao userDao = new UserDao();

            BigDecimal currentMaxAmount = itemDAO.getMaxBidAmount(itemId);
            BigDecimal currentMinAmount = itemDAO.getMinBidAmount(itemId);

            Item item = itemDAO.getItemById(itemId);
            User user = userDao.getUserById(userId);

            newBid = item.placeBid(user, newAmount, currentMaxAmount,
                currentMinAmount);

            HibernateUtil.commitTransaction();

        } catch (InfrastructureException ex) {
            // Ponownie zgłoś wyjątek weryfikowalny.
            throw new CommandException(ex);

        } catch (BusinessException ex) {
            // Ponownie zgłoś wyjątek weryfikowalny.
            throw new CommandException(ex);

        } finally {
            HibernateUtil.closeSession();
        }
    }
}

```

Kilka pierwszych wierszy nie jest zbyt interesujących: wykorzystujemy standarde atrybuty i metody dostępowe klas JavaBean, by zadeklarować parametry wejściowe i wyjściowe. Metoda execute() wygląda znajomo, ponieważ zawiera logikę sterującą i obsługę wyjątków z wcześniejszego ziarenka sesyjnego. Metodę execute() można łatwo rozszerzyć, na przykład inicjalizując część grafu obiektów zwracaną następnie przez parametry wyjściowe wykorzystywane do renderingu widoku.

Uwaga

Biblioteki Hibernate u klienta. Ponieważ klasa BidForItemCommand potrzebuje klas DAO, musimy dołączyć wszystkie biblioteki związane z trwałością do ścieżki wyszukiwania serwletu (nawet jeśli polecenie wykonuje tak naprawdę tylko warstwa biznesowa). To istotna wada wzorca polecenia. Rozwiązaniem może być traktowanie polecień tylko jako mechanizmu transportującego i przechowywanie logiki biznesowej w bezstanowym ziarenku sesyjnym. Niestety, zaczyna to przypominać (anty-) wzorzec DTO, więc trzeba samemu zdecydować, które podejście w danej sytuacji okaże się najlepsze.

Ponieważ mamy tylko jedno polecenie, wzorzec polecenia wydaje się wymagać więcej pracy niż wzorzec fasady sesyjnej. Wraz z rozwojem systemu dodawanie nowych poleceń staje się prostsze dzięki umieszczeniu wspólnego kodu obsługi wyjątków w uchwycie polecenia. Polecenia są łatwe do implementacji i wielokrotnego użycia (łatwo dokonać kompozycji lub rozszerzenia polecenia za pomocą delegacji lub dziedziczenia). Wzorzec polecenia ma też i inne przyjemne cechy. Ziarenko sesyjne to nie jedyny sposób tworzenia uchwytu polecenia! Można wykonać uchwyt polecenia bazujący na JMS i w ten sposób zapewnić asynchroniczność wykonywania poleceń. Można na przykład umieścić polecenie w bazie danych i zaharmonogramować jego wykonanie. Polecenia nie muszą działać w środowisku EJB — mogą działać w procesach wsadowych lub testach jednostkowych JUnit. W praktyce rozwiązanie to sprawdza się doskonale.

Na tym zakończymy opis warstw. Istnieje wiele odmian przedstawionych rozwiązań. Nie wspominaliśmy o użyciu Hibernate w lekkich kontenerach, na przykład Spring Framework lub PicoContainer, bo choć kod wygląda inaczej, to podstawowa zasada jego działania nadal jest podobna.

Przypadek użycia „złożenie oferty na przedmiot” był prosty w jednym aspekcie — transakcja aplikacyjna obejmowała tylko jedno żądanie klienta, więc mogła zostać zaimplementowana za pomocą dokładniej jednej transakcji bazodanowej. Rzeczywiste transakcje aplikacyjne mogą dotyczyć wielu żądań klienta. W takiej sytuacji aplikacja (i baza danych) musi przechowywać stan transakcji w oczekiwaniu na odpowiedź użytkownika. W kolejnym podrozdziale przedstawimy sposoby implementacji transakcji aplikacyjnych w architekturach warstwowych, na przykład tych właśnie omówionych.

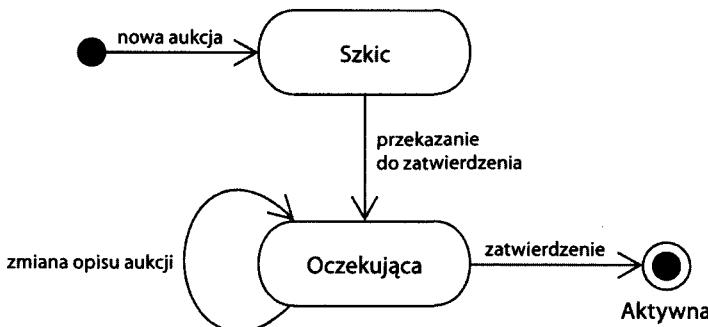
8.2. Implementacja transakcji aplikacyjnych

Ogólne znaczenie transakcji aplikacyjnej wskazaliśmy w podrozdziale 5.2. Wskazaliśmy również, w jaki sposób Hibernate wykrywa konflikty między współbieżnymi transakcjami aplikacyjnymi, stosując wersjonowanie zarządzane. W podrozdziale nie wskazaliśmy jednak, w jaki sposób używać w Hibernate tego rodzaju transakcji, więc tu zajmiemy się tym niezwykle istotnym tematem.

Istnieją trzy sposoby implementacji transakcji aplikacyjnych w systemie wykorzystującym Hibernate: **długa sesja, obiekty odłączane i sposób trudny**. Zaczniemy od tego ostatniego, ponieważ osoby używające ziarenek sesyjnych EJB powinny znać go od podszewki. Najpierw przedstawmy przypadek użycia ilustrujący całe zagadnienie.

Aukcja wykorzystuje cykl zatwierdzenia. Nowy przedmiot aukcji powstaje w stanie *Szkic*. Użytkownik zlecający aukcję może przenieść ją do stania *Oczekująca*, gdy będzie zadowolony z wyglądu i danych szczegółów przedmiotu. Administrator systemu przegląda oczekujące aukcje i gdy nie ma zastrzeżeń, uaktywnia je (przenosi do stanu *Aktywna*). W dowolnym momencie przed uaktywnieniem (zatwierdzeniem) aukcji, zlecający ją użytkownik i administrator może edytować jej szczegóły. Po zatwierdzeniu modyfikacje stają się niemożliwe

do przeprowadzenia. Najważniejsze, by administrator podejmujący decyzję zawsze widział najnowszą wersję opisu przedmiotu przed zatwierdzeniem, bo nie można tej samej aukcji zatwierdzić dwukrotnie. Rysunek 8.1 przedstawia cykl zatwierdzania aukcji.



Rysunek 8.1. Diagram stanów zatwierdzania aukcji w aplikacji CaveatEmptor

Transakcją aplikacyjną jest zatwierdzenie aukcji, które wymaga dwóch żądań użytkownika. Najpierw administrator wybiera oczekującą aukcję, by zobaczyć jej szczegóły; następnie zatwierdza aukcję, przenosząc ją do stanu aktywnego. Drugie żądanie musi kontrolować wersję aukcji, by sprawdzić, czy inny użytkownik nie zmienił jej lub nie zatwierdził po jej wyświetleniu w przeglądarce internetowej.

Logika biznesowa zatwierdzania aukcji powinna (jak zwykle) znaleźć się w modelu dziedzinowym. Dodajmy do klasy Item metodę approve().

```

public void approve(User byUser) throws BusinessException {
    if ( !byUser.isAdmin() )
        throw new PermissionException("Nie jesteś administratorem!");

    if ( !state.equals(ItemState.PENDING) )
        throw new IllegalStateException("aukcja przedmiotowa jest jeszcze szkicem.");

    state = ItemState.ACTIVE;
    approvedBy = byUser;
    approvalDatetime = new Date();
}
  
```

Tak naprawdę jesteśmy zainteresowani kodem, który wywoła przedstawioną metodę.

Odpowiedzi na pytania

Czy transakcje aplikacyjne rzeczywiście są transakcjami? Większość książek definiuje transakcję jako cztery właściwości ACID: niepodzielność (atomowość), spójność, izolację i trwałość. Czy transakcja aplikacyjna naprawdę zasługuje na miano transakcji w świetle tej definicji? Spójność i trwałość nie wydają się problemem, ale co z niepodzielnością i izolacją? Przedstawiony przykład jest zarówno niepodzielny, jak i izolowany, ponieważ cała aktualizacja zachodzi w ostatnim cyklu żądanie-odpowiedź (czyli w ostatniej transakcji bazodanowej). Przedstawiona wcześniej

definicja transakcji aplikacyjnej dopuszcza aktualizację w dowolnym momencie cyklu. Jeżeli transakcja aplikacyjna przeprowadza aktualizację w innym stanie niż ostatni, nie jest niepodzielna i może nie być izolowana. Mimo to sądzimy, iż termin **transakcja** pozostaje odpowiedni, bo systemy stosujące tego rodzaju zadania biznesowe najczęściej wprowadzają dodatkową funkcjonalność rekompensującą brak niepodzielności (na przykład możliwość ręcznego wycofania wcześniejszych kroków transakcji aplikacyjnej).

Po omówieniu przypadku użycia prześledźmy różne sposoby jego implementacji. Zaczniemy od podejścia, którego nie polecamy.

8.2.2. Trudny sposób

Utrudniająca życie implementacja transakcji aplikacyjnych polega na porzuceniu wszystkich trwałych obiektów między kolejnymi żądaniami. Usprawiedliwienie takiego postępowania wynika z faktu, iż po zakończeniu transakcji bazodanowej nie mamy żadnej pewności, że obiekt trwały ma ten sam stan co baza danych. Im dłużej administrator zastanawia się nad zatwierdzeniem aukcji, tym większe jest ryzyko, że użytkownik dokona zmian w aukcji i tym samym obiekt Item stanie się nieaktualny.

Przypuśćmy, że pierwsze żądanie wykonuje następujący kod, by pobrać szczegóły aukcji:

```
public Item viewItem(Long itemId) {  
    return ItemDAO.getItemId(itemId);  
}
```

Ten sposób myślenia zalecałby usunięcie zwróconego obiektu Item tuż po zrenderowaniu widoku — na potrzeby następnego żądania zapamiętana zostałaby tylko wartość identyfikatora. Tylko pozornie pobieranie nowego egzemplarza Item na początku nowego żądania ma sens (choć takie podejście stosują ziarenka encyjne dla trwałych danych), bo mamy pewność, że egzemplarz w drugiej transakcji bazodanowej na pewno zawiera aktualne informacje.

Z drugiej strony ten sposób myślenia ma poważną wadę: administrator zapewne użył niekoniecznie aktualnych danych przy podejmowaniu decyzji o zatwierdzeniu! Wczytywanie obiektu Item w drugim żądaniu jest bezcelowe, bo wczytany stan nigdzie nie zostanie użyty — a przynajmniej nie można go wykorzystać do podjęcia decyzji, czy zatwierdzić aukcję.

W celu zapewnienia, że szczegóły opisu widziane przez administratora systemu są zgodne z obecną wersją opisu w drugiej transakcji bazodanowej, musimy przeprowadzić jawne i ręczne sprawdzenie wersji. Następujący kod przedstawia sposób implementacji tego zadania w serwecie kontrolera:

```
public void approveAuction(Long itemId, int itemVersion, Long adminId)  
throws BusinessException {  
  
    Item item = new ItemDAO().getItemById(itemId);  
    if ( !(itemVersion==item.getVersion()) )
```

```

        throw new StaleItemException();

User admin = new UserDAO().getUserById(adminId);
item.approve(admin);
}

```

W przedstawionym przykładzie ręczne sprawdzanie wersji nie jest trudne do implementacji.

Czy rzeczywiście słusznie nazwaliśmy ten sposób **trudnym**? W bardziej złożonych przypadkach używających związków trudno ręcznie przeprowadzić sprawdzenie wersji dla wszystkich obiektów poddawanych aktualizacji. To ręczne sprawdzenie staje się szumem — wynika z wyboru sposobu implementacji systemu, a nie z problemu biznesowego.

Co gorsza, przedstawiony fragment kodu zawiera i inny niepotrzebny szum. Pobraliśmy już obiekty Item i User w poprzednim żądaniu. Czy naprawdę musimy wczytywać je ponownie w obecnym? Kod sterujący warto byłoby sprowadzić do następującej sytuacji:

```

public approveAuction(Item item, User admin)
    throws BusinessException {
    item.approve(admin);
}

```

W ten sposób nie tylko zaoszczędziliśmy trzy wiersze kodu, ale i zwiększyliśmy obiektowość rozwiązania — system korzysta wyłącznie z obiektów modelu dziedzinowego i nie przekazuje wartości identyfikatorów. Co więcej, kod wykoną się znacznie szybciej, bo nie zawiera dwóch zapytań SELECT wczytujących zbędne dane. Jak osiągnąć tę prostotę, używając Hibernate?

8.2.3. Odłączone obiekty trwałe

Przypuśćmy, że zapamiętujemy obiekt Item jako odłączony w sesji HTTP użytkownika. Możemy z niego skorzystać w drugiej transakcji bazodanowej, dołączając go do nowej sesji Hibernate metodą lock() lub update(). Sprawdźmy, jak wygląda kod stosujący metody wymienione.

W przypadku metody lock() modyfikujemy metodę approveAuction() w sposób następujący:

```

public void approveAuction(Item item, User admin)
    throws BusinessException {

    try {
        HibernateUtil.getSession().lock(item, LockMode.NONE);
    } catch (HibernateException ex) {
        throw new InfrastructureException(ex);
    }
    item.approve(admin);
}

```

Wywołanie metody Session.lock() ponownie przypisuje przedmiot do sesji Hibernate i zapewnia, że od tego momentu każda wprowadzona w nim zmiana zostanie przeniesiona do bazy danych w momencie zatwierdzania (poszczególne tryby

blokad LockMode opisuje punkt 5.1.7). Ponieważ obiekt Item stosuje wersjonowanie (na przykład z powodu użycia właściwości <version> w odwzorowaniu), Hibernate sprawdzi numer wersji obiektu w pamięci z numerem w bazie danych, stosując mechanizm opisany w punkcie 5.2.1. Z tego powodu nie musimy zakładać blokady pesymistycznej, jeśli inne współbieżne transakcje powinny móc odczytywać dane przedmiotu w momencie jego zatwierdzania.

Najlepiej byłoby ukryć kod Hibernate w nowej metodzie DAO, więc dodajmy do klasy ItemDAO nową metodę lock(). Uprości to znacząco kod metody approveAuction().

```
public approveAuction(Item item, User admin)
    throws BusinessException {
    new ItemDAO().lock(item, false); // Nie bądź pesymistyczny.
    item.approve(admin);
}
```

Alternatywne podejście to metoda update(). W przedstawianym przykładzie różnica między lock() i update() polega na tym, że update() można wywołać po zmianie stanu przedmiotu, czyli zmianach dokonanych przez administratora przed jego zatwierdzeniem.

```
public approveAuction(Item item, User admin)
    throws BusinessException {
    item.approve(admin);
    new ItemDAO().saveOrUpdate(item);
}
```

Nowa metoda saveOrUpdate() klasy ItemDAO wywołuje metodę `HibernateUtil.getSession().saveOrUpdate(item)`. Także i tu Hibernate sprawdzi wersję obiektu przedmiotu w momencie aktualizacji.

Czy ta implementacja stosująca obiekty odłączone rzeczywiście okazuje się prostsza od pierwszego rozwiązania? Nadal potrzebujemy jawnego wywołania metody obiektu ItemDAO, więc zysk jest niewielki. W bardziej złożonym przykładzie zawierającym asocjacje zysk byłby większy, gdyż wywołanie lock() lub update() miałoby kaskadowy wpływ na powiązane obiekty. Pamiętaj, że przedstawiona implementacja okazuje się wydajniejsza — nie wymaga niepotrzebnych wywołań SELECT.

Nadal nie jesteśmy usatysfakcjonowani. Czy można uniknąć ponownego jawnego przypisywania obiektu do sesji? Jeden ze sposobów polega na wykorzystaniu tej samej sesji Hibernate dla dwóch transakcji bazodanowych. Wzorzec ten opisaliśmy w rozdziale 5. jako **jedną sesję na transakcję aplikacyjną**.

8.2.4. Długa sesja

Długa sesja to sesja Hibernate obejmująca swym zasięgiem całą transakcję aplikacyjną, co umożliwia ponowne użycie tych samych obiektów trwałych w wielu transakcjach bazodanowych. Unika się ponownego przypisywania odłączonych obiektów utworzonych lub pobranych we wcześniejszych transakcjach bazodanowych.

Sesja zawiera dwa istotne rodzaje stanu: bufor obiektów trwałych i połączenie JDBC. Wcześniej wskazaliśmy, dlaczego nie należy przetrzymywać otwartych połączeń bazodanowych między transakcjami. Z tego powodu sesja musi zwolnić połączenie bazodanowe między żądaniami, jeśli ma pozostać otwarta przez dłuższy czas.

Metoda `disconnect()` zwalnia połączenie JDBC sesji bez zamykania sesji. Metoda `reconnect()` uzyskuje nowe połączenie bazodanowe dla istniejącej sesji. W ten sposób sesja może trwać bardzo długo (kilka żądań użytkownika) bez zajmowania cennych zasobów.

Obecnie sesję przechowujemy tylko w obiektach `ThreadLocal`. Ponieważ poszczególne żądania najczęściej trafiają do różnych wątków, a jedna sesja dotyczy kilku żądań, potrzebujemy innego rozwiązania. W środowisku wyłącznie serwletowym idealnym miejscem do przechowywania sesji Hibernate między żądaniami okazuje się obiekt `HttpSession`.

Można łatwo zmienić przedstawiony wcześniej filtr serwletu, by odłączał sesję między żądaniami, zamiast ją zamykać. Ten sam filtr idealnie nadaje się również do ponownego przyłączenia sesji. Nową metodę `doFilter()` przedstawia listing 8.6 (choć w przykładzie stosujemy filtr serwletu, ta sama zasada działania dotyczy dowolnego innego kodu przechwytyującego).

Listing 8.6. Metoda `doFileter()` dla długich sesji

```

public void doFilter(ServletRequest request,
                      ServletResponse response,
                      FilterChain chain)
throws IOException, ServletException {

    // Spróbuj pobrać sesję Hibernate z obiektu HttpSession.
    HttpSession userSession = ((HttpServletRequest) request).getSession();
    Session hibernateSession = (Session)
        userSession.getAttribute("HibernateSession");

    // Podłącz sesję do aktualnego wątku.
    if (hibernateSession != null)
        HibernateUtil.reconnect(hibernateSession);
    try {
        chain.doFilter(request, response);

        // Zatwierdź dowolną oczekującą transakcję.
        HibernateUtil.commitTransaction();
    } finally {
        // Odlacz sesje.
        hibernateSession = HibernateUtil.disconnectSession();

        // Następnie zapamiętaj ją w obiekcie HttpSession.
        userSession.setAttribute("HibernateSession", hibernateSession);
    }
}

```

W bloku finally zamiast metody closeSession() wywołujemy metodę disconnectSession(). Przed uruchomieniem łańcucha filtrów sprawdzamy, czy sesja użytkownika zawiera istniejący obiekt Session. Jeśli tak, dołączamy go metodą reconnect() i wiążemy z aktualnym wątkiem. Operacje odlączenia i podłączenia dla sesji Hibernate odpowiednio kończą i uruchamiają nowe połączenie JDBC.

Niestety, przedstawiona implementacja nigdy nie zamknie sesji Hibernate — uzyskuje ona taki sam czas życia jak sesja użytkownika. Wszystkie kolejne transakcje aplikacyjne ponownie trafią do tej samej sesji Hibernate. Sesja przechowuje bufor trwałych obiektów, który rośnie w czasie. Tej sytuacji nie możemy zaakceptować.

Na początku każdej nowej transakcji aplikacyjnej musimy zapewnić stosowanie całkowicie nowej sesji Hibernate. Potrzebujemy nowej metody wskazującej początek transakcji aplikacyjnej, która zamknie aktualną sesję i rozpocznie nową. Poza metodami disconnect() i reconnect() do klasy HibernateUtil dodajemy również metodę newApplicationTx().

```
public static void reconnect(Session session)
    throws InfrastructureException {
    try {
        session.reconnect();
        threadSession.set(session);
    } catch (HibernateException ex) {
        throw new InfrastructureException(ex);
    }
}

public static Session disconnectSession()
    throws InfrastructureException {

    Session session = getSession();
    try {
        threadSession.set(null);
        if (session.isConnected() && session.isOpen())
            session.disconnect();
    } catch (HibernateException ex) {
        throw new InfrastructureException(ex);
    }
    return session;
}

public static void newApplicationTx() {
    closeSession();
}
```

Metodę newApplicationTx() wywołujemy na początku transakcji aplikacyjnej tuż przed wyświetleniem administratorowi przedmiotu do zatwierdzenia (przed wywołaniem metody viewItem(itemId) w serwecie kontrolera).

```
HibernateUtil.newApplicationTx();
viewItem(itemId);
```

Metoda viewItem() pozostaje taka sama jak wcześniej:

```
public Item viewItem(Long itemId) {
    return ItemDAO.getItemById(itemId);
}
```

Zgodnie z obietnicą metodą approveAction() ulega uproszczeniu:

```
public approveAuction(Item item, User admin)
    throws BusinessException {
    item.approve(admin);
}
```

Bardziej złożone transakcje aplikacyjne niejednokrotnie zmieniają model dziedzinowy w kilku kolejnych żądaniach. Ponieważ kod przechwytyujący (interceptor) zapisuje stan sesji w bazie danych na końcu każdego żądania, transakcja aplikacyjna może przestać być niepodzielna w momencie zapisu danych w środkowym żądaniu. Aplikacje wymagające odtworzenia niepełnej transakcji aplikacyjnej w przypadku błędu w systemie nie muszą przejmować się opisaną sytuacją. Inne aplikacje wymagają niepodzielności — w takiej sytuacji zmiany powinny trafić do bazy danych tylko w ostatnim żądaniu. Nietrudno zapewnić to działanie w obiektach odłączanych, ale w długiej sesji okazuje się ono bardziej złożone.

Rozwiążanie polega na ustaleniu sesji Hibernate w trybie FlushMode.NEVER i jawnym przekazaniu zmian do bazy danych na końcu transakcji aplikacyjnej. Do tego momentu wszystkie zmiany istnieją jedynie w pamięci (w zasadzie w obiekcie HttpSession). Pamiętaj, że wykonywane w międzyczasie zapytania nic nie wiedzą o zmianach i tym samym mogą zwracać nieaktualne informacje.

To nie ostatnia komplikacja związana z długimi sesjami. Obiekt Session nie jest bezpieczny pod kątem wątków, natomiast mechanizm serwów dopuszcza współbieżne wykonywanie wielu żądań od tego samego użytkownika. Możliwe jest, że współbieżne żądania otrzymają ten sam obiekt Session z HttpSession, jeśli na przykład użytkownik przypadkiem dwukrotnie kliknie przycisk zatwierdzenia. Trudno w takiej sytuacji przewidzieć zachowanie aplikacji. Przedstawiony problem dotyczy również obiektów odłączonych (one także nie są zabezpieczone przed współbieżnymi wątkami). W zasadzie problem dotyczy dowolnej aplikacji internetowej przechowującej możliwy do zmiany stan w obiekcie HttpSession.

Ponieważ jest to ogólny problem niemal wszystkich aplikacji internetowych, nie będziemy poszukiwać rozwiązania w niniejszej książce. Jedne aplikacje mogą po prostu odrzucać kolejne żądania przychodzące od tego samego użytkownika, jeśli realizują inne jego żądanie. Nic nie stoi na przeszkodzie, by takie zabezpieczenie wprowadzić w filtrze serwletów. Inne aplikacje mogą wymagać szeregowania żądań pochodzących od tego samego użytkownika (efekt ten łatwo uzyskać, synchronizując obiekt HttpSession w filtrze serwletu).

Niektóre aplikacje (na przykład programy z wielookienkowym interfejsem) mogą potrzebować wielu współbieżnych transakcji aplikacyjnych. W takiej sytuacji aplikacja przydziela każdemu użytkownikowi więcej niż jedną długą sesję, zapewniając odwzorowanie między oknem będącym źródłem żądania a długą sesją obsługującą powiązaną transakcję aplikacyjną.

Wszystkie zagadnienia działały poprawnie w architekturze czysto serwletowej. Czy w aplikacjach EJB będzie podobnie? Jeśli z sesji Hibernate korzysta

wyłącznie warstwa EJB, przechowywanie jej w sesji użytkownika warstwy internetowej nie wchodzi w grę. W jakiej sytuacji atrakcyjne stają się opisane wcześniej sposoby obsługi transakcji aplikacyjnych?

8.2.5. Wybór odpowiedniej implementacji transakcji aplikacyjnych

Zapewne zgadujesz, że należy unikać rozwiązania, które nazwaliśmy trudnym sposobem. Nigdy nie zastosowalibyśmy go we własnych aplikacjach. Jeśli jednak architektura systemu wymaga, by warstwa internetowa nigdy nie miała bezpośredniego dostępu do modelu dziedzinowego (model ten zostaje całkowicie ukryty przed warstwą prezentacyjną przez pośrednią warstwę DTO) lub też nie można przechować stanu sesji użytkownika na warstwie EJB (stosujesz tylko bezstanowe ziarenka sesyjne), w zasadzie nie masz wyboru. Nic nie stoi na przeszkodzie, by w ten sposób tworzyć aplikacje używające Hibernate (ORM obsługuje je bez problemów). Zaletami tego podejścia są: brak zmartwień co do różnic między obiektem trwałym i odłączonym oraz eliminacja zgłaszania wyjątków `LazyInitializationException` przez obiekty odłączone.

Obecnie większość aplikacji Hibernate stosuje podejście z obiektami odłączonymi, czyli tworzenie nowej sesji dla każdej transakcji bazodanowej. W szczególności można je polecić aplikacjom, gdzie logika biznesowa i dostęp do danych wykonywane są na warstwie EJB, ale z modelu dziedzinowego korzysta też warstwa internetowa (unika się wtedy żmudnego tworzenia obiektów DTO). Co więcej, podejście to sprawdziło się również w aplikacjach wykorzystujących tylko serwety. Nie wydaje nam się jednak, że to najlepsze rozwiązanie dla aplikacji serwetowych.

Dla tego rodzaju aplikacji polecamy długie sesje. Trudno wyjaśnić ich implementację i nie są one dobrze rozumiane przez społeczność Hibernate. Wynika to może ze słabego zrozumienia sposobu działania transakcji aplikacyjnych przez społeczność programistów języka Java. Po prostu, wiele osób nie myśli jeszcze w tych kategoriach. Mamy nadzieję na poprawę sytuacji w krótkim okresie czasu, bo transakcje aplikacyjne przydają się nie tylko w kontekście długich sesji.

Stosowanie długich sesji w aplikacji EJB wymaga odnalezienia sposobu na powiązanie sesji Hibernate z konkretnym użytkownikiem bez opuszczania warstwy EJB. Ale czy w ogóle należy przechowywać stan powiązany z użytkownikiem na warstwie EJB? Najczęściej odrzuca się takie pomysły.

Wydaje się nam, że nie ma jakichś **wstępnych** przeciwwskazań, by przechowywanie stanu powiązanego z sesją użytkownika na warstwie biznesowej było mniej wydaje od przechowywania go na warstwie internetowej. Wręcz przeciwnie, wydaje się, że **największą** wydajność oferuje właśnie pamiętanie stanu na warstwie średniej, gdyż tam właśnie przynależy, niż ciągła serializacja stanu w celu jego przesyłu do i z bazy danych i warstwy internetowej na początku każdego żądania. Właśnie z tego powodu specyfikacja EJB definiuje stanowe ziarenko sesyjne. Ziarenko stanowe przechowuje sesję Hibernate w zmiennej egzemplarza, odłączając sesję między kolejnymi żądaniami. Niestety, wiele serwerów

aplikacji niewystarczająco wydajnie implementuje stanowe ziarenka sesyjne. Sprawdź docelowy serwer, zanim wybierzesz to rozwiązanie.

Zakończyliśmy opis podstawowych sposobów projektowania aplikacji. Przejedźmy do bardziej egzotycznych tematów pojawiających się w momencie obsługi przestarzałych baz danych lub innych nietypowych rodzajów informacji.

8.3. Obsługa specjalnych rodzajów danych

Niektóre rodzaje danych wymagają szczególnego potraktowania, które nie pasuje do ogólnych zasad opisywanych w pozostałe części książki. W niniejszym rozdziale zajmiemy się specjalnymi rodzajami danych zwiększającymi złożoność kodu Hibernate.

Pierwszym i najważniejszym problemem są **przestarzałe** bazy danych. Relatywnie niewiele projektów ma ten luksus, by od nowa rozpoczynać projektowanie bazy danych; większość aplikacji musi współdzielić dane ze starszymi aplikacjami i systemami. W takiej sytuacji zmiana schematu bazy danych często nie wchodzi w grę, choć z naszego punktu widzenia jest nieoptymalna. Co gorsza, można trafić na tak egzotyczny projekt bazy danych, że zastosowanie Hibernate okaże się niemożliwe (na szczęście zdarza się to niezwykle rzadko).

Drugim interesującym zagadnieniem są dane wymagające **audytu**. Dowolna zmiana w tych danych musi wiązać się z utworzeniem wpisu w dzienniku wskazującym zmianę, czas i datę jej wykonania oraz nazwę użytkownika ją zlecającego. Hibernate zawiera specjalny mechanizm współpracujący z dziennikami audytowymi (i innymi zagadnieniami wymagającymi reagowania na zdarzenia zmiany trwałych danych).

Najpierw przyjrzyjmy się odwzorowywanie starszych danych i wymuszany przez to zmianom w projektach aplikacji.

8.3.1. Starsze schematy baz danych i klucze złożone

Gdy aplikacja musi używać istniejącego schematu bazy danych, chce się go modyfikować w jak najmniejszym stopniu. Każda zmiana mogłaby uniemożliwić prawne działanie innych starszych aplikacji lub wymagać bardzo kosztownej migracji danych. Z drugiej strony rzadko się zdarza, by nowa aplikacja nie wymagała żadnych zmian w schemacie bazy danych — przecież najczęściej jej powstanie ma związek z nowymi wymaganiami biznesowymi, które w sposób naturalny prowadzą do ewolucji bazy danych.

Z tego względu rozważymy dwa rodzaje problemów: mające związek ze zmianą wymagań biznesowych (które najczęściej wymuszają modyfikacje schematu bazy danych) i wynikające wyłącznie z chęci unowocześnienia aplikacji rozwiązucej ten sam problem biznesowy (który w większości przypadków można rozwiązać bez zmian w schemacie). Pierwszy rodzaj problemu najczęściej analizuje się, sprawdzając **logiczny** model danych. Drugi rodzaj problemu dotyczy raczej implemen-

tacji logicznego modelu danych na podstawie konkretnego fizycznego schematu bazy danych.

Zauważwszy ten fakt, nietrudno odgadnąć, które rodzaje konkretnych problemów wymagają zmian w schemacie bazy danych: dodające nowe rodzaje encji, modyfikujące istniejące encje, dodające atrybuty do aktualnych encji lub zmieniające związki między encjami. Problemy **niewymagające zmiany** w schemacie najczęściej dotyczą nietrafionych definicji kolumn konkretnych encji.

Na razie skupmy się na drugim rodzaju problemów. Nietrafiona (lub mało poręczna) definicja kolumny najczęściej dotyczy dwóch przypadków:

- ◆ użycia klucza naturalnego (szczególnie złożonego),
- ◆ nieodpowiedniego typu kolumny.

Wspominaliśmy wcześniej, że klucze obce nie są dobrym pomysłem. Niejednokrotnie utrudniają refaktoryzację modelu danych po zmianie wymagań biznesowych. Niekiedy ujemnie wpływają na wydajność. Niestety, wiele starszych systemów korzysta z (naturalnych) kluczy złożonych i niezależnie od ogromu zalet kluczy sztucznych nie można ich wprowadzić. Z tego powodu Hibernate obsługuje klucze naturalne. Jeśli klucz naturalny jest kluczem złożonym, odwzorowanie musi zawierać element <composite-id>.

Nieodpowiedniego typu kolumny najprościej ukryć, stosując własne typy odwzorowania Hibernate (UserType lub CompositeUserType) omówione dokładnie w rozdziale 6.

Prześledźmy kilka przykładów ilustrujących rozwiązywanie obu problemów. Zaczniemy od odwzorowań kluczy naturalnych.

Odwzorowanie tabeli z kluczem naturalnym

Tabela USER stosuje klucz sztuczny USER_ID oraz ograniczenie wprowadzające unikatowość wartości kolumny USERNAME. Oto fragment stosowanego odwzorowania Hibernate:

