Федеральное государственное автономное образовательное учреждение

высшего образования

«Санкт-Петербургский политехнический университет Петра Великого»

Институт компьютерных наук и технологий

Кафедра «Распределенные вычисления и компьютерные сети»

**ОТЧЕТ**

Миграция схемы данных

по дисциплине

«Архитектура систем управления реляционными базами данных»

Выполнил

студент гр. 63507/1 И.А. Рапенок

Руководитель

асс. С.Г. Попов

Санкт-Петербург

2016

Содержание

[Анализ задания 2](#_Toc471689987)

[Терминология 2](#_Toc471689988)

[Постановка задачи 2](#_Toc471689989)

[Варианты использования 3](#_Toc471689990)

[Объектно-ориентированное проектирование 3](#_Toc471689991)

[Диаграмма классов 3](#_Toc471689992)

[Диаграммы последовательностей 4](#_Toc471689993)

[Описание программы 5](#_Toc471689994)

[Методика и результаты тестирования 7](#_Toc471689995)

[Выводы 8](#_Toc471689996)

[Приложение 9](#_Toc471689997)

[Код программы 9](#_Toc471689998)

# Анализ задания

## Терминология

База данных — совокупность всех объектов БД (таблиц, процедур, триггеров и т.д.), статических данных (неизменяемых данных, хранящихся в lookup-таблицах) и пользовательских данных (которые изменяются в процессе работы с приложением).

Система управления базами данных — совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных.

Структура базы данных — совокупность всех объектов БД и статических данных. Пользовательские данные в понятие структуры БД не входят.

Data Definition Language (DDL) (язык описания данных) — это семейство компьютерных языков, используемых в компьютерных программах для описания структуры баз данных.

Миграция, в данном контексте, — на основании существующей схемы данных создание запроса на языке DDL описывающего эту схему данных в другой СУБД.

## Постановка задачи

В данной работе разрабатывается приложение для миграции схемы данных из СУБД Oracle в СУБД PostreSQL. На данном этапе требуется миграция структуры таблиц и ограничений таких как: PRIVATE KEY, UNIQUE, NOT NULL, CHECK, FOREIGN KEY и REFERENCES. Приложение должно разрабатываться с учетом масштабируемости, а именно должен быть предусмотрен механизм добавления новых классов работы с другими базами данных без существенного изменения уже имеющих.

Клиентом данного приложения является пользователь. Пользователь может указать атрибуты соединения с базой данных и получить DDL описание для выбранной СУБД.

## Варианты использования

Миграция данных:

Основной сценарий:

1. Пользователь вводит параметры соединения в конфигурационном файле и целевую СУБД
2. Система создает sql файл с DDL описанием структуры данных для целевой СУБД

Расширения:

* 1. Указаны неверные параметры соединения или целевая СУБД не поддерживается
  2. Приложение возвращает ошибку

