Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Московский государственный технический университет имени Н.Э. Баумана» (МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления (ИУ)

КАФЕДРА Теоретическая информатика и компьютерные технологии (ИУ-9)

Объектно-реляционное отображение

Тришин А.А.

Группа ИУ9-71

Руководитель

Коновалов А.В.

Москва, 2018г.

Оглавление

Введение

Сегодня любая IT-разработка очень тесно связана с использованием баз данных. Чаще всего приходится работать с реляционными СУБД. Поэтому приходится использовать различные драйвера к базам данных для необходимых языков программирования, что приводит к постоянному написанию SQL-запросов внутри кода программы, к использованию большого количества необходимых классов, чтобы выполнить достаточно простые операции получения/изменения информации внутри базы.

Таким образом, появляется задача обеспечения работы с данными в терминах классов/объектов, а не таблиц. Необходимо преобразовать термины и данные классов в данные, пригодные для хранения в СУБД, а также обеспечить интерфейс для операций создания, получения, обновления, удаления (create, read, update, delete – CRUD) данных, и в целом избавиться от написания SQL-запросов в коде программы для взаимодействия с СУБД.

Решением проблемы является объектно-реляционное отображение (object relational mapping – ORM) – технология, связывающая объекты в рамках понятий баз данных и объекты в рамках объектно-ориентированного программирования. Применение объектно-реляционного отображения в настоящее время является общим средством в процессе разработки сложных систем, позволяющим объединить объектно-ориентированную модель представления данных с реляционной.

ORM является промежуточным слоем между базой данных и кодом программы, которая позволяет выполнять CRUD-операции в базе данных, в частности, в таблицах, с помощью использования специального API и ООП-объектов.

Целью данной работы является реализация подобного отображения, которое можно будет использовать на практике.

1. Обзор существующих решений

1.1 Технология Hibernate

Hibernate[1] – одна из наиболее популярных технологий для решения задач объектно-реляционного отображения, то есть освобождает разработчика от значительного объёма сравнительно низкоуровневого программирования при работе с объектно-ориентированными языками программирования и реляционными базами. Hibernate можно использовать как для работы с уже существующей базой данных, так и для работы «с нуля». Поддерживает только язык Java.

Непосредственное проецирование Java-классов с таблицами осуществляется при помощи конфигурации XML-файлов и Java-аннотаций. В первом случае классы сами генерируется на основе файлов.

Таким образом, чтобы выполнять простейшие CRUD операции необходимо сконфигурировать XML-файл, пример представлен в листинге 1.

```
<hibernate-mapping>
    <class name="ru.bmstu.BankBillingServer.AccountsEntity.AccountsEntity"
table="accounts" schema="billing_schema"
           catalog="billingdb">
        <id name="id">
            <column name="id" sql-type="integer"/>
        </id>
        <property name="money">
            <column name="money" sql-type="double precision" precision="-1"/>
        </property>
    </class>
</hibernate-mapping>
```

Листинг 1. Пример маппинг-файла.

Затем генерируется или пишется самостоятельно Java класс, пример показан в листинге 2.

```
@Entity
@Table(name = "accounts", schema = "billing_schema", catalog = "billingdb")
public class AccountsEntity {
    private int id;
    private double money;

    @Id
    @Column(name = "id", nullable = false)
    public int getId() {…}

    public void setId(int id) {…}
```

```
    @Basic
    @Column(name = "money", nullable = false, precision = 0)
    public double getMoney() {…}

    public void setMoney(double money) {…}

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        AccountsEntity that = (AccountsEntity) o;
        return id == that.id;
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, money);
    }
}
```

Листинг 2. Пример файла с маппинг-классом для таблицы.

После необходимо реализовать фабрику сессий и только потом можно выполнять необходимые операции. Чтобы создать объект на основе строки из таблицы базы данных предлагается получить его по первичному ключу или на основе результата специального языка запросов Hibernate HQL.

Также стоит отметить, что Hibernate не позволяет работать с таблицами, в которых нет первичного ключа, а доступ к хранимым процедурам и функциям получается очень громоздким (пример на листинге 3).

```
StoredProcedureQuery query = entityManager
    .createStoredProcedureQuery("count_comments")
    .registerStoredProcedureParameter(
        "postId", Long.class, ParameterMode.IN)
    .registerStoredProcedureParameter(
        "commentCount", Long.class, ParameterMode.OUT)
    .setParameter("postId", 1L);

query.execute();

Long commentCount = (Long) query.getOutputParameterValue("commentCount");
```

Листинг 3. Пример вызова функции в Hibernate.

Таким образом, явными преимуществами является возможность гибкой настройки Hibernate’а под необходимую базу данных, однако это влечет за собой дополнительное написание соответствующих классов и файлов. Существует механизм работы с базой данных с помощью транзакций. API CRUD операций достаточно простой в примитивных ситуациях, когда известен первичный ключ строки, но всё равно приходится работать со

строками и писать запросы. Еще одним недостатком является то, что пользователю необходимо самому писать так называемые Data Access Objects (объекты для получения доступа к данным), что при большом количестве схем и баз данных приводит к такому же большому количеству однотипных классов и объектов.

1.2 Технология JOOQ

Java Object Oriented Querying (JOOQ[2]) – библиотека для языка программирования Java, которая позволяет сопоставлять таблицы базы данных и ООП-объекты. Поддерживает языки Java, Scala, Kotlin.

