Лекция 16

JDBC

JDBC – это стандартный прикладной интерфейс (API) языка Java для организации взаимодействия между приложением и СУБД.

Это взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов.

В JDBC определяются четыре типа драйверов:

1. Тип 1 – драйвер, использующий другой прикладной интерфейс, в частности ODBC, для работы с СУБД (так называемый JDBC-ODBC – мост). Стандартный драйвер первого типа **sun.jdbc.odbc.JdbcOdbcDriver** входит в JSDK
2. Тип 2 – драйвер, работающий через нативные библиотеки (т.е. клиента) СУБД.
3. Тип 3 – драйвер, работающий по сетевому и независимому от СУБД протоколу с промежуточным Java-сервером, который, в свою очередь, подключается к нужной СУБД.
4. Тип 4 – сетевой драйвер, работающий напрямую с нужной СУБД и не требующий установки native-библиотек.

Предпочтение естественным образом отдается второму типу, однако если приложение выполняется на машине, на которой не предполагается установка клиента СУБД, то выбор производится между третьим и четвертым типами.

Причем четвертый тип работает напрямую с СУБД по ее протоколу, поэтому драйвер четвертого типа будет более эффективным по сравнению с третьим типом с точки зрения производительности.

Первый же тип, как правило, используется редко, т.е. в тех случаях, когда у СУБД нет своего драйвера JDBC, зато есть драйвер ODBC.

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД.

С помощью JDBC отсылаются SQL-запросы только к реляционным базам данных (БД), для которых существуют драйверы, знающие способ общения с реальным сервером базы данных.

Рассмотрим обычную последовательность действий при работе с базой данных:

1. Необходимо подключить библиотеку

   import java.sql.*;

2. Загрузка класса драйвера базы данных при отсутствии экземпляра этого класса(для JavaDB)

String driverName = "org.apache.derby.jdbc.EmbeddedDriver";

После этого выполняется собственно загрузка драйвера в память:

Class.forName(driverName).newInstance();

и становится возможным соединение с СУБД.

Эти же действия можно выполнить, импортируя библиотеку и создавая объект явно (например для DB2):

import COM.ibm.db2.jdbc.net.DB2Driver;

а затем,

new DB2Driver();

3. Установка соединения с БД в виде

```
String connectionURL=
      “jdbc:derby:NeuronNetwork;create=true";
//create=true –означает, что БД будет создана, если до этого она не создавалась
Connection conn =
                  DriverManager.getConnection(
                    connectionURL, "login", "password");
```

В результате будет возвращен объект Connection и будет одно установленное соединение с БД NeuronNetwork.

Класс **DriverManager** предоставляет средства для управления набором драйверов баз данных.

Методу **getConnection()** необходимо передать тип и физическое месторасположение БД, а также логин и пароль для доступа.

С помощью метода **registerDriver()** драйвера регистрируются, а методом **getDrivers()** можно получить список всех драйверов.

4. Создание объекта для передачи запросов

Statement st = conn.createStatement();

Объект класса **Statement** используется для выполнения запроса и команд SQL.

Могут применяться также операторы для выполнения подготовленных запросов и хранимых процедур **PreparedStatement** и **CallableStatement**.

Созданный объект можно использовать для выполнения запроса.

Основные методы класса **Statement:**

- **public void addBatch(String sql) throws SQLException**

Добавляет заданную команду SQL к текущему пакету команд.

- **public void cancel() throws SQLException**

В многопоточной среде с помощью этого метода можно потребовать прекращения всякой обработки, связанной с данным Statement.

В этом смысле метод аналогичен методу stop() для объектов Thread.

- **public boolean execute(String sql) throws SQLException**

public ResultSet executeQuery(String sql) throws SQLException
public int executeUpdate(String sql) throws SQLException

Выполняет Statement, передавая базе данных заданную SQL-строку.

Первый метод, execute(), позволяет вам выполнить Statement, когда неизвестно заранее, является SQL-строка запросом или обновлением.

Метод возвращает true, если команда создала результирующий набор.

Метод executeQuery() используется для выполнения запросов (на извлечение данных).

Он возвращает для обработки результирующий набор.

Метод executeUpdate() используется для выполнения обновлений.

Он возвращает количество обновленных строк.

- **public int[ ] executeBatch(String sql) throws SQLException**

Посылает базе данных пакет SQL-команд для выполнения.

И возвращает ко-во строк, которые были изменены

- **public ResultSet getResultSet() throws SQLException**

Метод возвращает текущий ResultSet.

Для каждого результата его следует вызывать только однажды.

Его не нужно вызывать после обращения к executeQuery(), возвращающему единственный результат.

- **public void close() throws SQLException**
  Вручную закрывает объект Statement.
  Обычно этого не требуется, так как Statement автоматически закрывается при закрытии связанного с ним объекта Connection.

Класс ResultSet

Этот класс представляет результирующий набор базы данных.

Он обеспечивает приложению построчный доступ к результатам запросов в базе данных.

Во время обработки запроса ResultSet поддерживает указатель на текущую обрабатываемую строку.

Приложение последовательно перемещается по результатам, пока они не будут все обработаны или не будет закрыт ResultSet.

Рассмотрим методы класса ResultSet

- **public boolean absolute(int row) throws SQLException**
  Метод перемещает курсор на заданное число строк от начала, если число положительно, и от конца - если отрицательно.

- **public void afterLast() throws SQLException**
  Этот метод перемещает курсор в конец результирующего набора за последнюю строку.

- **public void beforeFirst() throws SQLException**
  Этот метод перемещает курсор в начало результирующего набора перед первой строкой.

- **public void deleteRow() throws SQLException**
  Удаляет текущую строку из результирующего набора и базы данных.

- **public ResultSetMetaData getMetaData() throws SQLException**
  Предоставляет объект метаданных для данного ResultSet.
  Класс ResultSetMetaData содержит информацию о результирующие таблице, такую как количество столбцов, их заголовок и т.д.

- **public int getRow() throws SQLException**
  Возвращает номер текущей строки.

- **public Statement getStatement() throws SQLException**
  Возвращает экземпляр Statement, который произвел данный результирующий набор.

- **public boolean next() throws SQLException**
  public boolean previous() throws SQLException
  Эти методы позволяют переместиться в результирующем наборе на одну строку вперед или назад.

  Во вновь созданном результирующем наборе курсор устанавливается перед первой строкой, поэтому первое обращение к методу next() влечет позиционирование на первую строку.

  Эти методы возвращают true, если остается строка для дальнейшего перемещения.

  Если строк для обработки больше нет, возвращается false.

  Если открыт поток InputStream для предыдущей строки, он закрывается.

  Также очищается цепочка предупреждений SQLWarning.
- **public void close() throws SQLException**
  Осуществляет немедленное закрытие ResultSet вручную.

  Обычно этого не требуется, так как закрытие Statement, связанного с ResultSet, автоматически закрывает ResultSet.

Рассмотрим пример:

```
package mssql;
import java.sql.*;
public class Main {
    private static Connection con = null;
    private static String username = "name";
    private static String password = "pass";
    private static String URL ="jdbc:mysql://localhost:3306/mybase?profileSQL=true";
   // profileSQL=true –означает,  делать трассировку запросов и их
    выполнения с использованием логгера, который должен быть
    сконфигурирован
    public static void main(String[] args) throws SQLException{
    //Загружаем драйвер
    String driverName = "com.mysql.jdbc.Driver";
    Class.forName(driverName);
    //соединяемся
    con = DriverManager.getConnection(URL, username, password);
    if(con!=null)  System.out.println("Connection Successful !\n");
    if (con==null) System.exit(0);
```

```
   //Statement позволяет отправлять запросы базе данных
   Statement st = con.createStatement();
   //ResultSet получает результирующую таблицу
  ResultSet rs = st.executeQuery( "select hd from pc group by hd
                                                    having count(hd)>=2"
);
  //Resultset.getMetaData() получаем информацию   о
    результирующей таблице
  int x = rs.getMetaData().getColumnCount();
  while(rs.next()){
      for(int i=1; i<=x;i++){
        System.out.print(rs.getString(i) + "\t");
      }
     System.out.println();
   }
System.out.println();
 if(rs!=null)rs.close();
 if(st!=null)st.close();
 if(con!=null)con.close();   } }
```

Интерфейс **CallableStatement** расширяет возможности интерфейса **PreparedStatement** и обеспечивает выполнение хранимых процедур.

Стоит заметить, что **PreparedStatement** расширяет интерфейс **Statement.**

Хранимая процедура – это в общем случае именованная последовательность команд SQL, рассматриваемых как единое целое, и выполняющаяся в адресном пространстве процессов СУБД, который можно вызвать извне (в зависимости от политики доступа используемой СУБД).

В данном случае хранимая процедура будет рассматриваться в более узком смысле как последовательность команд SQL, хранимых в БД и

Интерфейс **CallableStatement** позволяет исполнять хранимые процедуры, которые находятся непосредственно в БД.

Одна из особенностей этого процесса в том, что **CallableStatement** способен обрабатывать не только входные (**IN**) параметры, но и выходящие (**OUT**) и смешанные (**INOUT**) параметры.

Тип выходного параметра должен быть зарегистрирован методом **registerOutParameter()**.

После установки входных и выходных параметров вызываются методы **execute()**, **executeQuery()** или **executeUpdate()**.

Рассмотрим пример использования **CallableStatement.**

Пусть в БД существует хранимая процедура **getempname**, которая по уникальному для каждой записи в таблице **employee** числу SSN будет возвращать соответствующее ему имя:

```
Delimiter //
CREATE PROCEDURE `getempname`
  (IN emp_ssn INT, OUT emp_name VARCHAR(30))
                                                AS
BEGIN
SELECT name
INTO emp_name
FROM employee
WHERE SSN = EMP_SSN;
END;
//
```

Тогда для получения имени служащего **employee** через вызов данной процедуры необходимо исполнить java-код вида:

```
String SQL = "{call getempname (?,?)}";
CallableStatement cs = conn.prepareCall(SQL);
int ssn=822301;
cs.setInt(1,ssn);
//регистрация выходящего параметра
cs.registerOutParameter(2,
                        java.sql.Types.VARCHAR);
cs.execute();
String empName = cs.getString(2);
System.out.println("Employee with SSN:" + ssn
                        + " is " + empName);
```

В результате будет выведено, приблизительно следующее:

Employee with SSN:822301 is Spiridonov

В JDBC также существует механизм batch-команд, который позволяет запускать на исполнение в БД массив запросов SQL вместе, как одну единицу.

```
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO employee VALUES
                                    (10, 'Joe ')");
stmt.addBatch("INSERT INTO location VALUES
                                    (260, ‘Kiev’)");
stmt.addBatch("INSERT INTO emp_dept VALUES
                                    (1000, 260)");
```

//выполняем запрос

int[] updateCounts = stmt.executeBatch();

В массиве updateCounts содержаться числа:

1.>=0 – означает, что запрос выполнился успешно и в данном элементе массива содержится ко-во строк в таблице, измененные запросом.