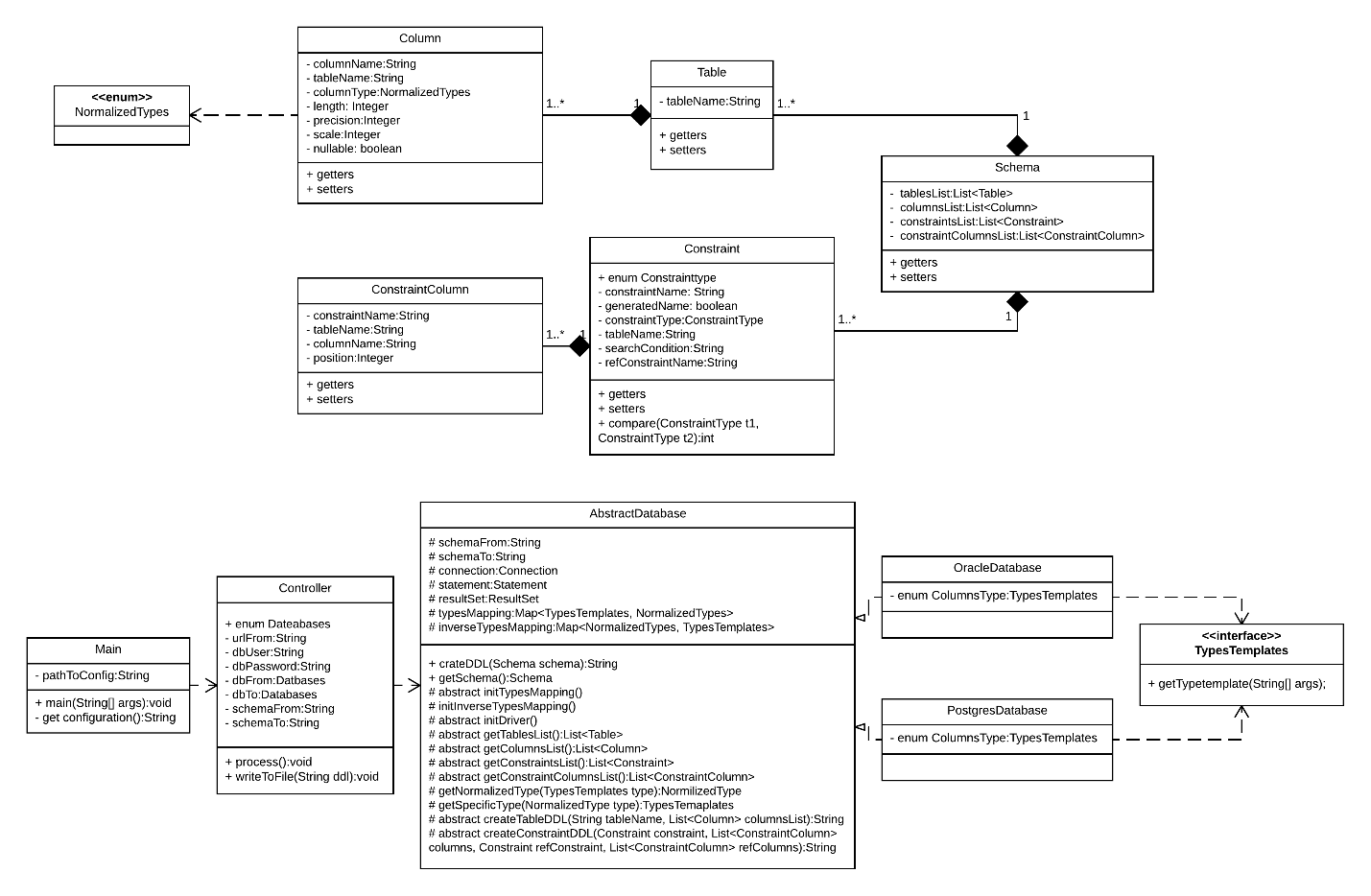
# Объектно-ориентированное проектирование

## Диаграмма классов

На рисунке 1 представлена диаграмма классов. На диаграмме:

Сущностные классы:

* Класс *Column* – предназначен для хранения информации о полях таблиц, а так же предоставляет интерфейс для доступа к ним. Так же содержит в себе перечисление содержащие нормализованные наименования всех типов данных, может пополняться при добавлении новы СУБД.



* Класс *Table* – предназначен для хранения информации о таблицах базы данных, а так же для предоставление интерфейса доступа к ним.
* Класс *ConstraintColumn* – предназначен для хранения информации о полях таблиц на которые наложены ограничения. Атрибут position предназначен для упорядочивания нескольких полей в одном отношении.
* Класс *Constraint* – предназначен для хранении информации об отношениях. Содержит в себе перечисление типов отношений:
  + PRIVATE KEY;
  + UNIQUE;
  + NOT NULL;
  + CHECK;
  + FOREIGN KEY и REFERENCES.

Для того чтобы отношения FOREIGN KEY и REFERENCES создавались после отношений PRIARY KEY необходимо реализовать метод compare который будет использоваться при сортировки.