Основной особенностью данной библиотеки, которая отличает её от остальных, является оперирование классическими методами ООП программирования вместо работы со строками таким образом, каким пришлось бы писать запрос на SQL, пример представлен на листинге 4.

```
SELECT AUTHOR.FIRST_NAME,
AUTHOR.LAST_NAME, COUNT(*)
FROM AUTHOR
JOIN BOOK ON AUTHOR.ID =
BOOK.AUTHOR_ID
WHERE BOOK.LANGUAGE = 'DE'
AND BOOK.PUBLISHED > DATE
'2008-01-01'
GROUP BY AUTHOR.FIRST_NAME,
AUTHOR.LAST_NAME
HAVING COUNT(*) > 5
ORDER BY AUTHOR.LAST_NAME
ASC NULLS FIRST
LIMIT 2
OFFSET 1
```

```
create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME,
count())
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.equal(BOOK.AUTHOR_ID))
      .where(BOOK.LANGUAGE.eq("DE"))
      .and(BOOK.PUBLISHED.gt(date("2008-01-01")))
      .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .having(count().gt(5))
      .orderBy(AUTHOR.LAST_NAME.asc().nullsFirst())
      .limit(2)
      .offset(1)
```

Листинг 4. Пример SQL кода (слева), и Java кода с использованием JOOQ (справа).

JOOQ сама генерирует необходимые классы для работы с базой данных, а именно отдельно классы для таблиц и классы для записей в таблицах.

Таким образом, достоинствами данной библиотеки являются:

- пользователю не нужно самому писать классы для работы с базой;
- пользователь не работает со строками;
- есть строенный оптимизатор, который может оптимизировать запросы;

– вызов функций проще, чем в Hibernate (выполняется также как простой select запрос), однако он не выглядит как естественный вызов метода в рамках ООП.

Присутствуют следующие недостатки:

– существует специальный «язык» запросов, который реализует лишь подмножество SQL;
– генерируемые классы очень большие, содержат большое количество однотипных и нефункциональных методов.

1.3 Технология Apache Cayenne

Apache Cayenne[3] – платформа с открытым исходным кодом, которая обеспечивает объектно-реляционное отображение данных и позволяет создавать удалённые сервисы для доступа к ним. Первое достигается благодаря тому, что Cayenne легко связывает одну или несколько баз данных напрямую с Java-объектами, которые автоматически управляют транзакциями и последовательностями, генерируют SQL-запросы и т.д. Второе достигается благодаря постоянству удалённых объектов Cayenne (Cayenne's Remote Object Persistence), что позволяет выгружать такие объекты клиентам через веб-сервисы.

Cayenne спроектирован таким образом, чтобы максимально облегчить работу с данными без потери гибкости или структуры этих данных. Для достижения этой цели, Cayenne поддерживает как проектирование и генерацию базы данных по классам, так и обратную генерацию классов по базе данных.

Чтобы получить объект, достаточно получить контекст (ObjectContext) и вызвать необходимый метод, пример представлен в листинге 5.

```
ServerRuntime cayenneRuntime = new ServerRuntime("cayenne-project.xml");
            ObjectContext context = cayenneRuntime.getContext();
SelectQuery select1 = new SelectQuery(Painting.class);
List paintings1 = context.performQuery(select1);
```

Листинг 5. Пример получения объекта в Apache Cayenne.

Таким образом, можно выделить следующие достоинства:

- простое API работы с объектами;
- классы генерируются сами, и они просты для понимания и работы с ними;
- есть GUI для упрощённого использования;
- не нужно создавать DAO классы.
- механизм вызова функций и хранимых процедур схож с SQL (query.addParameter("paramter1", "abc"); QueryResponse result = context.performGenericQuery(query);)

Недостатки:

- меньшая гибкость по сравнению с Hibernate.

2. Разработка собственной ORM библиотеки

2.1 Архитектура библиотеки

Библиотека объектно-реляционного отображения в первую очередь должна решать две задачи:

– генерирование классов на целевом языке программирования по описанию таблицы и функций;

– предоставление определённого API, с помощью которого можно работать с ORM-объектами вместо написания запросов в базе.

Таким образом, можно выделить две части всей библиотеки, которые отвечают поставленным задачам соответственно – генератор классов для объектно-реляционного отображения и определённый набор классов, с помощью которых можно будет взаимодействовать с базой данных, используя объекты сгенерированных классов. Получается схема, представленная на рисунке 1.

