Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Ярославский государственный университет им. П. Г. Демидова
Кафедра компьютерных сетей

И. В. Парамонов

Разработка приложений баз данных с использованием средств объектно-реляционного отображения

Методические указания

Рекомендовано
Научно-методическим советом университета
для студентов, обучающихся по специальности
Прикладная математика и информатика

Ярославль 2008

УДК 004.65
ББК 3 973.2–018.1я73
П 18

Рекомендовано
Редакционно-издательским советом университета
в качестве учебного издания. План 2008 года

Рецензент:
кафедра компьютерных сетей Ярославского
государственного университета им. П. Г. Демидова

П 18 **Парамонов, И. В.** Разработка приложений баз данных с использованием средств объектно-реляционного отображения: метод. указания / И. В. Парамонов; Яросл. гос. ун-т. — Ярославль: ЯрГУ, 2008. — 44 с.

Методические указания посвящены применению технологии объектно-реляционного отображения при разработке приложений баз данных. Рассмотрены правила определения классов-сущностей в соответствии со спецификацией JSR 220, а также возможности ORM-средства с открытым кодом Hibernate.

Предназначено для студентов IV курса факультета информатики и вычислительной техники ЯрГУ, обучающихся по специальности 010501 Прикладная математика и информатика (дисциплина «Базы данных и информационная безопасность», блок ДС), очной формы обучения.

Библиогр.: 7 назв.

УДК 004.65
ББК 3 973.2–018.1я73

Оглавление

Объектно-реляционное отображение (object-relational mapping, ORM) — это современная технология, позволяющая приложениям естественным образом взаимодействовать с реляционными базами данных, используя для этого высокоуровневые объектно-ориентированные абстракции вместо традиционных представлений о реляционной БД как о совокупности связанных таблиц.

Интерфейс программиста (application programming interface, API) средств объектно-реляционного отображения позволяет сохранять состояние объектов приложения в БД, а также манипулировать хранимыми данными в терминах изменения состояний объектов, тогда как необходимые SQL-запросы, осуществляющие перенос данных между объектами приложения и полями таблиц БД, генерируются автоматически.

Цель данного издания — дать базовые представления об основных механизмах ORM и продемонстрировать основные возможности этого удобного и мощного инструмента. Изложение в основном строится на примерах работы с конкретной базой данных, состоящей из четырёх таблиц и содержащей одну связь типа «один ко многим» и одну связь типа «многие ко многим».

В первой части рассматриваются правила определения классов-сущностей (entities), объекты которых подлежат хранению в БД. Соблюдение этих правил необходимо для того, чтобы средство объектно-реляционного отображения могло корректно загружать и сохранять объекты классов-сущностей. Изложенный в этой части материал соответствует спецификации JSR 220: Enterprise JavaBeans, Version 3.0. Java Persistence API [1] и потому остаётся применимым независимо от того, какое конкретное средство объектно-реляционного отображения используется.

Во второй и третьей частях описывается библиотека Hibernate, которая представляет собой зрелое ORM-средство с открытым кодом, предоставляющее большое количество возможностей и пригодное для профессиональной разработки приложений на языке Java (в том числе на платформе J2EE), а также на C#.NET.

Во второй части рассматриваются основные концепции, предоставляемые Hibernate для управления объектами, а также API для выполнения конкретных операций. В третьей части рассмотрены средства построения запросов на языке HQL (Hibernate Query Language), заимствующем многие черты языка запросов SQL, однако ориентированном на описание свойств объектов, а не полей таблиц БД.

К сожалению, по тематике объектно-реляционного отображения практически отсутствует русскоязычная литература. Обзорное изложение соответствующих вопросов можно найти в книге [2], но эта книга уже несколько устарела (несмотря на то что она была издана в 2008 году!). В частности, в ней рассматривается механизм определения связей между сущностями посредством файлов XML, а не аннотаций в отличие от настоящего издания.

Что касается англоязычных источников основными по данной тематике можно считать спецификацию JSR 220 [1], а также руководство пользователя Hibernate [3].

Отсутствием русскоязычной литературы отчасти обусловлена ещё одна проблема, связанная с тем, что применяемая в данной области русскоязычная терминология является неустоявшейся. Такие термины, как entity, persistence и многие другие, просто не имеют на данный момент общепринятого перевода, а те переводы, которые встречаются в литературе, едва ли можно признать удачными. В связи с этим в данных методических указаниях для таких спорных терминов оставлено английское написание.

В приложении приведён код законченной программы, демонстрирующей основные возможности Hibernate. Большинство примеров, приведённых в частях 1–3, представляют собой фрагменты кода этой программы. Полный код программы позволяет увидеть, как все эти фрагменты вписываются в общий контекст.

Схема данных, используемая в примерах

Используемая в примерах схема данных содержит три сущности **Student** (студент), **Teacher** (преподаватель) и **Subject** (предмет), а также две связи. Связь между сущностями **Teacher** и **Student** обозначает научное руководство и имеет тип «один ко многим». Связь между сущностями **Student** и **Subject** имеет тип «многие ко многим» и обозначает изучение студентом соответствующего предмета. Концептуальная ER-модель, определяющая сущности и связи между ними, приведена на рис. 1.

При реализации концептуальной модели каждой сущности будет соответствовать таблица БД. Поскольку реляционные СУБД непосредственно не поддерживают связи типа «многие ко многим», то выполняется декомпозиция соответствующей связи, в результате которой образуется дополнительная таблица БД и две новые связи типа «один ко многим». Полная атрибутивная модель схемы данных, включающая все таблицы и связи БД, приведена на рис. 2[1].

При разработке приложения каждой сущности соответствует свой собственный класс. В первой части такие классы названы *классами-сущностями*[2]. Состояние объектов этих классов при использовании ORM хранится в таблицах БД, причём каждому объекту соответствует одна строка таблицы.

Кроме данных классы-сущности могут (и на самом деле должны) содержать методы обработки этих данных, что обозначает применение известного в объектно-ориентированном программировании принципа инкапсуляции. К сожалению, в приведённых простейших примерах реализацию бизнес-логики показать трудно без существенного увеличения размеров примера. Более подробную информацию по данному вопросу можно получить в книге [5].

[1] Более подробную информацию о проектировании баз данных и ER-моделях можно найти, например, в классическом учебнике [4].

[2] Термин необщепринятый. В англоязычной литературе вообще не разделяются концептуальные сущности и классы, им соответствующие, — для тех и для других используется один и тот же термин entity.

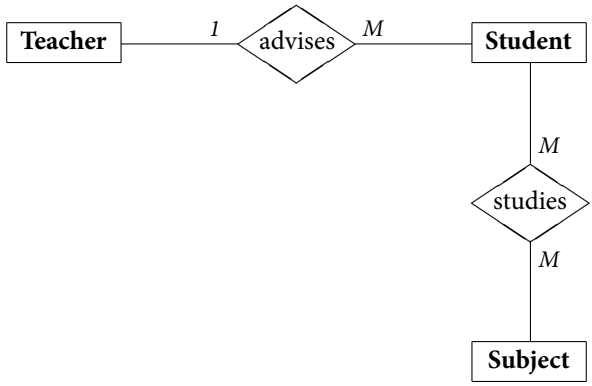

Рис. 1. Концептуальная модель схемы данных

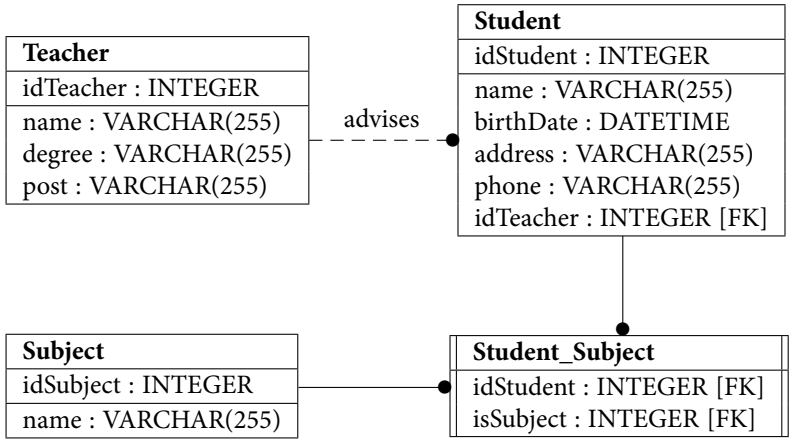

Рис. 2. Полная атрибутивная модель схемы данных

1. Классы-сущности

1.1. Определение классов-сущностей. Существует два способа установления соответствия между классами приложения и столбцами реляционной базы данных. Первый из них основан на использовании XML-файлов. Этот способ является традиционным и подробно рассматривается в литературе (см., например, [2, 3]). Однако он является достаточно громоздким и не очень удобным в поддержке приложения. Более удобным и современным способом установления соответствия является использование аннотаций.

