Java. Aplikacje bazodanowe. Najlepsze rozwiązania

Autor: George Reese
Tłumaczenie: Radosław Meryk
ISBN: 83-7361-260-2
Tytuł oryginału: Java Database Best Practices
Format: B5, stron: 296
Przykłady na ftp: 54 kB

Aplikacje biznesowe dotyczą danych — niezależnie od tego, czy są to dane o produkcie, szczegóły dotyczące kart kredytowych użytkowników czy preferowanego koloru kupowanych samochodów. Wraz ze wzrostem znaczenia informacji wzrosła także złożoność dostępu do nich. Programiści Javy mogą wybierać teraz spośród różnego rodzaju interfejsów API i technologii — EJB, JDO, JDBC, SQL, RDBMS, OODBMS i innych. Do tej pory byli oni zdani na siebie przy podejmowaniu decyzji o tym, który model najlepiej pasuje do ich aplikacji i jak w najlepszy sposób korzystać z wybranego API.

Książka „Java. Aplikacje bazodanowe. Najlepsze rozwiązania" przychodzi z pomocą programistom. Teraz nie muszą już oni przeszukiwać kilku książek na temat różnych API, aby zdecydować o odpowiedniej metodzie. Ten obszerny przewodnik omawia podstawy wszystkich wiodących interfejsów API (Enterprise JavaBeans, Java Data Objects, JDBC, a także innych, mniej znanych opcji), objaśnia metodologię i komponenty projektowe wykorzystujące wspomniane interfejsy oraz prezentuje rozwiązania najbardziej dostosowane do różnych typów aplikacji.

Książka omawia także zagadnienia dotyczące projektowania baz danych, począwszy od architektury tabel, skończywszy na normalizacji. Autor przedstawia najlepsze rozwiązania rozmaitych problemów. Nauczysz się w jaki sposób przeprowadzać różne rodzaje normalizacji, a także dowiesz się, kiedy warto przeprowadzić denormalizację. Uzyskasz także szczegółowe instrukcje dotyczące optymalizacji zapytań SQL w celu najlepszego wykorzystania struktury bazy danych. Zaprezentowano także praktyczne zastosowania omawianych technik dostarczając informacje, które Czytelnik może zastosować natychmiast we własnych projektach aplikacji biznesowych.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

8

Alternatywne wzorce trwałości

Rozum musi we wszystkich swych poczynaniach poddawać się krytyce. W przypadku gdy ograniczy wolność krytyki przez zakaz, wyrządza krzywdę sam sobie, narażając się na niszczące podejrzenia. Nic nie jest tak ważne z powodu swojej roli ani tak święte, aby mogło być zwolnione z tych analiz, nie znających respektu dla żadnej z osób. Rozum zależy od tej wolności przez samo swoje istnienie. Rozum nie ma dyktatorskiego autorytetu, zatem jego werdykt jest zawsze porozumieniem wolnych obywateli, z których każdy musi mieć prawo do wyrażania, bez nakłaniania lub przeszkadzania, swoich wątpliwości lub nawet sprzeciwu.

— Immanuel Kant

Krytyka czystego rozumu

Wachlarz możliwości wyboru modelu trwałości dostępny dla projektanta aplikacji o skali przedsiębiorstwa zawiera o wiele więcej możliwości, niż tylko zastosowanie modelu trwałości polecanego przez firmę Sun lub opracowanie własnego modelu. Ostatnio popularność zyskują alternatywne modele trwałości. Dzieje się tak z kilku powodów:

- komponenty EJB są skomplikowane, mają duży rozmiar i wymagają zastosowania serwera aplikacji;
- zastosowanie interfejsu JDBC jest czasochłonne i wymaga doświadczenia w programowaniu baz danych;
- obiekty JDO to rozwiązanie nowe, które ciągle jeszcze nie jest dostępne we wszystkich implementacjach.