2. SUCCESS_NO_INFO – означает, что команда выполнилась успешно, но ко-во строк в таблице, которые были изменены запросом, неизвестно.

Если один из запросов выполнен не будет, то будет возбуждено исключение BatchUpdateException.

При этом JDBC может продолжить выполнение остальных запросов.

3. EXECUTE_FAILED – означает, что выполнение запроса провалилось.

Если используется объект PreparedStatement, batch-команда состоит из параметризованного SQL-запроса и ассоциируемого с ним множества параметров.

Рассмотрим пример:

```
try {
Employee[] employees = new Employee[10];
PreparedStatement statement=
                                    con.prepareStatement(
  "INSERT INTO employee VALUES (?,?,?,?,?)");
for (int i = 0; i < employees.length; i++) {
  Employee currEmployee = employees[i];
  statement.setInt(1, currEmployee.getSSN());
  statement.setString(2, currEmployee.getName());
  statement.setDouble(3, currEmployee.getSalary());
  statement.setString(4,currEmployee.getHireDate());
  statement.setInt(5, currEmployee.getLoc_Id());
  statement.addBatch();} }
```

```
int [] updateCounts = statement.executeBatch();
 } catch (BatchUpdateException e) {
   e.printStackTrace(); }
```

Транзакции

Транзакцию (деловую операцию) определяют как единицу работы, обладающую свойствами ACID:

- **Атомарность** – две или более операций выполняются все или не выполняется ни одна.

  Успешно завершенные транзакции фиксируются, в случае неудачного завершения происходит откат всей транзакции.

- **Согласованность** – при возникновении сбоя система возвращается в состояние до начала неудавшейся транзакции.

  Если транзакция завершается успешно, то проверка согласованности удостоверяется в успешном завершении всех операций транзакции.
- **Изолированность** – во время выполнения транзакции все объекты-сущности, участвующие в ней, должны быть синхронизированы.
- **Долговечность** – все изменения, произведенные с данными во время транзакции, сохраняются, например, в базе данных.

Для фиксации результатов работы SQL-операторов, логически выполняемых в рамках некоторой транзакции, используется SQL-оператор **COMMIT**.

В API JDBC эта операция выполняется по умолчанию после каждого вызова методов **executeQuery()** и **executeUpdate().**

Если же необходимо сгруппировать запросы и только после этого выполнить операцию COMMIT, сначала вызывается метод **setAutoCommit(boolean param)** интерфейса Connection с параметром false, в результате выполнения которого текущее соединение с БД переходит в режим неавтоматического подтверждения операций.

После этого выполнение любого запроса на изменение информации в таблицах базы данных не приведет к необратимым последствиям, пока операция COMMIT не будет выполнена непосредственно.

Подтверждает выполнение SQL-запросов метод **commit()** интерфейса Connection, в результате действия которого все изменения таблицы производятся как одно логическое действие.

Если же транзакция не выполнена, то методом rollback() отменяются действия всех запросов SQL, начиная от последнего вызова commit().

```
try{
  Connection cn=…….;
  cn.setAutoCommit(false);
  Statement st = cn.createStatement();
 String upd =
       "INSERT INTO student (id, name) VALUES
                                ('"+ id + "', '" + name + "')";
st.executeUpdate(upd);
……..
cn.commit(); //подтверждение.
} catch(SQLException e){
    cn.rollback();
……..
}
```

Для транзакций существует несколько типов чтения:

- **Грязное чтение** (dirty reads) происходит, когда транзакциям разрешено видеть несохраненные изменения данных.

  Иными словами, изменения, сделанные в одной транзакции, видны вне ее до того, как она была сохранена.

  Если изменения не будут сохранены, то, вероятно, другие транзакции выполняли работу на основе некорректных данных;
- **Непроверяющееся чтение** (nonrepeatable reads) происходит, когда транзакция А читает строку, транзакция В изменяет эту строку, транзакция А читает ту же строку и получает обновленные данные;

- **Фантомное чтение** (phantom reads) происходит, когда транзакция A считывает все строки, удовлетворяющие WHERE-условию, транзакция B вставляет новую или удаляет одну из строк, которая удовлетворяет этому условию, транзакция A еще раз считывает все строки, удовлетворяющие **WHERE**-условию, уже вместе с новой строкой или недосчитавшись старой.

JDBC удовлетворяет четырем уровням изоляции транзакций, определенным в стандарте SQL.

Уровни изоляции транзакций определены в виде констант интерфейса **Connection** (по возрастанию уровня ограничения):

- **TRANSACTION_NONE** – информирует о том, что драйвер не поддерживает транзакции;
- **TRANSACTION_READ_UNCOMMITTED** – позволяет транзакциям видеть несохраненные изменения данных, что разрешает грязное, непроверяющееся и фантомное чтения;

- **TRANSACTION_READ_COMMITTED** – означает, что любое изменение, сделанное в транзакции, не видно вне неё, пока она не сохранена.

  Это предотвращает грязное чтение, но разрешает непроверяющееся и фантомное;
- **TRANSACTION_REPEATABLE_READ** – запрещает грязное и непроверяющееся, но фантомное чтение разрешено;
- **TRANSACTION_SERIALIZABLE** – определяет, что грязное, непроверяющееся и фантомное чтения запрещены.

Метод

boolean supportsTransactionIsolationLevel (int level)

интерфейса **DatabaseMetaData** определяет, поддерживается ли заданный уровень изоляции транзакций.

В свою очередь, методы интерфейса **Connection** определяют доступ к уровню изоляции:

int getTransactionIsolation() – возвращает текущий уровень изоляции;

void setTransactionIsolation(int level) – устанавливает нужный уровень.

Точки сохранения

Точки сохранения дают дополнительный контроль над транзакциями.

Установкой точки сохранения обозначается логическая точка внутри транзакции, которая может быть использована для отката данных.

Таким образом, если произойдет ошибка, можно вызвать метод **rollback()** для отмены всех изменений, которые были сделаны после точки сохранения.

Метод **boolean supportsSavepoints()** интерфейса **DatabaseMetaData** используется для того, чтобы определить, поддерживает ли точки сохранения драйвер JDBC и сама СУБД.

Методы **setSavepoint(String name)** и **setSavepoint()** (оба возвращают объект **Savepoint**) интерфейса **Connection** используются для установки именованной или неименованной точки сохранения во время текущей транзакции.

Рассмотрим пример:

```
……….
Class.forName ("com.mysql.jdbc.Driver")
                                        .newInstance();
Connection cn = null;
Savepoint savepoint = null;
cn = DriverManager.getConnection(
          "jdbc:mysql://localhost/db3","root","pass");
cn.setAutoCommit(false);
Statement stmt = cn.createStatement();
String trueSQL =  "INSERT INTO emp
                              (id,name,surname,salary)" +
                  "VALUES(2607,'Петя','Иванов',540)";
stmt.executeUpdate(trueSQL);
```

```
//установка точки сохранения
savepoint = cn.setSavepoint("savepoint1");
//выполнение некорректного запроса
String wrongSQL =
        "INSERT INTO (id,name,surname,salary) "
            + "VALUES(2607,'Петя','Иванов',540)";
stmt.executeUpdate(wrongSQL);
} catch (SQLException ex){
    cn.rollback(savepoint);
 ……….
}
```

Рассмотрим пример, двухмерный массив на основе базы данных.

```
import java.sql.*;
import java.util.*;
  //здесь хранится имя массива, ко-во строк и ко-во столбцов.
  class MasSize  {
      private int columns;
      private int rows;
      private String name;
    public void setName(String n){name=new String(n);}
    public void setColumns(int n){columns=n;}
    public void setRows(int n){rows=n;}
    public String getName(){return name;}
    public int getColumns(){return columns;}
    public int getRows(){return rows;}
    public boolean equals(Object o){
      MasSize o1=(MasSize) o;
     if(name.equals(o1.name)) return true;
      else  return false; }  };
```

```
public class DB{
    private String name;  //имя базы данных
    private LinkedList tables=new LinkedList();//список имен
                                                массивов в базе данных
    private Connection conn;
 public DB(String preambula, String n, String driver){
    name=new String(n);
    String connectionURL=preambula+”””
                                          +name+";create=true";
    try{
     Class.forName( driver);
     conn = DriverManager.getConnection(connectionURL);
   } catch(Exception e){
        System.out.println("In function DB");
        e.printStackTrace();
  }
```

```
  public void createMas(String name_of_massive,
                                        int number_of_rows,
                                        int number_of_columns){
      try {
//добавляем имя массива в список имен массивов
          MasSize m=new MasSize();
          m.setName(name_of_massive);
          m.setColumns(number_of_columns);
          m.setRows(number_of_rows);
          if(!tables.contains(m)) { tables.add(m);}
          else  {
           System.out.println("Massive with such name
                                                 already  exists");
            return;
}
```

```
//создание таблицы
 String createString="CREATE TABLE
                                        "+name_of_massive+
          " ( NUMBER int  generated by default as identity
                (START WITH 1, INCREMENT BY 1), "+
                                          "w0 DOUBLE )";
Statement stmt = conn.createStatement();
stmt.executeUpdate(createString);
for(int i=1; i< number_of_columns; i++)  {
   //добавление столбцов в таблицу
  String stradd="ALTER TABLE "+name_of_massive+
                  “ADD"+"w"+String.valueOf(i)+" DOUBLE";
   stmt.executeUpdate(stradd);    }
```

```
for(int j=0;j<number_of_rows;j++){
  //добавление строк в таблицу
 String str_query="insert into “
                        +name_of_massive+" ( w0 )"+
                                    " values  "+"( ?)";
 PreparedStatement psInsert =
                conn.prepareStatement(str_query);
 psInsert.setDouble(1,0.0);
 psInsert.executeUpdate();}}
catch (Exception e) {
    System.out.println("In function createMas");
    e.printStackTrace();} }
```

```
public void set(String name_of_massive, int row,
                                        int column, double value){
  //изменение значения элемента с координатами row и
  column на значение value
  try{
  //проверка существует ли массив с именем
  name_of_massive
     MasSize m=new MasSize();
     m.setName(name_of_massive);
     m.setColumns(0);
     m.setRows(0);
     if(!tables.contains(m)) {
       System.out.println("there is no such massive");
       return; }
```

```
String str="update "+name_of_massive+
        "set"+"w"+String.valueOf(column)+"=?"+
      "where NUMBER="+String.valueOf(row+1);
PreparedStatement psInsert =
                        conn.prepareStatement(str);
psInsert.setDouble(1,value);
psInsert.executeUpdate();}
catch(Exception e){
      System.out.println("In function set");
      e.printStackTrace();}
```

```
public double get(String name_of_massive,
                                    int row, int column){
  //получение элемента массива с
 координатами  row и column
 try{
  MasSize m=new MasSize();
      m.setName(name_of_massive);
      m.setColumns(0);
      m.setRows(0);
     if(!tables.contains(m)){
        System.out.println("there is no such
                                          massive");
         return -1; }
```

```
String str="select"
        +"w"+String.valueOf(column)+"from"
                                    +name_of_massive+
     "where NUMBER="+String.valueOf(row+1);
  Statement stmt2 = conn.createStatement();
  ResultSet rs = stmt2.executeQuery(str);
  rs.next();
  double d=rs.getDouble(1);
  rs.close();
  return d;}
catch(Exception e){
          System.out.println("In function get");
          e.printStackTrace();
          return -1;}}
```

```
 public int Ncolumns(String name_of_massive) {
   //кол-во столбцов в массиве  name_of_massive
 try{
    MasSize m=new MasSize();
     m.setName(name_of_massive);
     m.setColumns(0);
     m.setRows(0);
    if(!tables.contains(m)) {
     System.out.println("there is no such massive");
     return -1; }
    MasSize m1=(MasSize)
                              tables.get(tables.indexOf(m));
    return m1.getColumns();}
catch(Exception e){
   System.out.println("In function Ncolumns");
   e.printStackTrace();
   return -1;}}
```

```
public int Nrows(String name_of_massive){
    //кол-во строк в массиве  name_of_massiv
   try{
      MasSize m=new MasSize();
      m.setName(name_of_massive);
      m.setColumns(0);
      m.setRows(0);
      if(!tables.contains(m)) {
        System.out.println("there is no such massive");
        return -1;    }
     MasSize m1=(MasSize)
                              tables.get(tables.indexOf(m));
     return m1.getRows();}
catch(Exception e){
    System.out.println("In function Ncolumns");
    e.printStackTrace();
    return -1;}}
```

```
 public void delete_mas(String name_of_massive){
  //удаление массива name_of_massive из базы данных
  try{
      MasSize m=new MasSize();
      m.setName(name_of_massive);
      m.setColumns(0);
      m.setRows(0);
      if(!tables.contains(m)) {
        System.out.println("there is no such massive");
        return; }
     String str="drop table "+name_of_massive;
     Statement stmt2 = conn.createStatement();
     stmt2.executeUpdate(str);}
catch(Exception e){
     System.out.println("In function delete_mas");
     e.printStackTrace();}}}
```