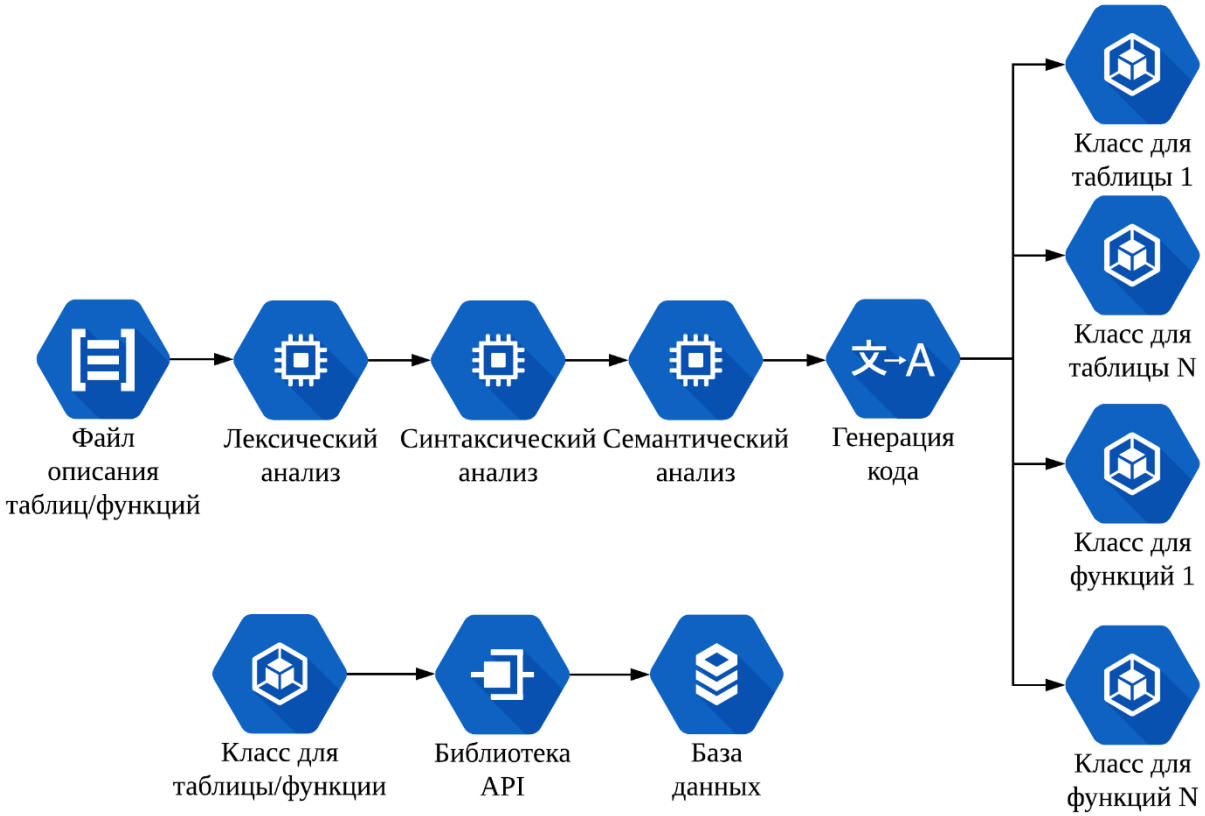


Рисунок 1. Схема работы библиотеки.

2.2 Генератор классов

Генератор классов представляет собой обычный транслятор, состоящий из четырёх последовательных этапов:

1. лексический анализ – объединение последовательных кодовых точек в лексемы;
2. синтаксический анализ – объединение лексем в синтаксические структуры и построение синтаксического дерева;
3. семантический анализ – проверка синтаксического дерева на соответствие его компонентов контекстным ограничениям (проверка соответствия типов данных, областей видимости и т.д.)
4. Генерация кода – генерация классов на целевом языке.

На первом этапе программе подаётся файл с кодом, написанным на SQL. Лексер по требованию парсера считывает кодовые точки и возвращает новую лексему. Парсер строит дерево разбора и передаёт его семантическому анализатору, который выполняет проверки на соответствие типов данных в выражениях, в столбцах таблиц, а также проверки на видимость объектов (столбцов, переменных и т.д.). После этого кодогенератор создаёт классы на основе полученных данных.

2.3 Грамматика поддерживаемого подмножества SQL

Лексическое описание поддерживаемого подмножества SQL (подмножество PostgreSQL[4]):

- ключевые слова: CREATE, TABLE, IF, NOT, EXISTS, ARRAY, RECORD, BOOLEAN, INT, INTEGER, SMALLINT, BIGINT, REAL, FLOAT, DOUBLE, PRECISION, DECIMAL, NUMERIC, CHARACTER, CHAR, VARCHAR, TIMESTAMP, TIME, DATE, CONSTRAINT, UNIQUE, PRIMARY, FOREIGN, KEY, REFERENCES, NOT, NULL, CHECK, DEFAULT, ON, UPDATE, DELETE, NO, ACTION, RESTRICT, CASCADE, SET, NULL, DEFAULT, OR, AND, TRUE, FALSE,

BETWEEN, LIKE, REPLACE, FUNCTION, RETURNS, VOID, IN, OUT, INOUT, AS, LANGUAGE, PLPGSQL;

- идентификаторы вида: [a-zA-Z]*[0-9]*;
- специальные символы: ‘;’, ‘.’, ‘[‘, ‘]’, ‘(‘, ‘)’, ‘,’, ‘+’, ‘-’, ‘*’, ‘/’, ‘’’, ‘<’, ‘>’, ‘=’, ‘!=’, ‘<=’, ‘>=’, ‘::’, ‘$$’;
- числа: десятичные и с плавающей точкой;
- строковые литералы вида: ‘.*’;
- комментарии вида: --.*\n;

Синтаксическое описание в формате РБНФ:

- S ::= (CREATE CreateTableFunction)*
- CreateTableFunction ::= (CreateTableStmt ';') | (CreateFunctionStmt ';')
- CreateTableStmt ::= TABLE (IF NOT EXISTS)? QualifiedName '(' (TableElement (',' TableElement)*)? ')'
- QualifiedName ::= IDENT('.'IDENT)*
- TableElement ::= ColumnDef | TableConstraint
- ColumnDef ::= IDENT Typename ColConstraint*
- Typename ::= SimpleTypename ArrayType?
- ArrayType ::= ('[' intConst? ']')+ | ARRAY ('[' intConst ']')?
- SimpleTypename ::= NumericType | CharacterType | DateTimeType | RECORD | BOOLEAN
- NumericType ::= INT | INTEGER | SMALLINT | BIGINT | REAL | FLOAT ('('intConst')')? | DOUBLE PRECISION | DECIMAL | NUMERIC
- CharacterType ::= CharacterKeyword ('(' intConst ')')?
- CharacterKeyword ::= CHARACTER | CHAR | VARCHAR
- DateTimeType ::= TIMESTAMP ('(' intConst ')')? | TIME ('(' intConst ')')? | DATE
- TableConstraint ::= CONSTRAINT IDENT ConstraintElem | ConstraintElem