Krótkie wyszukiwanie w internecie pozwoli na odnalezienie wielu alternatywnych systemów trwałości. Dwa najpopularniejsze projekty to *Castor JDO* (niebędący implementacją

Wymagania programowe

Systemy Castor i Hibernate nie są standardową częścią platformy Javy. Z tego powodu potrzebny jest zestaw tych narzędzi, obsługujący ich interfejsy API. Każdy interfejs API charakteryzuje się innym zbiorem wymagań, o których więcej informacji można znaleźć w macierzystych witrynach obu projektów, odpowiednio:

Castor

http://castor.exolabs.com

Hibernate

http://hibernate.sf.net

specyfikacji Sun JDO) oraz *Hibernate*[*]. W tym rozdziale opisano projekty Castor i Hibernate jako alternatywne narzędzia odwzorowań obiektowo-relacyjnych, wykorzystujące język XML.

Dlaczego warto stosować alternatywne szablony?

Filozofia Javy jest oparta na zasadzie przestrzegania standardów i rywalizowania w dziedzinie implementacji. W świecie Javy należy przyjąć zasadę unikania odchodzenia od standardów, chyba że standardy te nie spełniają wymagań. O modelach trwałości w Javie trudno powiedzieć, że są tworzone według przyjętych standardów. Jednym z powodów rozbieżności poszczególnych modeli jest fakt, iż trudno stworzyć model trwałości, który spełniałby wymagania każdej aplikacji. Każde podejście do trwałości wymaga podejmowania decyzji projektowych, które zmuszają do rezygnacji z właściwości funkcjonalnych udostępnianych przez inne systemy.

Każdy alternatywny szablon ma określone zalety, z których warto skorzystać w programowaniu baz danych. Ogólnie, alternatywne szablony charakteryzują się następującymi właściwościami:

- Możliwość spełnienia specyficznych wymagań, niespełnionych w standardach EJB i JDO.
- Systemy Castor i Hibernate, podobnie jak EJB i JDO do odwzorowań trwałości wykorzystują pliki deskryptorów XML. Takie rozwiązanie doskonale nadaje się do utworzenia zrozumiałego pliku odwzorowania, opisującego złożone relacje w bazie danych.
- Alternatywne systemy zaprezentowane w tym rozdziale pozwalają na zminimalizowanie rozmiaru kodu, jaki musi utworzyć programista po to, aby utrwalić obiekty.

[*] Obydwa są projektami *open source* dostępnymi w witrynie *SourceForge* (*http://www.sourceforge.net*).

Model „open source"

Model *open source* to sposób rozpowszechniania oprogramowania, oparty na filozofii głoszącej, że użytkownicy oprogramowania mają prawo do posiadania kodu źródłowego tego oprogramowania oraz prawo do jego modyfikowania. Chociaż często model ten jest utożsamiany z oprogramowaniem darmowym, to oprogramowanie tego typu może, ale nie musi być darmowe.

Oprogramowanie *open source* ma następujące zalety:

- Możliwość większej kontroli nad oprogramowaniem włącznie z możliwością jego usprawniania bez konieczności oczekiwania na poprawę błędu przez producenta.
- Ze względu na dużą liczbę programistów zaangażowanych w tworzenie takich programów, oprogramowanie *open source* zwykle charakteryzuje się bardziej zróżnicowanym zapleczem programowym od oprogramowania komercyjnego.
- Oprogramowanie *open source*, nawet jeśli nie jest darmowe, jest zazwyczaj tańsze od komercyjnego. Wynika to stąd, że modele biznesowe firm zajmujących się produkcją takiego oprogramowania są zorientowane przede wszystkim na usługi związane z obsługą systemów.

Z drugiej strony, oprogramowanie *open source* nie jest pozbawione wad:

- Serwis techniczny oprogramowania często zależy od woli jego programistów, a nie jest gwarantowany umową serwisową.
- Wydania oprogramowania *open source*, nad którym nie pracuje zbyt duża grupa programistów, są niespójne a nawet podatne na długie okresy niestabilności.
- Korzyści wynikające z większej kontroli nad oprogramowaniem mogą okazać się fikcyjne, jeżeli utracimy możliwość utrzymania wprowadzonych modyfikacji.

- Systemy alternatywne opisane w tym rozdziale to produkty *open source*. W związku z tym charakteryzują się wszystkimi zaletami i wadami narzędzi tego typu.