Использование данного класса может выглядеть следующим образом:

```
class m1{
    public static void main(String[] args){
 DB data=new   DB("jdbc:derby:","NeuronNetwork",
                                        "org.apache.derby.jdbc.EmbeddedDriver");
      data.createMas("mas",3,3);
      data.set("mas",1,1,3.56);
      double d=data.get("mas",1,1);
      System.out.println(d);
      data.set("mas",1,2,4.78);
      d=data.get("mas",1,2);
      System.out.println(d);
      int r=data.Ncolumns("mas");
      int c=data.Nrows("mas");
      System.out.println("rows= "+r);
      System.out.println("cols= "+c);
      data.set("mas",1,2,5.5);
      d=data.get("mas",1,2);
      System.out.println(d);  }
}
```

Метаданные

С помощью методов интерфейсов **ResultSetMetaData** и **DatabaseMetaData**

можно получить список таблиц, определить типы, свойства и количество столбцов БД.

Получить объект **ResultSetMetaData** можно следующим образом:

ResultSetMetaData rsMetaData=rs.getMetaData();

где rs имеет тип ResultSet.

Рассмотрим некоторые методы интерфейса ResultSetMetaData:

int getColumnCount() – возвращает число столбцов набора результатов объекта ResultSet;

String getColumnName(int column) – возвращает имя указанного столбца объекта ResultSet;

int getColumnType(int column) – возвращает тип данных указанного столбца объекта ResultSet.

Получить объект **DatabaseMetaData** можно следующим образом:

DatabaseMetaData dbMetaData = conn.getMetaData();

где cn объект класса Connection.

Рассмотрим некоторые методы интерфейса **DatabaseMetaData**:

String getDatabaseProductName() – возвращает название СУБД;

String getDatabaseProductVersion() – возвращает номер версии СУБД

String getDriverName() – возвращает имя драйвера JDBC;

String getUserName() – возвращает имя пользователя БД;

String getURL() – возвращает местонахождение источника данных;

ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types) – возвращает набор типов таблиц, доступных для данной БД

catalog - Значение String, содержащее имя каталога. Задание значения NULL для этого параметра указывает на то, что имя каталога использовать не нужно.

schema - Значение String, содержащее шаблон имени схемы. Задание значения NULL для этого параметра указывает на то, что имя схемы использовать не нужно.

tableName - Значение String, содержащее шаблон имени таблицы.

types - Массив строковых значений, содержащий включаемые типы таблиц. Значение NULL показывает, что нужно включать все типы таблиц. Типы таблиц можно получить вызвав функцию getTableTypes(). Бывают следующие типы "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM".

Рассмотрим пример.

Возврат описания таблицы Person.Contact в базы данных AdventureWorks.

```
public static void executeGetTables(Connection con) {
  try {
     DatabaseMetaData dbmd = con.getMetaData();
     ResultSet rs = dbmd.getTables("AdventureWorks",
                                   "Person", "Contact", null);
    ResultSetMetaData rsmd = rs.getMetaData();
    int cols = rsmd.getColumnCount();
    while(rs.next()) {
        for (int i = 1; i <= cols; i++) {
            System.out.println(rs.getString(i)); }
     }
     rs.close();
   } catch (Exception e) { e.printStackTrace(); }
}
```

Результирующий набор, возвращаемый методом **getTables**, включает следующие данные:

TABLE_CAT - Имя базы данных, в которой расположена указанная таблица

TABLE_SCHEM- Имя схемы для таблицы.

TABLE_NAME- Имя таблицы.

TABLE_TYPE - Табличный тип.

REMARKS- Описание таблицы.

Пул соединений

При большом количестве клиентов приложения к БД этого приложения выполняется большое количество запросов.

Соединение с БД является дорогостоящей (по требуемым ресурсам) операцией.

Эффективным способом решения данной проблемы является организация пула используемых соединений, которые не закрываются физически, а хранятся в очереди и предоставляются повторно для других запросов.

Разделяемый доступ к источнику данных можно организовать путем объявления статической переменной типа **DataSource** из пакета **javax.sql**.

Источник данных типа DataSource – это компонент, предоставляющий соединение с приложением СУБД.

Класс **InitialContext**, как часть JNDI API, обеспечивает работу с каталогом именованных объектов.

В этом каталоге можно связать объект источника данных DataSource с некоторым именем (не только с именем БД, но и вообще с любым), предварительно создав объект DataSource.

Затем созданный объект можно получить с помощью метода **lookup()** по его имени.

Методу lookup() передается имя, всегда начинающееся с имени корневого контекста(Если пул организовывается с помощью Tomcat)

```
javax.naming.Context ct =
                    new javax.naming.InitialContext();
DataSource ds =
          (DataSource)ct.lookup(“java:jdbc/mybd”);
Connection cn = ds.getConnection(“name”, “pass”);
```

После выполнения запроса соединение завершается и его объект возвращается обратно в пул вызовом:

```
cn.close();
```

Пул соединений Tomcat

Для организации пула необходимо изменить файл server.xml, который находится в папке conf (в директории, в которой установлен Tomcat):

```
<Context docBase="FirstProject" path="/FirstProject“
                                                    reloadable=“true“
            source="com.ibm.etools.webtools.server:FirstProject">
<!—создание пул соединений для СУБД MySQL -->
<Resource auth="Container" name="jdbc/db1“
                                                    type="javax.sql.DataSource"/>

<ResourceParams name="jdbc/db1">
<parameter>
    <name>factory</name>
    <value>
        org.apache.commons.dbcp.BasicDataSourceFactory
    </value>
</parameter>
<parameter>
    <name>driverClassName</name>
    <value>org.gjt.mm.mysql.Driver</value>
</parameter>
```

```
<parameter>
  <name>username</name>
   <value>root</value>
</parameter>
<parameter>
  <name>password</name>
  <value>pass</value>
</parameter>
<parameter>
<!--maxActive - максимальное количество
соединений, которые будут содержаться в пуле.-->
  <name>maxActive</name>
   <value>500</value>
</parameter>
<parameter>
<!--maxIdle - максимальное количество простаивающих соединений,
    которые будут оставаться в пуле. При значении этого параметра
    равным нулю, ограничений не будет.-->
  <name>maxIdle</name>
  <value>10</value>
</parameter>
```

```
<parameter>
<!--maxWait - если время ожидания соединения превысит значение параметра maxWait
    (в миллисекундах), пользователь получит Exception. При значении maxWait равным
    -1, время ожидания не ограничено.-->
  <name>maxWait</name>
  <value>10000</value>
</parameter>
<parameter>
<!-- removeAbandoned - при установке этого парамера в true, заброшенные
    соединения будут освобождены, когда истечет время, учтановленное в
    параметре  removeAbandonedTimeout .-->
  <name>removeAbandoned</name>
  <value>true</value>
</parameter>
<parameter>
<!-- removeAbandonedTimeout - этот параметр указывает в секундах время, через
    которое любое простаивающее соединение будет считаться заброшенным.-->
  <name>removeAbandonedTimeout</name>
  <value>60</value>
</parameter>
<parameter>
  <name>logAbandoned</name>
  <value>true</value>
</parameter>
</ResourceParams>
</Context>
```

Некоторые производители СУБД для облегчения создания пула соединений определяют собственный класс на основе интерфейса DataSource.

В этом случае пул соединений может быть создан, например, следующим образом:

```
import COM.ibm.db2.jdbc.DB2DataSource;
...
DB2DataSource ds = new DB2DataSource();
ds.setServerName("//localhost:6061/mybd");
Connection cn =
                ds.getConnection("db2adm","pass");
```

Драйвер определяется автоматически в объекте DB2DataSource.

Также существует специальный пакет для создания пула соединений с БД **DBPool (необходимы пакеты DBPool-5.0 и commons-logging-1.1.1-bin).** Можно также использовать пул DBCP

Рассмотрим пример использования DBpool 5.0

```
……..
Connection conn = null;
  try{
    String userName = "muser";
    String password = "qwertyui";
    String url = "jdbc:mysql://localhost/mybase";
    Class.forName ("com.mysql.jdbc.Driver").newInstance();
//Создаем пул
   ConnectionPool pool=new ConnectionPool("local",1,10,20,180000,url,
                                                    userName,password);
//Здесь “local”- имя пула, 1-минимальное количество соединений в пуле, 10-
    максимальное ко-во соединений в пуле, 20-максимальное количество
    соединений, которое может быть создано. Избыточные соединения будут
    уничтожены, когда их вернут в пул.
//Создаем соединение, если оно не было создано или достаем из пула
  conn=pool.getConnection();
```

```
String SQL="{call getmname(?,?)}";
CallableStatement cs = conn.prepareCall(SQL);
int ssn=251;
cs.setInt(1, ssn);
cs.registerOutParameter(2, Types.VARCHAR);
cs.execute();
String name=cs.getString(2);
System.out.println("Employee with SSN:" + ssn+
                                    " is " + name);
conn.close();
……..//снова используем соединение
//закрываем пул
pool.release();
……..
```

Паттерн Data Access Object

Data Access Object (DAO) используется для абстрагирования и инкапсулирования доступа к источнику данных.