- ConstraintElem ::= UNIQUE '(' IDENT (',' IDENT)* ')' | PRIMARY KEY '(' IDENT (',' IDENT)* ')' | FOREIGN KEY '(' IDENT (',' IDENT)* ')' REFERENCES QualifiedName ('(' IDENT (',' IDENT)* ')')? KeyActions?
- ColConstraint ::= CONSTRAINT IDENT ColConstraintElem | ColConstraintElem
- ColConstraintElem ::= NOT NULL | NULL | UNIQUE | PRIMARY KEY | CHECK '(' BoolExpr ')' | DEFAULT ConstExpr | REFERENCES QualifiedName ('(' IDENT ')')? KeyActions?
- KeyActions ::= ON UPDATE KeyAction (ON DELETE KeyAction)? | ON DELETE KeyAction (ON UPDATE KeyAction)?
- KeyAction ::= NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT
- ArithmExpr ::= ArithmExprTerm ({'+' | '-'} ArithmExprTerm)*
- ArithmExprTerm ::= ArithmExprFactor ({'*' | '/'} ArithmExprFactor)*
- ArithmExprFactor ::= IDENT | NumericValue | '-' ArithmExprFactor
- BoolExpr ::= BoolExprTerm (OR BoolExprTerm)*
- BoolExprTerm ::= BoolExprFactor (AND BoolExprFactor)*
- BoolExprFactor ::= BoolConst BoolRHS? | NOT BoolExprFactor BoolRHS? | '(' BoolExpr ')' BoolRHS? | IDENT RHS? | ArithmExpr ArithmRHS
- BoolConst ::= TRUE | FALSE | NULL
- RHS ::= BoolRHS | DateRHS | StringRHS
- ArithmRHS ::= '<' ArithmExpr | '<=' ArithmExpr | '>' ArithmExpr | '>=' ArithmExpr | '=' ArithmExpr | '!=' ArithmExpr | NOT? BETWEEN ArithmExpr AND ArithmExpr
- BoolRHS ::= IS NOT? BoolConst
- DateRHS ::= '<' DateTimeCast | '<=' DateTimeCast | '>' DateTimeCast | '>=' DateTimeCast | '=' DateTimeCast | '!=' DateTimeCast | NOT? BETWEEN DateTimeCast AND DateTimeCast

- StringRHS ::= LIKE CharacterValue
- ConstExpr ::= ArithmConstExpr | NOT? BoolConst | CharacterValue | DateTimeCast
- DateTimeCast ::= DateValue'::'DATE | TimeValue'::'TIME | TimestampValue'::'TIMESTAMP
- ArithmConstExpr ::= ArithmConstExprTerm ({'+'|'-'} ArithmConstExprTerm)*
- ArithmConstExprTerm ::= ArithmConstExprFactor ({'*'|'/'} ArithmConstExprFactor)*
- ArithmConstExprFactor ::= NumericValue | '-' ArithmConstExprFactor | '(' ArithmConstExpr ')'
- CreateFunctionStmt ::= (OR REPLACE)? FUNCTION QualifiedName '(' funcArgsWithDefaultsList? ')' RETURNS CreateFunctionReturnStmt CreateFuncBody
- CreateFunctionReturnStmt::= Typename|TABLE '(' TableFuncColumnList ')' | VOID
- TableFuncColumnList ::= IDENT Typename (',' IDENT Typename)*
- funcArgsWithDefaultsList ::= funcArgWithDefault(',' funcArgWithDefault)*
- funcArgWithDefault ::= funcArg funcArgDefault?
- funcArgDefault ::= DEFAULT constExpr | '=' constExpr
- funcArg ::= argClass IDENT? Typename | IDENT argClass? Typename | Typename
- argClass ::= IN OUT? | OUT | INOUT
- CreateFuncBody ::= AS '$$' /*skip*/ '$$' LANGUAGE plpgsql | LANGUAGE plpgsql AS '$$' /*skip*/ '$$'

2.4 Реализация лексического и синтаксического анализатора

В качестве языка реализации, а также целевого языка был выбран язык программирования Java. Для написания лексического анализатора использовался объектно-ориентированный подход. Для синтаксического

анализа был написан парсер РБНФ грамматики рекурсивным спуском. Схема взаимодействия всех компонент представлена на рисунке 2.