Najlepsze rozwiązanie Alternatywne szablony trwałości należy stosować w aplikacjach przeznaczonych dla odbiorców, których podstawowym wymaganiem są niskie koszty oraz aplikacjach o specyficznych wymaganiach, jeżeli takie wymagania spełniają systemy alternatywne.

Alternatywne systemy trwałości mają również swoje wady:

- Ich interfejsy API zapewniające funkcje trwałości są tym, na co wskazuje nazwa — alternatywą. Interfejsy te nie przestrzegają uznanych standardów, takich jak EJB 1.1, EJB 2.0, czy JDO.

- Alternatywne systemy trwałości nie zapewniają stosowania systemu transakcji zarządzanych przez kontenery, podobnego do tego, jaki oferuje kontener EJB. W przypadku komponentów EJB wyznaczenie ram transakcji odbywa się automatycznie po zdefiniowaniu atrybutów transakcji w deskryptorze instalacji a kontener wymusza ścisłe przestrzeganie tych ram.

Najlepsze rozwiązanie Nie należy stosować alternatywnych szablonów trwałości dla aplikacji, które są przeznaczone do wdrożenia w wielu środowiskach korporacyjnych. W takich środowiskach przestrzeganie standardów jest kluczowym czynnikiem, pozwalającym na zdobycie zaufania osób odpowiedzialnych za obsługę aplikacji.

Sposób realizacji funkcji trwałości

Podobnie jak we wszystkich innych zautomatyzowanych systemach trwałości opisywanych w tej książce, w systemach Castor i Hibernate atrybuty komponentów są utrwalane na podstawie pliku konfiguracyjnego XML, zawierającego definicje odwzorowania trwałości. W tabeli 8.1 wyszczególniono atrybuty dwóch obiektów biznesowych, które mogą zostać utrwalone w bazie danych.

Tabela 8.1. Atrybuty dwóch obiektów biznesowych wraz z typami odwzorowania

Klasa	Atrybut	Typ
`Author`	`AuthorID`	`Long`
	`firstName`	`String`
	`lastName`	`String`
	`Publications`	`List`
`Book`	`bookID`	`Long`
	`Author`	`Author`
	`Title`	`String`

Najlepsze rozwiązanie Chociaż w każdym z szablonów wszystkie odwzorowania trwałości można umieścić w jednym pliku XML, należy ograniczać objętość każdego pliku XML, tak aby opisywał odwzorowania pojedynczej klasy.

Kod klasy `Author` opisanej w tabeli 8.1 znajduje się na listingu 8.1.

Listing 8.1. Trwała klasa Author

```
package book;

import java.util.Set;

public class Author {
    private long  authorID;
```

```
    private String firstName;
    private String lastName;
    private Set  publications;

    public long getAuthorID () {
        return authorID;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public Set getPublications () {
        return publications;
    }

    public void setAuthorID (long id) {
        authorID = id;
    }

    public void setFirstName(String fn) {
        firstName = fn;
    }

    public void setLastName(String ln) {
        lastName = ln;
    }

    public void setPublications (Set pubs) {
        publications = pubs;
    }
}
```

Ta klasa biznesowa jest bardzo prosta — zawiera wyłącznie metody pobierające i ustawiające atrybuty. Nie zawiera kodu utrwalania. Zasadniczy kod dla klasy `Book` przedstawiono na listingu 8.2.

Listing 8.2. Obiekt wartości Book zawierający odwołanie do obiektu wartości Author

```
package book;

public class Book {
    private Author author;
    private long bookID;
    private String title;

    public Author getAuthor () {
        return author;
    }

    public long getBookID () {
        return bookID;
    }

    public String getTitle () {
```

```
        return title;
    }

    public void setAuthor (Author auth) {
        author = auth;
    }

    public void setBookID (long id) {
        bookID =id;
    }
    public void setTitle(String ttl) {
        title = ttl;
    }
}
```

Odwzorowania pól w systemie Castor

Kluczowym elementem trwałości w systemach Castor i Hibernate jest plik XML. Każdy interfejs API zawiera własny język, służący do definiowania odwzorowań pomiędzy obiektami a tabelą. Na listingu 8.3 pokazano sposób odwzorowania obiektów biznesowych do bazy danych.