* Класс *Schema* – предназначен для хранения информации о схеме данных.
* Класс *Main* – предназначен для получение всей необходимой для миграции схемы информации.
* Класс *Controller* – предназначен для обработки входных параметров и запуска необходимой для миграции логики. Так же в классе реализована логика записи полученного DDL-описания схемы данных в файл. Методы:
  + *void process()* – миграция схемы данных;
  + *void writeToFile(String resultDDL)* – запись результата в файл.
* Класс *AbstractDatabase* – абстрактный класс реализующий логику считывания метаданных из базы и создание DDL-описания схемы. Для поддержки новой СУБД, необходимо добавить класс наследник AbstractDatabase и определить необходимые абстрактные методы, при необходимости переопределить не абстрактные. Методы:
  + *crateDDL(Schema schema) –* создание DDL описания по схеме;
  + *getSchema()* – получение схемы данных;
  + *abstract initTypesMapping() –* определение соответствия между типами из СУБД и нормализованными типами приложения;
  + *initInverseTypesMapping() –* определение соответствия между нормализованными типами приложения и типами из СУБД;
  + *abstract initDriver() –* определение необходимого для работы с СУБД jdbc-драйвера;
  + *abstract getTablesList() –* получение списка таблиц из СУБД;
  + *abstract getColumnsList() –* получение списка полей таблиц из СУБД;
  + *abstract getConstraintsList() –* получение списка ограничений из СУБД;
  + *abstract getConstraintColumnsList() –* получение списка полей на которые накладываются ограничения из СУБД;
  + *getNormalizedType(TypesTemplates type)* – получение нормализованного названия типа;
  + *getSpecificType(NormalizedType type) –* получение специфичного для СУБД названия типа;
  + *abstract createTableDDL(String tableName, List<Column> columnsList) –* возвращает DDL-описание таблиц с полями;
  + *abstract createConstraintDDL(Constraint constraint, List<ConstraintColumn> columns, Constraint refConstraint, List<ConstraintColumn> refColumns) –* возвращает DDL-описание ограничений.
* Классы *OracleDatabase* и *PostrgreDatabase* – наследники *AbstractDatabase*. Содержат перечисление типов специфичных для конкретной СУБД, реализующий интерфейс *TypesTemplates.*
* Интерфейс *TypesTemplates* - предоставляет интерфейс для получения шаблона. Для некоторых типов данных СУБД необходимо задавать параметры. Шаблон представляет из себя строку с %s в местах, где необходимо подставить значение. Пример “NUMBER(%s, %s)”, в таком случае методу getTemplate необходимо передать массив из двух параметров.

## Диаграммы последовательностей

Успешное создание файла с DDL-описанием (из СУБД Oracle в PostgreSQL):

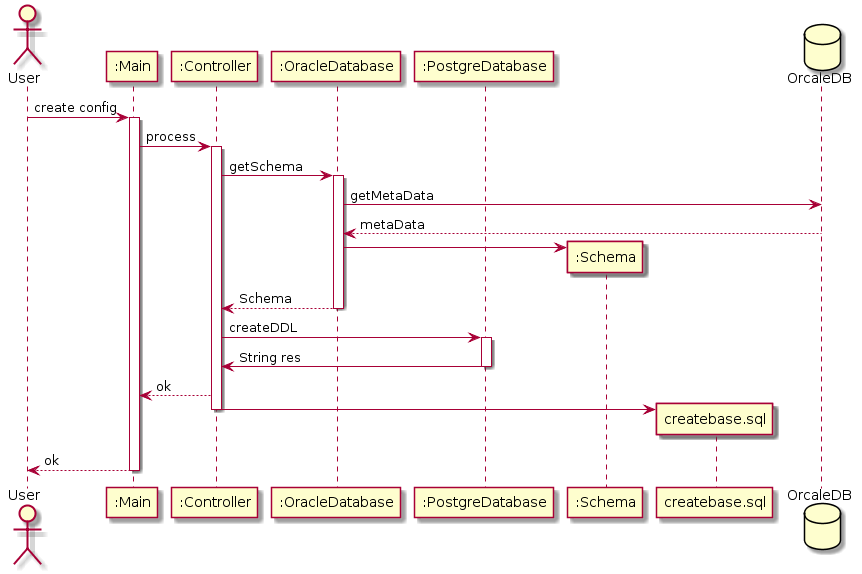


Рисунок 2. Диаграмма последовательности. Успешная миграция.