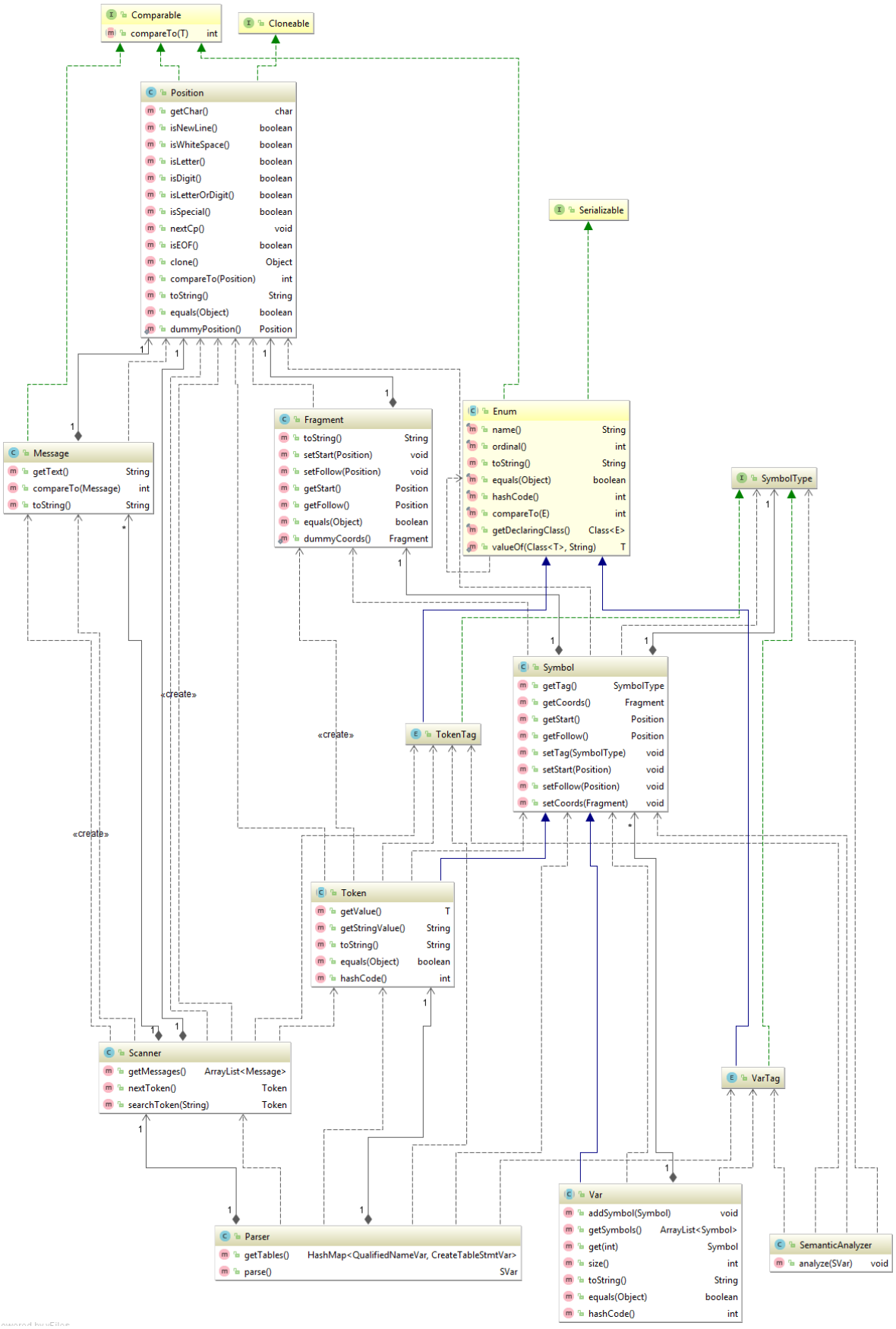


Рисунок 2. Схема взаимодействия компонент анализатора.

Классы Token и Var – абстрактные, отвечают за терминальные и нетерминальные символы соответственно, являются наследниками их общего класса-предка – Symbol. У любого символа есть свой доменный тег – интерфейс SymbolType. Перечисления TokenTag и VarTag реализуют данный интерфейс. Для хранения информации о текущей кодовой точке и её координатах используются классы Position и Fragment соответственно. Класс Message хранит информацию о сообщениях, порожденных компилятором. Класс Scanner является итератором по токенам. Его метод nextToken осуществляет лексический анализ. Синтаксический анализ осуществляет объект класса Parser, он хранит в себе объект класса Scanner и вызывает его метод nextToken. После синтаксического анализа SemanticAnalyzer проводит семантический анализ.

3. Кодогенерация и API библиотеки

3.1 Разработка шаблонов генерируемого кода

Результатом работы компилятора является Java-класс (для таблиц унаследованный от Entity, для функций – от FunctionsExecutor), размеченный специально разработанными аннотациями:

- @Table(String db, String schema, String name) – помечается класс, db – название базы данных, schema – название схемы, name – название таблицы;
- @Column(String name, boolean unique, boolean nullable, int length) – помечается каждое поле-столбец генерируемого класса. Name – название столбца, unique – необязательный флаг, указывающий на уникальность поля, по умолчанию – false, nullable – необязательный флаг, указывающий на возможность хранить значение null, по умолчанию – true, length – для строковых полей-столбцов – длина хранимой строчки, по умолчанию – 255;
- @PK – помечается поле-столбец, которое является первичным ключом или частью составного первичного ключа;
- @FK(String table, String referencedColumn) – помечается поле-столбец, которое является внешним ключом или частью внешнего ключа, table – таблица, на которую ссылается внешний ключ, referencedColumn – столбец, на который ссылается внешний ключ;
- @Default – помечается поле-столбец, это флаг, который обозначает, что поле имеет значение по умолчанию.
- @FO(String table) – помечается поле, тип которого является другой ORM-класс;
- @Routines(String db, String schema) – помечается класс, который используется для работы с функциями в базе данных.

Таким образом, SQL-код описание таблиц и функций преобразуется в специальные размеченные Java-классы, на листинге 6 приведён пример результата работы компилятора.

```
CREATE TABLE IF NOT EXISTS ShopDB.shopschema.Employee
(
  employeeCode INT PRIMARY KEY NOT NULL,
  name VARCHAR(25) NOT NULL,
  dateOfBirth DATE NOT NULL,
  phone CHAR(11) NOT NULL UNIQUE,
  isFired BOOLEAN DEFAULT FALSE NOT NULL,
  salary REAL NOT NULL,
  shopCode INT,

  FOREIGN KEY (shopCode) REFERENCES ShopDB.ShopSchema.Shop
(shopCode) ON  DELETE CASCADE
);

CREATE or replace function shopdb.shopschema.go(c int = 2, k
int) returns int as $$
begin
  if (1>2) then
    return 1;
  else
    return 2;
end;
$$ language plpgsql;
```

а)

```
@Table(db = "shopdb", schema = "shopschema", name = "employee")
public class Employee implements Entity {
	@PK
	@Column(name = "employeecode", nullable = false)
	private Integer employeecode;

	@Column(name = "name", nullable = false, length = 25)
	private String name;

	@Column(name = "dateofbirth", nullable = false)
	private Date dateofbirth;

	@Column(name = "phone", unique = true, nullable = false,
length = 11)
	private String phone;

	@FK(table = "shop", referencedColumn = "shopcode")
	@Column(name = "shopcode")
	private Integer shopcode;

	@FO(table = "shop")
	private Shop shop;
```

```
        public Integer getEmployeecode() {
                return this.employeecode;
        }

        /*get-еры и set-еры для всех других полей*/

        public void setShopcode(Integer shopcode) {
                this.shopcode = shopcode;
        }

        @Override
        public boolean equals(Object obj) {
                /* основан на PK */
        }

        @Override
        public int hashCode() {
                return Objects.hashCode(this.employeecode);
        }


        @Override
        public String toString() {
                ...
        }
}

@Routines(db = "shopdb", schema = "shopschema")
public class GoFunction extends FunctionsExecutor {
        public GoFunction(Connection connection) {
                super(connection);
        }

        public Integer go(Integer c, Integer k) {
                return (Integer)
super.executeScalarFunction(this.getClass(),
String.format("go(%s, %s)", c, k));
        }

        public Integer go(Integer k) {
                return (Integer)
super.executeScalarFunction(this.getClass(),
String.format("go(%s)", k));
        }

}
```

б)

Листинг 6. Пример работы компилятора: а) SQL-код, подающийся на вход, б) получаемый Java-код.