Listing 8.3. Plik XML deskryptora odwzorowania w systemie Castor

```
<?xml version= "1.0"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0
➥//EN" "http://castor.exolab.org/mapping.dtd">

<mapping>
    <class name="book.Author" identity="authorID" key-generator="MAX">
        <map-to table="AUTHOR"/>
        <field name="authorID" type="long">
            <sql name="AUTHORID"/>
       </field>
       <field name="firstname" type="java.lang.String">
            <sql name="FIRSTNAHE"/>
       </field>
       <field name="lastname" type="java.lang.String">
            <sql name="LASTNAME"/>
       </field>
       <field name="publications" type="book.Book" collection="set">
            <sql many-key="authorid"/>
       </field>
    </class>
    <class name="book.Book" identity="bookID" key-generator="MAX">
        <map-to table="BOOK"/>
        <field name="bookID" type="long">
            <sql name="BOOKID"/>
        </field>
        <field name="title" type="java.lang.String">
            <sql name="TITLE"/>
        </field>
        <field name="author" type="book.Author">
            <sql name="AUTHORID"/>
        </field>
    </class>
</mapping>
```

Dla każdej klasy, która ma zostać odwzorowana w deskryptorze XML, umieszcza się znacznik `class`[*]. Znacznik ten ma następujące znaczenie:

- określa nazwę odwzorowywanej klasy Javy;
- określa nazwę atrybutu tożsamości (atrybut niepowtarzalnie identyfikujący egzemplarz klasy);
- jest narzędziem generowania kluczy, które można wykorzystać do otrzymywania identyfikatorów. W obu systemach, zarówno Castor, jak i Hibernate istnieje kilka sposobów generowania niepowtarzalnych wartości.

Najlepsze rozwiązanie Należy wybrać taką metodę generowania sekwencji, która najlepiej odpowiada wybranemu szablonowi trwałości.

System Castor obsługuje następujące algorytmy generowania kluczy:

HIGH-LOW

W tym algorytmie wykorzystano mechanizm podobny do opracowanego przez Autora algorytmu generowania sekwencji, który opisano w rozdziale 4. Algorytm wymaga specjalnej tablicy sekwencji, której klucze są nazwami tabel oraz której kolumny są wartościami zarodków używanych do generowania sekwencji. Więcej informacji na temat wartości zarodków można znaleźć w części poświęconej generowaniu sekwencji, w rozdziale 4.

IDENTITY

Generowana wartość wykorzystuje liczbę uzyskaną przez zastrzeżony mechanizm generowania kluczy dla wybranej bazy danych. Do obsługiwanych baz danych należą Hypersonic SQL, MS SQL Server, MySQL oraz Sybase ASE/ASA.

MAX

Wygenerowana wartość jest o jeden większa od maksymalnej wartości przechowywanej w bazie danych.

SEQUENCE

Generowana wartość wykorzystuje mechanizm `SEQUENCE` baz danych Interbase, Oracle, PostgreSQL oraz SAP DB.

UUID

Ten algorytm generuje globalną unikalną wartość na podstawie adresu IP, bieżącego czasu systemowego w milisekundach oraz statycznego licznika.

---

[*] Użycie słowa `class` powoduje, że ten dialekt XML jest technicznie niedozwolony, gdyż `class` jest zastrzeżonym słowem języka XML.

Algorytmy *SEQUENCE* oraz *HIGH-LOW* wymagają podania parametrów. Parametry te można określić wykorzystując znacznik `key-generator`, umieszczony poza znacznikiem `class`:

```
<key-generator name = "HIGH-LOW">
    <param name = "table" value = "Sequence"/>
    <param name = "key-column" value = "name"/>
    <param name = "value-column" value = "seed"/>
    <param name = "grab-size" value = "1000000"/>
</key-generator>
```

Najlepsze rozwiązanie Należy stosować algorytm generowania kluczy HIGH-LOW.