```
<class name="User" table="USER">
    <id name="id" column="USER_ID" unsaved-value="null">
        <generator class="native"/>
    </id>

    <version name="version"
        column="VERSION"/>

    <property name="username"
        column="USERNAME"
        unique="true"
        not-null="true"/>
    ...
</class>
```

Zauważ, że sztuczny identyfikator wykorzystuje wartość podaną w atrybucie unsaved-value do wskazania Hibernate, czy egzemplarz klasy jest obiektem odłączonym czy nowym. Poniższy fragment kodu tworzy nowy obiekt trwały dla użytkownika:

```
User user = new User();
user.setUsername("jan");
user.setFirstname("Jan");
user.setLastname("Kowalski");
session.saveOrUpdate(user); // Jako efekt uboczny generuje wartość identyfikatora.
System.out.println(session.getIdentifier(user)); // Wyświetla wartość 1.
session.flush();
```

Jeżeli na tabelę USER natknęliśmy się w starszym schemacie, najprawdopodobniej kluczem głównym jest kolumna USERNAME. Nie mamy sztucznego identyfikatora, musimy więc w pliku odwzorowania klasy wskazać Hibernate strategię generowania identyfikatora o nazwie assigned, by oczekivał wskazania klucza naturalnego przez aplikację przed zapisem obiektu.

```
<class name="User" table="USER">
  <id name="username" column="USERNAME" >
    <generator class="assigned"/>
  </id>

  <version name="version"
    column="VERSION"
    unsaved-value="0"/>
  ...
</class>
```

Atrybut unsaved-value zniknął z elementu <id>. Przypisany przez aplikację identyfikator nie może posłużyć do sprawdzenia, czy egzemplarz klasy jest odłączony czy ulotny (aplikacja dba, by nigdy nie był równy null). Atrybut umieściliśmy więc we właściwości <version>. W ten sposób uzyskaliśmy dokładnie ten sam efekt. Kod zapisujący użytkownika nie zmienił się:

```
User user = new User();
user.setUsername("jan"); // Przypisanie klucza głównego.
user.setFirstname("Jan");
user.setLastname("Kowalski");
session.saveOrUpdate(user); // Zapisze obiekt, bo numerem wersji jest 0
System.out.println(session.getIdentifier(user)); // Wyświetla „jan”.
session.flush();
```

Jeśli klasa z kluczem naturalnym **nie** deklaruje właściwości wersji ani znacznika czasowego, znacznie trudniej uzyskać poprawne działanie saveOrUpdate() i kaskadowego zapisu. Można skorzystać z własnego obiektu Interceptor opisywanego w dalszej części rozdziału. Z drugiej strony, jeżeli komuś nie przeszkał jawnie stosowanie metody save() przy zapisie i metody update() przy aktualizacji zamiast jednej metody saveOrUpdate(), Hibernate nie musi sam rozróżnić, czy obiekt znajduje się w stanie odłączenia czy stanie ulotnym, więc obiekt Interceptor staje się zbędny.

Złożone klucze naturalne stosują podobne podejście.

Odwzorowanie tabeli z kluczem złożonym

Z punktu widzenia Hibernate klucz złożony można obsługiwać jako przypisywany identyfikator typu wartości (typ Hibernate jest komponentem). Przypuśćmy, że klucz główny tabeli użytkowników składa się z kolumn USERNAME i ORGANIZATION_ID.

W takiej sytuacji wystarczy do klasy User dodać właściwość organizationId i zastosować odwzorowanie:

```
<class name="User" table="USER">

    <composite-id>
        <key-property name="username"
                      column="USERNAME"/>

        <key-property name="organizationId"
                      column="ORGANIZATION_ID"/>

    </composite-id>

    <version name="version"
              column="VERSION"
              unsaved-value="0"/>

    ...
</class>
```

Tym razem kod zapisujący obiekt User ma postać:

```
User user = new User();

// Przypisanie klucza głównego.
user.setUsername("jan");
user.setOrganizationId(42);

user.setFirstname("Jan");
user.setLastname("Kowalski");
session.saveOrUpdate(user);      // Zapisze obiekt, bo numerem wersji jest 0.
session.flush();
```

Jaki element zastosować jako identyfikator, gdy chcemy załadować obiekt metodą load() lub get()? Można użyć egzemplarza klasy User. Oto przykład:

```
User user = new User();

// Przypisanie klucza głównego.
user.setUsername("jan");
user.setOrganizationId(42);

// Załadowanie stanu obiektu trwałego.
session.load(User.class, user);
```

W przedstawionym kodzie klasa User działa jako własny identyfikator. Pamiętaj, że w tej sytuacji klasa musi implementować interfejs Serializable oraz metody equals() i hashCode(). Bardziej eleganckie rozwiązanie definiuje osobną klasę identyfikatora złożonego zawierającą tylko właściwości klucza złożonego. Nadajmy tej klasie nazwę UserId:

```
public class UserId implements Serializable{
    private String username;
    private String organizationId;

    public UserId(String username, String organizationId) {
        this.username = username;
```

```

        this.organizationId = organizationId;
    }

    // Metody dostępowe...

    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof UserId)) return false;
        final UserId userId = (UserId) o;
        if (!organizationId.equals(userId.organizationId)) return false;
        if (!username.equals(userId.username)) return false;
        return true;
    }

    public int hashCode() {
        int result;
        result = username.hashCode();
        result = 29 * result + organizationId.hashCode();
        return result;
    }
}

```

Niezwyczajna jest odpowiednia implementacja metod `equals()` i `hashCode()`, ponieważ Hibernate używa tych metod do wyszukiwania obiektu w buforze. Klasy kluczowe powinny dodatkowo implementować interfejs `Serializable`.

Teraz usuniemy właściwości `userName` i `organizationId` z klasy `User`, by wstawić jedną właściwość `userId`. Nowe odwzorowanie będzie miało postać:

```

<class name="User" table="USER">

    <composite-id name="userId" class="UserId">
        <key-property name="userName"
                      column="USERNAME"/>

        <key-property name="organizationId"
                      column="ORGANIZATION_ID"/>

    </composite-id>

    <version name="version"
              column="VERSION"
              unsaved-value="0"/>
    ...
</class>

```

Zamianie ulega również kod zapisujący dane użytkownika:

```

UserId id = new UserId("jan", 42);

User user = new User();

// Przypisanie klucza głównego.
user.setUserId(id);

user.setFirstname("Jan");
user.setLastname("Kowalski");

```

```
session.saveOrUpdate(user);      // Zapisze obiekt, bo numerem wersji jest 0.  
session.flush();
```

Następujący kod przedstawia sposób wczytywania obiektu na podstawie jego identyfikatora:

```
UserId id = new UserId("jan", 42);  
User user = (User) session.load(User.class, id);
```

Przypuśćmy, że kolumna ORGANIZATION_ID to klucz obcy tabeli ORGANIZATION i chcemy w modelu Javy odpowiednio odwzorować ten związek. zalecamy użycie odwzorowania asocjacji <many-to-one> z argumentami insert="false" i update="false".

```
<class name="User" table="USER">  
  
<composite-id name="userId" class="UserId">  
    <key-property name="userName"  
        column="USERNAME"/>  
  
    <key-property name="organizationId"  
        column="ORGANIZATION_ID"/>  
  
</composite-id>  
  
<version name="version"  
        column="VERSION"  
        unsaved-value="0"/>  
  
<many-to-one name="organization"  
        class="Organization"  
        column="ORGANIZATION_ID"  
        insert="false" update="false"/>  
    ...  
</class>
```

Argumenty insert="false" i update="false" informują Hibernate, by pomijał tę właściwość w trakcie aktualizacji lub wstawiania obiektu User, ale zapewniał odczyt właściwości metodą jan.getOrganization().

Alternatywne podejście polega na użyciu elementu <key-many-to-one>.

```
<class name="User" table="USER">  
  
<composite-id name="userId" class="UserId">  
    <key-property name="userName"  
        column="USERNAME"/>  
  
    <key-many-to-one name="organization"  
        class="Organization"  
        column="ORGANIZATION_ID"/>  
  
</composite-id>  
  
<version name="version"  
        column="VERSION"
```

```
        unsaved-value="0" />
    ...
</class>
```

Najczęściej istnienie asocjacji w klasie identyfikatora złożonego nie jest wygodne, więc nie polecamy tego podejścia, chyba że wymaga tego sytuacja.

Ponieważ klasa USER stosuje klucz złożony, dowolny klucz obcy dotyczący tej klasy również jest złożony. Na szczęście, Hibernate potrafi ukryć szczegóły przed kodem Javy. Odwzorowanie asocjacji w klasie Item ma postać:

```
<many-to-one name="seller" class="User">
    <column name="USERNAME"/>
    <column name="ORGANIZATION_ID"/>
</many-to-one>
```

Dowolna kolekcja należąca do klasy User również zawiera złożony klucz obcy. Przykładem może być asocjacja odwrotna z listą przedmiotów użytkownika:

```
<set name="items" lazy="true" inverse="true">
    <key>
        <column name="USERNAME"/>
        <column name="ORGANIZATION_ID"/>
    </key>
    <one-to-many class="Item"/>
</set>
```

Kolejność występowania kolumn ma znaczenie — powinna odpowiadać kolejności ich podania w elemencie `<composite-id>`.

Przejedźmy do drugiego problemu ze starszymi bazami danych — nieodpowiednich typów kolumn.

Własne typy odwzorowujące nieodpowiednie kolumny

Stwierdzenie **nieodpowiedni typ kolumny** obejmuje swym zasięgiem wiele problemów, na przykład: użycie kolumny CHAR zamiast VARCHAR, użycie kolumny VARCHAR dla danych numerycznych, użycie specjalnej wartości zamiast NULL z SQL itp. Łatwo zaimplementować własny UserType obsługujący wartości typu CHAR (przez usuwanie zbędnych spacji z tekstów zwracanych przez sterownik JDBC), konwertujących typy tekstowe na liczbowe i specjalne wartości na typ null Javy. Nie będziemy przedstawiać przykładowego kodu dla żadnego z tych typowych problemów — wykonanie go pozostawiamy Czytelnikowi (wskaźówka co do implementacji szukaj w punkcie 6.1.3).

Zamiast tego przyjrzyjmy się bardziej interesującemu problemowi. Do tej pory klasa User wykorzystywała dwie właściwości dla imienia i nazwiska użytkownika: `firstname` i `lastname`. Gdy jeszcze dodamy inicjał lub tytuł, klasa stanie się mało przejrzysta. Dzięki obsłudze komponentów w Hibernate możemy łatwo poprawić model obiektowy, stosując jedną właściwość `name` typu Name (zawierającą wszystkie szczegóły nazwiska).

Załóżmy również, że baza danych zawiera tylko jedną kolumnę o nazwie NAME. Musimy odwzorować trzy właściwości klasy Name na jedną kolumnę. Poniższy UserType obrazuje, w jaki sposób można tego dokonać (upraszczamy implementację, zakładając, że `initial` nigdy nie jest równe null).

```
public class NameUserType implements UserType {  
  
    private static final int[] TYPES = { Types.VARCHAR };  
    public int[] sqlTypes() { return TYPES; }  
    public Class returnedClass() { return Name.class; }  
  
    public boolean isMutable() {  
        return true;  
    }  
  
    public Object deepCopy(Object value) throws HibernateException {  
        Name name = (Name) value;  
        return new Name(name.getFirstname(), name.getInitial(),  
                        name.getLastname());  
    }  
  
    public boolean equals(Object x, Object y) throws HibernateException {  
        // Wykorzystuje implementację equals z klasy Name.  
        return x==null ? y ==null : x.equals(y);  
    }  
  
    public Object nullSafeGet(ResultSet resultSet, String[] names,  
                             Object owner)  
        throws HibernateException, SQLException {  
  
        String dbName = (String) Hibernate.STRING.nullSafeGet(resultSet,  
                    names);  
  
        if (dbName==null) return null;  
  
        StringTokenizer tokens = new StringTokenizer(dbName);  
        Name realName = new Name(tokens.nextToken(),  
  
        String.valueOf(tokens.nextToken().charAt(0)),  
                        tokens.nextToken());  
        return realName;  
    }  
  
    public void nullSafeSet(PreparedStatement statement, Object value,  
                           int index)  
        throws HibernateException, SQLException {  
  
        Name name = (Name) value;  
  
        String nameString = (name==null) ?  
            null :  
            name.getFirstname()  
            + ' ' + name.getInitial()  
            + ' ' + name.getLastname();  
        Hibernate.STRING.nullSafeSet(statement, nameString, index);  
    }  
}
```

Zauważ, że klasa wykonanie części zadań zleca wbudowanemu typowi Hibernate. To częste podejście, choć niewymagane.

Mamy nadzieję, że dostrzegasz, jak wiele problemów z nieodpowiednimi definicjami kolumn można rozwiązać, stosując własne typy Hibernate. Pamiętaj,

że za każdym razem, gdy Hibernate odczytuje dane z obiektu ResultSet lub zapisuje je do obiektu PreparedStatement, używa odpowiedniego obiektu Type. Niemal zawsze Type może być typem użytkownika (dotyczy to również asocjacji — typ ManyToOneType deleguje zadania do typu identyfikatora powiązanej klasy, który może być typem użytkownika).

Jeszcze jeden problem często pojawia się w kontekście obsługi starszych danych — integracja z wyzwalaczami bazodanowymi.

Wyzwalacze

Istnieją sensowne motywy stosowania wyzwalaczy nawet w całkowicie nowej bazie danych, więc starsze bazy danych to nie jedyne miejsce, w którym mogą się pojawić. Wyzwalacze i ORM najczęściej mają poważne problemy, by ze sobą współpracować. Trudno zsynchronizować efekt działania wyzwalacza z pamięciową reprezentacją danych.

Przypuśćmy, że tabela ITEM zawiera kolumnę CREATED odwzorowywaną na właściwość created typu Date inicjalizowaną wyzwalaczem wstawiania. Właściwość powinna stosować następujące odwzorowanie:

```
<property name="created"
          type="timestamp"
          column="CREATED"
          insert="false"
          update="false"/>
```

Zauważ użycie atrybutów insert="false" i update="false", by tej właściwości Hibernate nie uwzględniał w poleceniach SQL INSERT i UPDATE.

Po zapisaniu nowego obiektu Item Hibernate nie będzie wiedział, że wyzwalacz przypisał kolumnie wartość, gdyż zdarzenie to ma miejsce po poleceniu INSERT wstawiającym wiersz danych. Gdy chcemy użyć wartości w aplikacji, musimy jawnie poinformować Hibernate, by ponownie załadował obiekt polecienniem SQL SELECT. Oto przykład:

```
Item item = new Item();
...
HibernateUtil.beginTransaction();
Session session = HibernateUtil.getSession();

session.save(item);
session.flush();
session.refresh(item);

System.out.println(item.getCreated());

HibernateUtil.commitTransaction();
HibernateUtil.closeSession();
```

Wiele problemów z wyzwalaczami łatwo rozwiązać, jawnie wywołując metodę flush(), co powoduje natychmiastowe wykonanie wyzwalacza. Metoda refresh() pobierze do pamięci efekt działania wyzwalacza.

Musimy ostrzec przed szczególnym problemem pojawiającym się w momencie użycia obiektów odłączonych i bazy danych z wyzwalaczami. Ponieważ Hiber-

nate synchronizuje się z obrazem zawartości bazy danych w momencie dołączania odłączonego obiektu metodą `update()` lub `saveOrUpdate()`, może wykonać niepotrzebną instrukcję SQL `UPDATE`, by zapewnić synchronizację stanu bazy i sesji. W konsekwencji doprowadzi to do niepotrzebnego wzbudzenia wyzwalacza `UPDATE`. By uniknąć tego problemu, warto dodać atrybut `select-before-update` do odwzorowania klasy, która stosuje wyzwalacz. Jeśli tabela `ITEM` używa wyzwalacza aktualizacji, odwzorowanie powinno mieć postać:

```
<class name="Item"
      table="ITEM"
      select-before-update="true">
    ...
</class>
```

Atrybut wymusza na Hibernate pobranie migawki aktualnego stanu bazy danych poleceniem SQL `SELECT`, co pozwala uniknąć polecenia SQL `UPDATE`, jeśli stan obiektu w pamięci jest ten sam.

Podsumujmy dyskusję na temat starszych modeli danych. Hibernate oferuje kilka strategii radzenia sobie z (naturalnymi) kluczami złożonymi i kolumnami o niepoprawnych typach. Zalecamy dokładną analizę, czy jednak zmiana schematu bazy danych nie okaże się możliwa do przeprowadzenia. Z doświadczenia wiemy, iż wielu programistów odrzuca zmiany w bazie danych jako zbyt trudne i czasochłonne, więc poszukują ratunku w Hibernate. Niejednokrotnie okazuje się, że zmiany są prostsze, niż można było początkowo sądzić. Modyfikacje schematu bazy danych warto traktować jako element cyklu życia danych. Jeśli zmiana tabel bazy danych lub eksport i import danych rozwiązuje problem, jeden dzień pracy teraz może zaoszczędzić wiele dni w przyszłości, bo wiele przypadków szczególnych i dodatkowy kod nie ułatwia kreowania aplikacji.

Przyjrzyjmy się bliżej dziennikom audytowym i śledzeniu zmian stanu obiektów w bazie danych.

8.3.2. Dziennik audytowy

Dziennik audytowy to tabela bazy danych zawierająca informacje o zmianach w innych danych, w szczególności zawierająca informację o **zdarzeniu** powodującym zmianę. Przykładowo możemy przechowywać informacje o tworzeniu i aktualizacji obiektów Item. Najczęściej dziennik zawiera datę i czas wykonania zmiany, nazwę użytkownika ją zlecającego, typ zdarzenia i modyfikowany element.

Dzienniki audytowe najczęściej powstają na podstawie wyzwalaczy bazodanowych. Uważamy to za idealne podejście. Niestety, czasem aplikacja musi przejąć odpowiedzialność za kreowanie dziennika, szczególnie jeśli istotna jest przenośność między różnymi bazami danych.

W celu implementacji dziennika audytowego, należy wykonać kilka kroków:

1. Oznaczyć klasy trwałe, dla których ma być tworzony dziennik.
2. Zdefiniować zapamiętywane informacje: użytkownika, datę, czas, rodzaj modyfikacji itp.

3. Połączyć to wszystko obiektem Interceptor, który automatycznie utworzy wpisy dziennika.

Tworzenie interfejsu znacznika

Najpierw tworzymy interfejs znacznika, Auditable. Interfejsem tym oznaczamy wszystkie klasy trwałe, które powinny automatycznie trafiać do dziennika.

```
package org.hibernate.auction.model;

public interface Auditable {
    public Long getId();
}
```

Interfejs wymaga, by encyjne klasy trwałe udostępniały swój identyfikator metodą pobierającą. Potrzebujemy tej informacji do zapamiętania śladu zmian. Włączenie dziennika audytowego dla konkretnej klasy trwałe jest banalnie proste — wystarczy dołączyć interfejs do deklaracji klasy. Oto przykład dla klasy Item:

```
public class Item implements Serializable, Auditable {
    ...
}
```

Utworzenie i odwzorowanie rekordu dziennika

Tworzymy nową klasę trwałą o nazwie AuditLogRecord. Klasa reprezentuje informacje, które mają się znaleźć w tabeli bazy danych dziennika audytowego.

```
public class AuditLogRecord {

    public String message;
    public Long entityId;
    public Class entityClass;
    public Long userId;
    public Date created;

    AuditLogRecord() {}

    public AuditLogRecord(String message, Long entityId,
                          Class entityClass, Long userId) {
        this.message = message;
        this.entityId = entityId;
        this.entityClass = entityClass;
        this.userId = userId;
        this.created = new Date();
    }
}
```

Klasy tej nie należy traktować jako części modelu dziedzinowego. Z tego powodu nie trzeba zwracać uwagi na upublicznenie atrybutów. Klasa AuditLogRecord stanowi część warstwy trwałości i prawdopodobnie współdzieli ten sam pakiet z innymi klasami powiązanymi z trwałością, na przykład HibernateUtil lub własnymi typami odwzorowań.

Następnie odwzorowujemy klasę na tabelę bazy danych AUDIT_LOG.