Аннотации — это метаданные, которые вставляются в код программы и непосредственно не оказывают влияния на ход её выполнения. Это означает, что аннотированные классы, поля и методы сами по себе функционируют так же, как и неаннотированные. Однако аннотации хранятся внутри скомпилированного байт-кода и могут быть доступны в ходе выполнения программы посредством специального интерфейса программиста (API).

Применительно к ORM, аннотации представляют собой механизм, который позволяет Hibernate распознать классы, подлежащие хранению в БД, на этапе инициализации и установить соответствие между полями класса и столбцами таблиц БД.

Приведём фрагмент кода аннотированного класса-сущности:

```
package entities;
import java.io.Serializable;
import java.util.Date;
import javax.persistence.*;
@Entity
public class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    // Поля, подлежащие хранению в БД
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer idStudent;
    private String name;
    private String address;
    @Temporal(TemporalType.TIMESTAMP)
    private Date birthDate;
    private String phone;
```

```
    // Конструктор по умолчанию
    public Student() { }
    // Конструктор
    public Student(String name, Date birthDate, String address, String phone) {
        this.name = name;
        this.birthDate = birthDate;
        this.address = address;
        this.phone = phone;
    }
    // Getter'ы и setter'ы
    public Integer getIdStudent() { return idStudent; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getAddress() { return address; }
    public void setAddress(String address) { this.address = address; }
    // ...
}
```

Дадим пояснения к приведённому коду. Все классы-сущности помечаются аннотацией **@Entity**, определённой в пакете **javax.persistence**. По умолчанию объекты классов-сущностей будут храниться в таблицах, имена которых совпадают с именами соответствующих классов (в таблице Student в приведённом выше примере). При необходимости имя таблицы можно указать принудительно с помощью аннотации **@Table**:

```
@Entity
@Table(name = "students")
class Student implements Serializable {
// ...
```

Каждый класс-сущность должен иметь конструктор по умолчанию с уровнем доступа **public** или **protected**.

По умолчанию считается, что все поля класса подлежат хранению в БД. Обычно для каждого из таких полей определяются методы получения и изменения значения, что диктуется бизнес-логикой приложения. Для того чтобы получить доступ к данным, средству объектно-ориентированного отображения эти методы не требуются[3].

[3]Это утверждение относится к применяемому в этих методических указаниях способу доступа к экземплярам через поля (field-based access). Существует также способ доступа через свойства (property-based access), который явно использует setter'ы и getter'ы, определённые в классе. Более подробно об этом способе можно прочитать в спецификации [1].

Описания полей типа **java.util.Date** и **java.util.Calendar** следует предварять аннотацией **@Temporal**.

Класс может содержать поля, значения которых не должны храниться в БД (*временные поля*). Такие поля необходимо помечать аннотацией **@Transient**. Пользоваться временными полями следует с осторожностью и только в случае крайней необходимости, поскольку они имеют тенденцию нарушать концептуальную целостность класса-сущности.

Каждый класс-сущность должен содержать поле, являющееся первичным ключом. Это поле помечается аннотацией **@Id**. По умолчанию предполагается, что значение первичного ключа устанавливается вручную перед сохранением объекта в БД. Поддерживается также автоматическая генерация значения ключа. Например:

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private Integer idStudent;
```

Здесь автоматическая генерация значения полагается на возможность автоинкрементных полей используемой БД. Существуют также другие стратегии.

В качестве типа данных простого первичного ключа могут использоваться как примитивные типы (обычно целочисленные), так и их обёртки. В ситуации, когда первичный ключ объекта не назначен, соответствующее поле должно содержать 0 в первом случае и **null** во втором. Как правило, назначают только положительные значения для первичных ключей, поэтому можно использовать примитивные типы.

Для любого из полей можно уточнить спецификацию столбца таблицы БД, на который будет проектироваться соответствующее поле. Для этого используется аннотация **@Column**. Например:

```
@Column(name = "student_name", nullable = false, length = 64)
private String name;
```

Здесь определяется, что поле **name** проектируется на столбец student_name таблицы БД (а не на столбец name, как это было бы по умолчанию), а также устанавливается максимальная длина поля (64 символа) и ограничение обязательности на значение поля.

1.2. Однонаправленные ассоциации. Связям между таблицами БД соответствуют *ассоциации* между классами. Для любого из типов связей («один ко многим» и «многие ко многим») поддерживаются однонаправленные и двунаправленные ассоциации.

Рассмотрим пример однонаправленной ассоциации «научное руководство» (тип «один ко многим») между сущностями «Студент» и «Преподаватель»[4].

```
@Entity
public class Teacher implements Serializable {
    // ...
}

@Entity
public class Student implements Serializable {
    // ...
    @ManyToOne
    @JoinColumn(name = "idTeacher", referencedColumnName = "idTeacher")
    private Teacher adviser;
    public Teacher getAdviser() { return adviser; }
    pubilc void setAdviser(Teacher adviser) { this.adviser = adviser; }
    // ...
}
```

В классе **Student** определяется поле типа **Teacher**, которое помечается аннотацией **@ManyToOne** с двумя аргументами: аргумент **name** задаёт имя столбца, являющегося *внешним ключом* в таблице **Student**, а аргумент **referencedColumnName** — имя столбца *первичного ключа* таблицы **Teacher**. Кроме того, для связи можно указать обязательность с помощью атрибута **nullable** (аналогично аннотации **@Column**, см. выше).

Движение по однонаправленной связи возможно только в одном направлении (в данном примере от студента к преподавателю). Например, если класс **Teacher** содержит свойство **name**, то вывести фамилию руководителя некоторого студента **s** можно следующим образом:

```
System.out.println(s.getAdviser().getName());
```

1.3. Двунаправленные ассоциации. Однонаправленную ассоциацию легко превратить в двунаправленную, добавив соответствующее поле в класс **Teacher**:

```
@Entity
public class Teacher implements Serializable {
    // ...
```

[4]Пример однонаправленной ассоциации для связи типа «многие ко многим» приведён в п. 1.4.

```
    @OneToMany(mappedBy = "adviser")
    private Set<Student> students;
    // ...
}
```

Поскольку рассматриваемая связь имеет тип «один ко многим», то на стороне преподавателя соответствующее поле объявляется как коллекция (в данном случае множество[5]) студентов. Для объявления ассоциации используется аннотация **@OneToMany** с аргументом **mappedBy**, определяющим *имя поля на противоположном конце ассоциации*.

Помимо множеств (**Set**) поддерживаются обычные коллекции (**Collection**), индексируемые коллекции (**List**) и ассоциативные массивы (**Map**). Порядок следования элементов при выборке их из БД можно указать с помощью аннотации **@OrderBy**. Следующий фрагмент кода определяет, что выборка студентов любого руководителя должна осуществляться в алфавитном порядке:

```
@OneToMany(mappedBy = "adviser")
@OrderBy("name")
private Set<Student> students;
```

При использовании двунаправленных ассоциаций следует учитывать важную их особенность:

> *При внесении изменений на одной стороне ассоциации ответственность за согласованное изменение другой стороны лежит на программисте.*

Для поддержания согласованности обеих сторон ассоциации можно применять различные методы [7]. В рассмотренном примере с классами **Student** и **Teacher** можно сделать setter для поля **adviser** недоступным извне пакета, а все изменения ассоциации выполнять только со стороны коллекции, предусмотрев для этого методы, которые помимо изменения коллекции будут устанавливать значение поля **adviser**. Примерная реализация[6]:

[5] При использовании коллекций типа «множество» следует помнить, что такие коллекции требуют определения методов **hashCode()** и **equals()** в классах-сущностях [6]. Определить метод **equals()** можно различными способами. Самый простой — сравнивать значения первичных ключей объектов. Существуют (и часто более предпочтительны) и другие подходы, рассматриваемые в документации Hibernate [3].

[6] Предполагается, что оба класса **Student** и **Teacher** принадлежат одному и тому же пакету.