Использующие DAO бизнес-компоненты работают с более простым интерфейсом, предоставляемым объектом DAO своим клиентам.

DAO полностью скрывает детали реализации источника данных от клиентов.

Поскольку при изменениях реализации источника данных представляемый DAO интерфейс не изменяется, этот паттерн дает возможность DAO принимать различные схемы хранилищ без влияния на клиенты или бизнес-компоненты.

Диаграмма классов для DAO имеет вид:

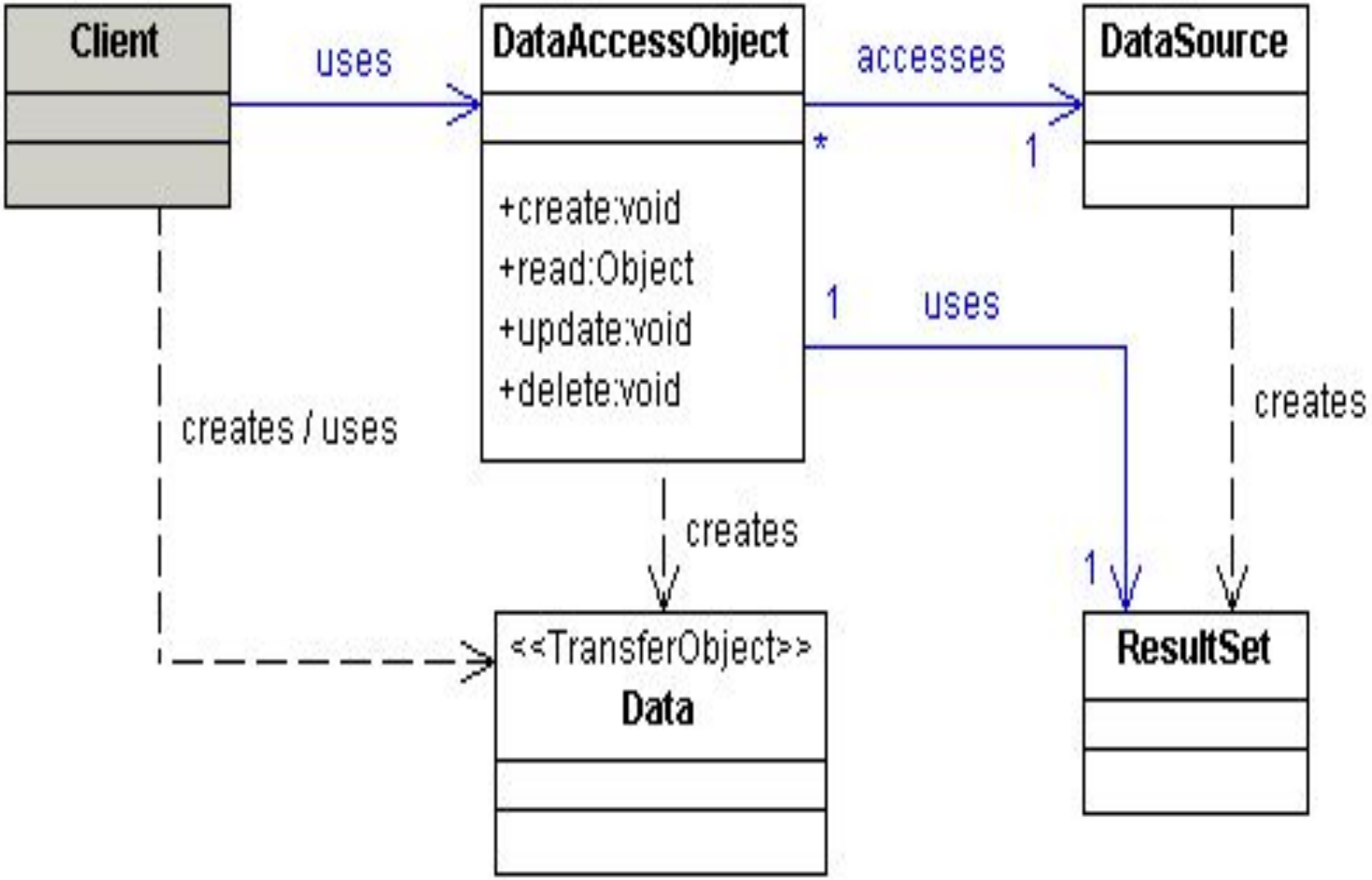
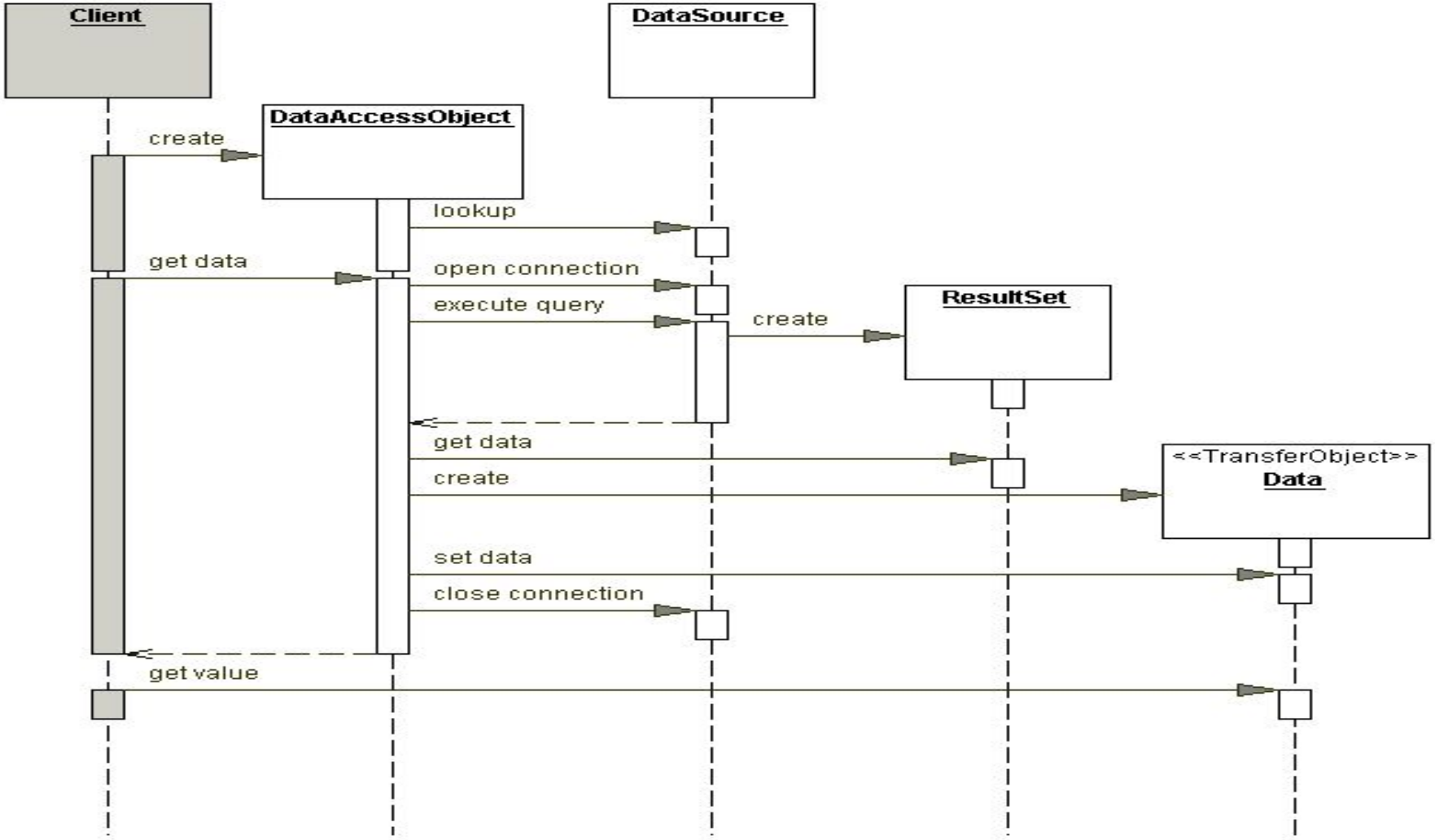


Диаграмма последовательности, показывающая взаимодействия между участниками в данном паттерне имеет вид:



BusinessObject

BusinessObject представляет клиента данных. Это объект, который нуждается в доступе к источнику данных для получения и сохранения данных.

BusinessObject может быть реализован как сессионный компонент, компонент управления данными или другой Java-объект, сервлет или вспомогательный компонент.

DataAccessObject

DataAccessObject является первичным объектом данного паттерна.

DataAccessObject абстрагирует используемую реализацию доступа к данным для BusinessObject, обеспечивая прозрачный доступ к источнику данных.

BusinessObject передает также ответственность за выполнение операций загрузки и сохранения данных объекту DataAccessObject.

DataSource

Представляет реализацию источника данных.

Источником данных может быть база данных, XML-репозиторий и др.

Источником данных может быть также другая система (традиционная/мэйнфрейм), служба (B2B-служба или система обслуживания кредитных карт), или какой-либо репозиторий (LDAP).

TransferObject

Представляет собой объект, используемый для передачи данных.

DataAccessObject может использовать TransferObject для возврата данных клиенту.

DataAccessObject может также принимать данные от клиента в объекте TransferObject для их обновления в источнике данных.

Стратегии

Стратегия Automatic DAO Code Generation

Поскольку BusinessObject соответствует конкретному DAO, есть возможность установить взаимоотношения между BusinessObject, DAO, и применяемыми реализациями (например, таблицы в RDBMS).

После установления взаимоотношений появляется возможность написать простую утилиту для генерации кода, зависящую от приложения, которая может генерировать код для всех нужных приложению объектов DAO.

Метаданные для генерации DAO могут определяться разработчиком в файле-дескрипторе.