3.2 Описание API ORM-библиотеки

Работа с базой осуществляется путём использования класса Session, который представляет собой непосредственное подключение к ней и реализует интерфейс AutoCloseable. Объекты данного класса создаются только с помощью фабрики класса SessionFactory, конструктор которого принимает на вход хост, порт, пользователя и его пароль для подключения к базе. С помощью метода openSession() и порождается необходимый объект и автоматически открывается соединение с базой (пример на листинге 7).

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

Листинг 7. Пример создания объекта класса Session.

final class Session implements AutoCloseable имеет следующие доступные методы:

- void close() – закрывает соединение с базой данных.
- boolean isClosed() – возвращает булевское значение, является ли соединение закрытым;
- boolean contains(Entity entity) – возвращает булевское занчение, существует ли таблица entity в базе;
- void save(Entity entity) – сохраняет объект класса Entity в базу (аналог INSERT INTO);
- void update(Entity entity) – обновляет значения в базе данных объекта entity (аналог UPDATE … SET);
- void delete(Entity entity) – удаляет объект entity в базе данных (аналог DELETE FROM);
- <T extends Entity> SelectClause<T> selectFrom(Class<T> table) – возвращает объект SelectClause<T>, с помощью которого можно получать объекты Entity (аналог SELECT … FROM);
- <T extends FunctionsExecutor> T getFunctions(Class<T> clazz) – возвращает объект класса-наследника FunctionsExecutor, который используется для работы с функциями в базе данных.

abstract class Fetchable<T extends Entity> имеет следующие доступные методы:

- String getSelectStmt() – возвращает текущий построенный SQL-запрос;
- java.sql.Connection getConnection() – возвращает текущее подключение к базе;
- List<T> fetchAll() – возвращает все ORM-объекты, которые получаеются при выполнении текущего SQL-запроса;
- T fetchFirst() – возвращает первый ORM-объект, который является первой строкой в результате выполнения текущего SQL-запроса.

class SelectClause<T extends Entity> extends Fetchable<T> implements WhereAble<T>, GroupByAble<T>, OrderByAble<T> имеет доступные методы:

- WhereClause<T> where(String whereClause) – добавляет WHERE … к текущему SQL-запросу;
- GroupByClause<T> groupBy(String groupByClause) – добавляет GROUP BY к текущему SQL-запросу;
- OrderByClause<T> orderBy(String orderByClause) – добавляет ORDER BY к текущему SQL-запросу;
- T getById(Map<String, Object> id) – возвращает строчку таблицы в виде ORM-объекта по составному первичному ключу;
- T getById(Serializable id) – возвращает строчку таблицы в виде ORM-объекта по первичному ключу.

class WhereClause<T extends Entity> extends Fetchable<T> implements GroupByAble<T>, OrderByAble<T> имеет следующие методы:

- WhereClause<T> and(String whereClause) – добавляет AND к текущему SQL-запросу;
- GroupByClause<T> groupBy(String groupByClause) – добавляет GROUP BY к текущему SQL-запросу;
- OrderByClause<T> orderBy(String orderByClause) – добавляет ORDER BY к текущему SQL-запросу.

class GroupByClause<T extends Entity> extends Fetchable<T> implements HavingAble<T>, OrderByAble<T> имеет следующие методы;

– HavingClause<T> having(String havingClause) – добавляет HAVING к текущему SQL-запросу;
– OrderByClause<T> orderBy(String orderByClause) – добавляет ORDER BY к текущему SQL-запросу.

class HavingClause<T extends Entity> extends Fetchable<T> implements OrderByAble<T> имеет следующие методы:

– OrderByClause<T> orderBy(String orderByClause) – добавляет ORDER BY к текущему SQL-запросу.

Класс OrderByClause<T extends Entity> extends Fetchable<T> имеет только конструктор и методы Fetchable.

class FunctionsExecutor имеет следующие методы:

– <T extends FunctionsExecutor> Object executeScalarFunction(Class<T> clazz, String function) – возвращает результат выполнения скалярной функции;
– <T extends FunctionsExecutor, P extends ReturnedTable> ArrayList<P> executeTableFunction(Class<T> clazz, Class<P> tableClass, String function) – возвращает результат выполнения табличной функции (класс ReturnedTabe является упрощенным классом Entity, так как необходим только для хранения результатов табличных функций);
– <T extends FunctionsExecutor> void executeVoidFunction(Class<T> clazz, String function) – выполняет хранимую процедуру.

Таким образом, предполагается следующий порядок действий при работе с данной библиотекой (пример на листинге 8):

1. получить объект класса Session;
2. получить объект класса Entity;
3. сохранить/обновить/удалить объект класса Entity.

```
Session session = sessionFactory.openSession();
List<Shop> shop = session.selectFrom(Shop.class).where("shopCode
= 101").fetchAll();
shop.get(0).setName("new Name");
session.update(shop.get(0));
session.close();
```

Листинг 8. Пример работы с ORM-библиотекой.