```
@Entity
public class Student implements Serializable {
    // ...
    @ManyToOne
    @JoinColumn(name = "idTeacher", referencedColumnName = "idTeacher")
    private Teacher adviser;
    public Teacher getAdviser() { return adviser; }
    // Setter с доступом в пределах пакета, снаружи недоступен
    void setAdviser(Teacher adviser) { this.adviser = adviser; }
    // ...
}

@Entity
public class Teacher implements Serializable {
    // ...
    @OneToMany(mappedBy = "adviser")
    private Set<Student> students;
    // Getter для коллекции без возможности её изменения
    public Set<Student> getStudents() {
        return Collections.unmodifiableSet(students);
    }
    // Методы изменения коллекции выполняют также согласованное
    // изменение объекта класса Student
    public void addStudent(Student s) {
        if(s.getAdviser() != null) {
            s.getAdviser().students.remove(s);
        }
        students.add(s);
        s.setAdviser(this);
    }
    public void removeStudent(Student s) {
        if(students.contains(s)) {
            students.remove(s);
            s.setAdviser(null);
        }
    }
    // ...
}
```

Похожим образом можно реализовать возможность изменения ассоциации с противоположной стороны или даже с любой из сторон:

```
@Entity
public class Student implements Serializable {
    // ...
    @ManyToOne
    @JoinColumn(name = "idTeacher", referencedColumnName = "idTeacher")
    private Teacher adviser;
    public Teacher getAdviser() { return adviser; }
    // Setter изменяет поле adviser и выполняет также согласованное
    // изменение коллекции
    public void setAdviser(Teacher adviser) {
        if(this.adviser != null) {
            this.adviser.getStudentsInternal().remove(this);
        }
        if(adviser != null) {
            adviser.getStudentsInternal().add(this);
        }
        this.adviser = adviser;
    }
    // ...
}

@Entity
public class Teacher implements Serializable {
    // ...
    @OneToMany(mappedBy = "adviser")
    private Set<Student> students;
    // Getter для коллекции без возможности её изменения
    public Set<Student> getStudents() {
        return Collections.unmodifiableSet(students);
    }
    // Внутренний getter коллекции с возможностью её изменения
    // из методов того же самого пакета
    Set<Student> getStudentsInternal() { return students; }
    // Методы изменения коллекции передают вызов в объект класса Student
    public void addStudent(Student s) { s.setAdviser(this); }
    public void removeStudent(Student s) { s.setAdviser(null); }
    // ...
}
```

В некоторых задачах согласование обеих сторон ассоциации может быть некритичным. В частности, в web-приложениях часто можно безболезненно изменять только одну сторону ассоциации, т. к. время жизни

объектов, извлекаемых из БД, обычно ограничено обработкой одного запроса пользователя[7]. Использование несогласованных объектов в подобных случаях может сэкономить некоторое количество усилий. При этом следует учитывать следующий факт:

> *При сохранении изменений в БД всегда учитывается только одна сторона ассоциации.* Она называется *ведущей* (owner-side). Изменения на *обратной стороне* ассоциации (inverse-side) не учитываются.

Обратная сторона ассоциации — это та сторона, на которой определён атрибут **mappedBy**. Для связей типа «один ко многим» на обратной стороне ассоциации всегда находится коллекция, а для связей типа «многие ко многим» обратная сторона ассоциации может выбираться программистом произвольно.

Соответственно, при работе с несогласованными объектами следует изменять объект на ведущей стороне ассоциации, иначе изменения не будут сохранены.

В заключение вопроса о согласовании отметим, что в приложениях с долгоживущими объектами, а также в некоторых случаях при использовании кэширования наличие несогласованных объектов может быть недопустимым. Поэтому стратегию согласования изменений следует тщательно продумывать в каждой конкретной задаче.

1.4. Связи типа «многие ко многим». При определении ассоциаций для связей типа «многие ко многим» необходимо указать имя промежуточной таблицы, через которую происходит соединение, а также имена полей, являющихся внешними ключами в этой таблице. Например, связь между сущностями **Student** и **Subject** (предметы, изучаемые студентами) может быть определена со стороны класса **Student** следующим образом:

```
@ManyToMany
@JoinTable(name = "Student_Subject",
    joinColumns = {@JoinColumn(name = "idStudent")},
    inverseJoinColumns = {@JoinColumn(name = "idSubject")})
private Set<Subject> subjects;
```

Двунаправленная ассоциация определяется аналогично случаю связи типа «один ко многим». При её использовании оба класса будут содержать коллекции объектов другого класса. Выбор ведущей стороны (см. п. 1.3) при этом можно осуществлять произвольно.

[7] Более подробно о времени жизни объектов см. в п. 2.4.

2. Управление объектами

2.1. Фабрика сеансов Hibernate. Инициализация Hibernate заключается в создании *фабрики сеансов*. Фабрика сеансов — это объект класса **SessionFactory**, существующий в программе (как правило) в единственном экземпляре и обеспечивающий возможность получения *сеансов*. Сеансы представляют собой элементарные единицы взаимодействия приложения с базой данных и могут включать в себя *транзакции*.

Следующий фрагмент кода создаёт фабрику сеансов Hibernate:

```
SessionFactory sessionFactory = new AnnotationConfiguration()
        .configure().buildSessionFactory();
```

Здесь при создании фабрики сеансов происходит чтение конфигурации из файла **hibernate.hbm.xml** (по умолчанию), а также из аннотаций описанных в нём классов (благодаря использованию класса-фабрики **AnnotationConfiguration**). Hibernate поддерживает и другие способы конфигурирования, которые можно найти в документации. При создании фабрики сеансов происходит установление соединения с базой данных. При любых ошибках инициализации будет выброшено исключение типа **HibernateException**.

По окончании использования фабрики сеансов её необходимо закрыть с помощью метода **close()**:

```
sessionFactory.close();
```

2.2. Конфигурационный файл Hibernate. Конфигурационный файл Hibernate хранит параметры доступа к базе данных, информацию о классах-сущностях и отображении их полей на столбцы БД[8], а также некоторые другие параметры.

Рассмотрим пример конфигурационного файла:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
```

[8]Последнее только в том случае, когда объектно-реляционное отображение определяется посредством XML, а не аннотаций.

```
        <property name="connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <property name="connection.url">jdbc:mysql://localhost/test</property>
        <property name="connection.username">user</property>
        <property name="connection.password">password</property>
        <property name="dialect">
            org.hibernate.dialect.MySQLInnoDBDialect
        </property>
        <property name="cache.provider_class">
            org.hibernate.cache.NoCacheProvider
        </property>
        <property name="current_session_context_class">thread</property>
        <property name="show_sql">false</property>
        <property name="format_sql">false</property>
        <property name="hbm2ddl.auto">create</property>
        <mapping class="entities.Student"/>
        <mapping class="entities.Teacher"/>
        <mapping class="entities.Subject"/>
    </session–factory>
</hibernate–configuration>
```

Параметры **connection.driver_class**, **connection.url**, **connection.username**, **connection.password** и **dialect** ответственны за соединение с БД. Они определяют соответственно имя класса JDBC-драйвера (он должен находиться в одном из каталогов, перечисленных в переменной окружения **CLASSPATH**), URL базы данных, имя пользователя, пароль и SQL-диалект используемой БД. Приведённые в примере параметры используются для подключения к СУБД MySQL, использующей тип таблиц InnoDB.

Параметр **cache.provider_class** определяет имя класса, предоставляющего возможность кэширования объектов и/или результатов запросов к БД. Кэширование используется для повышения производительности работы приложения. В поставке Hibernate содержится несколько классов, которые поддерживают различные стратегии кэширования. Более подробную информацию об их использовании можно найти в руководстве Hibernate.

Параметр **current_session_context_class** определяет политику управления сеансами Hibernate в приложении. Более подробно данный вопрос рассмотрен в п. 2.3.

Параметры **show_sql** и **format_sql** определяют, следует ли выводить на консоль генерируемый Hibernate SQL-код. Такая возможность полезна для отладки.