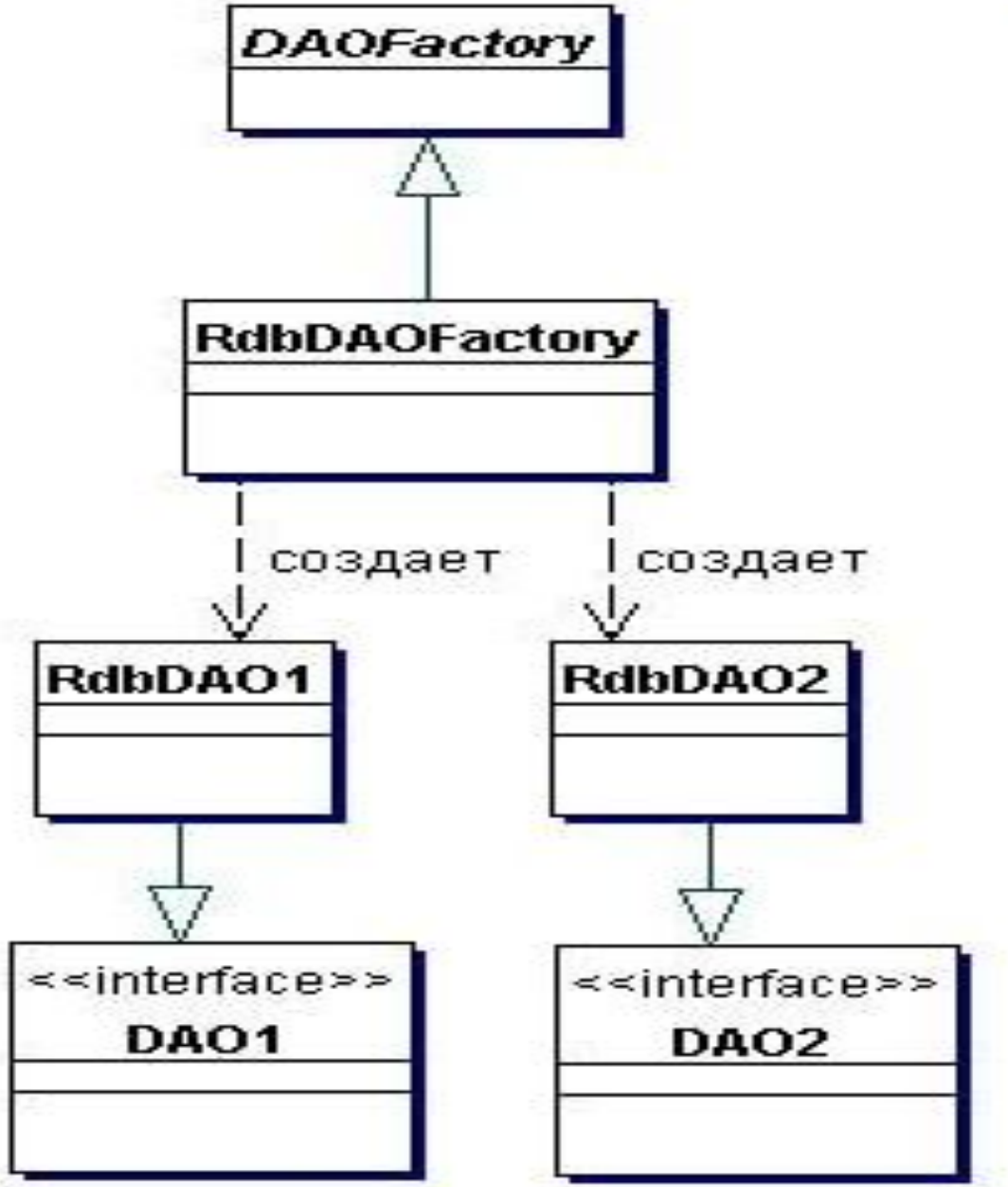
В качестве альтернативы генератор кода может автоматически проанализировать базу данных и предоставить необходимые для доступа к ней объекты DAO.

Стратегия Factory for Data Access Objects

Паттерн DAO может быть сделан очень гибким при использовании паттернов Abstract Factory и Factory Method.

Данная стратегия может быть реализована с использованием паттерна Factory Method для генерации нескольких объектов DAO, которые нужны приложению, в тех случаях, когда применяемое хранилище данных не изменяется при переходе от одной реализации к другой.

Диаграмма классов этого случая имеет вид:



Когда используемое хранилище данных может измениться при переходе от одной реализации к другой, данная стратегия может быть реализована с применением паттерна Abstract Factory.

Диаграмма классов этой стратегии имеет следующий вид:

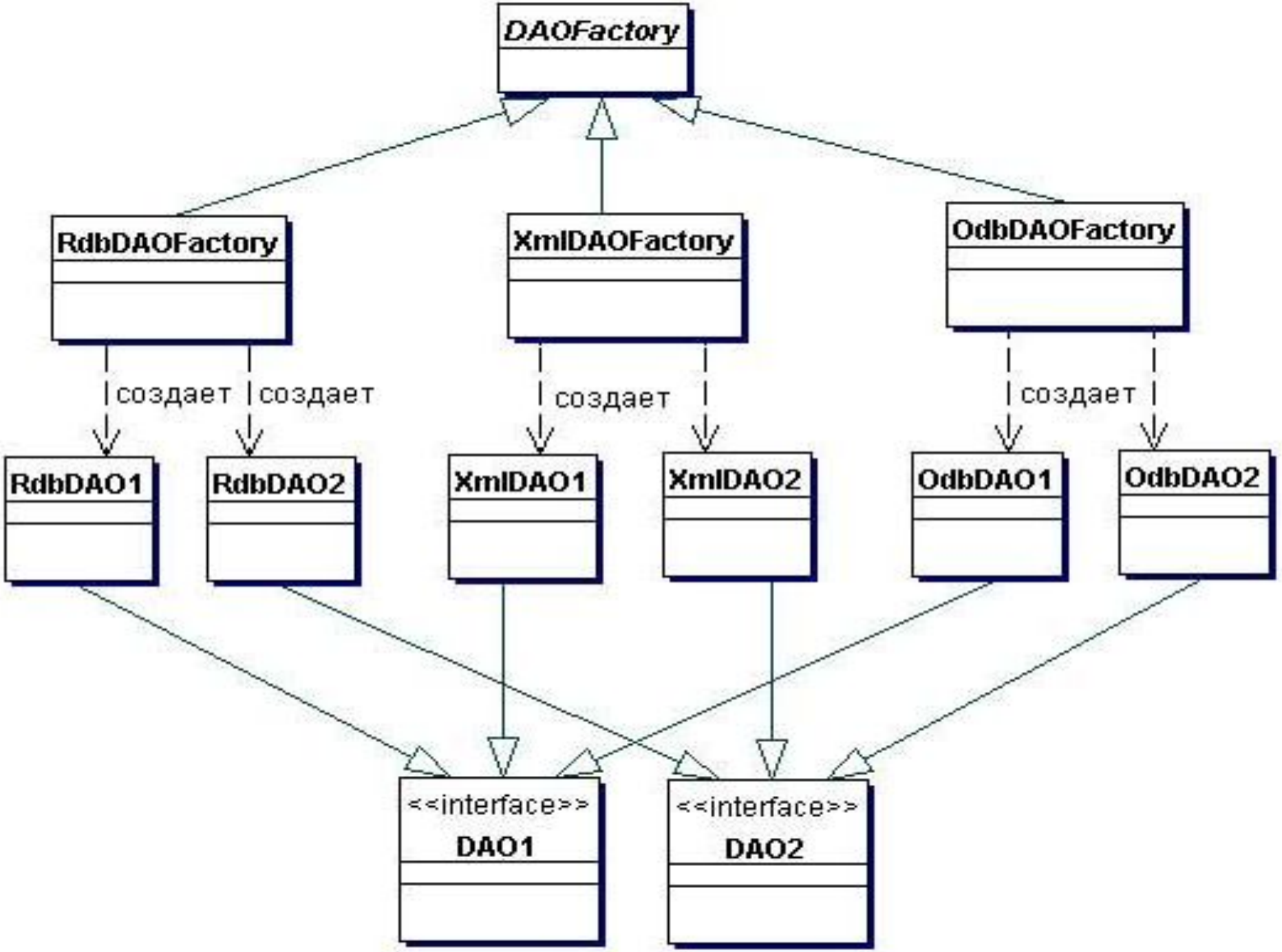
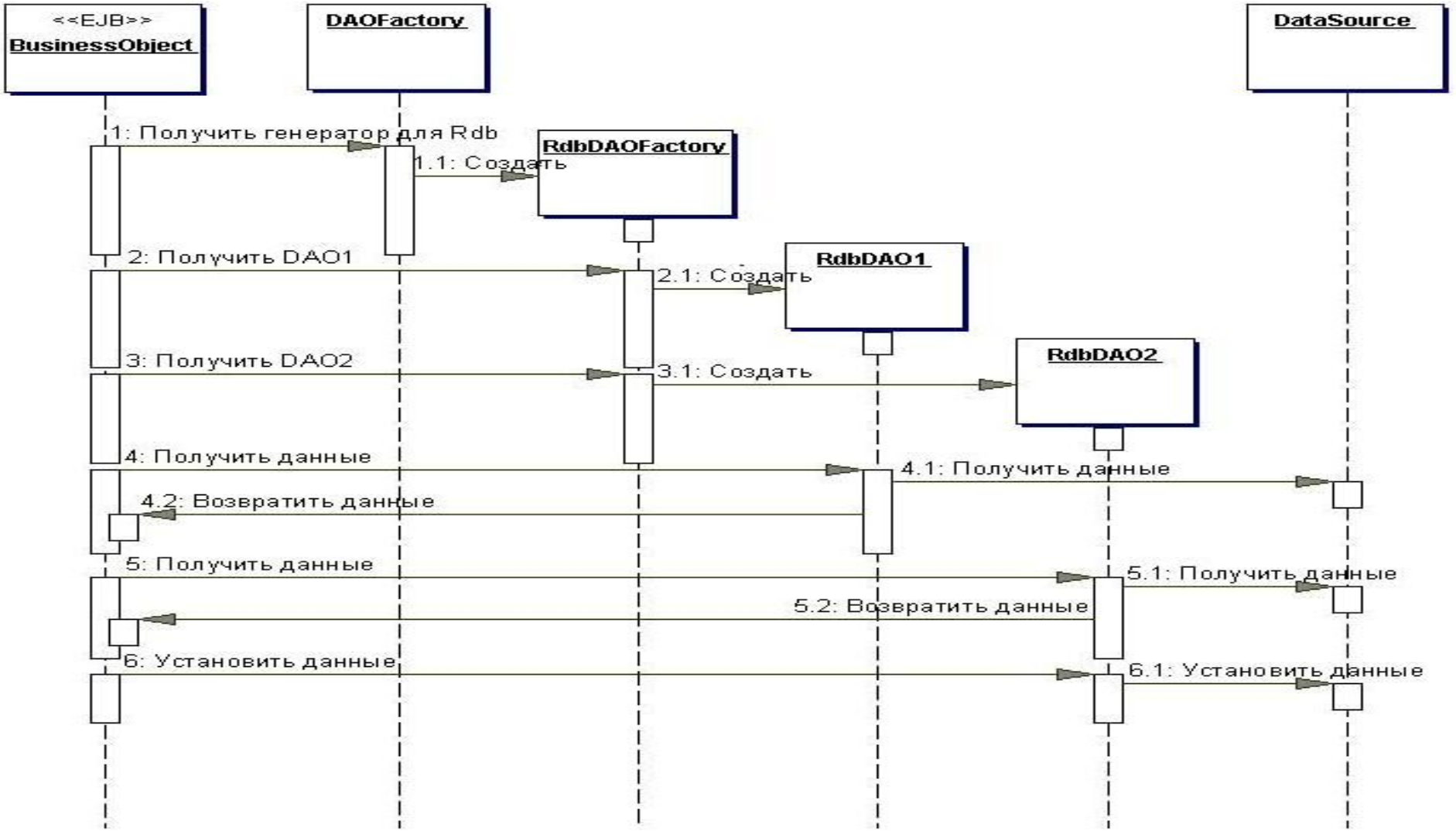