## Описание программы

Приложение реализовано на языке программирования Java версии 1.8 (Language level 8), в среде IntelliJ IDEA 2016.1.4. При создании проекта использовался Maven. В pom.xml файле описано из чего состоит проект. Также определены плагины – инструменты сборки проекта. В репозиториях артефактов хранятся продукты сборки программных модулей вместе с метаданными.

На данный момент реализовано получение схемы данных из СУБД Oracle и создание DDL-описания для PostgreSQL. В дальнейшем количество баз данных можно увеличивать. Для добавление новой базы данных необходимо создать класс наследник от AbstractDatabase. Зарегистрировать новую базу данных в классе Controller (добавить в перечисление Databases и добавить по одному case в соответствующие switch блоки), добавить jdbc-драйвер в “src\main\resources\jdbc\ ”.

Для осуществления миграции необходимо заполнить следующие параметры. Все параметры являются обязательными:

* URL доступа к БД - urlFrom = "jdbc:oracle:thin:@localhost:1521:orcl";
* Пользователь для доступа к БД - user = "ewa";
* Пароль пользователя БД - password = "\*\*\*\*\*";
* Наименование схемы для миграции - schemaFrom = "ewa";
* Наименование СУБД из перечисления Databases (которую необходимо мигрировать) - databaseFrom = Oracle;
* Наименование целевой схемы данных - schemaTo = "ewa";
* Наименование целевой СУБД из перечисления Databases - databaseTo = Databases.Postgres.

Для доступа к данным из СУБД Oracle использовался jdbc-драйвер ojdbc6.jar (расположение в проекте “src\main\resources\jdbc\ojdbc6.jar”). Для получение мета-описания схемы данных использовались следующие представления и запросы к ним:

* ALL\_TABLES – мета-описание таблиц

*select TABLE\_NAME*

*from SYS.ALL\_TABLES*

*where OWNER = [schemaFrom];*

* ALL\_TAB\_COLUMNS – мета-описание полей таблиц

*select COLUMN\_NAME, TABLE\_NAME, DATA\_TYPE, DATA\_LENGTH, DATA\_PRECISION, DATA\_SCALE, NULLABLE*

*from SYS.ALL\_TAB\_COLS*

*where OWNER = [schemaFrom]*

*order by COLUMN\_ID;*

* ALL\_CONSTRAINT – мета-описание ограничений

*select CONSTRAINT\_NAME, GENERATED, CONSTRAINT\_TYPE, TABLE\_NAME, SEARCH\_CONDITION\_VC, R\_CONSTRAINT\_NAME*

*from SYS.ALL\_CONSTRAINT*

*where owner = [schemafrom];*

* ALL\_CONS\_COLUMNS – мета-описание полей ограничений

*select CONSTRAINT\_NAME, TABLE\_NAME, COLUMN\_NAME, POSITION*

*from SYS.ALL\_CONS\_COLUMNS*

*where OWNER = [schemaFrom];*

Для сопоставления типов полей таблиц разных СУБД для каждой СУБД определяется коллекция вида «ключ – значение» соответствия () связывающая нормализованные атрибуты (*NormalizedTypes*) с специфичными для конкретной СУБД атрибутами (*ColumnTypes*). Все ключи и значения должны быть уникальны, это свойство проверяется на этапе создания обратной коллекции, в которой в качестве ключей начинают выступать значения (). Для всех типов требующих аргументы необходимо задать шаблон с методе *getTypeTemplate.*