```
<hibernate-mapping>

<class name="org.hibernate.auction.persistence.audit.AuditLogRecord"
      table="AUDIT_LOG"
      mutable="false">

    <id type="long" column="AUDIT_LOG_ID">
        <generator class="native"/>
    </id>

    <property name="message" column="MESSAGE"
              not-null="true" access="field"/>

    <property name="entityId" column="ENTITY_ID"
              not-null="true" access="field"/>

    <property name="entityClass" column="ENTITY_CLASS"
              not-null="true" access="field"/>

    <property name="userId" column="USER_ID"
              not-null="true" access="field"/>

    <property name="created" column="CREATED"
              type="java.util.Date" not-null="true"
              access="field"/>

</class>

</hibernate-mapping>
```

Klasę oznaczamy jako `mutable="false"`, ponieważ rekordy dziennika audytowego są niezmienne. Hibernate nie dopuści do aktualizacji rekordu, nawet gdy się go zmieni. Zauważ brak właściwości identyfikującej — Hibernate sam utworzy wewnętrznie odpowiedni klucz sztuczny.

Tworzenie dziennika audytowego jest niezależne od logiki biznesowej, która powoduje powstawanie zdarzeń. Choć można mieszać logikę dziennika z logiką biznesową, w wielu aplikacjach preferuje się rozdzielenie obu kwestii i tworzenie dziennika w jednym centralnym miejscu odseparowanym od logiki biznesowej. Innymi słowy, nie chcemy ręcznie tworzyć i zapisywać obiektu `AuditLogRecord` za każdym razem, gdy zmienimy obiekt `Item`.

Hibernate oferuje system rozszerzeniowy, do którego można dołączyć dowolny kod generujący dziennik (lub z innych powodów nasłuchujący zdarzeń). Rozszerzenie to nosi nazwę `Interceptor`.

Tworzenie kodu przechwytyującego

Chcielibyśmy, by metoda `logEvent()` była wywoływaną automatycznie po użyciu metody `save()`. Najlepiej wykonać to zadanie, implementując klasę z interfejsem `Interceptor`. Oto przykład:

```
public class AuditLogInterceptor implements Interceptor {

    private Session session;
```

```
private Long userId;

private Set inserts = new HashSet();
private Set updates = new HashSet();

public void setSession(Session session) {
    this.session=session;
}
public void setId(Long userId) {
    this.userId=userId;
}

public boolean onSave(Object entity, Serializable id, Object[] state,
                      String[] propertyNames, Type[] types)
                      throws CallbackException {

    if (entity instanceof Auditable)
        inserts.add(entity);

    return false;
}

public boolean onFlushDirty(Object entity, Serializable id, Object[] currentState,
                           Object[] previousState, String[] propertyNames,
                           Type[] types) throws CallbackException {

    if (entity instanceof Auditable)
        updates.add(entity);

    return false;
}

public void postFlush(Iterator iterator) throws CallbackException {
    try {
        for (Iterator it = inserts.iterator(); it.hasNext();) {
            Auditable entity = (Auditable) it.next();
            AuditLog.logEvent("create", entity, userId,
                              session.connection());
        }
        for (Iterator it = updates.iterator(); it.hasNext();) {
            Auditable entity = (Auditable) it.next();
            AuditLog.logEvent("update", entity, userId,
                              session.connection());
        }
    } catch (HibernateException ex) {
        throw new CallbackException(ex);
    } finally {
        inserts.clear();
        updates.clear();
    }
}
...
}
```

Interfejs Interceptor ma znacznie więcej metod, niż zostało przedstawionych w zaprezentowanym kodzie. Zakładamy, że zostaną zaimplementowane zgodnie z domyślną semantyką (czyli najczęściej będą zwracały wartości `false` lub `null` w zależności od wskazówek w dokumentacji interfejsu).

Przedstawiony kod elementu przechwytyującego (interceptora) ma dwa interesujące aspekty. Po pierwsze, do poprawnego działania potrzebuje odpowiednio ustawionych atrybutów `session` i `userId`. Ustawić je musi kod włączający przechwytywanie. Po drugie, za zapamiętanie zmian odpowiadają metody `onSave()` i `onFlushDirty()`, w których dodajemy lub aktualizujemy obiekty zawarte w kolekcji. Hibernate wywołuje metodę `onSave()`, gdy zapisuje encję, a metodę `onFlushDirty()`, gdy wykryje zabrudzenie obiektu. Za właściwy zapis do dziennika odpowiada metoda `postFlush()`, którą Hibernate wykonuje po synchronizacji zawartości pamięci z zawartością bazy danych. Do utworzenia zdarzenia dziennika kod używa metody statycznej `AuditLog.logEvent()` opisywanej dokładnie w dalszej części rozdziału. Zauważ, że nie możemy tworzyć dziennika w metodzie `onSave()`, ponieważ na tym etapie identyfikator obiektu niekoniecznie będzie już wypełniony. Hibernate gwarantuje poprawne ustawienie wszystkich identyfikatorów po przekazaniu obiektów do bazy danych, więc `postFlush()` okazuje się idealnym miejscem do zapisu dziennika.

Zwrócić uwagę na sposób użycia obiektu `session` — połączenie JDBC przekazanego obiektu `Session` trafia do statycznej metody `AuditLog.logEvent()`. Istnieją powody, dla których zadanie to wykonujemy w przedstawiony sposób. Sprawdźmy, jak połączyć wszystkie elementy układanki, czyli podłączyć klasę przechwytyującą do Hibernate.

Włączenie klasy przechwytyjącej

Musimy przypisać klasę implementującą `Interceptor` do obiektu `Session` tuż po rozpoczęciu sesji.

```
Interceptor interceptor = new AuditLogInterceptor();

Session session = HibernateUtil.getSessionFactory().openSession(interceptor);
Transaction tx = session.beginTransaction();

interceptor.setSession(session);
interceptor.detUserId(currentUser.getId());

session.save(newItem); // Powoduje wywołanie metody onSave() obiektu Interceptor.

tx.commit();           // Powoduje wywołanie metody postFlush() obiektu Interceptor.
session.close();
```

Zauważ, że w przykładzie nie stosujemy metody `HibernateUtil.getSession()`. Gdybyśmy tak zrobili, nie udałoby się włączyć przechwytywania. Z tego powodu sami obieramy obiekt `Session` z `SessionFactory` (to samo zadanie wykonuje metoda z `HibernateUtil`). Wbudowanie w klasę `HibernateUtil` obsługi obiektów `Interceptor` nie stanowi problemu, więc pozostawiamy to do wykonania jako ćwiczenie.

Dodaj również metodę `HibernateUtil.registerInterceptor()`, która zapamięta aktualny obiekt `Interceptor` w zmiennej lokalnowątkowej.

Powróćmy do istotnego kodu obsługi sesji z klasy przechwytyjącej i dowiedzmy się, dlaczego do metody `AuditLog.logEvent()` przekazaliśmy wartość zwróconą przez metodę `connection()` aktualnej sesji.

Sesja tymczasowa

Potrzeby użycia obiektu `Session` wewnętrz `AuditLogInterceptor` raczej nie musimy tłumaczyć. Klasa przechwytyująca tworzy i utrzymuje obiekt `AuditLogRecord`, więc niedoświadczona osoba mogłaby napisać kod metody `onSave()` w następujący sposób:

```
if (entity instanceof Auditable) {
    try {

        AuditLogRecord logRecord = new AuditLogRecord( ... );
        // ... ustaw dane dla dziennika

        session.save(logRecord);
    } catch (HibernateException ex) {
        throw new CallbackException(ex);
    }
}
```

Na pierwszy rzut oka wydaje się to poprawne — tworzymy obiekt `AuditLogRecord` i zapisujemy go, używając aktualnej sesji. Niestety, ten sposób nie działa.

Wywołanie oryginalnej sesji `Hibernate` z poziomu wywołania zwrotnego obiektu `Interceptor` nie jest poprawne. Sesja w momencie wywoływania klasy przechwytyującej znajduje się w bardzo delikatnym stanie. Ciekawa sztuczka polega na otwarciu nowej sesji tylko na potrzeby zapisu obiektu `AuditLogRecord`. Aby dodatkowo zwiększyć szybkość wykonania, możemy posłużyć się połączeniem JDBC oryginalnej sesji.

Obsługę sesji tymczasowej zawiera klasa pomocnicza `AuditLog`.

```
public class AuditLog {

    public static void logEvent(String message, Auditable entity,
                               Long userId, Connection connection)
        throws CallbackException {

        Session tempSession =
            HibernateUtil.getSessionFactory().openSession(connection);

        try {
            AuditLogRecord record = new AuditLogRecord(message,
                entity.getId(),
                entity.getClass(),
                userId);

            tempSession.save(record);
            tempSession.flush();
        } catch (Exception ex) {
```

```
        throw new CallbackException(ex);
    } finally {
        try {
            tempSession.close();
        } catch (HibernateException ex) {
            throw new CallbackException(ex);
        }
    }
}
```

Zauważ, że metoda nigdy nie zatwierdza ani nie rozpoczyna transakcji bazodanowej. Wykonuje tylko kilka dodatkowych polecen INSERT w istniejącym połączeniu JDBC wewnętrz działającej transakcji bazodanowej. Sesja tymczasowa dla niektórych operacji wykorzystująca to samo połączenie JDBC i transakcje bywa przydatna również w innych sytuacjach.

Zalecamy poeksperymentowanie i wypróbowanie innych wzorców projektowych klas przechwytyujących. Przykładowo, można tak zmodyfikować mechanizm przechwytyjący, by tworzył dzienniki dla wszystkich klas encyjnych, nie tylko tych implementujących interfejs AuditTable. Witryna Hibernate zawiera przekłady zagnieżdżonych klas przechwytyujących lub klas tworzących dziennik pełnej historii encji (włącznie z informacjami o aktualizowanej właściwości lub kolekcji).

8.4. Podsumowanie

W rozdziale skupiliśmy się na projektowaniu aplikacji i niektórych przypadkach szczególnych spotykanych w typowych zadaniach programistycznych wykorzystujących Hibernate. Najpierw prześledziliśmy projektowanie aplikacji stosującej proste serwlety. Klasa pomocnicza HibernateUtil idealnie pasuje do aplikacji warstwowych, gdyż separuje kod obsługi wyjątków oraz obiektów Session i HibernateSession od innych części aplikacji. Przestawiliśmy zalety „inteligentnych” modeli dziedzinowych przez implementację logiki biznesowej w klasie Item.

Wzorzec DAO posłużył do wykreowania fasady dla warstwy trwałości ukrywającej szczegóły działania Hibernate przez warstwami: sterującą, prezentacyjną i biznesową. Wczesniejszy przykład z prostym serwletem przeniesliśmy do trój-warstwowej architektury EJB, używając wzorców fasady sesyjnej i polecenia EJB. Dzięki obsłudze obiektów odłączonych możemy uniknąć wzorca DTO i tym samym wyeliminować wiele zbędnego kodu.

Obiekty odłączone przydają się również w momencie implementacji długiego działających transakcji aplikacyjnych. Filtr serwletu posłużył nam jako kod przechwytyujący zapewniający implementację transakcji aplikacyjnych za pomocą długiego działającej sesji Hibernate.

W drugiej części rozdziału przyjrzaliśmy się bardziej egzotycznym zagadniom dotyczącym aplikacji stosujących starsze schematy bazy danych. Przedstawiliśmy sposoby odwzorowania (naturalnych) kluczy złożonych i ich obsługi w kodzie aplikacji. Dodatkowo zajęliśmy się wykorzystaniem wyzwalaczy i własnych typów Hibernate do obsługi starszych danych.

Na końcu zaimplementowaliśmy dziennik audytowy dla trwałych encji, używając interfejsu Interceptor. Kod przechwytujący używa sztuczki z tymczasową sesją, by umożliwić zapis śledzonych zdarzeń w tabeli dziennika.

Narzędzia Hibernate

9

W rozdziale:

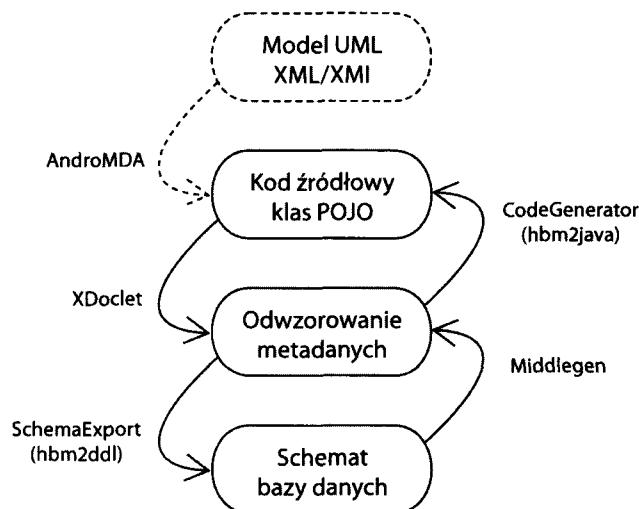
- ◆ Proces tworzenia aplikacji używających Hibernate
- ◆ Automatyczne generowanie schematu bazy danych
- ◆ Generowanie kodu POJO
- ◆ Import schematów starszych baz danych narzędziem Middlegen
- ◆ Programowanie atrybutowe z użyciem XDoclet

Dobre oprogramowanie ORM zawiera zestaw przydatnych narzędzi. Hibernate nie jest wyjątkiem, więc w tym rozdziale zajmiemy się narzędziami dostarczanymi wraz z systemem odwzorowania obiektowo-relacyjnego. Narzędzia automatycznie generują metadane odwzorowań, schematy baz danych SQL, a nawet kod źródłowy klas POJO. To, z którego z narzędzi się skorzysta, zależy od konkretnego procesu tworzenia aplikacji.

9.1. Procesy tworzenia aplikacji

W niektórych projektach kreowaniem modelu dziedzinowego kierują programiści analizujący pod kątem obiektowości model biznesowy. W innych projektach model silnie zależy od istniejącego modelu relacyjnego — starszej bazy danych lub całkowicie nowego schematu bazy danych wykonanego przez profesjonalistę od modelowania danych.

Ponieważ różne projekty zaczynają się z różnych miejsc, musimy rozważyć różne scenariusze i różne narzędzia stosowane w każdej z sytuacji. Wykaz narzędzi wraz z ich danymi źródłowymi i wynikami przedstawia rysunek 9.1. Warto powracać do diagramu w trakcie czytania rozdziału.



Rysunek 9.1.
Dane wejściowe i wyjściowe narzędzi używanych w trakcie tworzenia aplikacji z Hibernate

Uwaga

AndroMDA, narzędzia generującego kod źródłowy klas POJO na podstawie plików diagramów UML, najczęściej nie traktuje się jako typowego narzędzia dla Hibernate. Z tego powodu nie opisujemy go w tym rozdziale. Więcej informacji na temat modułów obsługi Hibernate w AndroMDA znajduje się na stronie społeczności użytkowników Hibernate.

Zanim przyjrzymy się dokładnie poszczególnym narzędziom mogąącym mieć zastosowanie w Hibernate, przedstawmy każdy ze scenariuszy do tworzenia oprogramowania i najlepsze dla niego narzędzia.

9.1.1. Podejście z góry na dół

Tworzenie oprogramowania z góry na dół zaczyna się od istniejącego modelu dziedzinowego w Javie (najlepiej zaimplementowanego jako obiekty POJO lub JavaBeans) z pełną swobodą co do schematu bazy danych. Należy wykonać dokument odwzorowania — ręcznie, używając edytora tekstu (zalecamy narzędzie IDE z automatycznymi podpowiedziami dla XML), lub automatycznie za pomocą XDoclet. Następnie wystarczy użyć narzędzia Hibernate hbm2ddl do wygenerowania schematu bazy danych. W przypadku braku istniejącego modelu danych jest to najwygodniejszy styl tworzenia oprogramowania dla większości programistów Javy. Można nawet skorzystać z narzędzi Hibernate, by automatycznie aktualizować schemat bazy danych po modyfikacjach modelu dziedzinowego.

9.1.2. Podejście z dołu do góry

W podejściu z dołu do góry tworzenie oprogramowania rozpoczyna się od istniejącego schematu bazy danych i modelu danych. W tym przypadku najwygodniej jest najpierw skorzystać z narzędzia Middlegen do wygenerowania plików odwzorowań Hibernate na podstawie istniejącej bazy danych, a następnie uruchomić narzędzie hbm2java do wygenerowania szkieletu klas trwałych POJO. Najczęściej wygenerowane pliki i klasy trzeba poprawiać i rozszerzać ręcznie, ponieważ nie wszystkie asocjacje klas i szczegóły metadanych można określić automatycznie na podstawie schematu bazy danych.

9.1.3. Podejście od środka

Ani klasy Javy (bez deklaracji XDoclet), ani schemat DDL nie zawierają wystarczających informacji, by móc określić rozwiązanie ORM. Aby wygenerować dokument odwzorowania Hibernate (bez tworzenia go ręcznie), potrzebne będą dodatkowe informacje od użytkownika. Gdy stosuje się XDoclet, informacje te najlepiej zawsze w dodatkowych atrybutach osadzonych w kodzie źródłowym. W Middlegen określa się je, używając graficznego interfejsu użytkownika.

Z drugiej strony, dokument odwzorowania zawiera wystarczająco dużo informacji, by na jego podstawie wykonać pełny schemat bazy danych i wygenerować działające klasy JavaBeans. Co więcej, dokument odwzorowania nie zawiera zbędnych danych. Podejście od środka wydaje się najlepsze, gdy samemu ręcznie tworzy się pliki odwzorowań. Wystarczy wtedy uruchomić narzędzia hbm2ddl (do wygenerowania schematu bazy danych) i hbm2java (do wygenerowania klas Javy).

9.1.4. Spotkanie w środku

Najtrudniejsza sytuacja występuje wówczas, gdy mamy istniejące klasy Javy i istniejący relacyjny schemat bazy danych. Wtedy automatyczne narzędzia na wiele się nie zdadzą. Nie można odwzorować dowolnego modelu dziedzinowego klas Javy na konkretny schemat, więc najczęściej trzeba zmodyfikować klasy Javy, schemat bazy lub oba te elementy. Dokument odwzorowania na ogólnie pisze się ręcznie (lub

z użyciem XDoclet). To wyjątkowo nieprzyjemny scenariusz tworzenia oprogramowania — na szczęście występuje rzadko.

9.1.5. Ścieżka przejścia

Ścieżka przejścia wykorzystuje sytuację, w której dowolny z trzech rodzajów zapisu (klasy, dokumenty odwzorowania lub schemat bazy danych) powinien wystarczyć do odbudowania dwóch pozostałych. Każde z narzędzi powinno zapewniać pełne i równoważne przejście między poszczególnymi postaciami. Wiemy, że tak nie jest. Jako całkowite minimum musimy do kodu Javy dołączyć adnotacje XDoclet. Co gorsza, pełne odzyskanie modelu dziedzinowego Javy, a nawet plików odwzorowania ze schematu bazy danych, nie jest możliwe.

Mimo to zespół Hibernate stara się osiągnąć nieco mniej ambitny cel dotyczący narzędzi Hibernate. Przypuśćmy, że rozpoczynamy tworzenie od istniejącego schematu bazy danych. Kolejne kroki powinny reprodukować ten schemat w sposób idealny przy minimalnych poprawkach po stronie użytkownika.

1. Użycie Middlegen do utworzenia dokumentu odwzorowania.
2. Użycie hbm2java do wygenerowania klas Javy z adnotacjami.
3. Użycie XDoclet do ponownego wygenerowania dokumentu odwzorowania.
4. Użycie hbm2ddl do wygenerowania DDL.

W chwili pisania niniejszej książki podejście to nie działa idealnie i wymaga dodatkowej pracy głównej dlatego, że stosuje wiele różnych narzędzi i sposobów konwersji.

Przyjrzymy się bliżej poszczególnym narzędziom wspomnianym na początku rozdziału. Zaczniemy od hbm2ddl, które automatycznie generuje polecenia DDL SQL na podstawie istniejącego dokumentu odwzorowania Hibernate. Zakładamy wcześniejsze wykonanie niektórych klas trwałych jako POJO i ich poprawne odwzorowanie na potrzeby Hibernate. Szukamy jedynie uproszczenia pozwalającej uniknąć ręcznego tworzenia schematu bazy danych.

9.2. Automatyczne generowanie schematu bazy danych

Schematy bazy danych SQL zapisuje się specjalnym podjęzyku nazwanym DDL SQL i składającym się przede wszystkim z dobrze znanych polecień CREATE i ALTER.

Narzędzie używane w progresie generowania nosi nazwę hbm2ddl. Jego klasa znajduje się w `net.sf.hibernate.tool.hbm2ddl.SchemaExport`, więc często używa się również nazwy SchemaExport.

Uwaga	Pakiet rozszerzeń Hibernate — zauważ, że hbm2ddl znajduje się w głównej dystrybucji Hibernate, a nie jak inne narzędzia w HibernateExtensions. Zespół Hibernate zdecydował, że hbm2ddl jest znacznie bliżej głównego
--------------	--

sposobu działania ORM niż inne narzędzia i z tego powodu powinien znać się w głównej dystrybucji. Co więcej, hbm2ddl można wywołać z poziomu aplikacji w trakcie jej działania, by dynamicznie wygenerować schemat bazy danych. Jest to szczególnie przydatna opcja, gdy chce się tworzyć nowy schemat przy każdym ponownym starcie zmodyfikowanej aplikacji.

Wymaganiami wstępymi do automatycznego wygenerowania DDL SQL jest zawsze plik odwzorowania Hibernate w postaci dokumentu XML. Zakładamy projektowanie i implementację klas POJO oraz metadanych odwzorowania bez zwracania szczególnej uwagi na szczegóły istotne z punktu widzenia baz danych (na przykład nazw tabel i kolumn).

Pliki odwzorowań mogą zawierać specjalne elementy i atrybuty. Większość z nich dotyczy nietypowego generowania schematu bazy danych. Hibernate stara się używać sensownych wartości domyślnych, gdy nie określi się własnych nazw i strategii. Pamiętaj, że profesjonalny administrator systemu bazodanowego może nie zaakceptować domyślnego schematu i zażądać modyfikacji. Wartości domyślne najczęściej okazują się wystarczające w środowisku prototypowym lub rozwojowym.

9.2.1. Przygotowanie metadanych odwzorowania

W niniejszym przykładzie oznaczyliśmy elementy i atrybuty odwzorowania klasy Item wykorzystywane przez narzędzie hbm2ddl do generowania schematu. Te opcjonalne definicje dobrze współpracują z innymi elementami odwzorowania, co przedstawia listing 9.1.

Listing 9.1. Dodatkowe elementy odwzorowania klasy Item dla SchemaExport

```
<class name="Item" table="ITEM">

    <id name="id" type="string">
        <column name="ITEM_ID" sql-type="char(32)"/> ❶
        <generator class="uuid.hex"/>
    </id>

    <property name="name" type="string">
        <column name="NAME"
               length="255"
               not-null="true"
               index="IDX_ITEMNAME"/> ❷
    </property>

    <property name="description"
              type="string"
              column="DESCRIPTION"
              length="4000"/> ❸

    <property name="initialPrice"
```

```

        type="customtype.MonetaryAmount">
    <column name="INITIAL_PRICE" check="INITIAL_PRICE > 0"/> ❸
    <column name="INITIAL_PRICE_CURRENCY"/>
</property>

<set name="categories" table="CATEGORY_ITEM" cascade="none">
    <key>
        <column name="ITEM_ID" sql-type="char(32)"/> ❹
    </key>
    <many-to-many class="Category">
        <column name="ITEM_ID" sql-type="char(32)"/>
    </many-to-many>
</set>
...
</class>
```

- ❶ Narzędzie hbm2ddl automatycznie generuje kolumnę typu VARCHAR, jeśli właściwość (nawet identyfikująca) jest typu odwzorowania string. Generator identyfikatorów uuid.hex zawsze generuje tekst o długości 32 znaków, więc warto użyć typu CHAR o stałej długości. Deklaracja wymaga zagnieźdzonego elementu <column>, ponieważ nie ma atrybutu określającego w elemencie <id> typ danych SQL.
- ❷ Atrybuty column, not-null i length są dostępne w elemencie <property>. Ponieważ jednak chcieliśmy utworzyć dodatkowy indeks, musielibyśmy użyć zagnieźdzonego elementu <column>. Indeks przyspieszy wyszukiwanie przedmiotów na podstawie nazwy. Jeśli ta sama nazwa indeksu pojawi się w odwzorowaniach innych kolumn, nowy indeks będzie obejmował wiele kolumn bazy danych. Podana w atrybucie nazwa indeksu stanie się również nazwą indeksu w samej bazie danych.
- ❸ Dla pola opisu stosujemy pobieranie leniwe, wykorzystując atrybuty elementu <property> zamiast osobnego elementu <column>. Kolumna DESCRIPTION zostanie wygenerowana jako typ VARCHAR(4000).
- ❹ Typ użytkownika MonetaryAmount wymaga dwóch kolumn bazy danych. Musimy użyć elementu <column>. Atrybut check powoduje utworzenie ograniczenia sprawdzającego, czy wartość kolumny odpowiada wskazanemu wyrażeniu SQL. Atrybut check istnieje także dla elementu <class>, by możliwe było tworzenie ograniczeń obejmujących wiele kolumn.
- ❺ Element <column> służy również do deklarowania pól kluczów obcych w odwzorowaniu asocjacji. Gdyby nie dodatkowy element, kolumna tabeli łączącej CATEGORY_ITEM byłaby typu VARCHAR(32) zamiast CHAR(32).

Wszystkie atrybuty istotne przy generowaniu schematu bazy danych zebraliśmy w tabeli 9.1. Niektóre z nich nie pojawiły się w przykładowym odwzorowaniu klasy Item.

Tabela 9.1. Atrybuty odwzorowania XML używane przez narzędzie hbm2ddl

Atrybut	Wartość	Opis
column	string	Dostępny w większości elementów odwzorowania. Określa nazwę kolumny tabeli. Jego pominięcie przy braku zagnieżdżonych elementów <column> powoduje użycie nazwy właściwości klasy Java przez hbm2ddl i rdzeń Hibernate. Zachowanie to można zmienić, implementując własną strategię nazewnictwa NamingStrategy (patrz rozdział 3.).
not-null	true lub false	Włącza generowanie ograniczenia NOT NULL. Dostępny w większości elementów odwzorowania oraz w elemencie <column>.
unique	true lub false	Wymusza wygenerowanie ograniczenia UNIQUE dla pojedynczej kolumny. Dostępny dla wielu elementów odwzorowania.
length	integer	Określa długość typu danych. Na przykład atrybut length="4000" dla typu string utworzy kolumnę typu VARCHAR(4000). Atrybut służy również do określania precyzji typów liczbowych.
index	string	Definiuje nazwę indeksu bazy danych (może być współdzielona przez wiele elementów). Nic nie stoi na przeszkodzie, by tworzyć indeks jednokolumnowy. Dostępny jedynie w elemencie <column>.
unique-key	string	Włącza unikatowość dotyczącą wielu kolumn bazy danych. Wszystkie elementy używające atrybutu muszą zawierać tę samą nazwę ograniczenia. Dostępny jedynie w elemencie <column>.
sql-type	string	Wyłącza automatyczne wykrywanie typu danych SQL przez hbm2ddl, co umożliwia użycie typów specyficznych dla danej bazy danych. Z drugiej strony zmniejsza przenośność bazodanową. Narzędzie domyślnie używa typu VARCHAR lub VARCHAR2 (Oracle), ale gdy wystąpi atrybut, zawsze z niego skorzysta. Dostępny jedynie w elemencie <column>.
foreign-key	string	Nadaje nazwę ograniczeniu klucza obcego. Dostępny w elementach <many-to-one>, <one-to-one>, <key> i <many-to-many>. Pamiętaj, że strona z inverse="true" nie będzie rozważana przy nadawaniu nazwy ograniczeniu klucza obcego (Hibernate zawsze używa nazwy podanej w elemencie bez tego atrybutu). System automatycznie wygeneruje unikatową nazwę, jeżeli nie zostanie podana.

Po przejrzeniu (prawdopodobnie razem z administratorem bazy danych) plików odwzorowań i dodaniu odpowiednich atrybutów dotyczących schematu możemy przystąpić do utworzenia wynikowego schematu bazy danych.

9.2.2. Utworzenie schematu

Narzędzie hbm2ddl można wywołać z poziomu wiersza poleceń:

```
java -cp ścieżki_klas net.sf.hibernate.tool.hbm2ddl.SchemaExport opcje
pliki_odwzorowań
```

Hibernate, jego biblioteki pomocnicze oraz skompilowane klasy trwałe muszą znaleźć się w ścieżce wyszukiwania klas.

Tabela 9.2 przedstawia opcje narzędzia hbm2ddl.

Tabela 9.2. Opcje konfiguracyjne wywołania hbm2ddl z poziomu wiersza poleceń

Opcja	Opis
--quiet	Nie wyświetla skryptu na standardowym wyjściu.
--drop	Jedynie tworzy kod usuwający tabele.
--text	Nie eksportuje DDL do bazy danych, ale na standardowe wyjście.
--output=nazwa_pliku	Przekazuje skrypt DDL do wskazanego pliku.
--config=nazwa_pliku	Odczytuje konfigurację bazy danych ze wskazanego pliku konfiguracyjnego XML.
--properties=nazwa_pliku	Odczytuje właściwości bazy danych z podanego pliku.
--format	Ładnie formatuje generowany kod SQL, zamiast umieszczać każde polecenie w jednym wierszu.
--delimeter=x	Ustawia znak końca wiersza dla skryptu (najczęściej jest to średnik). Domyslnie nie stosuje znaków końca wiersza. Opcję stosuje się tylko w połączeniu z opcją --text; bezpośredni zapis do bazy danych nie wymaga ogranicznika.

Jak nie trudno się domyślić po przejrzeniu dostępnych opcji, kod DDL można od razu wykonać. Narzędzie potrzebuje w takiej sytuacji ustawień połączenia z bazą danych z pliku właściwości (lub dokumentu XML). Kod generowany przez hbm2ddl zawsze usuwa wszystkie tabele i tworzy je od nowa. Okazuje się to szczególnie przydatne przy projektowaniu aplikacji. W pliku konfiguracyjnym trzeba wskazać dialekt SQL, bo polecenia DDL silnie zależą od rodzaju stosowanej bazy danych.

Jeden z powodów umieszczenia narzędzia hbm2ddl w podstawowym pakiecie Hibernate wynika z możliwości jego uruchomienia wewnątrz aplikacji, co przedstawia poniższy kod:

```
Configuration cfg = new Configuration();
SchemaExport schemaExport = new SchemaExport(cfg);
schemaExport.create(false, true);
```

Nowy obiekt SchemaExport powstaje na podstawie obiektu Configuration. Istniejące pliku *hibernate.cfg.xml* spowoduje wczytanie zawartych w nim danych dotyczących połączenia bazodanowego i dialekta do obiektu Configuration, który trafia do konstruktora SchemaExport. Metoda create(false, true) rozpoczęta proces generowania kodu DDL bez wyświetlania polecień SQL na standardowym wyjściu (false), ale z natychmiastowym wykonywaniem w bazie danych (true). Więcej informacji na ten temat znajduje się w dokumentacji SchemaExport. Wszystkie opcje wiersza poleceń są również dostępne programowo — nic nie stoi na przeszkodzie, by ustawić je dla obiektu SchemaExport.

Narzędzie hbm2ddl można także sterować globalnie za pomocą właściwości konfiguracyjnych Hibernate — na przykład zawartych w pliku *hibernate.properties*:

```
hibernate.hbm2ddl.auto create-drop
```

Ustawienie hibernate.hbm2ddl.auto na create-drop powoduje usunięcie i ponowne utworzenie schematu bazy danych w momencie wywołania metody buildSessionFactory(), czyli najczęściej w momencie uruchamiania aplikacji Hibernate. Wywołanie metody close() klasy SessionFactory ponownie usuwa schemat. Wartość opcji równa create powoduje tylko usunięcie i ponowne utworzenie schematu bazy danych w momencie tworzenia obiektu SessionFactory. Istnieje również ustawienie update powodujące automatyczną aktualizację schematu po zmianie wytycznych. Zagadnienie to opiszymy dokładniej w następnym podrozdziale wraz z narzędziem SchemaUpdate.

Trzy przedstawione metody uruchamiania rzadko wystarczają. Wykonanie hbm2ddl z poziomu wiersza poleceń nie jest typowym zadaniem programistycznym, natomiast uruchomienie wewnątrz aplikacji nie zawsze udaje się zastosować. Jeśli, podobnie jak większość programistów Javy, stosujesz Ant do budowania projektu, wystarczy dodać zadanie Ant automatycznie generujące schemat.

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="net.sf.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        config="${basedir}/etc/hibernate_export.cfg.xml"
        quiet="no"
        text="no" drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target>
```

Przykład wykorzystuje definicję zadania Ant. Zadanie może zawierać różne opcje. W przedstawionym przykładzie kod DDL trafia do pliku *schema-export.sql*. Polecenia oddziela znak średnika. Dodatkowo generowanie dotyczy wszystkich plików odwzorowań znalezionych w folderze *src*, które zostają automatycznie przekazane do bazy danych (*text="no"*). Dane dotyczące konfiguracji bazy danych i dialekta znajdują się odczytane z pliku *hibernate_export.cfg.xml* z podfolderu *etc*.

9.2.3. Aktualizacja schematu bazy danych

Po wdrożeniu aplikacji niełatwo zmienić stosowany przez nią schemat bazy danych. Podobna sytuacja występuje nawet w trakcie tworzenia aplikacji, kiedy to dane testowe powinny pozostać nienaruszone. Narzędzie hbm2ddl dopuszcza jedynie usunięcie struktury bazy danych i ponowne jej utworzenie, po którym musiałaby nastąpić czasochłonny import danych.

Hibernate zawiera wbudowane narzędzie przekształcania schematu bazy danych, SchemaUpdate, które pozwala aktualnić istniejących schemat bazy danych SQL. Usuwa stare tabele, kolumny i ograniczenia. Używa metadanych JDBC, by

utworzyć nowe tabele i ograniczenia na podstawie już istniejącej bazy danych. Pamiętaj, że SchemaUpdate silnie zależy od jakości metadanych uzyskiwanych ze sterownika JDBC, więc z niektórymi sterownikami i bazami danych może nie działać najlepiej (w momencie pisania tego tekstu SchemaUpdate nie był wystarczająco stabilny, by stosować go w sytuacjach innych niż eksperymenty).

Klasę SchemaUpdate można wywołać z poziomu aplikacji, używając następującego kodu:

```
Configuration cfg = new Configuration();
SchemaUpdate schemaUpdate = new SchemaUpdate(cfg);
schemaUpdate.execute(false, true);
```

Obiekt SchemaUpdate powstaje na podstawie obiektu Configuration. Wymaga tych samych ustawień (połączenia bazodanowego i dialekta) co hbm2ddl. Przykład aktualizuje jedynie bazę danych i nie wyświetla żadnych komunikatów na standardowym wyjściu (wartość false).

Klasę SchemaUpdate można także wykorzystać w skrypcie Ant.

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="net.sf.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target>
```

Zadanie Ant aktualizuje schemat bazy danych dla wszystkich plików odwzorowań odnalezionych w folderze src i dodatkowo wyświetla polecenia SQL na standardowym wyjściu. Ustawienia połączenia z bazą danych odczytuje z pliku hibernate.properties znajdującego się w ścieżce wyszukiwania klas.

Narzędzie hbm2ddl jest bardzo popularne w projektach Hibernate stosujących proces tworzenia z góry na dół. Używa metadanych z plików odwzorowań Hibernate, by wygenerować schemat bazy danych spełniający większość wymagań administratorów baz danych. Nie jest to jedyne narzędzie Hibernate używające metadanych odwzorowań. W procesach z dołu do góry lub od środka można wykorzystać dane odwzorowań do generowania klas trwałych.

9.3. Generowanie kodu klas POJO

Narzędzie Hibrnate do automatycznego generowania klas trwałych nosi nazwę hbm2java, a jego główna klasa nazwę net.sf.hibernate.tool.hbm2java.CodeGenerator, więc niejednokrotnie narzędzie nazywa się również CodeGenerator. Znajduje się w opcjonalnej dystrybucji HibernateExtensions.

Narzędzie hbm2java stosuje się do:

- ◆ generowania kodu źródłowego klas POJO na podstawie plików odwzorowań Hibernate przy podejściu od środka,
- ◆ generowania kodu źródłowego klas POJO na podstawie plików odwzorowań Hibernate uzyskanych automatycznie narzędziem Middlegen z istniejącego schematu bazy danych.

Narzędzie można dostosować do własnych potrzeb, korzystając na przykład z dodatkowych metadanych (jak to miało miejsce w narzędziu hbm2ddl). Dokumentacja narzędzi Hibernate wyjaśnia podstawowe użycie narzędzia oraz opisuje wszystkie możliwe parametry konfiguracyjne. Zamiast więc powtarzać dokumentację, przedstawimy praktyczny przykład.

9.3.1. Wprowadzenie metaatributów

Przypuśćmy, iż dysponujemy plikiem odwzorowania Hibernate dla klasy User i chcemy wygenerować źródła dla klasy zgodne z zasadami POJO. Zgodnie z rozdziałem 3. klasy POJO implementują interfejs Serializable, mają konstruktor bezparametrowy, metody ustawiające i pobierające dla wszystkich właściwości i stosują podejście hermetyzujące implementację.

Warto w miarę możliwości wykorzystywać wartości domyślne Hibernate i tym samym stosować jak najmniej metadanych. Niektóre wartości domyślne Hibernate odgaduje, używając mechanizmu refleksji na podstawie istniejących klas. Niestety, po użyciu narzędzia hbm2java klasy te jeszcze nie istnieją, więc musimy wprowadzić więcej informacji niż w standardowym pliku odwzorowania.

Pierwszy krok polega na uzupełnieniu odwzorowania o dodatkowe dane, by narzędzie hbm2java zadziałało bezbłędnie. Jeśli na przykład odwzorowanie właściwości username klasy User ma postać:

```
<property name="username"/>
```

trzeba je uzupełnić jawnym wskazaniem typu Hibernate lub typu klasy Java:

```
<property name="username" type="string"/>
```

Po uzupełnieniu odwzorowania informacjami o typie przechodzimy do innych ustawień usprawniających proces generowania kodu.

Domyślnie hbm2java tworzy prostą klasę trwałą POJO. Klasa implementuje interfejs Serializable, ma wymagany konstruktor i metody dostępowe, a także metody `toString()`, `equals()` i `hashCode()` z zalecanym domyślnym znaczeniem. Wszystkie właściwości domyślnie mają widoczność prywatną. Możemy ją zmienić, stosując elementy i atrybuty `<meta>`.

Jednym z pierwszych usprawnień będzie bardziej restrykcyjna widoczność właściwości klasy User. Domyślnie wszystkie metody dostępowe mają widoczność publiczną. Jeżeli klasa byłaby niezmienna, metody ustawiające nie powinny stanowić części interfejsu publicznego. Publicznymi powinny pozostać tylko metody pobierające właściwości. Zamiast modyfikować odwzorowania wszystkich

właściwości elementem <meta>, zadeklarujmy dodatkowy element na poziomie klasy, co spowoduje użycie tych samych ustawień dla wszystkich właściwości klasy.

```
<class name="User" table="USER">
    <meta attribute="scope-set">private</meta>
    ...
</class>
```

Atrybut scope-set definiuje widoczność metod ustalających właściwości. Narzędzie hbm2java obsługuje również metaatrybuty na wyższym poziomie, czyli w specjalnym pliku konfiguracyjnym narzędzia, o którym więcej powiemy w dalszej części podrzędu. Globalny metaatrybut wpływa na generowanie kodu źródłowego wszystkich klas. Szczegółowe metaatrybuty można stosować dla konkretnych właściwości, kolekcji i komponentów.

Jednym (choć niewielkim) ulepszeniem byłoby dołączanie wartości właściwości username do wyniku generowanego przez metodę `toString()`. Domyslnie metoda zwraca jedynie wartość identyfikującą obiekt. Nazwa użytkownika to dobra wartość dla wizualnej kontrolki stosowana w dzienniku aplikacji. Z tego powodu wzbogaćmy odwzorowanie klasy User o dołączanie nazwy użytkownika do wyniku metody `toString()`.

```
<property name="username" type="string">
    <meta attribute="use-in-tostring">true</meta>
</property>
```

Wygenerowany kod metody `toString()` z pliku *User.java* będzie miał postać:

```
public String toString() {
    return new ToStringBuilder(this)
        .append("id", getId())
        .append("username", getUsername())
        .toString();
}
```

Zauważ użycie przez hbm2java klasy narzędziowej `ToStringBuilder` pochodzącej z projektu *open source* commons-lang. Musisz dołączyć tę bibliotekę narzędziową do projektu, jeśli wygenerowany kod ma się poprawnie skompilować bez modyfikacji.

Wspomnialiśmy wcześniej o dziedziczeniu metaatrybutów. Zadeklarowanie `use-in-tostring` na poziomie klasy spowoduje dołączenie wartości wszystkich właściwości do wyniku zwracanego przez metodę `toString()`. Mechanizm dziedziczenia działa dla wszystkich metaatrybutów. Co więcej, można go selektywnie wyłączać:

```
<meta attribute="scope-class" inherit="false">public abstract</meta>
```

Ustawienie `inherit` na `false` w `scope-class` spowoduje utworzenie aktualnej klasy jako publicznej i abstrakcyjnej, ale ustawienie to nie przeniesie się na podklasy.

W momencie pisania tego tekstu hbm2java obsługuje 21 metaatrybutów zmieniających sposób generowania wynikowego kodu. Większość z nich dotyczy widoczności, implementacji interfejsu, rozszerzania klas i predefiniowanych komentarzy

Javadoc. Dwa metaatrybuty są szczególnie interesujące, gdyż sterują automatycznym generowaniem **metod odnajdujących**.

9.3.2. Metody odnajdujące

Metoda **odnajdująca** to metoda statyczna służąca kodowi aplikacji do pobrania obiektów z bazy danych. Stanowi część **klasy odnajdującej**, interfejsu klasy traktowanego jako część publicznie widocznego interfejsu dotyczącego warstwy trwałości.

Pełna warstwa trwałości wymaga wielu rodzajów interfejsów do zarządzania obiektami, na przykład pełnego interfejsu DAO opisanego w rozdziale 8. Automatycznie generowany kod metody odnajdującej może stanowić szkielet tej implementacji.

Narzędzie hbm2java generuje metody odnajdujące dla pojedynczych właściwości. Dodajmy metaatrybut `finder` do deklaracji odwzorowania:

```
<property name="username" type="string">
    <meta attribute="use-in-tostring">true</meta>
    <meta attribute="finder">findByUsername</meta>
</property>
```

Atrybut `finder` definiuje nazwę metody odnajdującej (`findByUsername`). Tworzona metoda statyczna ma postać:

```
public static List findByUsername(Session session, string username)
    throws SQLException, HibernateException {

    List finds = session.find("from User as user where user.username=?",
        username, Hibernate.STRING);

    return finds;
}
```

Metoda jako argument przyjmuje nazwę użytkownika i zwraca listę (obiekt `List`) wszystkich użytkowników o podanej nazwie. Klasa podobna do metody nosi nazwę `UserFinder`.

```
public class UserFinder implements Serializable {

    public static List findAll(Session session)
        throws SQLException, HibernateException {
        List finds = session.find("from User");
        return finds;
    }
}
```

Zauważ, że wygenerowana klasa odnajdywania zawiera przynajmniej jedną metodę, `findAll()`, zwracającą wszystkie obiekty danej klasy.

Zwróć również uwagę na sposób stosowania przez metody obiektu `Session`. Musi zostać przekazany do każdej z nich jako argument. Nie jest to wygodne, szczególnie w aplikacji stosującej dla sesji mechanizm `ThreadLocal` (patrz rozdział 8.).

Do określenia statycznej metody pomocniczej służącej do pobierania sesji zamiast atrybutu session zastosujmy metaatribut session-method na poziomie odwzorowania klasy:

```
<class name="User" table="USER">
    <meta attribute="session-method">
        HibernateUtil.getSession();
    </meta>
    ...
</class>
```

Wygenerowana metoda odnajdująca wykorzysta wskazane wywołanie od pobrania obiektu Session:

```
public static List findAll()
    throws SQLException, HibernateException {
    Session session = HibernateUtil.getSession();
    List finds = session.find("from User");
    return finds;
}
```

Polecamy to rozwiązanie zamiast żmudnego podawania za każdym razem argumentu w postaci obiektu sesji. Więcej informacji na temat sesji lokalnowątkowej i klasy `HibernateUtil` znajduje się w rozdziale 8.

Dodatkowe możliwości sterowania generowaniem prostych klas trwałych POJO i klas odnajdywania oferuje plik konfiguracyjny narzędzia `hbm2java`.

9.3.3. Konfiguracja `hbm2java`

Bez pliku konfiguracyjnego narzędzie `hbm2java` do generowania kodu stosuje wyłącznie metaatributy z pliku odwzorowania oraz klasę `BasicRenderer`. Domyślna klasa generująca nie tworzy klas odnajdywania, a jedynie klasę trwałe POJO. Musimy więc dodać do konfiguracji generator `FinderRender`.

```
<codegen>
    <meta attribute="implements">
        org.hibernate.auction.model.Auditable
    </meta>

    <generate renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>

    <generate renderer="net.sf.hibernate.tool.hbm2java.FinderRenderer"
        package="org.hibernate.auction.finder"
        suffix="Finder"/>

</codegen>
```

Konfiguracja zawiera dodatkowo globalny metaatribut, który dotyczy wszystkich klas we wszystkich plikach odwzorowań. Dla klas trwałych POJO używamy `BasicRenderer`. Generator `FinderRender` korzysta z dwóch ustawień, `package` i `suffix`, dla wygenerowanych klas. Pełna nazwa klasy odnajdywania dla klasy `User` będzie miała postać: `org.hibernate.auction.finder.UserFinder`.

Jedną z cech hbm2java jest możliwość wykorzystania systemu szablonów Velocity. Generatory BasicRenderer i FinderRenderer stosują na stałe zakodowane szablony klasy natomiast generator VelocityRenderer dostosuje się w dowolny sposób, przekazując własne szablony. Oto przykład zastąpienia nim dwóch pozostały generatorów:

```
<codegen>
    <generate renderer="net.sf.hibernate.tool.hbm2java.VelocityRenderer">
        <param name="template">pojo.vm</param>
    </generate>
</codegen>
```

Generator używa parametru template do określenia szablonu, na podstawie którego mają powstać pliki klas. Szablon musi być napisany w języku Velocity i dostępny w ścieżce wyszukiwania klas. Narzędzie hbm2java zawiera domyślna wersję szablonu pojo.vm, z której warto skorzystać, kreując własne szablony generowania klas. Generowanie kodu na podstawie szablonów Velocity to nie jedyna możliwość. Niejednokrotnie lepiej wykonać własną klasę generatora. Będzie ona miała zdecydowanie większą elastyczność od szablonów Velocity, które nie zawsze nadają się do generowania kodu. Przy jej wykonaniu warto wzorować się na kodzie źródłowym klas BasicRenderer lub FinderRenderer.

Narzędzie hbm2java uruchamiamy z poziomu wiersza poleceń lub jako zadanie Ant w procesie budowania aplikacji.

9.3.4. Uruchamianie narzędzia hbm2java

Uruchomienie hbm2java z poziomu wiersza poleceń nie jest trudne:

```
java -cp ścieżka_klas net.sf.hibernate.tool.hbm2java.CodeGenerator
opcje pliki_odwzorowań
```

Narzędzie obsługuje dwie opcje: output ustawia folder dla generowanego kodu, natomiast config określa plik konfiguracyjny. Należy jawnie wskazać wszystkie pliki odwzorowań mające uczestniczyć w generowaniu.

W większości projektów wygodniej zastosować zadanie Ant. Oto jego przykład:

```
<target name="codegen">
    <taskdef name="hbm2java"
        classname="net.sf.hibernate.tool.hbm2java.hbm2JavaTask"
        classpathref="class.path"/>

    <hbm2java config="codegen.cfg.xml"
        output="generated/src/"
        <fileset dir="mappings">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </hbm2java>

</target>
```

Wygenerowane pliki źródłowe kodu Javy trafiają do folderu generated/src. Narzędzie używa pliku codegen.cfg.xml z aktualnego katalogu do własnej konfiguracji

i odczyta wszystkie pliki odwzorowania Hibernate z folderu *mappings* (wraz z podfolderami).

Pamiętaj, by ścieżka wyszukiwania zadania zawierała nie tylko archiwum *hibernate-tools.jar* z dystrybucji HibernateExtensions, ale również podstawowe pliki Hibernate i inne wymagane biblioteki (na przykład Velocity).

Narzędzie hbm2java potrafi znacząco przyspieszyć proces tworzenia aplikacji, szczególnie jeśli ma się wiele istniejących tabel bazy danych i dla wszystkich automatycznie wygenerowało się metadane odwzorowania Hibernate. Generowaniem tych danych na podstawie istniejącego schematu bazy danych zajmuje się narzędzie Middlegen.

9.4. Istniejące schematy i Middlegen

Wielu programistów używa Hibernate w projektach, które wykorzystują starsze schematy baz danych i zawarte w nich dane. Rzadko można pozwolić sobie na zmianę schematu w sposób ułatwiający integrację z Hibernate. Bazy danych SQL tradycyjnie mają problemy z ewolucją schematu bazy; niektóre sprawiają problemy nawet przy zmianie nazw kolumn tabeli.

Jeśli jedyna możliwość to istniejący schemat bazy danych, warto spróbować wygenerować na jego podstawie pliki odwzorowań Hibernate. Oszczędność czasu jest tym większa, im więcej tabel zawiera schemat. Co więcej, aplikacja może zacząć działać szybciej i zawierać najbardziej aktualne metadane schematu. Narzędzie Middlegen generuje odwzorowanie Hibernate na podstawie istniejącego schematu bazy danych. Po wygenerowaniu pliki zawsze można poprawić ręcznie.

Middlegen nie ogranicza się tylko do obsługi Hibernate. Dzięki modułom dodatkowym obsługuje generowanie kodu ziarenek encyjnych EJB, akcji Struts, a nawet kodu JSP. W podrozdziale skupimy się na module obsługującym Hibernate. Narzędzie oferuje interfejs graficzny ułatwiający dostosowanie danych z tabel do własnych potrzeb i zmianę domyślnych opcji generowania.

9.4.1. Uruchomienie Middlegen

Zalecaną sposobem uruchomienia Middlegen wymaga utworzenia zadania Ant na podstawie klasy *middlegen.MiddlegenTask*. Zdanie umieszcza się w pliku *build.xml* po skopiowaniu biblioteki Middlegen i modułu Hibernate do ścieżki wyszukiwania klas (pamiętaj również o sterowniku JDBC!):

```
<taskdef name="middlegen"
        classname="middlegen.MiddlegenTask"
        classpathref="class.path"/>
```

Zadanie *middlegen* umieść w dowolnym celu budowania aplikacji i włącz wyświetlenie interfejsu graficznego Middlegen:

```
<middlegen appname="CaveatEmptor"
            prefmdir="${basedir}" guj="true"
            databaseurl="jdbc:oracle:thin:@localhost:1521:orac"
```

```

driver="oracle.jdbc.driver.OracleDriver"
username="test"
password="test"
schema="auction"
<hibernate destination="generated/src"
    package="org.hibernate.auction.model"/>
</middlegen>

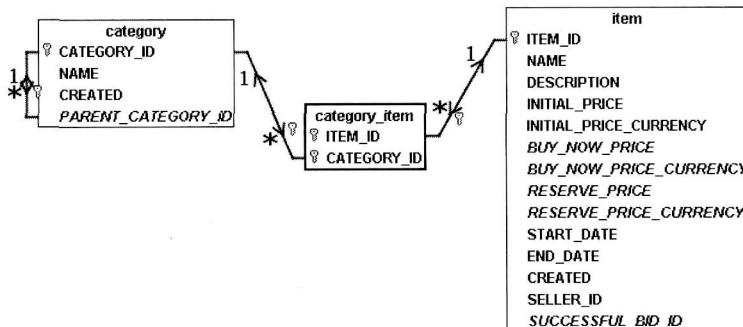
```

Przykład przedstawia minimalną liczbę opcji wymaganą przez Middlegen z modelem Hibernate. Określają one między innymi połaczenie bazodanowe, sterownik JDBC, nazwę użytkownika i hasło do bazy danych. Nazwa schematu bazy również ma znaczenie — w przeciwnym razie Middlegen pobierze tabele wszystkich schematów, do których ma dostęp wskazany użytkownikiem bazy danych.

Middlegen zapisuje ustawienia użytkownika (między innymi położenie tabel w interfejsie graficznym i zmienione opcje) w ścieżce bazowej projektu. Plik z właściwościami ma tę samą nazwę co aplikacja: *CaveatEmptor-prefs.properties*.

Opcji dotyczących modułu Hibernate jest niewiele. Ustawiamy docelowy folder dla generowanych plików odwzorowań. W przykładzie używamy tego samego folderu, do którego później trafią wygenerowane narzędziem *hbm2java* klasy POJO. W ten sposób pliki odwzorowań i klas znajdują się w tej samej lokalizacji. Atrybut *package* określa pakiet stosowany dla wszystkich klas z metadanych odwzorowań.

Uruchomienie budowania w Ant uruchamia Middlegen. Po automatycznym połączeniu się z lokalną bazą danych Oracle, Middlegen odczytuje metadane schematu bazy i uruchamia interfejs graficzny. Góra część okna zawiera widok tabel bazy danych, natomiast dolna zawiera opcje związane z aktualnie wybranym elementem. Rysunek 9.2 przedstawia widok tabel.



Rysunek 9.2. Middlegen przedstawiający tabele bazy danych CaveatEmptor

Po uruchomieniu Middlegen po raz pierwszy (nie istnieje plik konfiguracyjny) tabele i związki są chaotycznie rozmieszczone w górnej części okna. Ich ułożenie wymaga nieco pracy. Na szczęście w większości sytuacji zadanie to trzeba wykonać tylko raz, gdyż Middlegen zapamiętuje ustawienia tabel. Tabele przesywa się, klikając je i przeciągając. Związki także dają się modyfikować (na przykład kliknij związek między *CATEGORY* i *CATEGORY_ITEM*). Po odrobinie pracy uzyskujemy widok bazy danych CaveatEmptor. Na rysunku przedstawiamy tylko

fragment całego schematu (w rzeczywistości narzędzie przedstawia wszystkie tabele i związki).

Zajmijmy się najpierw dostosowaniem asocjacji między tabelami CATEGORY i ITEM. W aplikacji tabela CATEGORY_ITEM służy do powiązania obiektów klas Category i Item asocjację wiele-do-wielu. Jeśli nie zmienimy domyślnych ustawień Middlegen, powstanie osobna encja (klasa trwała CategoryItem).

Większość opcji narzędzia można zmienić graficznie. Niestety, opisana zmiana wymaga modyfikacji konfiguracji pliku Ant dla Middlegen.

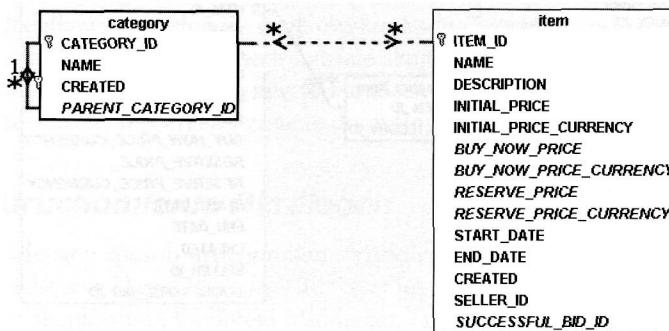
9.4.2. Ograniczanie tabel i związków

Tabele łączące stanowiące część związku wiele-do-wielu należy zadeklarować w zadaniu Ant jako jointable:

```
<many2many>
    <tablea name="CATEGORY"/>
    <jointable name="CATEGORY_ITEM" generate="false"/>
    <tableb name="ITEM"/>
</many2many>
```

Element <many2many> wskazuje asocjację wiele-do-wielu. Teraz Middlegen wygeneruje tylko jedną asocjację w plikach odwzorowań zamiast dodatkowej encji i dwóch asocjacji jeden-do-wielu. Atrybut generate="false" informuje narzędzie, by przeanalizowało tabelę (w celu generacji asocjacji), ale nie tworzyło dla niej osobnej encji.

Ponowne uruchomienie narzędzia powoduje uzyskanie innego układu tabel, patrz rysunek 9.3.



Rysunek 9.3. Związek wiele-do-wielu między CATEGORY i ITEM

Związek między tabelami CATEGORY i ITEM tym razem jest poprawny, ale z widoku znikły wszystkie pozostałe tabele bazy danych (i to niespodziewanie).

Po dodaniu elementu <many2many> w zadaniu Ant Middlegen rozpoznaje tylko nazwane tabele. Ponieważ pojawiły się nazwy tylko dwóch tabel (i tabeli łączące), narzędzie pominęło wszystkie inne. Element <table> zadania Ant pozwala ręczne określić pozostałe tabele.