Параметр **hbm2ddl.auto** позволяет использовать возможность Hibernate выполнять экспорт кода классов-сущностей схему БД. Данная возможность может быть полезна, если сначала разрабатывается структура классов приложения, а затем на её основе генерируется схема БД. Если схема БД уже создана (например, с помощью CASE-средства или путём непосредственного выполнения SQL-кода), то использовать данную возможность не следует. Возможные значения параметра:

- **create** — удалить старую схему данных и пересоздать её заново на основании кода классов, старое содержимое таблиц БД будет потеряно;
- **create-drop** — аналогично предыдущему варианту, но дополнительно удалить схему по окончании работы программы;
- **update** — обновить схему данных в соответствии с кодом классов-сущностей. Имеющиеся в БД данные будут по возможности сохранены. Следует иметь в виду, что данная возможность не всегда работает корректно, поскольку поддерживается не всеми JDBC-драйверами, а также поскольку выполнить инкрементное обновление схемы в некоторых случаях не представляется возможным.

Последняя часть конфигурационного файла перечисляет все классы-сущности, используемые приложением.

2.3. Управление сеансами и транзакциями. Когда фабрика сеансов создана, для получения сеансов можно использовать различные способы, однако самым распространённым является использование метода **getCurrentSession()**:

```
Session session = sessionFactory.getCurrentSession();
```

Этот метод возвращает сеанс Hibernate, связанный с текущим потоком выполнения[9]. Если такой сеанс к моменту вызова не существует, он будет создан. Такой способ работы с сеансами позволяет безболезненно решать проблему отсутствия поточной безопасности сеансов. Получен-

[9]Такой способ управления сеансами обусловлен используемой в данном издании конфигурацией Hibernate (в частности, параметром **current_session_context_class**). Он является наиболее простым и распространённым, хотя поддерживаются и другие возможности.

ный вызовом данного метода сеанс не следует использовать на других потоках выполнения.

Использование сеанса обычно сводится к выполнению на нём одной или нескольких транзакций. Транзакция начинается с вызова метода **beginTransaction()**, например:

```
session.beginTransaction();
```

Разумеется, в каждый момент времени на каждом сеансе может выполняться не более одной транзакции.

По окончании выполнения требуемых операций с БД транзакция либо принимается:

```
session.getTransaction().commit();
```

либо отклоняется:

```
session.getTransaction().rollback();
```

При закрытии транзакции автоматически закрывается и сеанс, в рамках которого она выполнялась.

2.4. Состояния объектов. Для управления данными, хранящимися в БД, Hibernate предлагает концепцию состояний объектов. Состояние объекта — это абстракция, описывающая отношение объекта к соответствующей ему записи в БД. При использовании средств объектно-реляционного отображения программист оперирует не конкретными SQL-операторами, а состояниями объектов. Изменение состояния приводит к автоматической генерации требуемого SQL-запроса к БД.

Hibernate определяет три состояния, в которых могут находиться объекты классов-сущностей:

- *Transient* (временное). Состояние объектов, не имеющих соответствующей им записи в БД. Временными являются объекты, создаваемые с помощью операции **new**.
- *Persistent* (постоянное). Состояние объектов, имеющих постоянное представление в виде записи в БД. Такие объекты *существуют только в пределах открытого сеанса Hibernate*. Содержимое полей этих объектов отслеживается, и по завершении сеанса внесённые изменения записываются в БД.
- *Detached* (отсоединённое). Состояние объектов, которые ранее находились в состоянии persistent, после закрытия соответствующего им сеанса. Содержимое объектов остаётся доступным и даже

изменяемым. При необходимости такие объекты могут быть переведены обратно в состояние persistent путём присоединения их к новому сеансу (см. п. 2.7).

Все операции, управляющие состоянием объектов, осуществляются путём вызова соответствующих методов объекта-сеанса (см. п. 2.3). Рассмотрим их.

2.5. Сохранение, получение и удаление объектов из БД. Метод сеанса **save()** сохраняет объект в БД[10], переводя его из состояния transient в состояние persistent. Например:

```
Student s = new Student("Иванов А. А.",
    new GregorianCalendar(1988, 3–1, 12).getTime(), // 12.03.1988
    "г. Ярославль, ...", "00–22–42");
session.save(s);
```

Для выполнения обратной операции — загрузки объекта из БД — может быть использован метод **get()** или **load()** объекта-сеанса. В качестве аргумента соответствующему методу передаётся объект класса **Class**, который используется для идентификации типа загружаемого объекта, и значение первичного ключа. Например, для загрузки объекта класса **Student** со значением первичного ключа **idStudent**, равным 1, можно использовать оператор:

```
Student s = (Student) session.get(Student.class, 1);
```

Отличие между этими методами состоит в том, что метод **get()** осуществляет немедленное обращение к БД в момент вызова и возвращает **null**, если соответствующего объекта в БД нет, тогда как метод **load()** всегда возвращает неинициализированный *объект-посредник* (proxy)[11]. Реальное получение данных из БД в этом случае будет происходить *в момент первого обращения* к методам объекта-посредника. Если требуемый объект не существует в БД, то в момент обращения будет выброшено исключение.

[10] Здесь и далее мы применяем абстракцию объектно-реляционного отношения, говоря о сохранении *объектов*, хотя на самом деле происходит сохранение *значений полей этих объектов* в соответствующих записях БД. Собственно, с точки зрения программиста, данная операция *выглядит* именно как сохранение объекта, позволяя абстрагироваться от реального SQL-запроса, посредством которого она выполняется. Аналогичное справедливо и для других операций.

[11] Если создание посредника невозможно (например, запрещено соответствующей аннотацией при определении класса-сущности), то выбрасывается исключение.

Объекты-посредники могут быть полезны в тех случаях, когда требуется выполнить какую-то операцию с объектом, не загружая его в память. В качестве примера рассмотрим удаление объекта с известным значением первичного ключа с помощью предназначенного для этой цели метода **delete()**:

```
Student s = (Student) session.load(Student.class, 1);
session.delete(s);
```

Применение в указанной ситуации **get()** вместо **load()** привело бы к выполнению ненужного SQL-запроса на получение полей удаляемого объекта. Сам бы объект **s** после выполнения **delete()** в этом случае перешёл из состояния persistent в состояние transient.

Существует также возможность принудительного перевода объекта из состояния persistent в состояние transient без его удаления из БД с помощью метода **evict()**.

2.6. Изменение объектов и синхронизация изменений. Внутри сеанса может существовать не более одного объекта с одним и тем же значением первичного ключа. Если объект был загружен посредством вызова метода **get()** или **load()**, то последующие попытки получения объекта с тем же самым значением первичного ключа будут возвращать тот же самый объект, что и первый вызов.

Все изменения таких объектов автоматически отслеживаются и сохраняются в БД. Операция обращения к БД в этом случае (а также во всех остальных случаях внесения изменений в БД) происходит не в момент изменения данных, а лишь при наступлении определённых событий, а именно:

- при фиксации активной транзакции с помощью метода **commit()** (см. также п. 2.3);
- при вызове метода принудительной синхронизации изменений:

  ```
  session.flush();
  ```

- при выполнении запроса на выборку данных (см. часть 3).

Учёт момента реального сохранения изменений может быть существенным, например, для обработки ошибок. Предположим, что в результате изменений объекта было нарушено ограничение на значение одного из полей. Исключение, сигнализирующее об этой ошибке, будет выброшено лишь в момент фиксации транзакции. Если же это нежелательно, необходимо выполнить принудительную синхронизацию изменений с помощью вызова метода **flush()** на объекте-сеансе.

В некоторых случаях возникает необходимость обратной синхронизации (перезагрузке из БД) объектов, находящихся в состоянии persistent. Для этого может использоваться метод **refresh()**:

```
session.refresh(s);
```

При его выполнении все внесённые и незафиксированные изменения объекта теряются, и все значения полей повторно загружаются из БД.

2.7. Использование отсоединённых (detached) объектов. Во многих случаях бывает необходимо выполнять изменения объектов не во время транзакции, а между ними. Например, в web-приложениях типичной является ситуация, когда объект загружается из БД и данные его передаются на редактирование пользователю в виде HTML-формы. После редактирования форма посылается на сервер и изменённые пользователем данные сохраняются в БД. Очевидно, что держать транзакцию открытой в течение всего времени редактирования формы пользователем нецелесообразно. Более того, специфика web-приложений такова, что пользователь может вообще не отправить заполненную форму на сервер (например, просто закрыв браузер). В этом случае транзакция может не быть закрыта вообще.