DAOFactory
RdbDAOFactory
XmlDAOFactory
OdbDAOFactory
создает
создает
создает
создает
RdbDAO1
RdbDAO2
XmlDAO1
XmlDAO2
OdbDAO1
OdbDAO2
<<interface>>
DAO1
<<interface>>
DAO2

Диаграмма последовательности, описывающая взаимодействия для этой стратегии имеет вид:



Рассмотрим пример кода для паттерна DAO.

```
// DAO Factory
public abstract class DAOFactory {
// Список DAO типов поддерживаемых фабрикой
 public static final int CLOUDSCAPE = 1;
 public static final int ORACLE = 2;
 public static final int SYBASE = 3;
...
// Методы для создания DAO
 public abstract CustomerDAO getCustomerDAO();
 public abstract AccountDAO getAccountDAO();
 public abstract OrderDAO getOrderDAO();
...
public static DAOFactory getDAOFactory( int whichFactory) {
   switch (whichFactory) {
      case CLOUDSCAPE: return new CloudscapeDAOFactory();
      case ORACLE : return new OracleDAOFactory();
      case SYBASE : return new SybaseDAOFactory();
      ...
      default : return null;
} } }
```

```
public class CloudscapeDAOFactory extends DAOFactory {
 public static final String DRIVER=
                    "COM.cloudscape.core.RmiJdbcDriver";

 public static final String DBURL=
     "jdbc:cloudscape:rmi://localhost:1099/CoreJ2EEDB";

 // метод для создания соединения к Cloudscape

 public static Connection createConnection() {…….}
 public CustomerDAO getCustomerDAO() {
                  return new CloudscapeCustomerDAO(); }

 public AccountDAO getAccountDAO() {
                   return new CloudscapeAccountDAO(); }

 public OrderDAO getOrderDAO() {
                       return new CloudscapeOrderDAO(); }
... }
```

Интерфейс для CustomerDAO

```
public interface CustomerDAO {
   public int insertCustomer(...);
   public boolean deleteCustomer(...);
   public Customer findCustomer(...);
   public boolean updateCustomer(...);
   public RowSet selectCustomersRS(...);
   public Collection selectCustomersTO(...);
   ...
}
```

Имплементация интерфейса CustomerDAO

```
public class CloudscapeCustomerDAO
                                implements CustomerDAO {
  public CloudscapeCustomerDAO() {…. }
  public int insertCustomer(...) {
  //метод возвращает номер нового созданного
 //Customer или -1 в случае ошибки
 }
  public boolean deleteCustomer(...) { …….. }
  ……….
}
```

Customer Transfer Object

```
public class Customer implements
                                    java.io.Serializable {
  int CustomerNumber;
  String name;
  String streetAddress;
  String city;
  ...
  // getter and setter
  ... ...
}
```

Использование DAO Factory

```
……..
DAOFactory cloudscapeFactory =
                        DAOFactory.getDAOFactory(
                    DAOFactory.DAOCLOUDSCAPE);
CustomerDAO custDAO =
              cloudscapeFactory.getCustomerDAO();

int newCustNo = custDAO.insertCustomer(...);

//Создаем Customer объект
Customer cust = custDAO.findCustomer(...);

//изменяем значение Customer объекта
cust.setAddress(...);
cust.setEmail(...);
custDAO.updateCustomer(cust);
……..
```

Hibernate

Hibernate - это механизм отображения в реляционной базе данных объектов java.

На практике наибольшую популярность получили реляционные модели баз данных, хотя в современных методологиях программирования пользуется популярностью объектно-ориентированное программирование.

Для стыковки данных технологий разработано множество технологий, спецификаций и фреймворков для маппинга объектов на таблицы реляционных баз данных.

Java разработчикам доступно множество технологий для работы с данными это может быть просто сериализация объектов, JDBC, JDO, JPA и множество других.

Java Persistence API (JPA), являющееся составной частью Hibernate сочетает в себе простоту сериализации объектов с возможностью работы с данными на уровне объектно-ориентированной модели.

При этом остается возможность комбинирования доступа к данным как в JDBC на уровне реляционных данных.

Рассмотрим примеры использования Hibernate.

В базе данных создадим таблицу **Student** с тремя полями:

1) ***id*** — идентификатор

2) ***name*** — имя студента

3) ***age*** — его возраст

Другими словами выполним запрос

```
CREATE TABLE Student(id NUMBER(10) NOT NULL,
name varchar(100) NOT NULL,
age NUMBER(3) NOT NULL,
CONSTRAINT pk_Student PRIMARY KEY(id));
```

Теперь создадим пакет *logic.*

В нем опишем класс-сущность, который будем хранить в БД:

```
package logic;


import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import org.hibernate.annotations.GenericGenerator;

@Entity
@Table(name="Student")
public class Student {

    private Long id;
    private String name;
    private Long age;

    public Student(){
        name = null;
    }

    public Student(Student s){
        name = s.getName();
    }
```

```
@Id
   @GeneratedValue (generator="increment")
   @GenericGenerator (name="increment", strategy = "increment")
   @Column(name="id")
   public Long getId() {
      return id;
   }

   @Column(name="name")
   public String getName(){
      return name;
   }

   @Column(name="age")
   public Long getAge(){
      return age;
   }

   public void setId(Long i){
      id = i;
   }

   public void setName(String s){
      name = s;
   }

   public void setAge(Long l){
      age = l; } }
```

Аннотации здесь используются для **Mapping Java** классов с таблицами базы данных.

Проще говоря для того, чтобы **Hibernate** знал, что данный класс является сущностью, то есть объекты данного класса мы будем хранить в базе данных.

Использованные здесь аннотации имеют следующий смысл:

@Entity — указывает на то, что данный класс является сущностью.

@Table — задает имя таблицы, в которой будут храниться объекты класса

@Id — обозначает поле id

@GeneratedValue(generator="increment") и ***@GenericGenerator(name="increment", strategy = "increment")*** — указывает на то, как будет генерироваться id (в данном случае по возростанию)

@Column — обозначает имя колонки, соответствующей данному полю.

Стоит отметить также, что все классы-сущности должны обязательно иметь *геттеры*, *сеттеры* и *конструктор по умолчанию*.

Теперь создадим главный конфигурационный файл **hibernate.cfg.xml** и поместим его в папку *bin* проекта.

Из этого файла **Hibernate** будет брать всю необходимую ему информацию:

```
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD
                3.0//EN"  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
 <hibernate-configuration>
   <session-factory>
    <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
    <property
                name="connection.url">jdbc:oracle:thin:@localhost:1521:MyDB</property>
    <property name="connection.username">Your_Login</property>
    <property name="connection.password">Your_Password</property>
    <property name="connection.pool_size">10</property>
    <property name="dialect">org.hibernate.dialect.OracleDialect</property>
    <property name="show_sql">true</property>
    <property name="hbm2ddl.auto">update</property>
    <property name="hibernate.connection.autocommit">false</property>
    <property name="current_session_context_class">thread</property>

    <mapping class="logic.Student" />

   </session-factory>
</hibernate-configuration>
```

Создадим пакет *util*, а в нем класс **HibernateUtil**, который будет отвечать за обработку данного *xml* файла и установление соединения с базой данных:

```
package util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
   private static SessionFactory sessionFactory = null;

   static {
      try {
            //creates the session factory from hibernate.cfg.xml
            sessionFactory = new Configuration().configure().buildSessionFactory();
      } catch (Exception e) {
           e.printStackTrace();
      }
   }

   public static SessionFactory getSessionFactory() {
      return sessionFactory;
   }
}
```

Теперь осталось разобраться со взаимодействием приложения с базой данных.

Тогда для класса-сущности, определим интерфейс **StudentDAO** из пакета *DAO*, содержащий набор необходимых методов:

```
package DAO;

import java.sql.SQLException;
import java.util.List;

import logic.Student;

public interface StudentDAO {
   public void addStudent(Student student) throws SQLException; //добавить
                                                                студента
   public void updateStudent(Student student) throws SQLException; //обновить
                                                                 студента
   public Student getStudentById(Long id) throws SQLException;   //получить
                                                              стедента по id
   public List getAllStudents() throws SQLException;   //получить всех студентов
   public void deleteStudent(Student student) throws SQLException; //удалить
                                                                студента
}
```

Теперь определим реализацию этого интерфейса в классе **SudentDAOImpl** в пакете *DAO.Impl*:

```
package DAO.Impl;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import javax.swing.JOptionPane;
import org.hibernate.Session;
import util.HibernateUtil;
import DAO.StudentDAO;
import logic.Student;

public class StudentDAOImpl implements StudentDAO {
    public void addStudent(Student stud) throws SQLException {
        Session session = null;
        try {
            session = HibernateUtil.getSessionFactory().openSession();
            session.beginTransaction();
            session.save(stud);
            session.getTransaction().commit();
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, e.getMessage(), "Ошибка I/O",
                                                    JOptionPane.OK_OPTION);
        } finally {
            if (session != null && session.isOpen()) {
                session.close();
            } }
    }
```

```
public void updateStudent(Student stud) throws SQLException {
        Session session = null;
        try {
            session = HibernateUtil.getSessionFactory().openSession();
            session.beginTransaction();
            session.update(stud);
            session.getTransaction().commit();
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, e.getMessage(), "Ошибка I/O",
                                                                JOptionPane.OK_OPTION);
        } finally {
            if (session != null && session.isOpen()) {
                session.close();
            }
        }
    }
```

```
public Student getStudentById(Long id) throws SQLException {
        Session session = null;
        Student stud = null;
        try {
            session = HibernateUtil.getSessionFactory().openSession();
            stud = (Student) session.load(Student.class, id);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, e.getMessage(), "Ошибка I/O",
                                                                JOptionPane.OK_OPTION);
        } finally {
            if (session != null && session.isOpen()) {
                session.close();
            }
        }
        return stud;
    }
```

```
public List<Student> getAllStudents() throws SQLException {
        Session session = null;
        List<Student> studs = new ArrayList<Student>();
        try {
            session = HibernateUtil.getSessionFactory().openSession();
            studs = session.createCriteria(Student.class).list();
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, e.getMessage(), "Ошибка I/O",
                                                                JOptionPane.OK_OPTION);
        } finally {
            if (session != null && session.isOpen()) {
                session.close();
            }
        }
        return studs;
    }
```

```
public void deleteStudent(Student stud) throws SQLException {
        Session session = null;
        try {
            session = HibernateUtil.getSessionFactory().openSession();
            session.beginTransaction();
            session.delete(stud);
            session.getTransaction().commit();
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, e.getMessage(),
                                          "Ошибка I/O", JOptionPane.OK_OPTION);
        } finally {
            if (session != null && session.isOpen()) {
                session.close();
            }
        }
    }
}
```