*public enum NormalizedTypes {*

*NUMBER,*

*VARCHAR,*

*. . .*

*}*

*private enum ColumnTypes implements TypesTemplates {*

*CHARACTER,*

*NUMERIC;*

*@Override*

*public String getTypeTemplate(String[] args) {*

*switch (this) {*

*case CHARACTER:*

*return String.format("CHARACTER(%s)", args[0]);*

*case NUMERIC:*

*return String.format("NUMERIC(%s,%s)", args[1], args[2]);*

*default:*

*return this.name();*

*}*

*}*

*}*

*@Override*

*protected void initTypesMapping() {*

*typesMapping = new HashMap<TypesTemplates, DataTypes.NormalizedTypes>() {{*

*put(ColumnTypes.NUMERIC, DataTypes.NormalizedTypes.NUMBER);*

*put(ColumnTypes.CHARACTER, DataTypes.NormalizedTypes.VARCHAR);*

*. . .*

*}};*

*}*

Типы ограничений определяются в перечислении *ConstraintType*. Для установки порядка. В котором будут создаваться ограничения при сортировки используется метод *compare*:

*public enum ConstraintType {*

*C, // CHECK*

*P, // PRIVATE KEY*

*U, // UNIQUE*

*R; //* FOREIGN KEY и REFERENCES

*// foreign keys must be created last*

*public static int compare(ConstraintType ct1, ConstraintType ct2) {*

*if (ct1 == ct2 || (ct1 != R && ct2 != R)) {*

*return ct1.name().compareTo(ct2.name());*

*} else if (ct1 == R) {*

*return 1;*

*} else {*

*return -1;*

*}*

*}*

*}*

Результат работы программы – DDL-описание схемы данных записывается в файл *createbase.sql* находящийся в корневом каталоге.

## Методика и результаты тестирования

В процессе функционального тестирования в классе Main вручную прописывались параметры необходимые для работы приложения. В качестве тестируемых данных использовалась схема данных для приложения «Электронная фондовая биржа». Запрос на создание схемы данных для Oracle и результат выполнения программы представленны в таблице 1.