```
<hibernate>
  ...
  <many2many>
  ...

  <table name="BID"/>
```

Po dodaniu ostatniego z przedstawionych wierszy Middlegen odczytuje i generuje kod dla tabeli BID (wraz z asocjacją między ITEM i BID). Element `<table>` nie zależy od elementu `<many2many>`, więc nic nie stoi na przeszkodzie, by użyć go do ograniczenia liczby tabel, dla których powstaną pliki odwzorowania.

Element `<table>` ma kilka atrybutów dotyczących nazewnictwa, na przykład określenie nazwy pojedynczej i mnogiej (wykorzystywane przy automatycznym nadawaniu nazw asocjacjom, by uzyskać poprawne `getCategories()` zamiast `getCategorys()`). Z pozostałych opcji korzysta się rzadko. Ich opis znajduje się w dokumentacji Middlegen.

Zajmijmy się na razie dostosowywaniem opcji konwersji za pomocą graficznego interfejsu Middlegen, bo wczytaliśmy wszystkie potrzebne tabele w poprawny sposób.

9.4.3. Dostosowanie generowania metadanych

Górna część okna pozwala zaznaczyć tabelę, asocjację lub konkretną kolumnę jednym kliknięciem danego elementu. Zmiana krotności lub kierunku związku następuje po kliknięciu odpowiedniego końca asocjacji z przytrzymanym klawiszem *Ctrl* lub *Shift*. Ustawienia krotności wskazują, czy właściwość będzie pojedynczą wartością czy kolekcją w asocjacji jeden-do-jednego lub wiele-do-jednego.

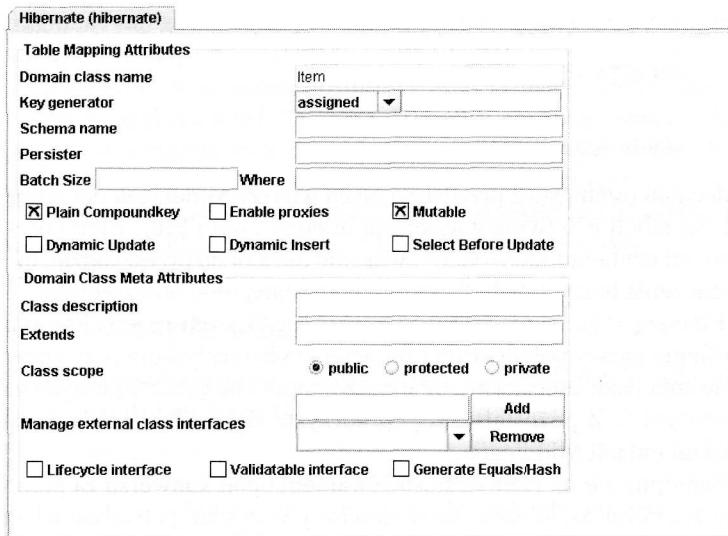
Można do modyfikacji wybrać nie tylko pojedynczą tabelę lub kolumnę, ale również wszystkie kolumny tabeli, by dostosować konwersję do własnych potrzeb opcjami z dolnej części okna. Rysunek 9.4 przedstawia opcje po wybraniu tabeli ITEM.

Middlegen automatycznie stosuje opcje domyślne, na przykład odgaduje nazwę odwzorowywanej klasy na podstawie nazwy tabeli. Pozostałe opcje odpowiadają elementom oraz atrybutom z pliku odwzorowania i określają strategię przypisywania klucza, widoczność, interfejsy klasy trwalej itp. Wszystkie opcje po wybraniu tabeli dotyczą elementu `<class>` pliku odwzorowania.

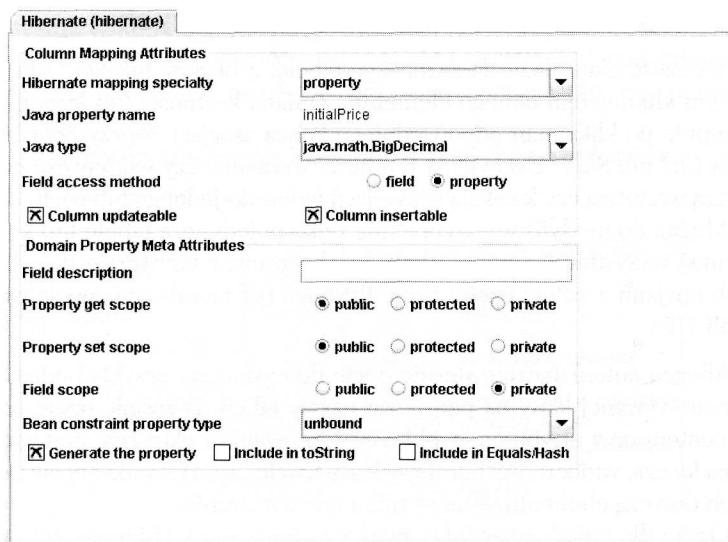
Opcje dla pojedynczej właściwości zmienia się, wybierając kolumnę tabeli. Dostępne opcje przedstawia rysunek 9.5.

W tym przypadku wybrana została kolumna INITIAL_PRICE tabeli ITEM. Jej domyślny typ Java to `BigDecimal`. Middlegen automatycznie sugeruje odpowiedni typ, analizując typ SQL (często uzależniony od rodzaju bazy danych) zawarty w metadanych schematu (dla bazy Oracle jest to `NUMBER(10, 2)`).

Niestety, domyślny mechanizm wykrywania typów nie jest idealny. Rozważmy kolumnę RANKING tabeli USER. Typ danych SQL w Oracle dla tej kolumny to `NUMBER(10, 0)`. Middlegen domyślnie wygeneruje następujący kod XML tworzący właściwość:

ROZDZIAŁ 9.
Narzędzia Hibernate


Rysunek 9.4. Opcje generowania dla tabeli ITEM w narzędziu Middlegen



Rysunek 9.5. Opcje generowania dla kolumny INITIAL_PRICE

```
<property name="ranking"
          type="long"
          column="RANGING"
          not-null="false"
          length="10"/>
```

Wszystko wydaje się poprawne, ale co, jeśli klasa Javy zamiast typu podstawowego stosuje jako typ właściwości klasę `java.lang.Long`? W zasadzie musimy zmienić typ na `java.lang.Long` po dokładnym przyjrzeniu się kolumnie RANKING, bo

typ podstawowy nie może zawierać wartości null, choć taką sytuację dopuszcza baza danych.

Wybór poszczególnych kolumn tabel i ręczna zmiana typu właściwości nie wydaje się najprzyjemniejsza. Na szczęście moduł Hibernate dla Middlegen dopuszcza zastosowanie osobnej klasy konwersji typów dla odwzorowań. Opcję włącza się w zadaniu Ant:

```
<hibernate destination="${gensrc.home}"  
    package="org.hibernate.auction.model"  
  
    javaTypeMapper="middlegen.plugins.hibernate.HibernateJavaTypeMapper"/>
```

Klasa konwersji odwzorowania automatycznie wykrywa kolumny mogące zawierać wartości null i zmienia typ Javy z podstawowego na taki, który obsługuje wartość null. Można również napisać własną klasę konwersji, wzorując się na HibernateJavaTypeMapper, bo jej kod źródłowy dostępny jest bezpłatnie.

Powróćmy do okna dialogowego narzędzia Middlegen. Sekcja *Domain Property Meta Attributes* nie dotyczy bezpośrednio sposobu odwzorowania danych w Hibernate ani głównego sposobu kreowania klas POJO. Jak sama nazwa wskazuje, dotyczy metaatributów używanych przez narzędzie hbm2java. Są to szczególnie przydatne opcje, jeśli na podstawie istniejącego schematu bazy danych powstaje nie tylko odwzorowanie, ale i klasy POJO.

9.4.4. Generowanie metadanych *hbm2java* i *XDoclet*

Jak wspomnieliśmy w poprzednim akapicie, Middlegen generuje metaatributy wykorzystywane przez narzędzie hbm2java. Jeśli na przykład ustawimy zasięg właściwości dotyczącej kolumny NAME tabeli CATEGORY na prywatny, Middlegen wygeneruje następujący kod XML dla właściwości name klasy Category:

```
<property name="name"  
    type="java.lang.String"  
    column="NAME"  
    not-null="true"  
    length="255">  
    <meta attribute="scope-get">private</meta>  
    <meta attribute="scope-set">private</meta>  
</property>
```

Narzędzie hbm2java wykorzysta element *<meta>* przy generowaniu klas POJO w sposób opisany we wcześniejszej części rozdziału. Wygenerowana klasa Category będzie zawierała prywatne metody dostępowe dla właściwości name.

To nie ostatnia zmiana — wystarczy dodać jeden atrybut do konfiguracji, by generowane klasy źródłowe POJO zawierały znaczniki XDoclet. Oznacza to, że na podstawie schematu bazy danych można wygenerować pliki odwzorowań, klasy źródłowe POJO na podstawie odwzorowań i ponownie utworzyć odwzorowania na podstawie kodu źródłowego po uruchomieniu XDoclet. Okazuje się to szczególnie przydatne, gdy Middlegen służy jedynie do wstępnego importu schematu, a wszystkie kolejne modyfikacje przeprowadza się na wygenerowanych klasach POJO lub odwzorowaniach. Przedstawiona sztuczka ponownie korzysta

z metaatributu dla hbm2java: metaatribut `description` umieszcza znaczniki XDoclet w wygenerowanym komentarzu Javadoc kreowanych plików źródłowych.

Najpierw włączamy opcję generowania XDoclet w zadaniu Ant:

```
<hibernate destination="generated/src"
    package="org.hibernate.auction.model"
    genXDocletTags="true"/>
```

Rozważmy wpis powstający dla kolumny NAME tabeli CATEGORY. Middlegen wygeneruje następujący element `<property>`:

```
<property name="name"
    ...
    <meta attribute="field-description">
        @hibernate.property
        column="NAME"
        length="255"
        not-null="true"
    </meta>
</property>
```

Uruchomienie hbm2java na podstawie uzyskanego dokumentu XML generuje kod źródłowy POJO ze znacznikami XDoclet w komentarzach:

```
/**
 * @hibernate.property
 * column="NAME"
 * length="255"
 * not-null="true"
 */
public String getName() { return name; }
```

Po wygenerowaniu w Middlegen szkieletu odwzorowania klas, a następnie użyciu kodu źródłowego klas POJO dzięki hbm2java, można przejść do tworzenia oprogramowania podejściem z góry na dół i jednocześnie lepiej dostosować klasy trwałe i odwzorowania.

9.5. XDoclet

Podejście do programowania z góry na dół zakłada rozpoczęcie prac od implementacji klas trwałych w Javie (lub wygenerowaniem ich automatycznie narzędzi AndroMDA bądź hbm2java). W tym podejściu preferuje się programowanie atrybutowe do automatycznego generowania metadanych. W rozdziale 3. wspomnialiśmy, że język Java do wersji JDK 1.5 nie posiadał obsługi metadanych. We wcześniejszych wersjach można posiłkować się znacznikami Javadoc (na przykład `@attribute`) do określania metadanych w kodzie źródłowym na poziomie klasy, pola i metody.

XDoclet to narzędzie, które odczytuje metaatributy i generuje pliki odwzorowań Hibernate. W zasadzie XDoclet nie ogranicza się tylko do Hibernate, bo

potrafi generować dowolne pliki opisowe bazujące na XML-u, na przykład deskryptory wdrożenia EJB i usług internetowych. W niniejszym podrozdziale wykorzystamy XDoclet 1.2, który potrafi generować zarówno stare pliki odwzorowań z Hibernate 1.x, jak i nowsze dla Hibernate 2.x.

Wady i zalety stosowania XDoclet omówiliśmy w punkcie 3.3.3. W tym podrozdziale nieco dokładniej przyjrzymy się zasadzie działania XDoclet i wykorzystamy go do wygenerowania plików odwzorowania klas trwałych aplikacji Caveat-Emptor. Zaczniemy od klasy User.

9.5.1. Ustawianie atrybutów typu wartości

Klasa User jest encją zawierającą właściwość identyfikującą, kilka innych zwykłych właściwości (oraz komponent Address) i asocjacje do innych encji. Najpierw zadeklarujmy podstawowe odwzorowanie klasy User.

```
/**  
 * @hibernate.class  
 * table="USER"  
 */  
public class User implements Serializable {  
...}
```

Znaczniki XDoclet dla Hibernate zawsze mają postać `@hibernate.znacznik atrybuty`. Znacznik ma najczęściej związek z elementem odwzorowania XML, na przykład znacznik `hibernate.class` dotyczy elementu `<class>`. Atrybut `table` wskazuje, że klasa ma trafić do tabeli USER.

Fragment wygenerowanego pliku odwzorowania dla tak podanego znacznika ma postać:

```
<hibernate-mapping>  
  <class name="User"  
        table="USER">  
  ...
```

Pamiętaj, że zmieniliśmy formatowanie wszystkich generowanych przykładów, by zwiększyć ich czytelność. Dodatkowo pomijamy atrybuty ustawiane przez XDoclet na wartości domyślne. Wyniki uzyskiwane w rzeczywistości mogą być inne, ale ich znaczenie będzie identyczne.

Klasa User jest encją, więc potrzebuje właściwości identyfikującej. W źródle klasy trwałej wszystkie właściwości (typu wartości lub asocjacje) oznacza się znacznikami XDoclet nad metodami pobierającymi tych właściwości. Dla właściwości `id` dodajemy komentarz do metody `getId()`.

```
/**  
 * @hibernate.id  
 * column="USER_ID"  
 * unsaved-value="null"  
 * generator-class="native"  
 */  
public Long getId() {  
    return id;  
}
```

Atrybuty znacznika hibernate.id odpowiadają atrybutom elementu <id>. Następny przykład dotyczy prostej właściwości username:

```
/**
 * @hibernate.property
 * column="USERNAME"
 * length="16"
 * not-null="true"
 * unique="true"
 * update="false"
 */
public Long getUsername() {
    return username;
}
```

Znacznik hibernate.property stosuje te same atrybuty co element <property>. Sytuacja powtarza się dla pozostałych elementów odwzorowania Hibernate. Pamiętaj, że nadal obowiązują wartości domyślne. Jeżeli do metody pobierającej dodamy znacznik @hibernate.property bez atrybutów, uzyskamy odwzorowanie <property name="username"/>. Wszystkie pozostałe atrybuty otrzymają wartości domyślne. Znaczco upraszcza to szybkie tworzenie prototypu modelu dziedziczonego stosującego XDoclet.

Przyjrzymy się komponentowi Address będącemu właściwością typu wartościowego w klasie User:

```
/**
 * @hibernate.component
 */
public Address getAddress() {
    return Address;
}
```

Tym razem domyślne wartości zostaną użyte dla deklaracji hibernate.component. Poza samym wskazaniem komponentu trzeba również odwzorować poszczególne właściwości klasy Address. W kodzie źródłowym Address dodajemy więc znaczniki hibernate.properties dla metod pobierających getStreet(), getZipcode() i getCity(). Samej klasy Address nie trzeba oznaczać — nie jest encją (a jedynie komponentem klasy User i być może innych klas) i nie zawiera właściwości identyfikującej. Znaczniki muszą znaleźć się jedynie przy metodach pobierających właściwości komponentu.

Zakończymy opis deklaracji klasy User na znacznikach związanych z odwzorowaniem asocjacji.

9.5.2. Odwzorowanie asocjacji encyjnych

Odwzorowanie asocjacji encyjnych w XDoclet przypomina w zasadzie oznaczanie właściwości typów wartości. Znaczniki XDoclet trafiają do metod pobierających wszystkich właściwości dotyczących asocjacji. Asocjacja od User do Item ma postać:

```
/**  
 * @hibernate.set  
 * inverse="true"  
 * lazy="true"  
 * cascade="save-update"  
 * @hibernate.collection-key  
 * column="SELLER_ID"  
 * @hibernate.collection-one-to-many  
 * class="Item"  
 */  
public Set getItems() {  
    return items;  
}
```

Pierwszą rzucającą się w oczy różnicą jest liczba elementów odwzorowania, które pojawiają się dla tej właściwości. Przedstawiony przykład odwzorowuje koniec „wiele” dwukierunkowej asocjacji jeden-do-wielu, więc pojawia się kolekcja. Atrybuty dla `hibernate.set` są takie same jak zwykle: `inverse` w celu uzyskania dwukierunkowości i `lazy` dla leniwego sprowadzania. Pozostałe dwa znaczniki odnoszą się do dobrze znanych z dokumentów odwzorowania elementów: `<key>` i `<one-to-many>`. Zauważ, że nazwa klucza obcego w tabeli przedmiotów to `SELLER_ID` (bardziej oczywiste byłoby `USER_ID`, choć nie podkreślałoby znaczenia osoby). Jawnie pojawia się też nazwa klasy, której encje zawiera kolekcja.

Drugi koniec asocjacji również należy odwzorować. W klasie `Item` odwzorowujemy właściwość `seller`.

```
/**  
 * @hibernate.many-to-one  
 * column="SELLER_ID"  
 * not-null="true"  
 * cascade="none"  
 */  
public User getSeller() {  
    return seller;  
}
```

Po stronie „jeden” możemy pominąć klasę wskazywaną przez encję, bo wynika ona niejawnie z typu właściwości. Po odwzorowaniu obu końców asocjacji, przystąpmy do wygenerowania plików XML, czyli uruchomienia XDoclet.

9.5.3. Uruchomienie XDoclet

Generowanie plików odwzorowań przy użyciu XDoclet nie jest trudne. Podobnie jak to miało miejsce w przypadku Middlegen, również używamy zadania Ant. W pliku `build.xml` definiujemy nowe zadanie:

```
<taskdef name="hibernatedoclet"  
        classname="xdoclet.modules.hibernate.HibernateDocletTask"  
        classpathref="class.path"/>
```

Ścieżka klas dla tego zadania powinna zawierać biblioteki: `xdoclet-X.X.X.jar`, `xdoclet-hibernate-X.X.X.jar`, `xdoclet-xdoclet-module-X.X.X.jar` i `xjavadoc-X.X.X.jar`. Biblioteki te znajdują się w głównej dystrybucji XDoclet. Potrzebują również

wielu innych bibliotek pomocniczych, między innymi *commons-lang.jar*, *commons-collections.jar* i *commons-logging.jar*. Proces XDoclet nie wymaga biblioteki Hibernate (ani jej bibliotek dodatkowych).

Kolejny krok to dołączenie utworzonego zadania do wywoływanego celu. Poniższy przykład wykorzystuje nowy cel o nazwie *xdoclet*.

```
<target name="xdoclet">
    <hibernatedoclet
        destdir="mappings/"
        excludetags="@version,@author,@todo"
        force="true"
        mergedir="merge"/>

    <fileset dir="src">
        <include name="**/org/hibernate/auction/*.java"/>
    </fileset>

    <hibernate version="2.0"/>

</hibernatedoclet>
</target>
```

Najpierw atrybut *destdir* definiuje docelowy folder dla generowanych plików odwzorowań. Niektóre standardowe znaczniki Javadoc pomijamy w procesie generowania. Wymuszamy również przetwarzanie plików źródłowych Javy przy każdym uruchomieniu XDoclet (w przeciwnym razie generowane byłyby tylko odwzorowania uaktualnionych plików źródłowych). Folder podany w *mergedir* służy do automatycznego dołączania do generowanych plików odwzorowań zdefiniowanych przez użytkownika.

XDoclet sprawdza cały kod źródłowy z folderu *src* i podfolderu (pakietu) *org.hibernate.auction*. Na końcu przełączamy moduł w tryb Hibernate 2, bo w przeciwnym razie wygenerowałby odwzorowanie dla Hibernate 1.x.

Generowanie metadanych w XDoclet dla Hibernate ma wpływ na środowisko programistyczne i na sposób współpracy wielu programistów. Warto poznać konsekwencje.

Atrybut *mergedir* z zadania Ant pomaga, jeśli trzeba wprowadzić przypadki szczególne w danym procesie programistycznym. Jeżeli plik o nazwie *hibernate-properties-class.xml* znajdzie się w folderze wskazanym w *mergedir*, jego zawartość trafi do pliku odwzorowania klasy *class*. W ten sposób można dodawać własne odwzorowania niezależnie od generatora XDoclet.

Ostatnie słowo na temat XDoclet: kusić może jego stosowanie w każdej sytuacji, nawet wtedy, gdy nie jest to odpowiednie. XDoclet dla Hibernate warto stosować tylko w przypadku programowania w idealnym podejściu z góry na dół. Nie sprawdza się najlepiej w sytuacji, gdy trzeba współpracować ze starszym schematem bazy danych. Bardzo trudne, a czasem wręcz niemożliwe, okazuje się odwzorowanie za pomocą XDoclet niektórych kluczów złożonych i asocjacji potrójnych. We wszystkich standardowych sytuacjach określanie asocjacji, właściwości i klas nie stanowi problemu.

9.6. Podsumowanie

Czasem nie wszystkie aspekty procesu tworzenia oprogramowania można ustalić dowolnie: korzystając z istniejącej bazy danych, najczęściej trzeba dostosować się do zawartego w niej schematu i z tego powodu na jego podstawie generuje się klasy POJO reprezentujące model danych. Do wygenerowania kodu źródłowego Javy na podstawie metadanych odwzorowania służy narzędzie hbm2java. Metadane odwzorowań na podstawie istniejącej bazy danych generuje narzędzie Middlegen. Oba narzędzia zapewniają proces programistyczny z dołu do góry.

Proces z góry na dół rozpoczyna się od utworzenia klas trwałych POJO. Zamiast ręcznie kreować pliki odwzorowań i schemat bazy danych, można w kodzie źródłowym umieścić specjalne znaczniki XDoclet, dzięki którym program o tej samej nazwie wygeneruje odwzorowania. Narzędzie hbm2ddl utworzy polecenia DDL SQL na podstawie metadanych odwzorowania. W ten sposób zautomatyzuje się cały proces programistyczny z góry na dół.

Stosując narzędzia Hibernate (i inne projekty *open source* typu AndroMDA, Middlegen lub XDoclet), pamiętaj o ich ograniczeniach koncepcyjnych: w pełni automatyczne generowanie klas POJO lub metadanych odwzorowań niezależnie od tego, z jakiego źródła nie jest możliwe. Zawsze trzeba ręcznie dostrajać proces konwersji lub modyfikować wyniki generatora.

Nie jest to ograniczenie samych narzędzi, które wydają się raczej dobre w tym, co robią, ale raczej z tego, że nie wszystkie informacje udaje się zebrać z poszczególnych postaci źródłowych. Wyjątek stanowi podejście z góry na dół (stąd jego popularność): odpowiednie klasy POJO wraz z metadanymi odwzorowań pozwalają wykonać w pełni działający skrypt DDL SQL generujący schemat bazy danych. Z doświadczenia wiemy, że skrypt ten niemal zawsze jest równie dobry jak skrypt tworzony ręcznie.

Warto rozpocząć naukę Hibernate bez wymienionych w rozdziale narzędzi. Podstawowym celem narzędzi jest zmnieszenie nakładu pracy związanego z wykonywaniem powtarzających się czynności przy pracy z projektem stosującym Hibernate. Nie przypominają jednak graficznych systemów kreowania, które pomagają szybko rozpocząć pracę, ale niejednokrotnie spowalniają ją na późniejszych etapach projektu. Poświęć czas na naukę podstaw, a dopiero później zwiększą wydajność, stosując narzędzia.

Podstawy języka SOL



DODATEK A

Podstawy języka SQL

Tabela ze swymi kolumnami i wierszami znana jest każdemu, kto kiedykolwiek korzystał z bazy danych SQL. Czasem tabele nazywa się relacjami, wiersze krótkimi, natomiast kolumny atrybutami. Jest to język relacyjnego modelu danych, czyli modelu matematycznego, który (nieidealnie) implementują bazy danych SQL.

Model relacyjny pozwala definiować strukturę danych i ograniczenia gwarantujące integralność danych, na przykład nie dopuszczając do wpisania wartości niezgodnych z regułami biznesowymi. Model relacyjny definiuje również operacje restrykcji, projekcji, iloczynu kartezjańskiego iłączenia [Codd 1970]. Operacje te ułatwiają wykonanie bardziej złożonych operacji na danych, na przykład podsumowywania i nawigacji.

Każda operacja tworzy nową tabelę na podstawie jednej lub kilku istniejących tabel. SQL to język opisujący wskazane operacje w aplikacjach (stąd też nazywany bywa językiem danych) oraz definiujący tabele bazowe, na których później wykonyuje się operacje.

Polecenia DDL SQL tworzą tabele i zarządzają nimi, czyli definiują schemat bazy danych. Do polecen DDL zalicza się polecenia CREATE TABLE, ALTER TABLE i CREATE SEQUENCE.

Polecenia DML SQL zarządzają danymi. Rozważmy te polecenia w kontekście tabel aplikacji CaveatEmptor.

Aplikacja obsługująca aukcje zawiera encje typu **użytkownik**, **przedmiot** i **oferta**. Założymy, że schemat bazy danych tego rodzaju aplikacji składa się z tabel ITEM i BID przedstawionych na rysunku A.1. Typy danych, tabele i ograniczenia określmy wcześniej, wykonując polecenia DDL SQL (operacje CREATE i ALTER).

ITEM

ITEM_ID	NAME	INITIAL_PRICE
1	Foo	2.00
2	Bar	50.00
3	Baz	1.00

BID

BID_ID	ITEM_ID	AMOUNT
1	1	10.00
2	1	20.00
3	2	55.50

Rysunek A.1. Tabele ITEM i BID systemu aukcyjnego

Wstawianie to operacja tworzenia nowej tabeli na podstawie starej przez dodanie kolejnego wiersza. Bazy danych SQL wykonują tę operację bez tworzenia nowej tabeli, czyli wstawiają nowy wiersz do istniejącej:

```
insert into ITEM values (4, 'Fum', 45.0)
```

Wstawiony wiersz modyfikuje operacja aktualizacji:

```
update ITEM set INITIAL_PRICE = 47.0 where ITEM_ID = 4
```

Operacja usunięcia wymazuje wiersz danych:

```
delete from ITEM where ITEM_ID = 4
```

Rzeczywista siła języka SQL objawia się w wyszukiwaniu danych. Pojedyncze zapytanie niejednokrotnie wykonuje wiele operacji relacji na kilku tabelach. Przyjrzyjmy się kilku operacjom.

Operacja **restrykcji** wybiera ze wszystkich wierszy tabeli tylko te spełniające wskazane kryterium. W języku SQL kryterium jest wyrażenie wskazane w klauzuli where:

```
select * from ITEM where NAME like 'F%'
```

Operacja **projekcji** wybiera wskazane kolumny tabeli i eliminuje z wyników powtórzenia. W języku SQL kolumny do pobrania określa klauzula select. Duplikacji wyników zapobiega słowo kluczowe distinct:

```
select distinct NAME from ITEM
```

Iloczyn kartezjański (nazywany też **złączeniem krzyżowym**) tworzy nową tabelę zawierającą wszystkie możliwe kombinacje wierszy dwóch istniejących kolumn. W języku SQL iloczyn kartezjański powstaje po podaniu kilku tabel w klauzuli from:

```
select * from ITEM i, BID b
```

Złączenie relacyjne tworzy nową tabelę, łącząc wiersze dwóch innych tabel. Dla każdej pary wierszy, dla których spełniony jest warunekłączenia, powstaje w nowej tabeli wiersz zawierający wartości pochodzące z obu złączanych wierszy. W ANSI SQL o złączeniu informuje klauzula join, natomiast o warunku złączenia słowo kluczowe on. Aby pobrać wszystkie przedmioty z ofertami, złączamy tabele ITEM i BID na podstawie wspólnego atrybutu ITEM_ID:

```
select * from ITEM i inner join BID b on i.ITEM_ID = b.ITEM_ID
```

Złączenie to w zasadzie iloczyn kartezjański, po którym występuje operacja restrykcji. Z tego powodu złączenie można również wyrazić w stylu theta, podając tabele w klauzuli from, a warunek złączenia w klauzuli where. Wynik poniższego zapytania będzie taki sam jak wcześniejszego:

```
select * from ITEM i, BID b where i.ITEM_ID = b.ITEM_ID
```

Poza podstawowymi operacjami istnieją jeszcze w bazach danych operacje grupowania (GROUP BY) i porządkowania wierszy (ORDER BY):

```
select b.ITEM_ID, max(b.AMOUNT)
from BID b
group by b.ITEM_ID
having max(b.AMOUNT) > 15
order by b.ITEM_ID asc
```

SQL został nazwany **strukturalnym** językiem zapytań, by odnieść się do możliwości stosowania **podzapytań**. Ponieważ każda operacja relacyjna tworzy nową tabelę na podstawie istniejącej (lub kilku istniejących), zapytanie SQL może korzystać z wynikowej tabeli innej operacji. Nic nie stoi na przeszkodzie, by jedno zapytanie zawrzeć w innym:

```
select *  
from (  
    select b.ITEM_ID as ITEM, max(b.AMOUNT) as AMOUNT  
    from BID b  
    group by b.ITEM_ID  
)  
where AMOUNT > 15  
order by ITEM asc
```

Wynik działania przedstawionego zapytania jest taki sam jak wcześniejszego.

Podzapytanie może pojawić się w dowolnym miejscu zapytania SQL. Najbardziej interesujący okazuje się przypadek umieszczenia go w klauzuli where.