Создадим класс **Factory** в пакете *DAO*, к которому будем обращаться за нашими реализациями DAO, от которых и будем вызывать необходимые нам методы:

```
package DAO;


import DAO.Impl.StudentDAOImpl;

public class Factory {
    private static StudentDAO studentDAO = null;
    private static Factory instance = null;

    public static synchronized Factory getInstance(){
        if (instance == null){
          instance = new Factory();
        }
        return instance;
    }

    public StudentDAO getStudentDAO(){
        if (studentDAO == null){
          studentDAO = new StudentDAOImpl();
        }
        return studentDAO;
    }
}
```

Метод main будет иметь вид:

```
import DAO.Factory;
public class Main {
    public static void main(String[] args) throws SQLException {
    //Создадим двух студентов
    Student s1 = new Student();
    Student s2 = new Student();

    //Проинициализируем их
    s1.setName("Ivanov Ivan");
    s1.setAge(21l);
    s2.setName("Petrova Alisa");
    s2.setAge(24l);

    //Сохраним их в бд, id будут сгенерированы автоматически
    Factory.getInstance().getStudentDAO().addStudent(s1);
    Factory.getInstance().getStudentDAO().addStudent(s2);

    //Выведем всех студентов из бд
    List<Student> studs = Factory.getInstance().getStudentDAO().getAllStudents();
    for(int i = 0; i < studs.size(); ++i) {
        System.out.println("Имя студента : " + studs.get(i).getName() + ", Возраст : " +
                                        studs.get(i).getAge() +",  id : " + studs.get(i).getId());
        System.out.println("==============================");
    }
  }
}
```

Запросы в Hibernate

Запросы возвращают набор данных из базы данных, удовлетворяющих заданному условию.

Библиотека **Hibernate** предлагает три вида запросов к БД:

1) **Criteria**
2) **SQL**
3) **HQL**

Запросы с использованием Criteria

Объект **Criteria** создается с помощью метода *createCriteria* экземпляра класса **Session**:

```
Criteria crit = session.createCriteria(Student.class); //создаем критерий запроса
crit.setMaxResults(50); //ограничиваем число результатов
List studs = crit.list(); //помещаем результаты в список
```

В данном примере был создан критерий запроса на основе класса **Student**

Сужение выборки осуществляется следующим образом:

```
List studs = session.createCriteria(Student.class)
    .add( Expression.like("name", "Ivanov%") )
    .add( Expression.between("age", 18, 25) )
    .list();

List studs = session.createCriteria(Student.class)
    .add( Expression.like("name", "_van%") )
    .add( Expression.or(
        Expression.eq( "age", new Integer(20) ),
        Expression.isNull("age")
    ) ).list();
```

```
List studs = session.createCriteria(Student.class)
    .add( Expression.in( "name", new String[] { "Ivanov Ivan",
                                                "Petrov Petia", "Zubin Egor" } ) )
        .add( Expression.disjunction()
        .add( Expression.isNull("age") )
        .add( Expression.eq("age", new Integer(20) ) )
        .add( Expression.eq("age", new Integer(21) ) )
        .add( Expression.eq("age", new Integer(22) ) )
    ) ).list();
```

Expression.like — указывает шаблон, где ‘_’ — любой один символ, ‘%’ — любое количество символов
Expression.isNull — значение поля равно NULL.
Expression.between — ‘age’ — имя поля, 18 — минимальное значение указанного поля, 25 — его максимальное значение
Expression.in — указывает диапазон значений конкретного поля
Expression.disjunction*, *Expression.or — дизъюнкция (OR) — объединяет в себе несколько других выражений оператором ИЛИ.
Expression.eq — определяет равенство поля какому-то значению.

Результаты также можно отсортировать:

```
List studs = sess.createCriteria(Student.class)
    .add( Expression.like("name", "Iv%")
    .addOrder( Order.asc("name") ) //по возрастанию
    .addOrder( Order.desc("age") ) //по убыванию
    .list();
```

Также есть возможность запроса по данным экземпляра класса:

```
Student s = new Student();
s.setName("Ivanov Ivan");
s.setAge(20l);
List results = session.createCriteria(Student.class)
    .add( Example.create(s))
    .list();
```

Поля объекта, имеющие значение *null* или являющиеся идентификаторами, будут игнорироваться.

Example также можно настраивать:

```
Example example = Example.create(s)
    .excludeZeroes()          //исключает поля с нулевыми значениями
    .excludeProperty("name")  //исключает поле "name"
    .ignoreCase()       //задает независимое от регистра сравнение строк
    .enableLike();            //использует like для сравнения строк
List results = session.createCriteria(Student.class)
    .add(example)
    .list();
```

Запросы с использованием SQL

Hibernate позволяет выражать запросы на родном для базы данных диалекте **SQL**. Выглядеть это будет, примерно, следующим образом:

```
sess.createSQLQuery("select * fromStudent").addEntity(Student.class).list();


sess.createSQLQuery("select id, name, age from Student")
                                                .addEntity(Student.class).list()
```

В запросах также можно указывать параметры:

```
Query query = session.createSQLQuery("select * from Student where
                                                name like ?").addEntity(Student.class);
List result = query.setString(0, "Ivan%").list();

query = session.createSQLQuery("select * from Student where name like
                                                :name").addEntity(Student.class);
List result = query.setString("name", "Ivan%").list();
```

В первом случае с помощью *query.setString* указывается порядковый номер параметра *(?)* и значение типа *String*, которое вместо него подставится.

Если значение типа *Long*, то будет *setLong*, если *Date*, то *setDate* и так далее.

Во втором случае имя параметра задано явно, поэтому значение задается параметру по имени.

Запросы с использованием HQL

Hibernate позволяет производить запросы на **HQL**(*The Hibernate Query Language* — Язык запросов **Hibernate**), который во многом похож на язык **SQL**, с той разницей, что является полностью объектно-ориентированным.

Если запрос с помощью **SQL** производился методом *createSQLQuery*, то в **HQL** будет просто *createQuery*.

Простой пример:

```
List<Student> studs = (List<Student>)session.createQuery("from Student
                                                          order by
name").list()
```

Видно, что *select* в начале запроса можно не указывать.

Поскольку **HQL** — объектно-ориентированный язык, то значение полей можно выбрать и так:

```
List<String> names = (List<String>)session.createQuery("select
                          stud.name from Student stud order by name").list();
```

А можно и так:

```
List result = session.createQuery("select new list(stud, name, stud.age)
                                                  from Student as stud").list();
```

Язык **HQL** относительно сложен, но зато богат и дает очень много возможностей.

Отношения в Hibernate

Помимо таблицы **Student**, создадим еще две таблицы **Test** и **Statistics**.

Они будут связаны следующим образом:

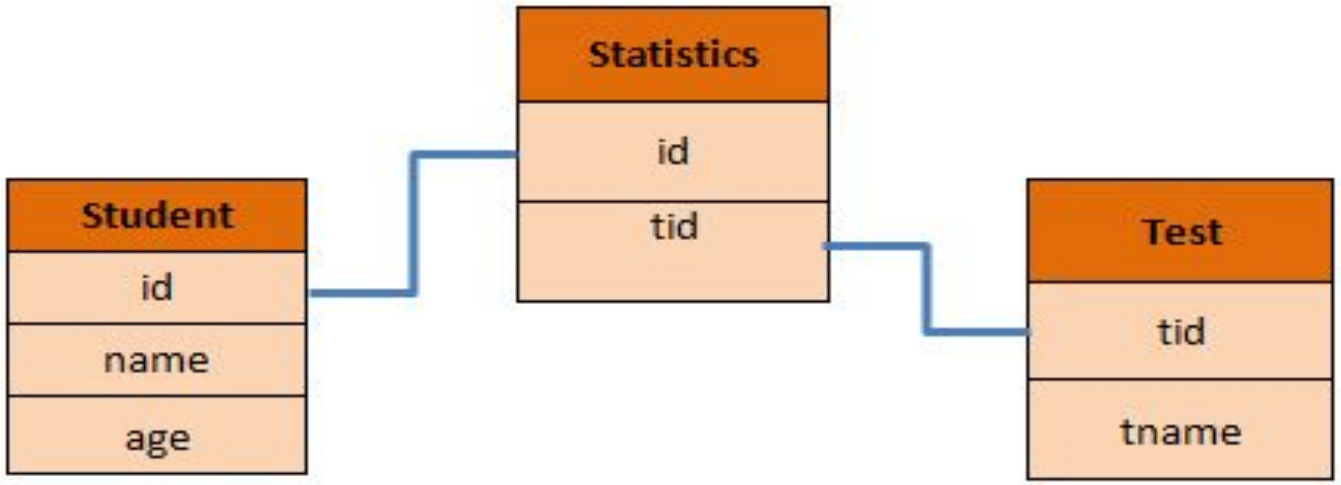


Таблица **Statistics** служит для связи таблиц **Student** и **Test**, чтобы избежать отношения многие ко многим.

Создадим эти две таблицы:

```
CREATE TABLE Test(tid NUMBER(10) NOT NULL,
tname varchar(100) NOT NULL, CONSTRAINT pk_Test PRIMARY KEY(tid));

CREATE TABLE Statistics(stid NUMBER(10) NOT NULL,
id NUMBER(10) NOT NULL,
tid NUMBER(10) NOT NULL,
CONSTRAINT pk_Statistics PRIMARY KEY(stid),
CONSTRAINT fk_Student FOREIGN KEY(id) REFERENCES Student(id),
CONSTRAINT fk_Test FOREIGN KEY(tid) REFERENCES Test(tid));
```

Также в файл **hibernate.cfg.xml** добавим маппинги новых классов:

```
<mapping class="logic.Test" />
<mapping class="logic.Statistics" />
```

Рассмотрим код.

Создаем в пакете *logic* классы-сущности:

Test

```
package logic;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinTable;
import javax.persistence.Table;
import javax.persistence.JoinColumn;

import org.hibernate.annotations.GenericGenerator;

@Entity
@Table(name="Test")
public class Test {

   private Long tid;
   private String tname;

   public Test(){
      tname = null;
   }
```

```
public Test(Test s){
        tname = s.getTName();
    }

    @Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy = "increment")
    @Column(name="tid")
    public Long getTid() {
        return tid;
    }

    @Column(name="tname")
    public String getTName(){
        return tname;
    }

    public void setId(Long i){
        tid = i;
    }

    public void setTName(String s){
        tname = s;
    }
}
```

Statistics

```
package logic;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

import org.hibernate.annotations.GenericGenerator;

@Entity
@Table(name="Statistics")
public class Statistics {

    private Long stid;
    private Long id;
    private Long tid;

    public Statistics(){
    }
```

```
@Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy = "increment")
    @Column(name="stid")
    public Long getStid(){
        return stid;
    }

    @Column(name="id")
    public Long getId(){
        return id;
    }

    @Column(name="tid")
    public Long getTid(){
        return tid;
    }
}
```

Классы **TestDAO** и **TestDAOImpl** создаются аналогично как для сущности **Student**.