|  |  |
| --- | --- |
| Исходная схема данных Oracle | Результат выполнения createbase.sql |
| create table SHARES\_TABLE  (  share\_id VARCHAR2(100) not null,  share\_type VARCHAR2(100),  price NUMBER(20,4)  );  alter table SHARES\_TABLE  add primary key (SHARE\_ID);  alter table SHARES\_TABLE  add constraint PRICE\_NOT\_NULL  check ("PRICE" IS NOT NULL);  create table USERS\_TABLE  (  user\_id VARCHAR2(100) not null,  user\_pswd VARCHAR2(100),  group\_id VARCHAR2(100),  money NUMBER(20,4)  );  alter table USERS\_TABLE  add primary key (USER\_ID);  alter table USERS\_TABLE  add constraint MONEY\_NOT\_NULL  check ("MONEY" IS NOT NULL);  create table OWNERSHIPS  (  user\_owner\_id VARCHAR2(100) not null,  share\_own\_id VARCHAR2(100) not null,  share\_count NUMBER(10)  );  alter table OWNERSHIPS  add primary key (USER\_OWNER\_ID, SHARE\_OWN\_ID);    alter table OWNERSHIPS  add constraint OWNERFK foreign key (USER\_OWNER\_ID)  references USERS\_TABLE (USER\_ID);  alter table OWNERSHIPS  add constraint OWNSHAREFK foreign key (SHARE\_OWN\_ID)  references SHARES\_TABLE (SHARE\_ID);  create table TRADES  (  trade\_id NUMBER(10) default "EWA"."TRADES\_SEQ"."NEXTVAL" not null,  trade\_type NUMBER(2),  share\_share\_id VARCHAR2(100) not null,  share\_count NUMBER(10) not null,  user\_initiator\_id VARCHAR2(100) not null,  user\_client\_id VARCHAR2(100),  price NUMBER(20,4) not null  );  alter table TRADES  add primary key (TRADE\_ID);  alter table TRADES  add constraint CLIENTFK foreign key (USER\_CLIENT\_ID)  references USERS\_TABLE (USER\_ID);  alter table TRADES  add constraint INITIATORFK foreign key (USER\_INITIATOR\_ID)  references USERS\_TABLE (USER\_ID);  alter table TRADES  add constraint SHAREFK foreign key (SHARE\_SHARE\_ID)  references SHARES\_TABLE (SHARE\_ID);  alter table TRADES  add check (trade\_type in (0, 1)); | CREATE TABLE ewa.OWNERSHIPS (  USER\_OWNER\_ID CHARACTER(100) not null,  SHARE\_OWN\_ID CHARACTER(100) not null,  SHARE\_COUNT NUMERIC(10,0)  );  CREATE TABLE ewa.TRADES (  TRADE\_ID NUMERIC(10,0) not null,  TRADE\_TYPE NUMERIC(2,0),  SHARE\_SHARE\_ID CHARACTER(100) not null,  SHARE\_COUNT NUMERIC(10,0) not null,  USER\_INITIATOR\_ID CHARACTER(100) not null,  USER\_CLIENT\_ID CHARACTER(100),  PRICE NUMERIC(20,4) not null  );  CREATE TABLE ewa.USERS\_TABLE (  USER\_ID CHARACTER(100) not null,  USER\_PSWD CHARACTER(100),  GROUP\_ID CHARACTER(100),  MONEY NUMERIC(20,4) not null  );  CREATE TABLE ewa.SHARES\_TABLE (  SHARE\_ID CHARACTER(100) not null,  SHARE\_TYPE CHARACTER(100),  PRICE NUMERIC(20,4) not null  );  ALTER TABLE ewa.TRADES ADD CHECK (trade\_type in (0, 1));  ALTER TABLE ewa.TRADES ADD PRIMARY KEY (TRADE\_ID);  ALTER TABLE ewa.OWNERSHIPS ADD PRIMARY KEY (USER\_OWNER\_ID, SHARE\_OWN\_ID);  ALTER TABLE ewa.SHARES\_TABLE ADD PRIMARY KEY (SHARE\_ID);  ALTER TABLE ewa.USERS\_TABLE ADD PRIMARY KEY (USER\_ID);  ALTER TABLE ewa.TRADES ADD CONSTRAINT CLIENTFK FOREIGN KEY (USER\_CLIENT\_ID) REFERENCES ewa.USERS\_TABLE (USER\_ID);  ALTER TABLE ewa.TRADES ADD CONSTRAINT INITIATORFK FOREIGN KEY (USER\_INITIATOR\_ID) REFERENCES ewa.USERS\_TABLE (USER\_ID);  ALTER TABLE ewa.TRADES ADD CONSTRAINT SHAREFK FOREIGN KEY (SHARE\_SHARE\_ID) REFERENCES ewa.SHARES\_TABLE (SHARE\_ID);  ALTER TABLE ewa.OWNERSHIPS ADD CONSTRAINT OWNERFK FOREIGN KEY (USER\_OWNER\_ID) REFERENCES ewa.USERS\_TABLE (USER\_ID);  ALTER TABLE ewa.OWNERSHIPS ADD CONSTRAINT OWNSHAREFK FOREIGN KEY (SHARE\_OWN\_ID) REFERENCES ewa.SHARES\_TABLE (SHARE\_ID); |

Таблица 1.

Затем получившийся DDL-запрос исполнялся в pgAdmin 4. В результате выполнения запроса создались четыре таблицы со всеми ограничениями. Query returned successfully in 224 msec.

При вводе некорректных данных выполнение программы завершается ошибкой.

# Выводы

В результате выполнения работы был рассмотрен и применен на практике общий механизм взаимодействия с базой данных с помощью языка программирования Java. Рассмотрен механизм хранения мета-данных для СУБД Oracle, а так же DDL-описание схемы данных для СУБД PostreSQL.

Было разработано приложение для миграции схемы данных. На данном этапе приложение удовлетворяет всем заявленным требованиям. Поскольку миграция схемы данных довольно серьезная и обширная задача, приложение было разработано с учетом возможности дальнейшего масштабирования. Масштабирование можно производить в двух направлениях: увеличение количества поддерживаемых СУБД и увеличение полноты мигрируемых метаданных, причем это является более приоритетной задачей. Следует добавить последовательности, процедуры, триггеры и так далее. Так же существенной доработкой может стать миграция из DDL-описания схемы. Из не существенных доработок можно отметить создание некоторого UI, создание схемы данных напрямую, добавление более детальной обработки исключений.

# Приложение

## Код программы