```
select * from BID b where b.AMOUNT >= (select max(c.AMOUNT) from BID c)
```

Zapytanie zwróci największą (wartościowo) ofertę zawartą w bazie danych. Klauzulę where często uzupełnia **kwantyfikatorem**. Oto przykład:

```
select * from BID b where b.AMOUNT >= all(select c.AMOUNT from BID c)
```

Kryteria ograniczające języka SQL mogą wykorzystywać wyrażenia zawierające: działania matematyczne, wywołania funkcji, dopasowywanie ciągów znaków, a nawet pełne wyszukiwanie tekstu.

```
select * from ITEM i  
where lower(i.NAME) like '%gc%' or lower(i.NAME) like '%super%'
```

*Strategie implementacji
systemu ORN*



W niniejszym dodatku przedstawimy kilka szczegółów działania systemu Hibernate: jak Hibernate wykrywa zmiany stanów obiektów i jak do tego zadania podchodzą inne systemy ORM. Ogólnie użytkownik ORM nie powinien przejmować się tymi szczegółami, ale rzeczywistość bywa inna, bo mogą mieć one wpływ na sposób projektowania aplikacji.

W trakcie działania implementacja ORM współpracuje z egzemplarzami klas trwałych, by wypełnić wartości właściwości i wykryć zmianę stanu dokonaną przez aplikację (sprawdzanie zabrudzenia). Dodatkowo ORM musi wykrywać dostęp do leniwie sprowadzanych asocjacji. Tradycyjnie istnieje wiele sposobów implementacji tego zadania, większość z nich przeszkaźca w tworzeniu czystego modelu dziedzinowego. Wykorzystuje się dziedziczenie po generowanym kodzie źródłowym, przetwarzanie kodu źródłowego, modyfikację skompilowanego kodu bajtowego i introspekcji w trakcie działania aplikacji. Wybór strategii (lub ich mieszanek) wskazuje, jak przeźroczyste dla modelu będzie narzędzie ORM.

Zanim przejdziemy do tych technik wspomnijmy najpierw o dwóch podstawowych sposobach sprawdzania zabrudzenia i dwóch widokach na to, co powinno zostać utrwalone.

B.1. Właściwości czy pola?

W programowaniu obiektowym dobrą praktyką okazuje się dostęp do atrybutów klasy przy użyciu metod dostępowych. W ten sposób wewnętrzna implementacja danych nie musi być taka sama jak implementacja widziana przez klientów. Przykładowo klasa `Calendar` stosuje jedną wewnętrzną reprezentację danych, ale kilka zewnętrznych widoków. Z drugiej strony klasa `ComplexNumber` stosuje kilka wewnętrznych reprezentacji, choć na zewnątrz przedstawia tylko jedną spójną wersję. Dodatkowo metody ustawiające mogą przeprowadzać validację nowych wartości lub testować, czy użytkownik jest upoważniony do ich zmiany.

Nie ma zgody co do tego, czy rozwiązania ORM powinny korzystać z tych zewnętrznie widocznych właściwości za pomocą metod dostępowych czy raczej z wewnętrznej implementacji w klasie. Zespół Hibernate uważa, że należy rozdzielić trwałą reprezentację JavaBean od jego wewnętrznej struktury danych. Dzięki temu możliwe staje się między innym przesłonięcie sposobu implementacji trwałości właściwości w podklasach. Z tego powodu Hibernate domyślnie zapamiętuje wartości właściwości. By wymusić stosowanie wewnętrznej reprezentacji danych, należy w pliku odwzorowania użyć atrybutu `access="field"`. Niemniej nie polecamy tego rozwiązania.

Oczywiście nie wszystkie atrybuty klasy muszą być publiczne i tym samym dostępne dla użytkownika. Hibernate potrafi współpracować również z metodami dostępowymi zadeklarowanymi jako `protected`, `private` lub o zasięgu pakietu.

Niezależnie od sposobu dostępu trwałych atrybutów istnieją dwa możliwe sposoby implementacji wykrywania modyfikacji obiektów (sprawdzania zabrudzenia): przechwytywanie i inspekcja.

B.2. Strategie sprawdzania zabrudzenia

Pierwsze podejście, **przechwytywanie**, powoduje, że system ORM wsuwa się między aplikację i pola obiektów trwałych, by przechwytywać wszystkie próby ustawiania nowych wartości. Podejście drugie, **inspekcja**, porównuje wartości właściwości na końcu transakcji z wartościami zapamiętanymi w migawce stanu po pobraniu go z bazy danych. Niektórzy argumentują, że przechwytywanie powinno oferować większą wydajność niż inspekcja, ale nie udało nam się potwierdzić tego w testach — różnica okazała się w większości sytuacji pomijalna (a przynajmniej w porównaniu z innymi kosztami, na przykład dostępem do bazy danych).

Hibernate stosuje drugie z wymienionych podejść przy sprawdzaniu zabrudzenia, bo można je zaimplementować bez interwencji ORM w proces wczytywania lub budowania klas. Niestety, system wykonawczy Javy nie dopuszcza zahańczeń umożliwiających przechwytywanie wywołań metod lub zmian pól klasy i kierowanie ich do ogólnego kodu (co ciekawe, funkcjonalność tę oferują inne języki interpretowane). Wprowadzenie w Javie przechwytywania wymaga przetwarzania kodu źródłowego lub bajtowego w trakcie budowania aplikacji lub kodu bajtowego w momencie wczytywania klasy.

By uniknąć tych wymagań, Hibernate sprawdza wszystkie klasy trwałe powiązane z sesją w momencie opróżniania bufora. Hibernate (podobnie jak i inne rozwiązania ORM) wykorzystuje przechwytywanie do implementacji leniwego sprowadzania asocjacji.

Omówimy pokróćce, jak poszczególne systemy ORM implementują przechwytywanie i inspekcję. Nie jest to wyczerpująca lista wszystkich możliwych podejść, ale wskazuje rozwiązania popularne dawniej i obecnie.

B.2.1. Dziedziczenie po wygenerowanym kodzie

To najbardziej inwazyjne podejście. W trakcie prac programistycznych na podstawie metadanych odwzorowania powstaje automatycznie generowania abstrakcyjna superklasa. Podklaśa implementuje zachowanie i stan ulotny obiektu. Podejście to sprawdza się w językach obsługujących dziedziczenie wielobazowe, ale nie pasuje do modelu dziedziczenia języka Java. Co więcej, całkowicie nie pasuje do preferowanego modelu trwałości używającego obiektów POJO. Podejście to traktuje się jako przestarzałe. Nowoczesne systemy ORM używają innych sposobów zapewnienia trwałości.

B.2.2. Przetwarzanie kodu źródłowego

Implementację przechwytywania można przeprowadzić w trakcie komplikacji kodu źródłowego. Przed właściwym procesem komplikacji powstaje pośrednia postać kodu źródłowego Javy. Teoretycznie system mógłby zastępować oryginalny kod źródłowy kodem przetworzonym, ale to podejście byłoby mniej elastyczne dla programisty.

Strategię tę charakteryzują dwa problemy. Po pierwsze, numery wierszy wskazywane przez stos wywołań metod i debugger nie będą odpowiadały oryginalnemu kodowi źródłowemu — numery wierszy wskazują w takiej sytuacji kod z wersji pośredniej. Z drugiej strony zawsze można skorzystać w debuggerze z kodu pośredniego (jest dostępny). Po drugie, uruchomienie przetwarzania w środowisku programistycznym potrafi sprawić problemy, choć nie dotyczy to środowisk wykorzystujących Ant. Z drugiej strony sam kod pośredni nie zawsze sprawdza się w środowiskach stosujących komplikację inkrementacyjną.

B.2.3. Przetwarzanie kodu bajtowego

W praktyce przetwarzanie kodu źródłowego bywa narażone na błędy, więc obecnie popularność zyskuje przetwarzanie skompilowanego kodu bajtowego. Wykonyuje się je najczęściej jako dodatkowy etap budowania aplikacji po kroku pokompilacyjnym. Nic nie stoi na przeszkodzie, by modyfikacji kodu bajtowego dokonywać na etapie wczytywania klasy, jeśli tylko mechanizm trwałości potrafi uzyskać kontrolę nad mechanizmem wczytywania klas.

Przetwarzanie kodu bajtowego nie widać na poziomie kodu źródłowego. Działa całkiem sprawnie, jeśli środowisko programistyczne współpracuje z modyfikatorem klas. Wydaje nam się, że system ORM powinien całą pracę modyfikacyjną wykonywać w momencie wczytywania aplikacji zamiast na etapie jej budowania, gdyż minimalizuje to ewentualne problemy z integracją narzędzi edycyjnych. Ogólnie wszystko zależy od własnych preferencji. Uważamy, że modyfikacja kodu bajtowego jest najlepszą z technik inwazyjnych czasu budowania aplikacji.

B.2.4. Introspekcja

Introspekcja (nazywana też refleksją) na poziomie działania aplikacji ma złą republikację w społeczności programistów Javy, bo uważa się tego rodzaju systemy za powolne. W większości sytuacji nie jest to prawda — introspekcja w najnowszych maszynach wirtualnych jest znacznie szybsza niż w JDK 1.2 — i nie ma dużego wpływu na opisywany rodzaj zagadnienia. Nawet jeśli introspekcja, jak twierdzą niektórzy, spowalnia działanie aplikacji, narzut ten okazuje się znacznie mniejszy od narzutu związanego z dostępem do dysku twardego lub i komunikacji sieciowej (nawet międzyprocesowej), która dominuje w systemach obróbki danych. Nie ma więc powodów, by unikać introspekcji na warstwie trwałości.

Co więcej, istnieje jeden doskonały powód, by wybierać ją zamiast innych technik (na przykład generowania kodu): nie wpływa na cykl powstawania programu ani na wydajność inicjalizacji systemu. Ma pozytywny wpływ na wydajność **programisty**.

Wczesne wersje Hibernate stosowały wyłącznie introspekcję do obiektów modelu dziedzinowego. Rozwiążanie okazało się popularne wśród użytkowników i nie powodowało znaczącego ujemnego wpływu na wydajność. Niestety, podejście to nie sprawdza się, jeśli chodzi o problem sprawdzania leniwego.

By zaimplementować leniwe sprowadzanie asocjacji, Hibernate wykorzystuje **pośredniki**. Pośrednik to obiekt implementujący interfejs publiczny obiektu biznesowego i przechwytyujący wszystkie komunikaty przychodzące do tego obiektu od klientów. Hibernate używa przechwytywania tylko do wykrycia pierwszej próby odczytania danych asocjacji, by w tym momencie załadować potrzebne informacje z bazy danych.

Pośrednik asocjacji po stronie „wiele” używa implementacji interfejsu kolekcji zdefiniowanego w pakiecie `java.util`. Asocjacje po stronie „jeden” (referencja do obiektu klasy użytkownika) wymagają bardziej wyrafinowanych działań.

Od JDK 1.3 Java oferuje `java.lang.reflect.Proxy`. Egzemplarz `Proxy` można tworzyć dynamicznie w trakcie pracy programu wraz z listą implementowanych przez niego interfejsów. Rozwiązanie wydaje się idealne, jeśli klienci wykorzystują obiekty trwałe za pomocą interfejsów. Ale co, jeśli klasa trwała nie implementuje żadnego interfejsu? Nie chcemy przecież wymusić dziwacznych interfejsów lokalnych w stylu EJB dla obiektów POJO. Pamiętaj: wszystko powinno pozostać możliwie proste i nierozbudowane!

Skoro introspekcja nie rozwiązuje wszystkich problemów, co pozostaje?

B.2.5. Generowanie kodu bajtowego „w locie”

Na szczęście dokładnie wtedy, gdy potrzebował tego zespół Hibernate, pojawił się inny projekt *open source* i za jednym zamachem rozwiązał cały problem. Projektanci CGLIB opisują swoje dzieło jako „bibliotekę do generowania kodu”, ale lepiej traktować ją jako alternatywę dla standardowego pakietu introspekcji Javy (zastępnik `java.lang.reflect`).

CGLIB używa kodu bajtowego generowanego „w locie” do implementacji tych samych funkcji, co standardowy interfejs introspekcji, ale znacznie wydajniej. Co ważne, CGLIB oferuje pośredniki dziedziczone po danej klasie, a nie tylko implementujące wskazane interfejsy! W ten sposób Hibernate potrafi zapewnić niemalże niewidoczne leniwe sprowadzanie asocjacji. W trakcie działania aplikacji klient otrzymuje referencję do pośrednika będącego wygenerowaną podklassą klasy trwałej. Gdy klient wywoła metodę pośrednika, Hibernate przechwyci wywołanie i wczyta stan rzeczywistego obiektu z bazy danych.

Dla klas z nieprywatnymi metodami dostępowymi CGLIB oferuje pominięcie introspekcji i uzyskanie wartości właściwości przy użyciu tradycyjnych wywołań metod, stosując wygenerowaną klasę. Hibernate korzysta z tej opcji, gdy tylko jest to możliwe.

Niektórzy użytkownicy Hibernate zauważyl, że krok wstępnej generacji kodu bajtowego w momencie inicjalizacji systemu potrafi spowolnić starsze wersje ORM. Biblioteka CGLIB 2.0 stosowana w nowszych wydaniach Hibernate znaczco poprawiła wydajność generacji kodu bajtowego i tym samym zmniejszyła czas oczekiwania na uruchomienie aplikacji.

CGLIB to wyjątkowo użyteczna biblioteka. Jeśli programowanie uogólnione sprawia problemy, warto z niej skorzystać.

B.2.6. **Obiekty „uogólnione”**

Niektóre warstwy trwałości, które trudno nazwać systemami ORM, utrwalają obiekty zawierające dynamiczny zestaw właściwości — kolekcję par nazwa-wartość (niektóre nawet zalecają rozszerzanie dynamicznych klas własnymi, by zapewnić bezpieczeństwo typów). W książce nie zwracamy zbyt dużej uwagi na takie podejście, ponieważ interesuje nas trwałość obiektów POJO. Przedstawione rozwiązanie znalazło swoją niszę — aplikacje sterowane metadanymi. Modele dziedzinowe z bezpieczeństwem typów nie pasują do aplikacji, w których model biznesowy określają metadane.

Doskonały przykład opisanego podejścia to mechanizm encyjny OFBiz. Również Hibernate 3 obsługuje tego rodzaju aplikacje, dopuszczając modele dziedzinowe reprezentowane jako graf obiektów Map zawierających pary nazwa-wartość.



*Powrót do świata
rzeczywistego*

Hibernate jako projekt typu *open source* utrzymuje inne stosunki z użytkownikami niż produkt tradycyjnej firmy programistycznej, która sprzedaje licencje swoim klientom. Użytkownicy biorą odpowiedzialność za projekt, ale ważniejsze (a niejednokrotnie zabawne) jest to, że całe bezpłatne wsparcie jest realizowane w pełni publicznie. Poznajemy problemy użytkowników z rzeczywistymi aplikacjami dzięki zadawanym przez nich pytaniom. Nie można uniknąć wszystkich kłopotów, czytając dokumentację i książki przed rozpoczęciem projektu. Wszyscy napotykamy nierozwiązywalne problemy — wtedy najlepsze, co można zrobić, to poprosić o pomoc użytkowników forum.

Te same prawa dotyczą użytkowników Hibernate, więc chcemy przedstawić kilka typowych pytań oraz odpowiedzi. Często pojawia się ogromna rozbieżność między książkową teorią a praktycznym programowaniem aplikacji. Na pytania postaramy się odpowiadać z perspektywy wykonawcy aplikacji. Utwierdzisz się w przekonaniu, że w świecie Dilberta nie jesteś sam.

C.1. Dziwna kopia

Z powodu dziwnych wymagań administratora bazy danych Oracle nie mam dostępu do bazy w trybie „na żywo”. Muszę skopiować dane z bazy Oracle do lokalnej bazy danych MySQL (tak, to bardzo nietypowe, ale nic nie poradzę). Napisałem specjalną klasę narzędziową (stosuje zwykle połączenie JDBC zamiast Hibernate). Problem w tym, że wykonuję ją jako zadanie cron poza aplikacją internetową i poza Hibernate. Po wykonaniu aktualizacji nie mam dostępu do bazy danych przez pewien czas (do wygaśnięcia czasu ważności zbuforowanych danych?).

Widzimy tu dwa problemy. Pierwszym jest administrator bazy danych i wymóg kopiowania danych do lokalnej bazy MySQL. To dziwny wymóg i nie wiemy, dlaczego w ogóle istnieje. Pamiętaj, że większość komercyjnych baz danych udostępnia bezpłatne wersje dla programistów. Gdy zmiana bazy danych nie wchodzi w grę, do importu i eksportu używaj Hibernate. Hibernate zawiera specjalną klasę XMLDatabinder, która potrafi zserializować graf obiektów trwały do pliku XML. Klasa ReverseXMLDatabinder służy do odbudowania zapisanych danych i zapisania ich w bazie danych. Umożliwia to dowolne przenoszenie danych między dowolnymi bazami danych obsługiwanyimi przez Hibernate. Więcej informacji na ten temat znajduje się w witrynie Hibernate.

Drugi problem dotyczy blokowania (i niezwalniania) tabel po importie przez bazę MySQL. O powody najlepiej zapytać społeczność użytkowników tej bazy lub tworzącą ją firmę.

C.2. Im więcej, tym lepiej

Mamy tabelę z 700 kolumnami. Z tego, co widzę, tabela znajduje się w trzeciej postaci normalnej. Możemy część kolumn przenieść do innych tabel, ale będą one w związku 1:1 z tabelą główną. Czy można stosować jedną tabelę z 700 kolumnami?

Wydajność bazy danych zależy od wielu czynników, niekoniecznie od liczby kolumn tabeli. Najczęściej warto ograniczyć odczyty wierszy, a nie odczyty kolumn. Przy bardzo dużej liczbie kolumn zawsze możesz użyć zapytań z projekcją. Duże tabele miewają problemy z buforowaniem po stronie bazy danych, bo mniej wierszy zmieści się w bloku bufora, jeśli kolumn jest 700. W takiej sytuacji niezwykle istotny staje się również dostrojenie bazy, szczególnie indeksów kolumn.

Drugą kwestią jest ilość danych, z którą będzie musiał sobie poradzić sterownik JDBC: przy każdym odczycie lub zapisie obiektu musi wysłać bardzo długie (rzędu kilku kilobajtów) polecenie SQL. Pomyśl również o odwzorowaniu takiego wyniku na obiekty trwałe. Sterownik JDBC może utworzyć (w sposób niewidoczny dla programisty) wiele obiektów, gdy Hibernate wydobywa informacje ze zbioru wyników. Koszt ten najczęściej nie jest duży, przynajmniej w porównaniu z innymi operacjami na danych. Zaczyna mieć znaczenie dopiero przy dużej liczbie kolumn, ale przynajmniej rośnie względem ich liniowo.

Wykonaj kilka testów, używając standardowego JDBC i SQL. Sprawdź, ile czasu zajmą. Przeanalizuj również częstotliwość wykorzystania zapytań do tej tabeli w aplikacji, bo może się okazać, że optymalizacja nie ma sensu z racji rzadkiego użycia tabeli.

C.3. Nie potrzebujemy kluczy głównych

Mam proste pytanie. Jeśli mam tabelę bez zdefiniowanego klucza głównego i spróbuję ją odwzorować, co się stanie? Czy Hibernate potrafi obsłużyć takie tabele? W zasadzie spróbowałem wykonać to zadanie, bo mam kilka tabel bez kluczy głównych, których nie pozwolono mi zmodyfikować. Mogę zapisać nowe egzemplarze, ale mam problemy z ich pobraniem.

Rozwiązania relacyjne, które nie wymuszają określenia klucza głównego dla relacji, nie są poprawne, bo dopuszczały pojemniki krotek zamiast zbiorów. Wartość relacyjna to zbiór krotek, więc nie ma duplikatów. Niestety, wiele baz danych SQL dopuszcza tabele ze zduplikowanymi wierszami, bo klucz główny jest w nich opcjonalny.

Hibernate potrzebuje klucza głównego dla tabeli, bo inaczej nie potrafi rozróżnić poszczególnych wierszy. W zasadzie dotyczy to nie tylko Hibernate, ale i wielu innych systemów współpracujących z bazą danych. Zalecamy modyfikację bazy danych. W ostatczności możesz użyć odwzorowania z kluczem złożonym w taki sposób, by klucz obejmował wszystkie kolumny.

C.4. Czas nie jest liniowy

Zauważylem, że po zmianie czasu na letni każde wywołanie aktualizujące wpis w bazie danych wykonany przed zmianą czasu powoduje zgłoszenie wyjątku `StaleObjectStateException`.

Jeśli typ danych znacznika czasowego nie uwzględnia zmian czasu, automatycznie wersjonowanie użyte w Hibernate sprawi problemy.

Nie istnieją żadne poważne przesłanki, by stosować znaczniki czasowe zamiast numerowania. Z drugiej strony znaczników używa się często na późniejszym etapie do tworzenia raportów i zapytań ad hoc zawierających datę ostatniej aktualizacji. Wersja to nic nie znacząca liczba, natomiast dane typu data utworzenia, data modyfikacji i autor modyfikacji niejednokrotnie trafiają do własnych kolumn. W aplikacji wymagającej zbierania wszystkich tego rodzaju danych warto rozważyć tworzenie pełnego dziennika audytowego (patrz podrozdział 8.3.2).

Automatyczne wersjonowanie stosujące zwykłe liczby jest mniej problematyczne, więc polecamy jego stosowanie.

C.5. Dynamicznie niebezpieczne

Dopiero zaczynam pracę z Hibernate. Chciałbym również zastosować tak zwane Dynabeans. Naprawdę! Pochodzę ze środowiska, które przyjęło całkowicie inną koncepcję niż Hibernate: wszystko znajduje się w obiekcie bazodanowym, który zawiera obiekty tabel zawierające obiekty Record, które zawierają obiekty Field. W ten sposób potrzeba metod dostępowych. W zasadzie przyczyniałem się do ich braku. Czy można podobny efekt uzyskać w Hibernate?

Hibernate jako dynamiczne komponenty obsługuje obiekty Map. Pamiętaj jednak, że nie jest to model danych zapewniający integralność danych, który powinien być celem każdej bazy danych. Model danych z niechronionymi wartościami nadaje się tylko do szybkiego prototypowania. Z drugiej strony, prototypy rzadko odrzuca się całkowicie — najczęściej stanowią podstawę dla rzeczywistej implementacji aplikacji. Jeżeli początkowej wersji nie zamieni się na znormalizowany relacyjny model danych, w przyszłości na pewno pojawią się poważne problemy: dziwne zachowanie, utrata danych i kosztowna konserwacja. Są sytuacje (sporadyczne), w których to dynamiczne podejście bywa przydatne.

Pamiętaj, że dane (i baza danych) niejednokrotnie żyje znacznie dłużej niż dowolna aplikacja. Model dynamiczny, znaczenie struktury danych i pewne zakodowane na sztywno w kodzie zasady integracji danych ulegną straciennu, gdy użytkownicy przestaną używać aplikacji. Efektem jest kosztowana i długotrwała migracja danych. Jeśli czas wypuszczenia produktu na rynek ma znaczenie i stanoi główny powód zmniejszenia bezpieczeństwa danych, użądHibernate, by zaoszczędzić czas i uzyskać znaczną wygodę działania.

C.6. Synchronizować czy nie synchronizować?

Używam od jakiegoś czasu Hibernate i bardzo go lubię. Wydaje mi się, że zawiera pewną skazę — synchronizację. Tracimy dane! Stosujemy aplikację internetową, w której wiele osób jednocześnie dodaje i aktualizuje dane — potrzebujemy synchronizacji!

Nie blokuj ani w żaden inny sposób nie szereguj dostępu do obiektów w aplikacji. Nie musisz nic robić, by uniknąć zniszczenia danych przy współbieżnych modyfikacjach — to zadanie Hibernate i bazy danych.