4. Тестирование

4.1 Тестирование компилятора

Для тестирования были выбраны SQL-запросы для таблиц из курсовой работы по базам данных, которые демонстрируют поддержку возможных типов данных, а именно таблица, описывающая выписанный по покупке чек («Check»), и таблица, которая содержит в себе информацию о покупках по скидочной карте («Card_Check_INT»), а также функция проверки валидности вида товара. В приложении А.1 приведены запросы для создания данных таблиц, а в приложении А.2 результат работы компилятора. Видно, что все типы данных корректным образом преобразовались, в том числе, массивы – в списки (так как в PostgreSQL массивы работают именно как списки, а не как массивы в классическом понимании), тип времени. Также корректным образом сгенерировался класс для функции. Стоит заметить, что для каждого класса сгенерировались еще и методы equals() и hashCode(), которые позволяют более эффективно использовать объекты ORM классов в каких либо классах-коллекциях.

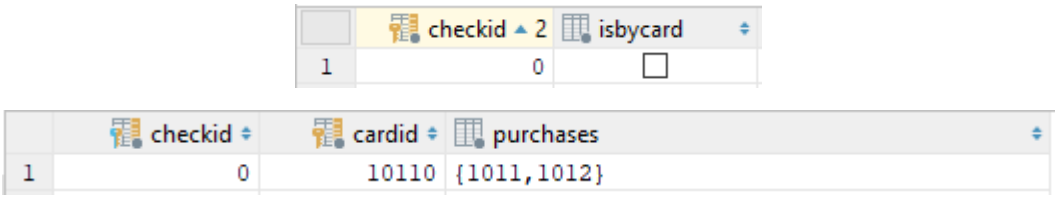
4.2 Тестирование библиотеки

Сгенерированные классы использовались для тестирования ORM-библиотеки. На листинге 9 приведен пример выполнения CRUD операций с помощью написанного API, а на рисунке 3 – результат выполнения Java-кода.

```
Session session = sessionFactory.openSession();
Check moscowCheck = session
        .selectFrom(Check.class)
        .where("shopCode = 100")
        .fetchFirst();
moscowCheck.setIsbycard(false);
session.update(moscowCheck);
Card_check_int card_check_int = new Card_check_int();
card_check_int.setCardid(10110);
card_check_int.setCheckid(moscowCheck.getCheckid());
```

```
ArrayList<Integer> purchases = new ArrayList<>();
purchases.add(1011);
purchases.add(1012);
card_check_int.setPurchases(purchases);
session.save(card_check_int);
```

Листинг 9. Пример использования API библиотеки.



	checkid ▲ 2	isbycard
1	0	☐

	checkid	cardid	purchases
1	0	10110	{1011,1012}

Рисунок 3. Результат выполнения кода. Данные взяты из приложения DataGrip от JetBrains.

Руководство пользователя

Данная курсовая работа состоит из двух частей, поэтому есть два .jar файла – с ORM-библиотекой и ORM-компилятором. ORM-библиотека для использования подключается как обычная jar-зависимость в java проекте.

Чтобы скомпилировать исходный файл SQL нужно запустить программу, указав первым аргументом путь до SQL-файла. Второй аргумент опциональный, он содержит путь до папки, в которую необходимо положить сгенерированные java-файлы, по умолчанию они кладутся в текущую директорию. Если аргументов нет, то программа выводит сообщение со справкой по использованию.

Чтобы собрать библиотеку из исходников, необходимо, склонировать проект, затем находясь в корне проекта, выполнить: mvn install -DskipTests. Тогда в папке target появится jar файл ORMc.jar, который содержит в себе компилятор и ORM библиотеку. Этот .jar файл, необходимо подключить в качестве зависимости к проекту, чтобы использовать библиотеку.

Рассмотрим пример, пусть есть SQL исходник с таблицей квадратов и кубов целых чисел (см. листинг 9).

```
CREATE TABLE powers (
  x INT PRIMARY KEY NOT NULL,
  square INT NOT NULL,
  cube INT NOT NULL,
  PRIMARY KEY (x)
);
```

Листинг 9. Пример исходного SQL файла.

Чтобы его скомпилировать в ORM-класс, необходимо запустить программу в командной строке: java -jar ORMc.jar powers.sql src/main/test. В папке src/main/test появится следующий файл, представленный на листинге 10:

```
@Table(db = "math", schema = "public", name = "powers")
public class Powers extends Entity {
   @PK
   @Column(name = "x", nullable = false)
   private Integer x;

   @PK
   @Column(name = "square", nullable = false)
   private Integer square;
```

```
    @PK
    @Column(name = "cube", nullable = false)
    private Integer cube;
         /*набор методов*/
    }
```

Листинг 10. Скомпилированный класс.

Чтобы заполнить таблицу при помощи библиотеки остается написать простой метод как на листинге 11.

```
public void testPowers() {
    SessionFactory factory = new SessionFactory("localhost", "5432",
"postgres", "math", "0212");
    Session session = factory.openSession();
    List<Powers> powers = new ArrayList<>();
    for (int i = 0; i < 100; i++) {
        Powers power = new Powers();
        power.setCube(i * i * i);
        power.setSquare(i * i);
        power.setX(i);
        powers.add(power);
    }

    for (int i = 0; i < 100; i++) {
        session.save(powers.get(i));
    }
}
```

Листинг 11. Пример метода, заполняющего таблицу

Заключение

В ходе данной курсовой работы был спроектирован и реализован компилятор из языка SQL в язык программирования Java для отображения таблиц и функций в классы и соответствующие функции. Также была реализована библиотека, которая позволяет разработчику использовать их для работы с базой данных и не писать SQL запросы в коде. Данная библиотека предоставляет API для выполнения необходимых CRUD операций с базой. Таким образом, код становится чище, проще и наглядней.