Для решения данной проблемы и используются отсоединённые объекты. После того как данные получены из БД, транзакция закрывается и полученный из БД объект переходит в состояние detached. Затем выполняются необходимые изменения объекта, после чего объект можно снова перевести в состояние persistent с помощью метода **update()** или **merge()**. Например:

```
session.beginTransaction();
Student s = (Student) session.get(Student.class, 1);
session.getTransaction().commit();
// ...
s.setName("Петров А. А."); // изменение объекта
session.beginTransaction();
session.update(s);
session.getTransaction().commit();
```

Отличие методов **update()** и **merge()** проявляется в том случае, когда новый сеанс *уже содержит объект, находящийся в состоянии persistent, с таким же значением первичного ключа, как и тот, который передан в качестве аргумента метода*. В этом случае метод **update()** просто

выбрасывает исключение, тогда как метод **merge()** *синхронизирует состояние объекта, уже находящегося в состоянии persistent, с состоянием объекта, переданного ему в качестве аргумента*. Следует иметь в виду, что передаваемый в метод **merge()** объект не переходит в состояние persistent, а остаётся в состоянии detached, а уже находящийся в состоянии persistent объект возвращается методом.

2.8. Отложенное получение (lazy fetch). Как уже отмечалось в п. 1.2–1.3, связи между таблицами при использовании ORM отображаются в ассоциации между классами-сущностями. На уровне приложения это позволяет с лёгкостью работать с данными разных таблиц без выполнения явных SQL-запросов на их соединение. Так, в примере п. 1.3 можно вывести на печать список всех студентов, для которых научным руководителем является преподаватель **t** (объект класса **Teacher**), следующим образом:

```
for(Student s : t.getStudents()) {
    System.out.println(s.getName());
}
```

Пример движения по этой же ассоциации в противоположном направлении приведён в п. 1.2.

В некоторых случаях (особенно при оптимизации приложения) требуется знать, каким образом и в какой момент происходит запрос данных связанных объектов, а также управлять этим процессом.

Hibernate предусматривает два режима получения связанных объектов, определяемых параметром **fetch** аннотаций, определяющих связь (***toOne** и ***toMany**).

- Значение параметра **fetch = FetchType.EAGER** означает, что получение содержимого связанных объектов будет производиться в момент получения запрошенного объекта.
- Значение параметра **fetch = FetchType.LAZY** означает использование так называемого *отложенного получения* (lazy fetch), при котором вместо связанных объектов возвращаются объекты-посредники. *Попытка обращения к любому методу такого объекта-посредника приводит к выполнению запроса на получение данных этого объекта, после чего объект-посредник становится неотличимым от реального объекта*. Если же доступ к данным объекта-посредника на протяжении всего времени жизни этого объекта не

осуществляется вообще, то соответствующий запрос к БД выполняться не будет.

По умолчанию используется следующая политика получения связанных объектов: для единичных связанных объектов используется режим **FetchType.EAGER**, а для связанных коллекций — режим **FetchType.LAZY**. В большинстве случаев эта политика является достаточно разумной, однако иногда может быть желательным её изменение с помощью соответствующих аннотаций.

2.9. Примеры использования различных режимов получения объектов. Пусть из БД получается объект-студент с известным значением первичного ключа и выводится фамилия этого студента и фамилия его руководителя:

```
Student student = (Student) session.get(Student.class, 1);
System.out.print("Студент " + student.getName());
System.out.println(" (Руководитель: " + student.getAdviser().getName() + ")");
```

В конфигурации по умолчанию выполнение такого кода будет приводить к выполнению следующего SQL-кода[12]:

```
select
    student0_.idStudent as idStudent0_1_,
    student0_.address as address0_1_,
    student0_.idTeacher as idTeacher0_1_,
    student0_.birthDate as birthDate0_1_,
    student0_.name as name0_1_,
    student0_.phone as phone0_1_,
    teacher1_.idTeacher as idTeacher1_0_,
    teacher1_.degree as degree1_0_,
    teacher1_.name as name1_0_,
    teacher1_.post as post1_0_
from Student student0_
left outer join Teacher teacher1_
     on student0_.idTeacher=teacher1_.idTeacher
where student0_.idStudent=?
```

Нетрудно видеть, что для получения данных связанного объекта используется один запрос с явным соединением таблиц (left outer join).

Если же в определении связи явно определить использование режима **FetchType.LAZY**:

[12]Включить вывод выполняемого SQL-кода в консоль можно с помощью параметров **show_sql** и **format_sql** конфигурационного файла Hibernate (см. п. 2.2).

```
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name = "idTeacher", referencedColumnName = "idTeacher")
private Teacher adviser;
```

то для получения связанного объекта-преподавателя будет выполнен отдельный SQL-запрос:

```
select
    student0_.idStudent as idStudent0_,
    student0_.address as address0_,
    student0_.idTeacher as idTeacher0_,
    student0_.birthDate as birthDate0_,
    student0_.name as name0_,
    student0_.phone as phone0_
from Student student0_
where student0_.name=?
select
    teacher0_.idTeacher as idTeacher1_0_,
    teacher0_.degree as degree1_0_,
    teacher0_.name as name1_0_,
    teacher0_.post as post1_0_
from Teacher teacher0_
where teacher0_.idTeacher=?
```

Этот второй запрос выполняется в момент вызова метода **getName()** на объекте-преподавателе.

Теперь рассмотрим случай получения связанной коллекции:

```
Teacher teacher = (Teacher) session.get(Teacher.class, 1);
for(Student s : teacher.getStudents()) {
    System.out.println(s.getName());
}
```

В этом случае по умолчанию используется режим **FetchType.LAZY**:

```
select
    teacher0_.idTeacher as idTeacher1_0_,
    teacher0_.degree as degree1_0_,
    teacher0_.name as name1_0_,
    teacher0_.post as post1_0_
from Teacher teacher0_
where teacher0_.idTeacher=?
select
    students0_.idTeacher as idTeacher1_,
```

```
    students0_.idStudent as idStudent1_,
    students0_.idStudent as idStudent0_0_,
    students0_.address as address0_0_,
    students0_.idTeacher as idTeacher0_0_,
    students0_.birthDate as birthDate0_0_,
    students0_.name as name0_0_,
    students0_.phone as phone0_0_
from Student students0_
where students0_.idTeacher=?
```

При явном задании режима **FetchType.EAGER** содержимое коллекции получается в том же запросе, что и основной объект:

```
select
    teacher0_.idTeacher as idTeacher1_1_,
    teacher0_.degree as degree1_1_,
    teacher0_.name as name1_1_,
    teacher0_.post as post1_1_,
    students1_.idTeacher as idTeacher3_,
    students1_.idStudent as idStudent3_,
    students1_.idStudent as idStudent0_0_,
    students1_.address as address0_0_,
    students1_.idTeacher as idTeacher0_0_,
    students1_.birthDate as birthDate0_0_,
    students1_.name as name0_0_,
    students1_.phone as phone0_0_
from Teacher teacher0_
left outer join Student students1_
on teacher0_.idTeacher=students1_.idTeacher
where teacher0_.idTeacher=?
```

Нетрудно заметить, что результат последнего запроса содержит много избыточной информации, так как в каждой его строке дублируется содержимое всех полей объекта-преподавателя. По этой причине использование режима **FetchType.EAGER** для коллекций не рекомендуется.

Установка режима получения с помощью аннотации распространяется на все случаи загрузки объектов соответствующего класса методом **get()**. Если же необходимо получить объект лишь при каком-то одном вызове, то можно вызвать произвольный метод этого объекта или коллекции.

Отметим также, что Hibernate поддерживает ещё более гибкий механизм получения объектов, чем здесь описано. В частности, возможно

отложенное получение отдельных полей объекта (полезно, например, в случае, когда соответствующее поле имеет тип BLOB и хранит большой бинарный объект), существуют также некоторые другие возможности.