Często pojawia się inne pytanie: „Jak Hibernate zapewnia, że obiekt zmodyfikowany w sesji A zostanie zsynchronizowany z tym samym obiektem załadowanym w sesji B?”. To nie jest ten sam obiekt. Oba obiekty dotyczą tej samej tożsamości bazodanowej, ale zostały wczytane przez inne obiekty Session i z tego powodu są różne. Jest to tak zwana identyczność na poziomie transakcyjnym. Nie ma potrzeby synchronizować dwóch obiektów, które tylko współdzielą tożsamość bazodanową. Innymi słowy, wszystkie operacje we współbieżnie wykonywanych transakcjach są od siebie niezależne. Jak rozwiązać współbieżny dostęp do zasobu?

Hibernate stosuje istniejący mechanizm dostępny w dowolnym systemie bazodanowym, by rozwiązywać konflikty. Obiekt Session z Hibernate używa tego samego znaczenia i poziomów izolacji transakcji co standardowe połączenie JDBC. Oznacza to odczuwanie tych samych problemów, co w przypadku nie posiadania w pełni izolowanego dostępu do współdzielonego zasobu (brudny odczyt, niepowtarzalny odczyt, fantomy). Wszystko zależy od wykorzystywanego poziomu izolacji transakcji bazodanowej. Konflikty aktualizacji są obsługiwane w momencie zatwierdzania przy użyciu podejścia optymistycznego (sprawdzanie wersji). Automatyczne wersjonowanie Hibernate zapewnia, że obiekt wczytany i zmodyfikowany w sesji A nie może nadpisać zmian wykonanych w obiekcie sesji B, jeśli transakcja bazodanowa sesji B (modyfikująca „ten sam” obiekt) została zatwierdzona wcześniej niż sesja A. W rzadkich przypadkach dotyczących wysoce krytycznych danych można użyć blokady pesymistycznej. Włączając taką blokadę (obiektem LockMode) w bazie danych, blokuje sesji B wczytanie „tego samego” obiektu — musi zaczekać (lub zgłosić błąd) aż do zwolnienia blokady przez sesję A.

Pytanie dotyczy niejednokrotnie automatycznego uaktualnienia załadowanego obiektu, gdy jego dane zostały zmienione w innym miejscu. Dotyczą sytuacji, w której sesja B wyświetla obiekt, natomiast sesja A go modyfikuje. Czy sesja B powinna go uaktualnić?

Hibernate nie stosuje automatycznego mechanizmu przekazywania zmian — aplikacja nie przechowuje puli modyfikacji bazodanowych. Współgra to ze znaczeniem transakcji bazodanowej i jej odpowiednika w Hibernate. Nic nie stoi na przeszkodzie, by samemu (stosując obiekt Interceptor) wywołać metodę refresh() wszystkich jednostek zadaniowych posiadających danych obiekt i tym samym zapewnić pełną synchronizację. Pamiętaj, że utrudni to skalowalność rozwiązania.

C.7. Naprawdę gruby klient

Nasz aplikacja stosuje technologię JWS. Zastosujemy Hibernate i architekturę J2EE. Wszystko związane z Hibernate znajdzie się po stronie serwera. Zauważliśmy, że użycie pośrednika powoduje braki w klasach Hibernate po odbudowaniu struktury obiektów po serializacji po stronie klienta. Czy rzeczywiście musimy umieszczać Hibernate i powiązane z nim biblioteki po stronie klienta? Są naprawdę duże.

Klient potrzebuje tylko następujących bibliotek (o łącznym rozmiarze około 1 MB), by poprawnie przywrócić obiekty po serializacji: *hibernate2.jar*, *odmg.jar*, *commons-logging.jar*, *cglib2.jar* (można nawet utworzyć osobną bibliotekę *hibernate-client.jar*).

Pamiętaj, że węzły liścia mogą być niezainicjalizowanymi kolekcjami lub asocjacjami (pośrednikami). Próba przejścia przez graf dla tych obiektów po stronie klienta zgłosi wyjątek. Rozwiązaniem jest pełna inicjalizacja wszystkich wymaganych obiektów przed ich wysłaniem do klienta. Nie zalecamy (nie implementuje też tego Hibernate) leniwego pobierania danych wyzwalanego przez kod klienta. Dostęp do danych, możliwe przejście przez graf i zakresy działania transakcji powinny być dobrze znane w każdej aplikacji.

Jeśli rozmiar klienta ma duże znaczenie, użyj wzorca DTO do przesyłania informacji. Utraci się jednak wtedy główną zaletę odłączonych obiektów POJO w systemie Hibernate — możliwość ich ponownego podłączenia do nowej transakcji i optymistycznego sprawdzenia współbieżności.

C.8. Wznawianie Hibernate

Uruchamiam komputer. Po wykonaniu pracy przechodzę w tryb hibernacji. Gdy ponownie muszę coś zrobić na komputerze, uruchamiam go wyjątkowo szybko (dzięki hibernacji). Problem polega na tym, że gdy wznawiam działanie komputera po braku zasilania (komputer działał w tym czasie), nie włącza się ponownie przy wykorzystaniu hibernacji? Czy w Windows XP mogę w jakiś sposób przywrócić stan komputera?

Szczerze mówiąc, nie wiemy. Nie sądzimy, że Hibernate stanowi złoty środek na wszystkie bolączki. Nie jest rozwiązaniem, które spowoduje, że wszystkie problemy bazodanowe (lub z Windows XP) nagle w magiczny sposób znikną.

Tworzenie aplikacji bazodanowych to jedno z bardziej wymagających zadań programistycznych. Zadaniem Hibernate jest **zredukowanie** ilości kodu, który trzeba napisać w 90 procentach przypadków (typowe operacje CRUD i raporty). Kolejne pięć procent zadań wymaga nieco trudu: wykonania złożonych zapytań, doskonałego zrozumienia znaczenia transakcji i trudnego wyszukiwania powodów problemów z wydajnością. Choć rozwiązanie tych zagadnień w Hibernate ułatwia przenoszenie kodu i pozwala uniknąć wielu pułapek, wymaga pewnego doświadczenia.

Szybkość uczenia Hibernate początkowo jest bardzo wysoka. Z doświadczenia wiemy, że programista potrzebuje od dwóch tygodni do miesiąca, by nauczyć się podstaw. Nie rozpoczynaj przygody z Hibernate na tydzień przed oddaniem projektu, bo Cię to nie uratuje. Przygotuj się, że pierwszy projekt tworzony z Hibernate pochłonie więcej czasu niż można początkowo przypuszczać.

Pozostałe pięć procent zadań wykorzystuje SQL, JDBC i procedury zapamiętane, czyli zagadnienia niemożliwe do zrealizowania w Hibernate: zbiorowa zmiana wielu wierszy danych i złożone zapytania raportujące stosujące funkcje specyficzne dla konkretnej bazy danych.

Zawsze używaj właściwego narzędzia do wykonania zdania (lub ponownie zainstaluj Windows XP).

Bibliografia

- [Ambler 2002] Ambler S., „Mapping Objects to Relational Databases”, materialy AmbySoft Inc., 2002, www.ambysoft.com/mappingObjects.html.
- [Codd 1970] Codd E. F., „A Relational Model of Data for Large Shared Data Banks”, Communications of ACM 13, nr 6 (1970), s. 377–387, <http://doi.acm.org/10.1145/362384.362685>.
- [Date 2004] Date C. J., *An Introduction to Database Systems*, 8th ed., Pearson/AddisonWesley, Boston, 2004.
- [Evans 2004] Evans E., *Domain-Driven Design*, AddisonWesley, Boston, 2004.
- [Fowler 1999] Fowler M., *Refactoring: Improving the Design of Existing Code*, AddisonWesley, Reading, 1999.
- [Fowler 2003] Fowler M., *Patterns of Enterprise Application Architecture*, AddisonWesley, Boston, 2003.
- [Fussel 1997] Fussel M. L., „Foundations of Object Relational Mapping”, ChiMu Corporation, 1997, www.chimu.com/publications/objectRelational.
- [Gamma i in. 1995] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, AddisonWesley, Reading, 1995.

- [Marinescu 2002] Marinescu F, *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*, Wiley, New Jork, 2002.
- [Tow 2003] Tow D., *SQL Tuning*, O'Reilly, Sebastopol, 2003.
- [Walls i Richards 2004] Walls C., Richards N., *XDoclet in Action*, Manning, Greenwich, 2004.

Skorowidz

@attribute, 101
<bag>, 221, 236, 238
<cache>, 200
<class>, 114, 120
<component>, 229
<composite-element>, 226
<element>, 221, 226
<hibernate-mapping>, 94
<idbag>, 221, 236
<joined-subclass>, 117, 119, 120
<key>, 221
<list>, 222
<many-to-many>, 161
<many-to-one>, 158, 159, 162, 228, 229, 237, 290
<nested-composite-element>, 228
<one-to-one>, 158, 162, 290
<param>, 232
<parent>, 226
<property>, 96, 120, 209
<subclass>, 115, 116, 119, 120
<version>, 181

A

access, 96
ACID, 320
addChildCategory(), 149
addClass(), 61
addResource(), 61, 65

Address.getUser(), 112
agregacja, 24, 44, 109, 276
aktualizacje, 97
 obiekt trwały, 143
 schemat bazy danych, 353
 stan trwałego obiektu odłączonego, 141
alias, 256, 266, 268
all, 285
all-delete-orphan, 127
analiza dziedziny biznesowej, 79
AndroMDA, 346, 366
ANSI SQL, 40
ANTLR, 46
any, 245, 285
Apache DBCP, 65
aplikacje
 internetowe, 298
 obiektowe, 25
 warstwowe, 296
approveAuction(), 322, 323, 326
architektura
 aplikacje, 35
 bufor Hibernate, 190
 Hibernate, 55
 warstwowa, 35
ArrayList, 221
asocjacja, 32, 121, 122, 228
 any, 245
 CMP, 121
 dwukierunkowa, 123

asocjacja

- encyjna, 228
- jeden-do-jednego, 32, 33, 229
- jeden-do-wielu, 32, 87, 238
- jeden-do-wielu z kolumną klucza obcego, 240
- jednoelementowa, 158
- jednokierunkowa, 121
- klucz główny, 231
- klucz obcy, 229
- POJO, 86
- polimorficzna, 30, 241, 244
- polimorficzna wiele-do-jednego, 241
- rekurencyjna, 80
- wiele-do-jednego, 123, 125
- wiele-do-wielu, 32, 89, 232, 236
- wiele-do-wielu stosująca komponent, 237
- zarządzana, 121

assemble()

Auditable, 338, 343

AuditLog, 342

AuditLog.logEvent(), 341

AuditLogInterceptor, 342

AuditLogRecord, 338, 342

audyt, 328

automatyczne generowanie schematu

bazy danych, 348

atrybuty odwzorowania XML, 351

hbm2ddl, 348, 349

ograniczenia sprawdzające, 350

przygotowanie metadanych

odwzorowania, 349

SchemaExport, 348

tworzenie schematu, 351

automatyczne sprawdzanie zabrudzenia, 132, 143

automatyczne wykrywanie zabrudzenia, 54

avg(), 276

B

BasicRenderer, 358, 359

batch-size, 160

baza danych, 37

JDBC, 24

SQL, 24

BEA WebLogic, 59

Bean Managed Persistence, 39

beginTransaction(), 141, 171, 172, 304

behawioralne aspekty niedopasowania

obiektowo-relacyjnego, 130

bezpośrednie zapytania SQL, 248

BidDAO, 308

bidForItem(), 315

BidFormItemCommand, 315, 318

Blob, 209

blok inicjalizacji statycznej, 298

blokady

optymistyczne, 176, 181, 183, 185

pesymistyczne, 177, 183

BMP, 39

bogaty model obiektowy, 80

brudny odczyt, 174

bufor

drugi poziom, 193

pierwszy poziom, 191

replikowany, 199

znaczniki czasowe, 293

buforowanie, 176, 186, 195, 204

architektura buforów Hibernate, 190

bufor drugiego poziomu, 193

bufor pierwszego poziomu, 191

bufor replikowany, 199

dostawca bufora, 194

dysk twardy, 199

identyczność obiektów, 188

izolacja transakcji, 189

komunikacja synchronizowana, 202

konfiguracja komunikacji

międzyklastrowej, 202

lokalny dostawca bufora, 198

modułowe zasady usuwania danych, 202

regiony, 197

spudlowanie, 187

sterowanie buforem drugiego poziomu, 203

strategia, 187

ustawienie lokalnego dostawcy bufora, 198

usuwanie elementu z bufora, 203

wbudowane strategie współbieżności, 193

współbieżność, 188

wybór dostawcy bufora, 194

zapytania, 292

zasady, 193

zasięg, 187

zasięg klastra, 187, 188

zasięg procesu, 187

zasięg transakcji, 187

buildSessionFactory(), 65, 71, 353

BusinessException, 315

byte[], 58

byty biznesowe, 34

C

C3P0, 63

Cache, 59

CacheProvider, 59, 195

Calendar, 208

cascade, 146

cascade="all", 235

cascade="all-delete-orphans", 235
cascade="delete", 235
cascade="none", 149
cascade="save-update", 150, 235
Category, 195
CaveatEmptor, 78
 analiza dziedziny biznesowej, 79
Auction, 80
 Category, 79, 80
 diagram stanów zatwierdzania aukcji, 320
 Item, 79, 80
 klasy trwałe modelu obiektów, 81
 model dziedzinowy, 79
 User, 79
CaveatEmptorFacade, 314
chodzenie po grafie obiektów, 33
ClassPersister, 59
Clob, 209
close(), 133
closeSession(), 325
CMP, 39, 83
CMT, 67, 170
CodeGenerator, 354
column, 351
commit(), 141, 143, 170, 171, 172
commons-logging, 72
Comparator, 223
Composite, 26
CompositeUserType, 56, 215, 217
concludeAuction(), 172
Configuration, 56, 57, 60, 65, 67, 102
 setProperties(), 62
configure(), 70
connection(), 287
ConnectionProvider, 59
constrained, 159, 232
Container Managed, 39
CORBA, 57
count(), 276
createAlias(), 269
createCriteria(), 248, 249, 270
createQuery(), 248, 249
createSQLQuery(), 248, 249
Criteria, 57, 152, 155, 249, 260, 268, 280
Criterion, 258
CRM, 121
CRUD, 22, 26, 58
cykl życia obiektu trwałego, 44, 58, 130
 equals(), 136
 hashCode(), 136
 obiekty odłączone, 133
 obiekty trwałe, 132
 obiekty ulotne, 131
 poza zasięgiem identyczności, 135

stany, 131
zasięg identyczności obiektów, 134
czas, 208, 209
czynienie obiektu trwałym, 140

D

dane referencyjne, 190
DAO, 37, 84, 307
Data Transfer Object, 311
Date, 208, 211
daty, 208, 209
DDL, 24, 347
definicja metadanych odwzorowujących, 92
deklaracja nazw klas, 100
deklaracja typu dokumentu, 69
delete(), 95, 143, 144, 206
deskryptor XML, 93
diagram klas UML, 79
Dialect, 59
disasemble(), 217
disconnect(), 185, 324, 325
disconnectSession(), 325
długa sesja, 323, 324
DML, 24, 208
dodawanie logiki do metod dostępowych, 90
doFilter(), 304, 324
dokument odwzorowania, 52
dokument XML, 53, 93
dopasowanie tekstów, 260
doPost(), 299
dostawca bufora, 194
dostęp do właściwości, 96
dowiązywanie
 dowolne argumenty, 253
 parametry, 251
drugi poziom buforowania, 57
DTD, 53, 69, 93
DTO, 39, 311
 równoległa hierarchia klas, 312
 składacz, 312
 zmiana na zasadzie rozprysku, 312
duplicacja, 312
duże obiekty, 209
dwukierunkowa asocjacja jeden-do-wielu, 243
dwukierunkowa asocjacja wiele-do-jednego, 125
dwukierunkowa asocjacja wiele-do-wielu, 234
dwuwarstwowa architektura buforująca, 191
dynamiczne tworzenie obiektów, 274
dyskryminator typu, 115, 116
działania atomowe, 168
dziedziczenie, 26, 29, 113
dziedzina biznesowa, 79

dziennik audytowy, 337
 Auditable, 338
 AuditLog, 342
 AuditLogRecord, 338
 Interceptor, 339
 interfejs znacznika, 338
 kod przechwytyujący, 339
 odwzorowanie rekordu dziennika, 338
 sesja tymczasowa, 342
 tworzenie rekordu dziennika, 338
 włączenie klasy przechwytyjącej, 341
 dzienniki, 71

E

efekt równoległej hierarchii klas, 312
 efekt zmiany na zasadzie rozprysku, 312
 EHCache, 194, 198, 202
 EJB, 22, 39, 66, 83, 311
 EJBException, 315
 element przechwytyujący, 314
 elements(), 285
 encje, 43, 79, 109, 206, 246
 biznesowe, 78
 korzeniowe, 249
 Enterprise JavaBeans, 39
 Environment, 65
 equals(), 31, 136, 137, 139, 212, 214, 227, 237
 evict(), 192, 203
 Exception, 37
 execute(), 299, 301, 302, 315, 316, 317
 Expression, 155, 258
 Expression.and(), 261
 Expression.conjunction(), 261
 Expression.disjunction(), 261
 Expression.or(), 261

F

fantom, 175
 fasada sesyjna, 311, 313
 filtr serwletu, 324
 filtry kolekcji, 282, 283
 find(), 52, 251
 FinderRenderer, 358, 359
 flush(), 173
 FlushMode.AUTO, 173
 FlushMode.COMMIT, 173
 FlushMode.NEVER, 173
 flushSession(), 305
 foreign-key, 351
 from, 264, 265, 274
 funkcje agregujące, 276

G

generator kluczny, 108
 generowanie
 identyfikatory, 107
 klucz główny, 59
 schemat bazy danych, 348
 generowanie kodu klas POJO, 354
 CodeGenerator, 354
 hbm2java, 355
 get(), 153, 179
 getId(), 106
 getIdentifier(), 104, 106
 getItem(), 122
 getNewBid(), 317
 getPropertyNames(), 217
 getPropertyTypes(), 217
 getPropertyValue(), 217
 getPropertyValues(), 217
 getSession(), 303, 304, 313
 getSessionFactory(), 51, 298, 313
 głębia sprowadzania danych, 161
 group by, 274, 277
 grupowanie, 24, 277

H

hashCode(), 136, 139, 212, 227, 237
 HashSet, 87
 having, 274, 278
 hbm2ddl, 348, 349
 aktualizacja schematu bazy danych, 353
 atrybuty odwzorowania XML, 351
 opcje konfiguracyjne, 352
 przygotowanie metadanych
 odwzorowania, 349
 SchemaExport, 352
 SchemaUpdate, 354
 właściwości konfiguracyjne Hibernate, 352
 hbm2java, 355, 365
 config, 359
 finder, 357
 klasy odnajdujące, 357
 konfiguracja, 358
 metaatrybuty, 355
 metody odnajdujące, 357
 output, 359
 scope-set, 356
 session-method, 358
 template, 359
 ToStringBuilder, 356
 uruchamianie narzędzia, 359

Hibern8IDE, 255
Hibernate, 22, 26, 50
dzienniki, 71
EJB, 66
interfejs programistyczny, 55
interfejsy podstawowe, 56
interfejsy rozszerzeń, 59
interfejsy wywołań zwrotnych, 58
konfiguracja bazującą na pliku XML, 69
konfiguracja podstawowa, 59
konfiguracja w środowisku niezarządzanym, 62
konfiguracja w środowisku zarządzanym, 66
opcje konfiguracyjne, 62
pliki odwzorowań, 61
pula połączeń w środowisku
niezarządzanym, 63
serwer aplikacji, 66
SessionFactory, 60
środowisko niezarządzane, 59
środowisko zarządzane, 59, 66
typy, 58
uruchamianie, 65
zaawansowane ustawienia konfiguracyjne, 68
Hibernate Query Language, 52
hibernate.cache.provider_class, 198
hibernate.cache.region_prefix, 197
hibernate.cache.use_minimal_puts, 203
hibernate.cfg.xml, 62, 69, 203
hibernate.connection.factory_class, 67
hibernate.connection.isolation, 177
hibernate.default_schema, 99
hibernate.hbm2ddl.auto, 353
hibernate.id, 368
Hibernate.initialize(), 162
Hibernate.isInitialized(), 163
hibernate.jndi.class, 71
hibernate.jndi.url, 71
hibernate.max_fetch_depth, 162, 163
hibernate.properties, 62, 63, 65, 368
hibernate.property, 368
hibernate.session_factory_name, 69, 70
hibernate.show_sql, 68, 72
Hibernate.STRING, 211
HibernateException, 172, 305
HibernateExtensions, 354, 360
HibernateService, 74
HibernateUtil, 302, 305, 313, 338
HibernateUtil.commitTransaction(), 315
HibernateUtil.newApplicationTx(), 325
Hibernator, 255
hierarchia dziedziczenia, 113
hilo, 108
HQL, 52, 114, 152, 154, 255
 alias, 256, 268
 from, 264, 265

ograniczenia, 257
operatory, 259
operatory logiczne, 261
pobieranie asocjacji, 265
select, 268
where, 264
zapytania, 256
zapytania polimorficzne, 257
złączenia, 264
złączenie niejawne, 270
złączenie w stylu theta, 271
złączenie wewnętrzne, 268
złączenie zewnętrzne, 266
znaki wieloznaczne, 260
HttpSession, 185, 324

I

iBATIS, 38
IdentifierGenerator, 56, 59, 107
identity, 108
identyczność, 44, 104
 obiekty, 30, 31, 103, 104
 zasięg procesu, 133, 134
identyfikatory SQL z apostrofami, 98
ignoreCase(), 281
ilike(), 281
iloczyn kartezjański, 24, 45, 271
Image, 225, 227, 228
implementacja
 asocjacja POJO, 86
 model dziedzinowy, 82
 uchwyt polecenia, 316
implementacja transakcji aplikacyjnych, 319, 321
 długa sesja, 323
 odłączone obiekty trwałe, 322
 trudny sposób, 321
 wybór odpowiedniej implementacji, 327
import statyczny, 258
in, 285
increment, 108
indeksowane kolekcje po inwersyjnej
 stronie asocjacji, 239
index, 351
indices(), 285
InfrastructureException, 315
inicjalizacja
 Hibernate, 69
 leniwe asocjacje, 162
INSERT, 35, 71
instrukcje SQL, 24
instrukcyjne podejście do przesyłania danych, 35
inteligentny model dziedzinowy, 306

intercepcja, 83
 Interceptor, 56, 58, 151, 296, 330, 339, 341
 isUnsaved(), 151
 interfejs
 Auditable, 338
 CompositeUserType, 215
 Configuration, 57
 Criteria, 57, 260
 IdentifierGenerator, 107
 Interceptor, 58, 151, 339
 Lifecycle, 58
 NamingStrategy, 98
 Query, 57, 293
 Serializable, 86
 Session, 56, 140
 SessionFactory, 57
 Transaction, 57, 171
 Validatable, 58
 interfejs programistyczny Hibernate, 55
 internetowy system aukcyjny, 78, 80
 inverse="true", 239
 isMutable(), 214
 istniejący schemat bazy danych, 360
 Item, 220
 Item.addBid(), 72
 ITEM_IMAGE, 220
 ItemDAO, 308
 ItemRow, 275
 iteracja przez listę wyników, 250
 iterate(), 251, 291
 izolacja transakcji, 168, 173, 174

J

Java, 40
 Java DataBase Connectivity, 24
 Java Management Extensions, 73
 Java. Grupa ODMG, 40
 java.io.Serializable, 58
 java.lang.Object, 136
 java.util.Calendar, 58
 java.util.Currency, 58
 java.util.HashSet, 84
 java.util.List, 84
 java.util.Properties, 62
 java.util.Set, 84, 87
 JavaBean, 254
 Javadoc, 101
 javax.jdbc.Datasource, 66
 JBoss, 59, 73
 JBossCache, 195, 199, 200, 202
 jboss-service.xml, 73
 JDBC, 24, 37, 56, 64, 170

JDBCTransactionFactory, 67
 JDK, 210
 JDO, 40
 jeden-do-jednego, 32, 122, 229
 jeden-do-wielu, 32, 122, 238
 jednokierunkowa asocjacja wiele-do-jednego, 123
 jednokierunkowa asocjacja wiele-do-wielu, 233
 jednostka zadaniowa, 168
 jest częścią, 109
 język
 ANSI SQL, 40
 definicja danych, 24
 HQL, 52, 154
 Java, 40
 modyfikacja danych, 24
 SQL, 24
 JGroups, 199
 JMX, 73
 deskryptor wdrożenia Hibernate, 73
 komponenty, 73
 serwer aplikacji, 73
 JMX MBean, 73
 JNDI, 56, 66, 70, 71
 JTA, 56, 67, 170, 301
 JTATransactionFactory, 67

K

kaskadowe usuwanie, 125
 kaskadowe zapisywanie, 125
 klasy
 bazowe, 29
 dziedziczenie, 29
 encyjne, 207
 identyfikator złożony, 331
 JDK, 210
 klasy typu użytkownika, 212
 komponenty, 225
 narzędziowe, 37
 odnajdujące, 357
 POJO, 84, 316
 pomocnicze, 37
 trwałe, 50, 86
 klient JMX, 73
 klucz
 biznesowy, 138
 czysto identyfikacyjny, 31, 107
 główny, 44, 105, 106, 231
 kandydujący, 106
 naturalny, 107, 329
 obcy, 229
 złożony, 107, 328, 330

kod
 przechwytyujący, 339
 rusztujący, 87, 90
 SQL, 24
 kolejność wyników zapytań, 262
 kolekcje, 87, 160
 komponenty, 225
 komponenty dla asocjacji wiele-do-wielu, 236
 polimorficzne, 243
 posortowane, 223
 uporządkowane, 223
 kolumny dyskryminatora, 245
 kolumny niepuste, 227
 komponenty, 109, 110
 EJB, 39
 MBean, 73
 programowe, 110
 zarządzane, 83
 kompozycja, 110
 jednokierunkowa, 112
 komunikaty, 50
 konfiguracja Hibernate, 57, 60
 środowisko niezarządzane, 62
 kontekst JNDI, 71
 kontener
 EJB, 311
 JMX, 73
 konwencje nazewnictwa, 98
 koszt niedopasowania, 34
 krotność, 122
 wiele-do-jednego, 53
 kwantyfikatory, 285

L

lazy, 160
LazyInitializationException, 301, 327
 lekkie odwzorowanie obiektów, 43
 length, 351
 leniwa asocjacja, 162
 leniwa inicjalizacja, 301
 liczba żądań kierowanych do bazy danych, 33
 Lifecycle, 56, 58
 like, 260, 281
 LinkedHashMap, 225
 LinkedListSet, 225
 List, 221
 list(), 250, 251