W obszarze otoczonym znacznikami `<class></class>` znajduje się zbiór znaczników, które definiują odwzorowania atrybutów klas do bazy danych. Pierwszy znacznik w grupie to `map-totable`. Tak jak sugeruje jego nazwa, określa on nazwę tabeli bazy danych, do której należy odwzorować klasę opisywaną przez znacznik.

Pozostałe znaczniki zawierają właściwe definicje odwzorowań pól na kolumny. Warto zwrócić szczególną uwagę na znacznik odwzorowujący relację klasy `Author` z klasą `Book`. Zamiast definiowania kolumny w tabeli `AUTHOR` zdefiniowano atrybut w obrębie klasy `Book`. System Castor wykorzystuje odwzorowanie tej kolumny w celu zrealizowania odpowiednich powiązań.

Odwzorowania pól w systemie Hibernate

Odwzorowania pól w systemie Hiberanate są podobne. Na listingu 8.4 pokazano deskryptor XML definiujący odwzorowanie przykładowych klas do bazy danych.

Listing 8.4. Deskryptor XML odwzorowań pól w systemie Hibernate

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
➥//EN" "http://hibernate.sourceforge.net/hibernate-mapping-1.1.dtd">

<hibernate-mapping >
    <class name="book.Author" table="AUTHOR">
        <id name="authorID" column="AUTHORID" type="long">
            <generator class="vm.long"/>
        </id>
        <property name="firstname" column="FIRSTNAME" type="string"/>
        <property name="lastname" column="LASTNAME" type="string"/>
        <set role="publications" lazy="true">
            <key column="AUTHORID"/>
            <one-to-many class="book.Book"/>
        </set>
    </class>
    <class name="book.Book" table="BOOK">
        <id name="bookID" column="BOOKID" type="long">
            <generator class="vm.long"/>
        </id>
```

```
            <property name="title" column="TITLE"/>
        </class>
    </hibernate-mapping>
```

Kod odwzorowań w systemie Hibernate jest podobny do kodu odwzorowań w systemie Castor. Istnieje jednak wiele istotnych różnic. Podobnie, jak w systemie Castor, do zdefiniowania odwzorowań trwałości dla określonych klas Javy, w systemie Hibernate wykorzystuje się znacznik `class`. Inaczej niż w systemie Castor, w systemie Hibernate zdefiniowano odwzorowania tabel jako atrybut znacznika. Wewnątrz znacznika `class` należy określić identyfikator, właściwości oraz zbiór klas. Znacznik `id` opisuje generator wykorzystywany do generowania kluczy. W znaczniku tym definiuje się klasę Javy zajmującą się wykonywaniem algorytmu generowania. Jeżeli algorytm wymaga podania parametrów, można je określić w treści znacznika `generator`, jako znacznik `param`:

```
<generator class="org.dasein.persist.Sequence">
    <param>sequence</param>
</generator>
```

Klasa `generator` jest implementacją klasy `cirrus.hibernate.id.IdentifierGene`
➥`rator`. W systemie Hibernate dostępne są następujące wbudowane generatory:

`assigned`

Umożliwia aplikacji generowanie własnych identyfikatorów.

`hilo.hex`

Algorytm identyczny z `hilo.long`, poza tym, że w wyniku jego działania uzyskuje się ciąg 16 znaków.

`hilo.long`

Generuje niepowtarzalne wartości typu `long` wykorzystując algorytm *HIGH-LOW*. Nie należy stosować tego generatora w środowiskach JTA lub do połączeń definiowanych przez użytkownika.

`native`

Generuje niepowtarzalną wartość na podstawie kolumn identyfikatorów dla baz danych DB2, MS SQL Server, MySQL, Sybase oraz Hypersonic SQL.

`seqhilo.long`

Generuje niepowtarzalną wartość typu `long` dla sekwencji, której przypisano nazwę, z wykorzystaniem algorytmu *HIGH-LOW*.

`sequence`

Generuje niepowtarzalną wartość wykorzystując konstrukcję `SEQUENCE` dostępną w bazach danych DB2, Interbase, Oracle, PostgreSQL i SAP DB.

`uuid.hex`

Generuje niepowtarzalny ciąg składający się z 32 znaków.

`uuid.string`

Działanie identyczne z `uuid.hex`, poza tym, że generowany jest szesnastoznakowy ciąg ASCII. Tego algorytmu nie należy stosować dla bazy danych PostgreSQL.

`vm.hex`

Generuje niepowtarzalne ciągi znaków na podstawie liczb heksadecymalnych. Tego algorytmu nie należy używać w architekturze klastra.

`vm.long`

Generuje niepowtarzalne wartości typu `long`. Algorytmu nie należy stosować w architekturze klastra.

Najlepsze rozwiązanie W większości platform trwałości dostępnych jest kilka mechanizmów generowania sekwencji. Ważne, aby wykorzystywać taki mechanizm generowania sekwencji, który najlepiej odpowiada wysokopoziomowym wymaganiom architektury (np. podział na klastry).

Właściwości są zasadniczymi atrybutami klasy. Z kolei zbiory reprezentują odwzorowania jeden-do-wielu lub wiele-do-wielu dla klasy. W opisywanym przypadku obiekty odwzorowują jednego autora na wiele książek. W systemie Hibernate relację tę można określić za pomocą znacznika `set`.

W systemie Hibernate dostępnych jest sześć różnych znaczników kolekcji:

- `<array>`
- `<bag>`
- `<list>`
- `<map>`
- `<primitive-array>`
- `<set>`

Dla tych wszystkich typów zbiorów, poza tablicami, można włączyć późne ładowanie wykorzystując zapis `lazy = "true"`. Zastosowanie późnego ładowania umożliwia znaczne przyspieszenie szybkości działania aplikacji, szczególnie takich operacji, jak wyszukiwanie. Aplikację można także przyspieszyć innymi sposobami, np. poprzez zastosowanie buforowania. Warto zwrócić uwagę, ze większość alternatywnych interfejsów API obsługujących trwałość jest wyposażona w pewnego rodzaju wbudowany mechanizm buforowania obiektów, który można włączyć lub wyłączyć.

Najlepsze rozwiązanie Wybierając interfejs API do obsługi trwałości ważne jest właściwe zrozumienie różnych typów odwzorowań obsługiwanych przez wybrany interfejs. Każdy system trwałości charakteryzuje się innymi niedoskonałościami i może nie zawierać obsługi niektórych złożonych rodzajów odwzorowań jak np. odwzorowania typu jeden-do-wielu oraz wiele-do-wielu.

Operacje dotyczące trwałości

Klasy `Author` i `Book`, zaprezentowane na listingach 8.1 i 8.2, nie zawierają metod obsługujących funkcje trwałości. Jednak zarówno w systemie Castor, jak i Hibernate wykorzystanie operacji obsługi trwałości wymaga dodania odpowiedniego kodu.

W aplikacjach, które często wykorzystują bazę danych, odświeżanie przesyłanie nowego stanu w bazie danych jest czasochłonnym procesem. Jedną z zalet kontroli nad tym, kiedy następuje to odświeżanie jest możliwość zarządzania tymi kosztami.

W obu opisywanych systemach trwałości, operacje obsługi trwałości wykonywane są poprzez załadowanie plików odwzorowań, uzyskanie połączenia z bazą danych, wywołanie obiektu, który ma zostać utrwalony i zamknięcie transakcji. Wydzielenie kodu obsługi trwałości za pomocą obiektu dostępu do danych pozwala na utworzenie bardziej przejrzystej implementacji.

Najlepsze rozwiązanie Operacje dotyczące trwałości należy umieścić w hermetycznej klasie, wykorzystując wzorce obiektów dostępu do danych, podobne do tych, które wykorzystano w przykładach innych modeli trwałości, zaprezentowanych w tej książce. Dzięki temu z łatwością można przekształcić kod na inny system trwałości, np. EJB.

Funkcje obsługi trwałości w systemie Castor

W naszym przykładzie potrzebujemy funkcji pozwalającej na dodania nowej książki do listy książek wybranego autora. W systemie Castor metoda służąca do wykonania tego zadania może mieć następującą postać:

```
public Book addBook(Book book) throws Exception {
   JDO jdo = new JDO("alternativepersistencedb");
   jdo.loadConfiguration("database.xml");
   Database db = jdo.getDatabase();
   db.begin();
   db.create(book);
   db.commit();
   db.close();
   return book;
}
```

Najlepsze rozwiązanie Zastosowanie wzorca projektowego *singleton* do zarządzania ładowaniem i analizowaniem plików konfiguracyjnych, opisujących operacje trwałości pozwoli na zwiększenie szybkości operacji obsługi trwałości w aplikacji.

W zaprezentowanym kodzie występuje odwołanie do pliku konfiguracyjnego XML *database.xml*. W pliku tym opisano połączenia z bazą danych, umożliwiające systemowi Castor uzyskanie dostępu do źródła danych JDBC. Na listingu 8.5 pokazano, jak mógłby wyglądać taki plik.

Listing 8.5. Deskryptor połączenia z bazą danych w systemie Castor

```
<!DOCTYPE databases
  PUBLIC "-//EXOLAB/Castor JDO Configuration DTD Version 1.0//EN"
         "http://castor.exolab.org/jdo-conf.dtd">

<database name="aps" engine="sql-server">
    <driver class-name="net.sourceforge.jtds.jdbc.Driver" url="jdbc:jtds:
    ➥sqlserver://localhost:1433/aps">
        <param name="user" value="aps"/>
        <param name="password" value="research"/>
    </driver>
    <mapping href="mapping.xml"/>
</database>
```

Dzięki zdefiniowaniu informacji o połączeniu następuje ustanowienie sesji z bazą danych poprzez wywołanie metody klasy JDO `getDatabase()`. W celu rozpoczęcia transakcji wywołuje się metodę `begin()` w sesji `Database`. Po rozpoczęciu transakcji klasa `Book` jest gotowa do utrwalenia. Wykonuje się to poprzez wywołanie w sesji `Database` metody `create()`. Po jej wywołaniu nastąpi utrwalenie nowej książki. Należy teraz wykonać porządkowanie poprzez zamknięcie transakcji za pomocą metod `commit()` i `close()` w sesji `Database`. W przypadku wystąpienia błędu transakcji sesja `Database` zgłosi wyjątek `TransactionAbortedException`.

Funkcje obsługi trwałości w systemie Hibernate

Obsługa trwałości w systemie Hibernate działa podobnie jak w systemie Castor. Należy wywoływać metody o podobnych nazwach umieszczone w innych klasach:

```
public Book addBook(Book book) throws Exception {
    Datastore ds = Hibernate.createDatastore();
    ds.storeFile("hibernate.xml");
    SessionFactory sessionFactory = ds.buildSessionFactory();
    Session session = sessionFactory.openSession();
    session.beginTransaction();
    session.saveOrUpdate(book);
    session.flush();
    session.connection().commit();
    session.close();
}
```

W pierwszym wierszu kodu wywołano metodę `createDatastore()`, która załadowała deskryptor połączenia systemu Hibernate (zobacz listing 8.6). Dostępny jest także deskryptor połączenia napisany w języku XML.

Listing 8.6. Deskryptor połączenia z bazą danych w systemie Hibernate