2.10. Отложенное получение и отсоединённые объекты. Следует иметь в виду, что Hibernate *не поддерживает отложенное получение полей (в т. ч. коллекций) отсоединённых объектов*. Это означает, что, например, при выполнении следующего фрагмента кода[13]

```
session.beginTransaction();
Teacher teacher = (Teacher) session.get(Teacher.class, 1);
session.getTransaction().commit();
for(Student s : teacher.getStudents()) { // здесь будет выброшено исключение
    System.out.println(s.getName());
}
```

в момент вызова метода **getStudents()** будет выброшено исключение типа **LazyInitializationException**, поскольку в момент фиксации транзакции методом **commit()** объект **teacher** переходит в состояние detached, а его коллекция **students** остаётся неинициализированной.

Данная особенность накладывает существенное ограничение на возможности использования отсоединённых объектов, и в каждом конкретном случае следует выбирать адекватный способ его обхода.

Например, можно отказаться от отложенного получения, установив режим **FetchType.EAGER**, если это не приведёт к каким-либо другим неприятным последствиям (см. п. 2.9). Другой способ решения проблемы состоит в принудительной инициализации коллекции путём вставки любого оператора, требующего наличия в коллекции элементов, перед фиксацией транзакции. Например, можно использовать оператор

```
teacher.getStudents().size();
```

В этом случае происходит запрос к БД, коллекция заполняется и её содержимое остаётся доступным даже после закрытия транзакции и перехода объекта **teacher** в состояние detached.

2.11. Обработка ошибок. При возникновении ошибок Hibernate выбрасывает исключение типа **HibernateException**. Этот тип исключения относится к иерархии исключений **RuntimeException** и, следовательно, не обязателен для декларации и обработки (unchecked

[13] В предположении, что используется режим получения коллекций по умолчанию (см. п. 2.9).

exception)[14]. Тем не менее обработка таких исключений обычно необходима, поэтому важно знать, каким образом её правильно осуществлять.

Самая существенная особенность политики управления исключениями в Hibernate состоит в том, что ситуации, когда исключение выброшено, следует трактовать как *невосстановимые применительно к текущему сеансу*. Это означает, что после того, как исключение выброшено, текущий сеанс не может использоваться для выполнения каких-либо операций. Всё, что можно сделать в такой ситуации, — это закрыть сеанс, предварительно выполнив откат транзакции, если она в этот момент была активна:

```
Session session = sessionFactory.getCurrentSession();
try {
    // выполнение операций с БД
} catch(Exception e) {
    if(session.getTransaction().isActive()) {
        session.getTransaction().rollback();
    }
    throw e; // перебрасывание исключения на более высокий уровень
} finally {
    session.close(); // гарантированное закрытие сеанса
}
```

Легко видеть, что приведённый код на самом деле не обрабатывает выброшенное исключение, а просто перебрасывает его на более высокий уровень, предварительно закрыв сеанс, так что дальнейшее взаимодействие с БД посредством открытия нового сеанса будет абсолютно безопасным.

[14]Этим Hibernate существенно отличается от JDBC, где исключение, возникающее при работе JDBC, обязательно для обработки и декларирования. Это связано с тем, что разработчики Hibernate считают, что исключения такого рода должны перехватываться и обрабатываться на более высоком уровне, нежели уровень доступа к данным. Эта точка зрения имеет под собой определённые основания, особенно применительно к приложениям платформы J2EE. Более подробную информацию можно получить в документации Hibernate [3].

3. Построение запросов на выборку данных

Hibernate поддерживает несколько способов построения запросов на получение данных из БД. Основным из них является использование языка HQL (Hibernate Query Language), напоминающего по синтаксису язык SQL, но использующего преимущественно объектную семантику. Возможно также использование Criteria API — интерфейса программиста, позволяющего формировать запросы путём наложения требуемых ограничений на поля объектов, подлежащих извлечению из БД. Поддерживаются также механизм «запросов по образцу», а также непосредственное выполнение запросов на языке SQL. Последнее обычно используется лишь в случаях крайней необходимости для запросов, к производительности которых предъявляются повышенные требования.

Ниже рассмотрен первый способа построения запросов как наиболее распространённый. Описание остальных можно найти в руководстве Hibernate [3].

3.1. Выполнение HQL-запросов. Интерфейс **Query** определяет методы, предназначенные для выполнения запросов к базе данных. Он напоминает интерфейс **Statement** в JDBC (см., например, [6]), методы которого предназначены для выполнения обычных SQL-запросов. Главное отличие рассматриваемого интерфейса от интерфейса **Statement** в том, что методы выполнения запроса в большинстве случаев *возвращают объекты или коллекции объектов классов-сущностей*, тогда как аналогичные методы интерфейса **Statement** возвращают результирующий набор данных (интерфейс **ResultSet**) в виде плоской таблицы.

Для создания запроса используется метод **createQuery**, вызываемый на объекте-сеансе. Например:

```
Query q = session.createQuery("from Student");
```

Данная строка создаёт запрос на выборку всех записей из таблицы Student.

Для выполнения запроса могут быть использованы: метод **list()**, возвращающий результат запроса в виде коллекции объектов, метод **uniqueResult()**, возвращающий единичный объект (должен применяться лишь в тех случаях, когда заранее известно, что в результате запроса может быть возвращено не более одной результирующей строки), а также методы **scroll()** и **iterate()**, допускающие блочное извлечение данных

посредством разбиения одного запроса на несколько (обычно используются при загрузке больших результирующих наборов).

Результат рассмотренного выше запроса можно получить следующим образом:

```
List<Student> result = q.list();
```

Заметим, что при компиляции указанного кода будет выдано предупреждение компилятора

[имя java-файла] uses unchecked or unsafe operations.

Это предупреждение связано с тем, что приведение результата запроса к параметризованному типу **List<Student>** в Java в принципе невозможно выполнить безопасно. Подавить указанное предупреждение можно, поместив перед заголовком функции, содержащей указанное приведение типов, аннотацию

```
@SuppressWarnings("unchecked")
```

Запросы могут содержать параметры. После создания таких запросов необходимо установить значения параметров и лишь после этого выполнять запрос. Параметры могут быть именованными (начинаются с двоеточия) либо нумерованными (обозначаются знаками вопроса; *в отличие от JDBC нумерация параметров ведётся с нуля!*). Например, выборка всех студентов с заданной фамилией и инициалами может осуществляться следующим образом:

```
String studentName = "Иванов А. А.";
Query q = session.createQuery("from Student where name = :studentName");
q.setParameter("studentName", studentName);
List<Student> result = q.list();
```

Можно также воспользоваться тем, что метод **setParameter()** возвращает объект, на котором он вызывался, и соединить вызовы методов установки параметра и выполнения запроса в один оператор:

```
List<Student> result = q.setParameter("studentName", studentName).list();
```

или даже так:

```
List<Student> result = session.createQuery(
        "from Student where name =:studentName"
    ).setParameter("studentName", studentName).list();
```

3.2. Простейшие запросы на выборку данных. В предыдущем пункте приведены два простейших запроса на языке HQL. В них очевидным образом прослеживаются заимствования синтаксиса языка SQL. Эта ситуация является типичной, многие конструкции языка SQL работают также и в HQL. Приведём ещё несколько примеров, иллюстрирующих возможности этого языка, особо отмечая его объектную семантику.

В отличие от SQL, при выборке объектов из одной таблицы можно опустить перечисление получаемых объектов (часть запроса перед ключевым словом **from**). В полной форме последний из приведённых в предыдущем пункте запросов выглядит так:

```
select s
from Student as s
where s.name = :studentName
```

Обратите внимание, что часть запроса после ключевого поля **select** содержит не список полей (как это было бы в SQL) а *список объектов*, получаемых из БД.

Параметры операторов HQL также могут быть объектами. Например, если при выполнении некоторых действий из БД был получен объект **t** типа **Teacher**, то список всех студентов соответствующего преподавателя можно получить, выполнив следующий код:

```
Query q = getCurrentSession().createQuery(
        "from Student where adviser = :adviser");
List<Student> result = q.setParameter("adviser", t).list();
```

При выполнении данного кода генерируется следующий SQL-запрос:

```
select
    student0_.idStudent as idStudent0_,
    student0_.address as address0_,
    student0_.idTeacher as idTeacher0_,
    student0_.birthDate as birthDate0_,
    student0_.name as name0_,
    student0_.phone as phone0_
from
    Student student0_
where
    student0_.idTeacher=?
```

причём вместо «**?**» Hibernate автоматически вставляет значение первичного ключа объекта **t**[15].