Осталось только показать **Hibernate**, как эти таблицы между собой связаны.

В **Hibernate** для этого предусмотрены следующие виды аннотаций: *@OneToOne*, *@OneToMany*, *@ManyToOne*, *@ManyToMany*.

Например, чтобы связать таблицы **Student** и **Statistics** связью многие к одному, следует добавить в класс **Student** следующей код:

```
private Statistics stat;

@ManyToOne
@JoinTable(name = "id")
public Statistics getStat(){
        return stat;
 }
```

В классе **Statistics** аннотируем связь один ко многим с классом **Student**:

```
private Set<Student> studs = new HashSet<Student>(0);

@OneToMany
@JoinTable(name = "id")
public Set<Student> getStuds() {
        return studs;
}
```

В классе **Student** объявили атрибут типа **Statistics** и обозначили, что данная таблица связана отношением многие к одному с таблицей, представленной классом-сущностью **Statistics**.

А в классе **Statistics** указали связь один ко многим с классом **Student**.

С помощью аннотации *@JoinTable* мы указываем, какое поле является внешним ключом к текущей таблице.

Так же обозначаем отношение таблицы **Test** и **Statistics**, просто добавив в класс **Test** код:

```
private Statistics stat;

@ManyToOne
@JoinTable(name = "id")
public Statistics getStat(){
        return stat;
 }
```

В классе **Statistics** аннотируем связь один ко многим с классом **Test**:

```
private Set<Test> tests = new HashSet<Test>(0);

@OneToMany
@JoinTable(name = "id")
public Set<Test> getTests() {
        return tests;
 }
```

Поскольку таблица **Statistics** является не просто таблицей, связанной со **Student** и **Test**, а она разбивает нежелательную связь многие ко многим, мы также можем показать это **Hibernate**.

Просто вместо того, чтобы отдельно обозначать связь в каждой таблице, мы обозначим всю связь в одной, к примеру, в таблице **Test** добавив код:

```
private Student stud;

@ManyToOne
@JoinTable(name = "Statistics",
   joinColumns = @JoinColumn(name = "tid"),
  inverseJoinColumns = @JoinColumn(name = "id"))
public Student getStud(){
        return stud;
 }
```

С помощью параметра *name* аннотации *@JoinTable* мы обозначаем связующую таблицу,

joinColumns = *@JoinColumn* — указываем через какой ключ связаны таблицы **Test** и **Statistics**,

inverseJoinColumns = *@JoinColumn* — указываем, через какие ключи связаны уже **Statistics** и **Student**.

Если бы мы обозначали эту связь в классе **Student**:

```
private Test test;

@ManyToOne
@JoinTable(name = "Statistics", joinColumns =
@JoinColumn(name = "id"), inverseJoinColumns =
@JoinColumn(name = "tid"))
public Test getTest(){
        return test;
 }
```

Другой пример реализации Hibernate JPA

Рассмотрим пример работы с базой данный с использованием объектно-ориентированной модели.

Пусть имеется СУБД MySQL.

Предположим, что создана база данный с именем mybase, а в ней имеется следующая таблица, с именем mytable:

ID	name	age

Создадим класс mytable, который соответствует записи в таблице:

```
import java.io.Serializable;
import javax.persistence.*;

@Entity
@Table(name= “mytable”)
public class mytable implements Serializable {

private static final long serialVersionUID = 1L;

@Id
@Column(name= “id”)
@GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;

@Column(name= “name”, length=64)
  private String name;

@Column(name= “age”, length=64)
  private Byte age;
```

```
public mytable() {};
public mytable(String name,Byte b){
   this.age=b;
   this.name=name;
}
public Long getId() {return id; }
public void setId(Long id) {this.id = id;}
public void setname(String name){
  this.name=name;
};
public void setAge(Byte age){
  this.age=age;
};
public String toString() {
   return "testproject.mytable[id=" + id + "]";
}
}
```

Hibernate для своей работы требует специальный файл конфигурации (если точнее, то он используется во время компиляции), который называется persistence.xml. Файл имеет вид:

```
<?xml version="1.0" encoding="UTF-8" ?>
 <persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

 <persistence-unit name="JavaApplication3PU"
                                        transaction-type="RESOURCE_LOCAL">
  <provider>
    org.eclipse.persistence.jpa.PersistenceProvider
 </provider>
 <class>mytable</class>
```

```
<properties>
 <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/mybase" />
 <property name="javax.persistence.jdbc.password“
                                        value="qwertyui" />
 <property name="javax.persistence.jdbc.driver“
                          value="com.mysql.jdbc.Driver" />
 <property name="javax.persistence.jdbc.user"
                                          value="alex" />
 <property name="eclipselink.ddl-generation“
                                      value="create-tables" />
</properties>
</persistence-unit>
</persistence>
```

Тогда работа с базой данных будет иметь вид:

```
import javax.persistence.*;
public class Main {
  public static final EntityManagerFactory emf =
                    Persistence.createEntityManagerFactory(
                                                    "JavaApplication3PU");
  public static EntityManager em = emf.createEntityManager();
  public static void main(String[] args) {
    mytable te=new mytable("aaaa",new Byte((byte)34));
    te.setname("bbb");
    te.setage(new Byte((byte)56));
    em.getTransaction().begin();
    em.persist(te);
    em.getTransaction().commit();
  }
}
```

Компиляция и запуск

Для компиляции удобно воспользоваться утилитой ant(разработка Apache foundation http://ant.apache.org/).

1. Создаем директорию, например base.
2. В base создаем поддиректорию src, в которую помещаем файлы Main.java и mytable.java
3. В base создаем поддиректорию META-INF и в нее помещаем файл persistence.xml
4. В base создаем поддиректорию lib и в нее помещаем библиотеки

eclipselink-2.0.2.jar

eclipselink-javax.persistence-2.0.jar

mysql-connector-java-5.1.6-bin.jar

5. В base помещаем файл build.xml, который имеет вид:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
   <project name="EJB3 Tutorial" basedir="."
                                          default="deploy">
<property name="deploy.dir“
                        value="C:/Java_Dev/WEB/dev/ejb/
                jboss-4.2.3.GA/server/default/deploy" />
<property name="sourcedir"
                                  value="${basedir}/src" />
<property name="targetdir"
                                   value="${basedir}/build" />
<property name="librarydir" value="${basedir}/lib" />
```

```
<path id="libraries">
<fileset dir="${librarydir}">
<include name="*.jar" />
  </fileset>
</path>
<target name="clean">
  <delete dir="${targetdir}" />
 <mkdir dir="${targetdir}" />
</target>
```

```
<target name="compile" depends=
                                    "copy-resources">
  <javac srcdir="${sourcedir}"
   destdir="${targetdir}" classpathref="libraries“
                                        debug="on" />
</target>
<target name="copy-resources">
    <copy todir="${targetdir}">
     <fileset dir="${sourcedir}">
        <exclude name="**/*.java" />
      </fileset>
     </copy>
</target>
```

```
<target name="deploy" description="JARs the
  Task" depends=
                    "clean,copy-resources,compile">
    <jar destfile="${deploy.dir}/java2s.jar">
    <metainf dir="${sourcedir}/META-INF" />
    <fileset dir="${targetdir}">
      <include name="**/*.class" />
    </fileset>
    </jar>
</target>
```

```
<target name="undeploy"
   description="Undeploy jar from server">
     <delete file="${deploy.dir}/java2s.jar" />
</target>
<target name="run" depends="compile">
   <java classname="Main"
                                  classpathref="libraries">
   <classpath path="${targetdir}" />
 <jvmarg value="-Djava.library.path=./lib" />
</java>
</target>
</project>
```

6. Перейти в директорию base и из консоли запустить ant

D:\base>ant

В результате в папке build будут находится скомпилированные файлы.

7. Запуск приложения будет иметь вид:

>java –cp .;d:\base\lib\eclipselink-2.0.2.jar; d:\base\lib\eclipselink-javax.persistence-2.0.jar;

d:\base\lib\ mysql-connector-java-5.1.6-bin.jar Main

Пакет JOOQ

Данный пакет представляет собой Linq для Java. С помощью данной библиотеки можно строить запросы прямо в Java коде.

Рассмотрим пример.

Пусть имеется база данных mybase и в ней таблица mytable, которая имеет вид:

ID	name	age
100	aaa	45

Создаем файл **mybase.properties**:

jdbc.Driver=com.mysql.jdbc.Driver

jdbc.URL=jdbc:mysql://localhost:3306/mybase

jdbc.Schema=mybase

jdbc.User=muser

jdbc.Password=qwertyui

#The default code generator. You can override this one, to generate your own code style

#Defaults to org.jooq.util.DefaultGenerator

generator=org.jooq.util.DefaultGenerator

#The database type. The format here is:

#generator.database=org.util.[database].[database]Database

generator.database=org.jooq.util.mysql.MySQLDatabase

#All elements that are generated from your schema (several Java regular expressions, separated by comma)

#Watch out for case-sensitivity. Depending on your database, this might be important!

generator.database.includes=.*

#All elements that are excluded from your schema (several Java regular expressions, separated by comma). Excludes match before includes

generator.database.excludes=

#Primary key / foreign key relations should be generated and used.

#This will be a prerequisite for various advanced features

#Defaults to false

generator.generate.relations=true

#Generate deprecated code for backwards compatibility

#Defaults to true

generator.generate.deprecated=false

#The destination package of your generated classes (within the destination directory)

generator.target.package=jooqs.Generated.test.generated

#The destination directory of your generated classes

generator.target.directory=C:/Java/Generated

Генерируем классы, соответствующие базе данных

java -classpath jooq-1.6.9.jar;jooq-meta-1.6.9.jar;
jooq-codegen-1.6.9.jar;mysql-connector-java-
5.1.18-bin.jar;. org.jooq.util.GenerationTool
/mybase.properties

Тогда получение записей из таблицы mytable
будет иметь следующий вид:

```
package jooqs;
import java.sql.*;
import jooqs.Generated.test.generated.*;
import jooqs.Generated.test.generated.tables.*;
import org.jooq.Record;
import org.jooq.Result;
```

```
public class Main {
    public static void main(String[] args) {
        Connection conn = null;
        try{
            String userName = "muser";
            String password = "qwertyui";
            String url = "jdbc:mysql://localhost/mybase";
            Class.forName ("com.mysql.jdbc.Driver").newInstance();
           conn = DriverManager.getConnection (url, userName,
                                                        password);
          //создаем объект соответствующий базе данных
            MybaseFactory mb=new MybaseFactory(conn);
```

```
//Получаем записи из таблицы mytable
Result<?> result =
            mb.select().from(Mytable.MYTABLE).fetch();
 for (Record r : result) {
            Integer id = r.getValue(Mytable.ID);
            String name = r.getValue(Mytable.NAME);
            Short ag = r.getValue(Mytable.AGE);
            System.out.println("ID: " + id + " name: " +
                                          name + " age: " + ag);
           conn.close();
  }
 } catch (Exception e) {
        e.printStackTrace();
 }
}
}
```