```
hibernate.connection.driver_class=net.sourceforge.jtds.jdbc.Driver
hibernate.connection.url=jdbc:jtds:sqlserver://localhost:1433/aps
hibernate.connection.username=aps
hibernate.connection.password=research
```

Po załadowaniu informacji o połączeniu metoda `storeFile()` wczytuje deskryptor odwzorowania. Do zarządzania połączeniami sesji w całej aplikacji wykorzystywana jest klasa `SessionFactory`. W opisywanym przykładzie obiekt klasy `SessionFactory`

tworzony jest dla każdego żądania. W celu rozpoczęcia transakcji wywoływana jest metoda `beginTransaction()` dla bieżącej sesji. Następnie wywoływana jest metoda `save➥OrUpdate()` w celu utworzenia lub aktualizacji utrwalanego obiektu. Na końcu cyklu każdej transakcji trzeba wywołać metodę `flush()`. Operacja wykonywana przez tę metodę ma na celu zsynchronizowanie bazy danych z obiektami w pamięci. Do zatwierdzenia transakcji służy metoda `commit()`, natomiast zamknięcie transakcji następuje po wywołaniu metody `close()`. W przypadku błędów w czasie transakcji zgłaszany jest wyjątek `SQLException`.

Najlepsze rozwiązanie	Obydwa opisane szablony obsługują grupowanie połączeń z bazą danych lub wykorzystanie źródeł danych JNDI. Warto wykorzystywać te możliwości w tworzonych aplikacjach.

Wyszukiwanie

Wyszukiwanie w obu systemach trwałości przebiega podobnie do wyszukiwania w architekturze JDO. Systemy te wykorzystują obiektowe języki zapytań o podobnych interfejsach API.

Wyszukiwanie w systemie Castor

System Castor wykorzystuje obiektowy język zapytań (ang. *Object Query Language* — OQL). Zapytania OQL mają zbliżoną postać do standardowych zapytań ANSI SQL, ale zamiast pól określających kolumny występują nazwy obiektów. Implementację wyszukiwania książek w systemie Castor pokazano w na listingu 8.7.

Listing 8.7. Wyszukiwanie książek według tytułu w systemie Castor

```
public Book findBookByTitle(String title) throws Exception {
    JDO jdo = new JDO("alternativepersistencedb");
    jdo.loadConfiguration("database.xml");
    Database db = jdo.getDatabase();
    db.begin();
    OQLQuery query = db.getOQLQuery("SELECT b FROM book.Book b WHERE
             title=$1");
    query.bind("Alternative Persistence Systems");
    OueryResults results = query.execute();
    // zakładamy, że pierwsza znaleziona książka jest tą, którą poszukiwano
    Book book = (Book) results.next();
    db.commit();
    db.close();
    return book;
}
```

Podobnie, jak w poprzednio omawianym przykładzie, przed wykonaniem operacji należy załadować informacje konfiguracyjne. Po ustanowieniu połączenia do wyszukiwania są wykorzystywane klasy `OQLQuery` oraz `QueryResults`. Najważniejszym wierszem kodu jest wywołanie metody `getOQLQuery()`. W zaprezentowanej operacji wyszukiwania

interesują nas wszystkie encje, dla których wartość atrybutu `title` odpowiada argumentowi `title` dostarczonemu do metody wywołującej. W przypadku braku obiektów spełniających kryteria nastąpi zgłoszenie wyjątku `NoSuchElementException`.

Wyszukiwanie w systemie Hibernate

Wyszukiwanie w systemie Hibernate przebiega niemal identycznie z wyszukiwaniem w systemie Castor. Implementację wyszukiwania w systemie Hibernate pokazano na listingu 8.8.

Listing 8.8. Wyszukiwanie książek według tytułu w systemie Hibernate

```
public Book findBookByTitle(String title) throws Exception {
    Datastore ds = Hibernate.createDatastore();
    ds.storeFile("hibernate.xml");
    SessionFactory sessionFactory = ds.buildSessionFactory();
    Session session = sessionFactory.openSession();
    Book book = null;
    List results = session.find("from o in class book.Book where title =
        ?", title, Hibernate.STRING);
    if (results.isEmpty()) {
        throw new Exception("Nie znaleziono encji: " + title);
    } else {
        book = results.get(0);
    }
    session.close();
    return book;
}
```

Różnica pomiędzy wyszukiwaniem w systemie Castor a Hibernate polega na tym, że w systemie Hibernate nie jest zgłaszany wyjątek w przypadku braku encji spełniających kryteria. Zamiast tego istnieje metoda `isEmpty()`, za pomocą której można sprawdzić, czy zapytanie zwróciło wyniki.

Dodatkowe informacje

Szczegółowy opis każdego z alternatywnych wzorców wykracza poza zakres tej książki. Poza tym systemy Castor i Hibernate nie są jedynymi dostępnymi systemami alternatywnymi. Dzięki zapoznaniu się z niniejszym rozdziałem Czytelnik uzyskał ogólny obraz sposobu działania opisanych szablonów. W rozdziale wskazano też elementy, które można przestudiować bardziej szczegółowo a także opisano rolę alternatywnych szablonów trwałości w programowaniu baz danych.