Поиск всех студентов, которыми руководит определённый преподаватель, можно осуществить по фамилии преподавателя:

```
from Student where adviser.name = :adviserName
```

Генерируемый при выполнении запроса SQL-код

```
select
    student0_.idStudent as idStudent0_,
    student0_.address as address0_,
    student0_.idTeacher as idTeacher0_,
    student0_.birthDate as birthDate0_,
    student0_.name as name0_,
    student0_.phone as phone0_
from
    Student student0_,
    Teacher teacher1_
where
     student0_.idTeacher=teacher1_.idTeacher
     and teacher1_.name=?
```

содержит неявное соединение таблиц, тогда как в исходном HQL-запросе присутствует лишь движение вдоль ассоциации от класса **Student** к классу **Teacher** посредством поля **adviser**. В итоге запрос на языке HQL выглядит проще и допускает естественную интерпретацию в рамках объектно-ориентированного подхода.

3.3. Агрегатные функции и соединения. В HQL можно использовать агрегатные функции способом, аналогичным их использованию в SQL. Например оператор

```
select count(*) from Student
```

выведет общее количество студентов в БД.

Предположим теперь, что мы хотим получить количество студентов, которыми руководит каждый из преподавателей. Данную операцию можно выполнить несколькими способами.

Первый из них связан с использованием левого соединения (left join) и реализован «по мотивам» языка SQL:

[15]Заметим, что в точности такой же запрос выполняется в том случае, когда происходит первое обращение к элементам коллекции в режиме доступа **FetchType.LAZY** — см. п. 2.9.

```
select teacher, count(student)
from Teacher as teacher
left join teacher.students as student
group by teacher
```

Здесь формируются пары объектов (преподаватель, студент) для каждого студента и (преподаватель, **null**)[16] для каждого преподавателя, который не является руководителем ни одного студента. Затем происходит разбиение полученного набора на группы студентов, которыми руководит один и тот же преподаватель, после чего вычисляется количество таких пар в каждой группе, для которых второй компонент не равен **null**. SQL-код, генерируемый по приведённому выше HQL-запросу:

```
select
    teacher0_.idTeacher as col_0_0_,
    count(students1_.idStudent) as col_1_0_,
    teacher0_.idTeacher as idTeacher1_,
    teacher0_.degree as degree1_,
    teacher0_.name as name1_,
    teacher0_.post as post1_
from Teacher teacher0_
left outer join Student students1_
    on teacher0_.idTeacher=students1_.idTeacher
group by teacher0_.idTeacher
```

Сравнивая HQL-запрос с соответствующем ему SQL-аналогом, отметим, что в HQL-запросе отсутствует условие соединения таблиц (секция **on** в операторе соединения **left join**). Вместо него указан признак вхождения объектов класса **Student** в коллекцию **students** объектов класса **Teacher**:

```
left join teacher.students as student
```

Последнее очередной раз подчёркивает объектную ориентированность языка HQL, синтаксис которого описывает выборку объектов, а не просто полей из таблиц БД.

Существует более простой способ решения задачи получения количества студентов, которыми руководит каждый из преподавателей. Он основан на использовании ключевого слова **size**, обозначающего размер коллекции:

[16] Появление таких пар обусловлено использованием левого соединения. В данном случае оно необходимо для того, чтобы все преподаватели попали в результирующий список.

```
select teacher, teacher.students.size
from Teacher as teacher
```

Код, генерируемый Hibernate, в этом случае использует не соединение таблиц, а вложенный подзапрос:

```
select
    teacher0_.idTeacher as col_0_0_, (
        select count(students1_.idTeacher)
        from Student students1_
        where teacher0_.idTeacher=students1_.idTeacher
    ) as col_1_0_,
    teacher0_.idTeacher as idTeacher1_,
    teacher0_.degree as degree1_,
    teacher0_.name as name1_,
    teacher0_.post as post1_
from Teacher teacher0_
```

Заметим, что при выполнении последних двух HQL-запросов получается не коллекция объектов, а коллекция пар объектов. Эти пары представлены массивами типа **Object[]**[17], первый элемент которых содержит объект типа **Teacher**, а второй — объект типа **Long**. Вывести список фамилий преподавателей вместе с количеством руководимых ими студентов можно следующим образом:

```
List<Object[]> result2 = q.list();
for(Object[] a : result2) {
    System.out.println(((Teacher) a[0]).getName() + " --- " + a[1]);
}
```

Операция **inner join** также может использоваться для принудительного получения элементов коллекции (о режимах получения см. п. 2.9). В этом случае используется ключевое слово **fetch**. Следующий запрос выбирает всех преподавателей, фамилии которых начинаются на букву «и» и не имеют учёной степени. Кроме того, у всех объектов будут заполнены коллекции **students**.

```
from Teacher as teacher
inner join fetch teacher.students
where teacher.name like 'И%' and teacher.degree is null
```

[17] Кроме обычных массивов Hibernate поддерживает возможность использования массивов ассоциативных и даже объектов классов, в которые результат запроса упаковывается посредством вызова соответствующего конструктора.

Соответствующий SQL-запрос:

```
select
    teacher0_.idTeacher as idTeacher1_0_,
    students1_.idStudent as idStudent0_1_,
    teacher0_.degree as degree1_0_,
    teacher0_.name as name1_0_,
    teacher0_.post as post1_0_,
    students1_.address as address0_1_,
    students1_.idTeacher as idTeacher0_1_,
    students1_.birthDate as birthDate0_1_,
    students1_.name as name0_1_,
    students1_.phone as phone0_1_,
    students1_.idTeacher as idTeacher0__,
    students1_.idStudent as idStudent0__
from Teacher teacher0_
inner join Student students1_
    on teacher0_.idTeacher=students1_.idTeacher
where (teacher0_.name like 'И%') and (teacher0_.degree is null)
```

В заключение отметим, что возможности языка HQL намного богаче, чем рассмотренные в данных методических указаниях. Детальное их изложение, сопровождаемое большим количеством примеров, можно найти в документации [3].

Литература

[1] JSR 220: Enterprise JavaBeans, Version 3.0. Java Persistence API. — Sun Microsystems, 2006. — *http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html*.

[2] *Хемраджани, А.* Гибкая разработка приложений на Java с помощью Spring, Hibernate и Eclipse / А. Хемраджани. — М.: Вильямс, 2008. — 352 с.

[3] Hibernate Reference Documentation. — *http://www.hibernate.org*.

[4] *Дейт, К.* Введение в системы баз данных / К. Дейт. — 8-е изд. — М.: Вильямс, 2005. — 1328 с.

[5] *Рамбо, Д.* UML 2.0. Объектно-ориентированное моделирование и разработка / Д. Рамбо, М. Блаха. — СПб.: Питер, 2006. — 544 с.

[6] *Парамонов, И. В.* Язык программирования Java и Java-технологии / И. В. Парамонов. — Ярославль: ЯрГУ, 2006. — 92 с.

[7] *Фаулер, М.* UML. Основы. 3-е издание / М. Фаулер. — М.: Символ-плюс, 2005. — 192 с.