 listy, 220, 222
 load(), 95, 134, 153, 179
 lock(), 142, 144, 322
 LockMode, 142, 178, 323
 log4j.properties, 72
 logEvent(), 339

logika
 biznesowa, 26, 306
 sterująca, 306
 trójstanowa, 258
 lokalizacja pliku odwzorowania, 60

L

łańcuch wywołań metod, 60
 łańcuchowe wykonywanie metod, 250

M

maksymalna głębia, 161
 mapowanie relacyjno-obiektowe, 41
 mapy, 222, 224
 MatchMode, 260
 max(), 276
 maxelement(), 285
 maxindex(), 285
 MBean, 73
 mediator, 26
 Mediator, 26
 menedżer trwałości, 84
 Message, 50, 51
 Message.hbm.xml, 60
 metadane, 53
 odwzorowanie obiektowo-relacyjne, 78, 92
 XML, 53, 92, 93
 metody
 biznesowe, 85
 odnajdujące, 357
 metody dostępowe, 86, 91
 logika, 90
 Middlegen, 360
 Domain Property Meta Attributes, 365
 dostosowanie generowania metadanych, 363
 generowanie metadanych hbm2java
 i XDoclet, 365
 hbm2java, 365
 ograniczanie tabel i związków, 362
 opcje generowania, 364
 tabele, 362
 uruchamianie, 360
 XDoclet, 365
 zadania, 360
 związki, 362
 middlegen.MiddlegenTask, 360
 min(), 276
 minelement(), 285
 minindex(), 285
 model biznesowy, 79
 model dziedzinowy, 26, 78, 79
 implementacja, 82

model metadanych, 103
 model obiektowy, 46
 model stylów kaskadowych ustalanych dla asocjacji, 147
 modelowanie związków encji, 42
 Model-View-Controller, 298
 moduły generatorów kluczy, 108
 modyfikacja metadanych w trakcie działania aplikacji, 102
 MVC, 299
 MVCC, 174

N

name, 210
 NamingStrategy, 98
 native, 108
 nawiasy grupujące, 261
 nawigacja, 32
 dwukierunkowa, 226
 po grafie obiektów, 33
 nazwane zapytania, 255
 nazwy, 98
 klasy, 100
 metody dostępowe, 86
 net.sf.hibernate, 56
 net.sf.hibernate.cfg.Environment, 65
 net.sf.hibernate.Hibernate, 211
 net.sf.hibernate.property.PropertyAccessor, 97
 net.sf.hibernate.tool.hbm2ddl.SchemaExport, 348
 net.sf.hibernate.transaction.
 → JDBCTransactionFactory, 67
 net.sf.hibernate.transaction.
 → JTATransactionFactory, 67
 newApplicationTx(), 325
 niedopasowanie paradygmatów, 25, 27
 niejawne złączenie, 264
 nieodpowiedni typ kolumny, 334
 niepowtarzalny odczyt, 174
 nonstrict-read-write, 194, 196
 not-null, 123, 351
 nullSafeGet(), 214
 nullSafeSet(), 214

O

obiektowe systemy bazodanowe, 40
 obiekty
 brudne, 132
 DTO, 39
 identyczność, 30, 104
 nowe, 132
 odłączone, 133

równość, 104
 SessionFactory, 60
 tożsamość, 30
 transakcyjne, 132
 trwałe, 132
 typy encyjne, 109
 typy wartościowe, 109
 ulotne, 25, 131
 wartości, 39
 obiekty dostępu do danych, 307
 obiekty transferu danych, 39
 object role modelling, 42
 obsługa
 dialekt języka SQL, 59
 sesje tymczasowe, 342
 specjalne rodzaje danych, 328
 strategia współbieżności przez bufora, 195
 UDT, 28
 współbieżny zapis danych do bazy danych, 181
 wyjątki, 37
 odczyt
 niezatwierdzony, 175, 176
 powtarzalny, 175
 szeregowalny, 175, 176
 zatwierdzony, 175
 odłączone obiekty trwałe, 322
 ODMG, 40
 odwzorowania, 220, 222
 asocjacje encyjne, 228
 asocjacje polimorficzne, 241
 jednokierunkowe, 239
 klasy, 95
 klasy trwałości danych, 77
 kolekcje, 226
 kolekcje typów wartości, 220
 obiektowe, 41
 obiektowo-relacyjne, Patrz ORM
 tabela z kluczem naturalnym, 329
 tabela z kluczem złożonym, 330
 typy proste Javy, 208
 właściwości, 95
 właściwości identyfikujące, 103
 odwzorowanie dziedziczenia klas, 113
 tabela na każdą hierarchię klas, 115
 tabela na każdą podklasę, 117
 tabela na klasę konkretną, 113
 wybór strategii, 120
 ogólne zadania, 82
 ograniczenia, 24, 257
 sprawdzające, 350
 one-to-one, 231
 onFlushDirty(), 341
 onSave(), 341
 OpenSymphony OSCache, 195

operacje
 atomowe, 169
 CRUD, 26
 kaskadowe, 149
operatory
 HQL, 259
 logiczne, 261
 porównania, 258
opróżnianie sesji, 133, 173
optymalizacja pobierania obiektów, 163, 288
 bufor znaczników czasowych, 293
 buforowanie zapytań, 292
 iterate(), 291
 problem n+1 pobrań danych, 288
 sprowadzanie wsadowe, 289
order by, 262, 274
order-by, 224
org.hibernate.auction.Category, 197
org.hibernate.auction.Category.items, 197
org.hibernate.auction.model, 100
ORM, 22, 26, 41, 42, 45, 192
 cykl życia obiektu trwałego, 44
 elementy, 42
 hierarchia dziedziczenia, 44
 implementacja, 45
 klasy trwałości, 44
 konserwacja, 46
 lekkie odwzorowanie obiektów, 43
 niezależność od dostawcy, 47
 pełna relacyjność, 43
 pełne odwzorowanie obiektów, 43
 problemy, 44
 produktywność, 45
 przekształcanie obiektowo-relacyjne, 44
 średnie odwzorowanie obiektów, 43
 wydajność, 46
OSCache, 195
otoczka kolekcji, 160
otrzymanie fantomu, 175
outer-join, 159, 160, 161, 289

P

parametry
 nazwane, 252
 pozycyjne, 252
pełna relacyjność, 43
pełne odwzorowanie obiektów, 43
PersistenceManager, 307
PersistentClass, 103
pielęgnowanie metadanych, 53
PlaceBidAction, 299
pliki konfiguracyjne, 70
pliki odwzorowań Hibernate, 61

pobieranie
 asocjacja, 265
 lista wyników, 250
 obiekt trwały, 142
pobieranie obiektów, 152, 163, 247
 HQL, 154
 pobieranie na podstawie identyfikatora, 153
 strategia sprowadzania danych, 156
 wybór strategii sprowadzania
 w odwzorowaniach, 158
 zapytania przez kryteria, 155
 zapytania przez przykład, 155
podejście od środka, 347
podejście z dołu do góry, 347
podejście z góry na dół, 347
podgrafy, 133
podklasy, 29
podtypy, 29
podzapytania, 24, 284
pojemniki, 220, 221
 zbiory, 238
POJO, 94, 316
 generowanie kodu klas, 354
 implementacja asocjacji, 86
 metody biznesowe, 85
 metody dostępowe, 86
 tworzenie klas, 84
 właściwości, 85, 86
polecenie EJB, 315
polimorficzna asocjacja wiele-do-jednego, 241
polimorfizm, 26, 30
połączenia JDBC, 214
porównywanie identyfikatorów, 272
porównywanie przez wartość, 138
posortowane mapy, 224
postFlush(), 341
poziomy izolacji transakcji, 174, 175
PreparedStatement, 47, 214
problemy
 asocjacja, 32
 identyczność obiektu, 30
 leniwa inicjalizacja, 301
 n+1 pobrań, 34, 45, 288
 nawigacja po grafie obiektów, 33
 niedopasowanie obiektowo-relacyjne, 34
 odwzorowanie obiektowo-relacyjne, 44
 ORM, 44
 podtypy, 29
 szczegółowość, 28
procedury zapamiętane, 287
proces tworzenia aplikacji, 346
 podejście od środka, 347
 podejście z dołu do góry, 347
 podejście z góry na dół, 347

proces tworzenia aplikacji
 spotkanie w środku, 347
 ścieżka przejścia, 348
 programowanie
 oparte na testach, 40
 zorientowane na atrybuty, 101
 projekcja, 24, 32, 154, 274
 projektowanie aplikacji, 295
 aplikacje warstwowe, 296
 propagowanie kontekstu transakcji, 170
 PropertyAccessor, 59, 97
 Proxool, 65
 ProxyFactory, 59
 przechodniość przez osiągalność, 145
 przechwytywanie, 83
 przekształcanie obiektowo-relacyjne, 44
 przesiąkanie zadań, 82
 przestarzałe bazy danych, 328
 przezroczysty zapis opóźniony, 173
 przypisane identyfikatory niezapisanych
 obiektów, 152
 pula połączeń, 63
 C3P0, 64
 JDBC, 63
 pula wątków, 59

Q

QBC, 155, 248
 QBE, 155, 248, 281
 Query, 56, 57, 84, 249, 293
 list(), 251
 Query By Criteria, 155
 Query By Example, 155
 QueryCache, 293

R

read-only, 194, 200
 read-write, 194, 196, 198
 reconnect(), 185, 325
 referencje, 32
 do obiektu odłączonego, 135
 obiektowe, 32
 refleksja, 210
 region buforowanych zapytań, 293
 region_prefix, 197
 regiony buforowania, 197
 relacyjna baza danych, 23
 RemoteException, 315
 reprezentacja hierarchii dziedziczenia, 113
 ResultSet, 214
 returnedClass(), 214

ręczne sprawdzenie wersji, 321
 ręczne tworzenie warstwy trwałości, 37
 RMI, 38
 rodzic-potomek, 126
 rollback(), 170, 172
 RowSet, 26
 rozróżnianie obiektów ulotnych i odłączonych, 151
 rozszerzenia, 59
 rozwiązywanie problemu n+1 pobrania danych, 288
 równoległa hierarchia klas, 312
 równość, 31, 44, 104
 identyfikatory bazodanowe, 137
 klucze biznesowe, 138
 na podstawie wartości, 31
 obiekty, 104
 przez wartość, 138
 równoważność, 104

S

save(), 52, 54, 95, 125, 140, 141, 206, 330
 saveOrUpdate(), 330
 scaffolding code, 87
 SchemaExport, 348, 349, 352
 schemat bazy danych, 348
 schemat SQL, 99
 SchemaUpdate, 353
 select, 268, 274, 275
 SELECT, 35
 SELECT ... FOR UPDATE, 178
 selektywne dołączanie wcześniej
 odłączonych obiektów, 136
 separacja logiki sterującej, 306
 separacja warstwy internetowej
 od warstwy logiki biznesowej, 311
 sequence, 108
 Serializable, 86, 331
 serializacja, 38
 serwer aplikacji, 66
 J2EE, 59
 JMX, 73
 serwlety, 296, 297, 299
 sesja lokalnowątkowa, 301
 sesje, 56, 140
 Hibernate, 56
 na każde żądanie, 184
 na każde żądanie z obiektami odłączonymi, 185
 przezroczysty zapis opóźniony, 173
 szczegółowość, 184
 tymczasowe, 342
 Session, 51, 56, 84, 125, 130, 132, 140, 184, 297
 clear(), 192
 close(), 133

- delete(), 144
find(), 251
flush(), 173
get(), 142
getIdentifier(), 104
lock(), 322
save(), 71, 140, 144
setFlushMode(), 173
update(), 141
- SessionFactory, 57, 60, 61, 62, 67, 70, 74, 103, 203, 297, 298, 313, 353
dowiązanie do JNDI, 70
- Set, 220
setAutoCommit(), 170
setDate(), 253
setEntity(), 253
setFetchMode(), 290
setFlushMode(), 173
setId(), 105
setInteger(), 253
setLocale(), 253
setMaxResults(), 249
setParameter(), 254
setProperties(), 254
setString(), 252, 253
setTimestamp(), 253
sieć zaufania, 80
size(), 285
składacz DTO, 312
słowa kluczowe HQL, 256
some, 285
sort="natural", 223
sortowanie, 24, 44
specjalne rodzaje danych, 328
specyfikacja ODMG, 40
sposoby obsługi współbieżnego zapisu danych do bazy danych, 181
spójność, 168
sprawdzanie danych, 156
leniwe, 156, 157
maksymalna głębia, 161
natychmiastowe, 156, 157
wsadowe, 157, 158, 289
wybór strategii w odwzorowaniach, 158
wyprzedzające, 156, 157, 264
- SQL, 22, 24, 37
Java, 24
sql-type, 351
sqlTypes(), 214
stan elementów, 80
starsze schematy baz danych, 328
sterowanie
aktualizacje, 97
bufor drugiego poziomu, 203
odłączanie podgrafów, 133
- proces wykonywania zapytań, 57
współbieżność przez wersjonowanie, 174
wstawienia, 97
- stosowanie obiektów trwałych, 129
- strategie, 26
buforowanie, 59, 187
dostęp do właściwości, 59, 96
sprawdzanie asocjacji w trakcie działania aplikacji, 266
sprawdzanie danych, 156, 288
współbieżność, 193
- Strategy, 26
string, 210
String, 210, 223
stronicowanie wyników, 249
Struts, 296
struts-config.xml, 61
styl theta, 264
style kaskadowe ustalone dla asocjacji, 147
sum(), 276
SwarmCache, 195
synchonizacja dostępu do obiektów trwałych, 134
system serwletowy, 297
system typów, 206
system zarządzania relacyjną bazą danych, 23
szczegółowe modele obiektów, 108
szczegółowość, 28
sesja, 184
- S**
- ścieżka przejścia, 348
ścieżki do plików odwzorowań, 60
średnie odwzorowanie obiektów, 43
środowisko
niezarządzane, 59, 62
zarządzane, 59, 66
- T**
- tabele
asocjacyjne, 33, 161
łączące, 33, 161
- Tapestry, 296
theta, 271
ThreadLocal, 302, 303, 324
Throwable, 298
timeToIdleSeconds, 198
timeToLiveSeconds, 198
Tomcat, 68
ToStringBuilder, 356
tożsamość
bazodanowa, 104, 132
obiekt, 30

- Transaction, 51, 56, 57, 59, 171
transactional, 194, 200
TransactionFactory, 59
TransactionManagerLookup, 59
transakcje, 57, 67, 132, 167, 168
 aplikacyjne, 133, 180, 319, 320
 aplikacyjne z sesją długoterminową, 185
 aplikacyjne z wieloma sesjami, 184
 atomowość, 168
 automatyczne zatwierdzanie, 170
 bazodanowe, 168, 169
 biznesowe, 180
 blokady optymistyczne, 176, 181, 183
 blokady pesymistyczne, 177, 183
 brudny odczyt, 174
 buforowanie, 176, 186
 CMT, 170
 CORBA, 57
 długo działające, 180
 fantom, 175
 implementacja blokady optymistycznej, 185
 izolacja, 168
 JDBC, 170
 JTA, 170
 LockMode, 178
 MVCC, 174
 niepowtarzalny odczyt, 174
 odczyt niezatwierdzony, 175, 176
 odczyt powtarzalny, 175
 odczyt szeregowalny, 175, 176
 odczyt zatwierdzony, 175
 opróżnianie sesji, 173
 otrzymanie fantomu, 175
 początek, 171
 poziomy izolacji, 174, 175
 problemy związane z izolacją, 174
 propagowanie kontekstu, 170
 sposoby obsługi współbieżnego zapisu
 danych do bazy danych, 181
 spójność, 168
 stany systemu, 169
 systemowe, 169
 szczegółowe, 179
 szczegółowość sesji, 184
 Transaction, 171
 trwałość, 168
 tryby blokad, 178
 ustawianie poziomu izolacji, 177
 utrata aktualizacji, 174
 utrata wcześniejszego zapisu, 174
 użytkownikowe, 180
 wersjonowanie zarządzane, 181
 współbieżny zapis danych do bazy danych, 181
wybór poziomu izolacji, 176
wycofanie, 169, 172
zarządcą, 170
zarządzane przez kontener, 67, 170
zatwierdzanie, 169
transakcyjny zapis opóźniony, 54, 132
transakcyjny zasięg identyczności, 134
TreeBag, 224
TreeCache, 202
treecache.xml, 200, 203
TreeMap, 224
TreeSet, 224
true_false, 208
trwałość, 23, 25, 95, 168
 aplikacje obiektowe, 25
 automatyczna, 82, 83
 bez zasięgu identyczności, 134
 dane, 22
 kaskadowa, 146
 obiekty, 47
 przez osiągalność, 145, 148, 164
 przezroczysta, 82, 83
 ulotna, 125, 127
 XML, 41
 zapewniania przez kontener, 39
 zarządzana przez ziarenko, 39
trwałość przechodnia, 144
 operacje kaskadowe, 149
 przechodniość przez osiągalność, 145
 rozróżnianie obiektów ulotnych
 i odłączonych, 151
 trwałość kaskadowa, 146
 zarządzanie kategoriami przedmiotów, 147
tryby blokad, 178
try-catch, 300
tworzenie
 aplikacje, 346
 asocjacja dwukierunkowa, 123
 interfejs znacznika, 338
 klasy komponentu, 225
 klasy POJO, 84
 kod przechwytyujący, 339
 łańcuch wywołań metod, 60
 pośrednik, 59
 SessionFactory, 60
 typy odwzorowań, 211
 warstwa trwałości, 37
 zaawansowane zapytania, 279
 zapytania raportujące, 273
 zapytania SQL, 249
typy, 58, 206
 binarne, 209
 encyjne, 109

proste, 208
wartości, 109, 206, 246
wyliczeniowe, 217
zdefiniowane przez użytkownika, 58
typy odwzorowań, 207, 210
 data i czas, 208
 duże obiekty, 209
 klasy JDK, 209

U

uaktualnianie, 24
uchwyt polecenia, 315
UDT, 28
UML, 79
unikanie kolumn niepustych, 227
unique, 351
unique-key, 351
uniqueResult(), 250
unsaved-value, 152
UPDATE, 35
update(), 141, 322
UpdateTimestampsCache, 293
uruchamianie Hibernate, 65
UserDAO, 309
UserTransaction, 57, 313
UserType, 56
ustawienie lokalnego dostawcy bufora, 198
ustawienie poziomu izolacji, 177
usuwanie, 24
utrata aktualizacji, 174
utrata wcześniejszego zapisu, 174
uuid.hex, 108

V

Validatable, 56, 58
VARCHAR, 218
Velocity, 359
VelocityRenderer, 359
viewItem(), 325

W

warstwowa architektura systemowej, 35
warstwy, 296
 alternatywy, 35
 biznesowa, 36
 prezentacji, 36
 trwałości, 35, 36, 37
warunki złączenia, 263
wbudowane strategie współbieżności, 193
wbudowane typy odwzorowań, 207

WebSphere, 59
WebWork, 296
wersjonowanie zarządzane, 181
węzeł zapasowy, 202
where, 264, 266, 274, 283
wiązanie obiektu z sesją bez wymuszania aktualizacji, 142
wiele-do-jednego, 53, 122
wiele-do-wielu, 32, 122, 232
własne typy odwzorowań, 211
nieodpowiednie kolumny, 334
właściwości
 identyfikujące, 104
 pochodne, 284
 wyliczane, 96
współbieżność, 188, 204
wstawianie, 24, 97
wybór
 implementacja transakcji aplikacyjnych, 327
 klucz główny, 106
 poziom izolacji, 176
 strategia sprowadzania w odwzorowaniach, 158
wycofanie transakcji, 172
wydajne pobieranie danych z asocjacjami, 45
wydajne pobieranie obiektów, 247
wydajny dostęp do danych w języku SQL, 33
wygrywa pierwsze zatwierdzenie, 183
wyjątki, 298, 300
wykonywanie zapytań, 249
wyliczenia zapewniające bezpieczeństwo typów, 217
wypełnianie stanu obiektu, 91
wysyłanie zapytań do bazy danych, 57
wyszukiwanie, 44
wywołania zwrotne, 58
wywołanie funkcji SQL, 275
wyzwalacze, 336
wzorce
 DAO, 37, 307
 DTO, 311
 element przechwytyujący, 314
 fasada sesyjna, 311, 313
 MVC, 298
 polecanie EJB, 315
 sesja lokalnowątkowa, 301, 302

X

XDoclet, 101, 102, 347, 365, 366
 atrybuty typu wartości, 367
hibernate.id, 368
hibernate.properties, 368
hibernate.property, 368

XDoclet

- odwzorowanie asocjacji encyjnych, 368
- uruchamianie, 369
- znaczniki, 367

XML, 53

Y

YACC, 46

yes_no, 208

Z

zaawansowane zapytania, 279

- zachowanie modelu biznesowego, 80
- zakończenie z pojedynczą wartością, 87
- zakończenie z wieloma wartościami, 87
- zamiana obiektu ulotnego na obiekt trwałego, 140
- zamykanie sesji Hibernate, 304
- zapamiętywanie komunikatów w bazie danych, 50
- zapis opóźniony, 72
- zapisywanie obiektu do bazy danych, 51
- zapytania**, 248, 249
 - buforowanie, 292
 - createQuery, 249
 - Criteria, 249
 - dotyczące obiektów, 255
 - dowiązywanie dowolnych argumentów, 253
 - dowiązywanie parametrów, 251
 - filtry kolekcji, 282
 - interfejsy, 249
 - iterate(), 291
 - kwantyfikatory, 285
 - nazwane, 254
 - nieksorelowane, 284
 - parametry nazwane, 252
 - parametry pozycyjne, 252
 - pobieranie listy wyników, 250
 - podzapytania, 284
 - polimorficzne, 30, 114, 257
 - przez kryteria, 155, 248
 - przez przykład, 155, 248
 - Query, 249
 - skorelowane, 284
 - stronicowanie wyników, 249
 - właściwości pochodne, 284
 - zaawansowane, 279
- zapytania dynamiczne**, 280
 - QBE, 281
- zapytania HQL**, 57, 248, 255
 - alias, 256, 266
 - dopasowanie tekstów, 260
 - from, 265

kolejność wyników zapytań, 262

- like, 260
- łączenie fragmentów tekstu, 260
- MatchMode, 260
- nawiasy grupujące, 261
- ograniczenia, 257
- operatory logiczne, 261
- operatory porównań, 258
- order by, 262
- polimorficzne zapytania, 257
- porównywanie identyfikatorów, 272
- składnia, 256
- where, 264, 266
- złączenia, 264, 266
 - złączenie niejawne, 270
 - złączenie w stylu theta, 271
 - złączenie wewnętrzne, 268
 - złączenie zewnętrzne, 266
 - znaki wieloznaczne, 260
- zapytania raportujące, 154, 273
- agregacja, 276
- dynamiczne tworzenie obiektów, 274
- funkcje agregujące, 276
- group by, 277
- grupowanie, 277
- having, 278
- ograniczanie grup, 278
- pobieranie różnych wyników, 275
- poprawa wydajności, 279
- projekcja, 274
- select, 274
 - wywołanie funkcji SQL, 275
- zapytania SQL**, 248, 249, 286
 - krotka z encjami, 287
 - procedury zapamiętane, 287
 - symbole zastępcze, 286
 - wyniki, 286
- zarządcą transakcji, 170
- zarządcą trwałosci, 140
- aktualizacja obiektu trwałego, 143
- aktualizacja stanu trwałego obiektu odłączonego, 141
- czynienie obiektu trwałym, 140
- pobieranie obiektów trwałych, 142
- zmiana obiektu odłączonego
 - na obiekt ulotny, 144
- zmiana obiektu trwałego na obiekt ulotny, 143
- zarządzane asocjacje**, 88
- zarządzanie**
 - bufor pierwszego poziomu, 192
 - kategorie przedmiotów, 147
 - połączenia JDBC, 59
 - transakcje, 57, 59, 168
 - trwałe dane, 23

- zasady buforowania, 193
zasady nazewnictwa metod dostępowych, 86
zasięg buforowania, 187
zasięg identyczności obiektów, 134
zbiory, 220
zdalne wywoływanie metod, 38
zdalny interfejs fasady sesyjnej, 314
ziarenko encyjne EJB, 39
ziarenko sesyjne EJB, 313
złączenia, 24, 32, 157, 262
 HQL, 264
 niejawne, 270
 sprowadzające, 264, 265, 288
 w stylu theta, 264, 271
 wewnętrzne, 264, 268
 zewnętrzne, 263, 266
złączenie asocjacji, 262
filtrowanie wyników, 263
warunki, 263
złożenie, 26
zmiana
 na zasadzie rozprysku, 312
obiekt odłączony na obiekt ulotny, 144
obiekt trwały na obiekt ulotny, 143
zmienne lokalnowątkowe, 302
znaczniki
 Javadoc, 101
 XDoclet, 367
znaki wieloznaczne, 260
związek rodzic-potomek, 126
związki między encjami, 32
związki zarządzane przez kontener, 88, 121

Ž

źródło danych, 66

Zamów tę książkę w dowolnej chwili



Zamówienia książek przez SMS pod numer 0 691 HELION

Wyślij SMS-em pod numer **0 691 HELION (0 691 435 466)** umieszczony na okładce numer katalogowy książki. **W ciągu 24 godzin wyślemy przesyłkę z książkami na wskazany adres.** Wraz z numerem katalogowym wpisz w treści SMS-a dane wysyłki, np.:

Przykład ➔ 0000 Adam Czerski ul. Kwiatowa 6/12 60-160 Poznań

zamówienia indywidualne

Przykład ➔ 0000 PPHU Webservice ul. Reymonta 3 60-160 Poznań NIP: 433-093-07-54

zamówienia dla firm i instytucji

W przypadku gdy planujesz zakup kilku egzemplarzy tego samego tytułu, przed numerem katalogowym dopisz liczbę zamówionych książek wraz ze znakiem *.

Przykład ➔ 2*0000 Adam Czerski ul. Kwiatowa 6/12 60-160 Poznań

Faktura VAT zostanie wysłana wraz z zamówionym towarem.

Koszty transportu i SMS-a

Domyślnie zamówienie będzie realizowane za pośrednictwem poczty polskiej, za zaliczeniem pocztowym (za książki zapłacisz przy ich odbiorze). W tym przypadku koszty transportu pokrywa wydawnictwo, do ceny książki doliczony zostanie jedynie koszt pobrania pocztowego. Jeśli życzysz sobie, by przesyłkę dostarczyła poczta kurierska, w treści SMS-a dopisz słowo **kurier**.

Przykład ➔ 0000 Adam Czerski ul. Kwiatowa 6/12 60-160 Poznań kurier

Ceny przesyłek są ustalane przez Poczту Polską i firmy spedycyjne:

- **4,90 zł** – przesyłka zwykłą pocztą za pobraniem na terenie Polski
- **11,99 zł** – przesyłka pocztą kurierską na terenie Polski

Koszty przesyłki mogą ulec zmianie. Koszt SMS-a określa operator Twojej sieci komórkowej.

Potwierdzenie przyjęcia zamówienia

Po otrzymaniu SMS-a z zamówieniem prześlemy informację zwrotną, potwierdzającą jego przyjęcie. W treści wiadomości znajdziesz numer zamówienia oraz całkowite koszty odbioru przesyłki książek: cena detaliczna książki + koszt pobrania pocztowego (lub koszt przesyłki kurierskiej).

Nie przyjmujemy zamówień wysłanych z internetowych bramek SMS.



Zamówienia telefoniczne:

0 801 33 99 00

Książki możesz zamówić telefonicznie, dzwoniąc pod numer **0 801 33 99 00**. Nasz konsultant doradzi Ci w wyborze książek, błyskawicznie realizując Twoje zamówienie. Koszt połączenia telefonicznego jest zgodny z taryfą operatora Twojej sieci telefonicznej.