Приложение. Полный исходный текст примера

Ниже приведён полный код примера, демонстрирующего возможности Hibernate. Конфигурационный файл **hibernate.hbm.xml** приведён в п. 2.2.

```
/*
 * Student.java
 */
package entities;
import java.io.Serializable;
import java.util.*;
import javax.persistence.*;
@Entity
public class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer idStudent;
    private String name;
    private String address;
    @Temporal(TemporalType.TIMESTAMP)
    private Date birthDate;
    private String phone;
    @ManyToOne
    @JoinColumn(name = "idTeacher", referencedColumnName = "idTeacher")
    private Teacher adviser;
    @ManyToMany
    @JoinTable(name = "Student_Subject",
    joinColumns = {@JoinColumn(name = "idStudent")},
    inverseJoinColumns = {@JoinColumn(name = "idSubject")})
    private Set<Subject> subjects;
    public Student() {
        subjects = new HashSet<Subject>();
    }
    public Student(String name, Date birthDate, String address,
            String phone, Teacher adviser) {
        this.name = name;
        this.birthDate = birthDate;
        this.address = address;
```

```
        this.phone = phone;
        this.adviser = adviser;
        subjects = new HashSet<Subject>();
    }
    public Integer getIdStudent() { return idStudent; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getAddress() { return address; }
    public void setAddress(String address) { this.address = address; }
    public String getPhone() { return phone; }
    public void setPhone(String phone) { this.phone = phone; }
    public Date getBirthDate() { return birthDate; }
    public void setBirthDate(Date birthDate) { this.birthDate = birthDate; }
    public Teacher getAdviser() { return adviser; }
    public void setAdviser(Teacher adviser) { this.adviser = adviser; }
    public Set<Subject> getSubjects() { return subjects; }
    public void setSubjects(Set<Subject> subjects) { this.subjects = subjects; }
    public int hashCode() {
        int hash = 0;
        hash += (idStudent != null ? idStudent.hashCode() : 0);
        return hash;
    }
    public boolean equals(Object object) {
        if(!(object instanceof Student)) {
            return false;
        }
        Student other = (Student) object;
        if((this.idStudent == null && other.idStudent != null)
            || (this.idStudent != null && !this.idStudent
                .equals(other.idStudent))) {
            return false;
        }
        return true;
    }
    public String toString() {
        return "entities.Student[idStudent=" + idStudent + "]";
    }
}

/*
 * Teacher.java
```

```
 */
package entities;
import java.util.*;
import javax.persistence.*;
import java.io.Serializable;
@Entity
public class Teacher implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "idTeacher", nullable = false)
    private Integer idTeacher;
    private String degree;
    private String name;
    private String post;
    @OneToMany(mappedBy = "adviser")
    private Set<Student> students;
    public Teacher() {
        students = new HashSet<Student>();
    }
    public Teacher(String name, String degree, String post) {
        this.name = name;
        this.degree = degree;
        this.post = post;
    }
    public Integer getIdTeacher() { return idTeacher; }
    public String getDegree() { return degree; }
    public void setDegree(String degree) { this.degree = degree; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getPost() { return post; }
    public void setPost(String post) { this.post = post; }
    public Set<Student> getStudents() {
        return Collections.unmodifiableSet(students);
    }
    Set<Student> getStudentsInternal() { return students; }
    public void addStudent(Student s) { s.setAdviser(this); }
    public void removeStudent(Student s) { s.setAdviser(null); }
    protected void setStudents(Set<Student> students) {
        this.students = students;
    }
```

```
    public int hashCode() {
        int hash = 0;
        hash += (idTeacher != null ? idTeacher.hashCode() : 0);
        return hash;
    }
    public boolean equals(Object object) {
        if(!(object instanceof Teacher)) {
            return false;
        }
        Teacher other = (Teacher) object;
        if((this.idTeacher == null && other.idTeacher != null)
            || (this.idTeacher != null
            && !this.idTeacher.equals(other.idTeacher))) {
            return false;
        }
        return true;
    }
    public String toString() {
        return name;
    }
}

/*
 * Subject.java
 */
package entities;
import java.io.Serializable;
import javax.persistence.*;
@Entity
public class Subject implements Serializable {
    private static final long serialVersionUID = 3L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    int idSubject;
    String name;
    public Subject() { }
    public Subject(String name) { this.name = name; }
    public int getIdSubject() { return idSubject; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

```
/*
 * Main.java
 */
package util;
import entities.*;
import java.util.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
import org.apache.commons.logging.*;
public class Main {
    /** Фабрика сеансов Hibernate */
    private SessionFactory sessionFactory = null;
    /** Получение текущего сеанса */
    private Session getCurrentSession() {
        return sessionFactory.getCurrentSession();
    }
    /** Конструктор */
    public Main() {
        sessionFactory = new AnnotationConfiguration().configure()
            .buildSessionFactory();
    }
    /** Закрытие фабрики сеансов */
    private void closeSessionFactory() {
        sessionFactory.close();
    }
    /** Заполнение базы */
    private void fillDB() {
        getCurrentSession().beginTransaction(); // начало транзакции
        Subject subj1 = new Subject("математический анализ");
        Subject subj2 = new Subject("алгебра");
        getCurrentSession().save(subj1);
        getCurrentSession().save(subj2);
        Teacher t = new Teacher("Иванов И. И.", "к. ф.–м. н.", "доцент");
        getCurrentSession().save(t); // сохранение объекта в БД
        t = new Teacher("Петров П. П.", null, "ст. преподаватель");
        getCurrentSession().save(t);
        Student s = new Student("Сидоров А. Ц.",
                new GregorianCalendar(1988, 3 – 1, 12).getTime(),
                "адрес 1", "99–22–42", t);
        s.getSubjects().add(subj1);
        s.getSubjects().add(subj2);
```

```
        getCurrentSession().save(s);
        s = new Student("Смирнов З. З.",
                new GregorianCalendar(1990, 10 – 1, 16).getTime(),
                null, null, t);
        s.getSubjects().add(subj2);
        getCurrentSession().save(s);
        getCurrentSession().getTransaction().commit();
    }
    /** Демонстрация модификации объектов */
    private void modifyObjects() {
        getCurrentSession().beginTransaction(); // начало транзакции
        Student s = (Student) getCurrentSession().get(Student.class, 1);
        System.out.print("Студент " + s.getName());
        if(s.getAdviser() != null) {
            System.out.print(" (Руководитель: " +
                    s.getAdviser().getName() + ")");
        }
        s.setAdviser(null);
        getCurrentSession().getTransaction().commit();
    }
    /** Примеры запросов к базе */
    @SuppressWarnings("unchecked")
    private void queriesDemo() {
        getCurrentSession().beginTransaction();
        // Пример 1
        System.out.print("1. Поиск студента по Ф. И. О.: ");
        String studentName = "Сидоров А. Ц.";
        q = getCurrentSession().createQuery(
                "FROM Student WHERE name = :studentName");
        result = q.setParameter("studentName", studentName).list();
        for(Student s : result) {
            System.out.println(s.getName() + " " + s.getBirthDate() + " " +
                    s.getAddress() + " " + s.getPhone());
        }
        // Пример 2
        System.out.print("2. Общее количество студентов: ");
        q = getCurrentSession().createQuery(
                "SELECT COUNT(*) FROM Student");
        // Т. к. в результате может быть лишь одна строка, используем
        // uniqueResult() вместо list()
        System.out.println(q.uniqueResult());
```

```
        // Пример 3
        System.out.println("3. Количество студентов у преподавателей");
        q = getCurrentSession().createQuery(
                "SELECT teacher, teacher.students.size " +
                "FROM Teacher AS teacher");
        List<Object[]> result2 = q.list();
        for(Object[] a : result2) {
            System.out.println(((Teacher) a[0]).getName() + " --- " + a[1]);
        }
        // Пример 4
        System.out.println("4. Поиск студентов заданного преподавателя");
        result = getCurrentSession().createQuery(
                    "FROM Student WHERE adviser.name = :adviserName")
                .setString("adviserName", "Петров П. П.").list();
        for(Student s : result) {
            System.out.println(s.getName());
        }
        // Пример 5
        System.out.println("5. Поиск студентов всех преподавателей без "
                + " степени, с фамилиями, начинающимися на букву И");
        q = getCurrentSession().createQuery("from Teacher as teacher " +
                "inner join fetch teacher.students " +
                "where teacher.name like 'И%' and teacher.degree is null");
        result = q.list();
        getCurrentSession().getTransaction().commit();
    }
    /** Точка входа в программу */
    public static void main(String[] args) {
        Main main = new Main();
        main.fillDB();
        main.queriesDemo();
        main.printStudentsAndTeachers();
        main.modifyObjects();
        main.closeSessionFactory();
    }
}
```

Учебное издание

Парамонов Илья Вячеславович

Разработка приложений баз данных с использованием средств объектно-реляционного отображения

Методические указания

Редактор, корректор И. В. Бунакова
Компьютерный набор, вёрстка И. В. Парамонова

Подписано в печать 20.10.2008 г. Формат 60×84/16.
Бумага тип. Усл. печ. л. 2,56. Уч.-изд. л. 2,0.
Тираж 100 экз. Заказ

Оригинал-макет подготовлен в редакционно-издательском отделе
Ярославского государственного университета.

Отпечатано на ризографе.

Ярославский государственный университет
150000, Ярославль, ул. Советская, 14.