Список литературы

1. Документация ORM библиотеки Hibernate. https://hibernate.org/orm/documentation/5.3/ – дата обращения 02.12.2018;
2. Документация ORM библиотеки JOOQ. https://www.jooq.org/doc/3.11/manual/ – дата обращения 02.12.2018;
3. Документация ORM библиотеки Apache Cayenne. https://cayenne.apache.org/docs/4.0/getting-started-guide/ – дата обращения 02.12.2018;
4. Документация PostgreSQL. https://postgrespro.ru/docs/postgresql/9.6/index – дата обращения 02.12.2018.

Приложение А. Пример работы компилятора

Содержание входного файла с SQL запросом представлено на листинге А.1, а результат работы компилятора на А.2.

```
CREATE TABLE IF NOT EXISTS ShopDB.shopschema.Card_Check_INT
(
  checkID INT NOT NULL ,
  cardID INT NOT NULL ,
  purchases INT[][2] NOT NULL ,
  PRIMARY KEY (checkID, cardID),

  FOREIGN KEY (checkID) REFERENCES shopdb.shopschema.Check1 (checkid)
ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS ShopDB.shopschema.Check1
(
  checkID INT PRIMARY KEY NOT NULL,
  date1 TIMESTAMP NOT NULL UNIQUE ,
  totalCostWithTax REAL NOT NULL,
  totalCostWithoutTax REAL NOT NULL,
  isByCard BOOLEAN NOT NULL DEFAULT FALSE,
  discount SMALLINT DEFAULT 0,
  employeeCode INT,
  shopCode INT,

  FOREIGN KEY (shopCode) REFERENCES ShopDB.shopschema.Shop (shopCode),
  FOREIGN KEY (employeeCode) REFERENCES ShopDB.ShopSchema.Employee
(employeeCode)
);

CREATE OR REPLACE FUNCTION shopdb.shopschema.item_type_code_check(id
INT, sex VARCHAR, type VARCHAR, model VARCHAR) RETURNS BOOLEAN AS $$
DECLARE
  code VARCHAR := '';
BEGIN
  IF sex = 'Man' THEN
    code := code || '1';
  ELSEIF sex = 'Woman' THEN
    code := code || '2';
  ELSE
    RAISE EXCEPTION 'Invalid sex --> %', sex;
  END IF;

  IF type = 'Jeans' THEN
    code := code || '1';
  ELSEIF type = 'T-shirt' THEN
    code := code || '2';
  ELSEIF type = 'Shirt' THEN
    code := code || '3';
  ELSEIF type = 'Boots' THEN
    code := code || '4';
  ELSEIF type = 'Accessory' THEN
    code := code || '5';
  ELSEIF type = 'Pants' THEN
    code := code || '6';
```

```
  ELSE
    RAISE EXCEPTION 'Invalid type --> %', type;
  END IF;

  IF model = '501' THEN
    code := code || '1';
  ELSEIF model = '502' THEN
    code := code || '2';
  ELSEIF model = '505' THEN
    code := code || '3';
  ELSEIF model = '511' THEN
    code := code || '4';
  ELSEIF model = '512' THEN
    code := code || '5';
  ELSE
    RAISE EXCEPTION 'Invalid model --> %', model;
  END IF;

  IF code::INT = id THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Листинг А.1. Пример входного файла.

```
@Table(db = "shopdb", schema = "shopschema", name = "check1")
public class Check extends Entity {
    @PK
    @Column(name = "checkid", nullable = false)
    private Integer checkid;

    @Column(name = "date1", unique = true, nullable = false)
    private Timestamp date;

    @Column(name = "totalcostwithtax", nullable = false)
    private Float totalcostwithtax;

    @Column(name = "totalcostwithouttax", nullable = false)
    private Float totalcostwithouttax;

    @Default
    @Column(name = "isbycard", nullable = false)
    private Boolean isbycard = false;

    @Default
    @Column(name = "discount")
    private Short discount = 0 ;

    @FK(table = "employee", referencedColumn = "employeecode")
    @Column(name = "employeecode")
    private Integer employeecode;

    @FO(table = "employee")
    private Employee employee;

    @FK(table = "shop", referencedColumn = "shopcode")
```

```
@Column(name = "shopcode")
private Integer shopcode;

@FO(table = "shop")
private Shop shop;

public Integer getCheckid() {
   return this.checkid;
}

public void setCheckid(Integer checkid) {
   this.checkid = checkid;
}

public Timestamp getDate() {
   return this.date;
}

public void setDate(Timestamp date1) {
   this.date1 = date;
}

public Float getTotalcostwithtax() {
   return this.totalcostwithtax;
}

public void setTotalcostwithtax(Float totalcostwithtax) {
   this.totalcostwithtax = totalcostwithtax;
}

public Float getTotalcostwithouttax() {
   return this.totalcostwithouttax;
}

public void setTotalcostwithouttax(Float totalcostwithouttax) {
   this.totalcostwithouttax = totalcostwithouttax;
}

public Boolean getIsbycard() {
   return this.isbycard;
}

public void setIsbycard(Boolean isbycard) {
   this.isbycard = isbycard;
}

public Short getDiscount() {
   return this.discount;
}

public void setDiscount(Short discount) {
   this.discount = discount;
}

public Integer getEmployeecode() {
   return this.employeecode;
}
```

```
    public void setEmployeecode(Integer employeecode) {
        this.employeecode = employeecode;
    }

    public Integer getShopcode() {
        return this.shopcode;
    }

    public void setShopcode(Integer shopcode) {
        this.shopcode = shopcode;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (this.getClass() != obj.getClass())
            return false;

        Check other = (Check) obj;
        return Objects.equals(this.checkid, other.checkid);
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.checkid);
    }

    @Override
    public String toString() {
        return "Check {\n" +
            "\tcheckid: " + this.checkid + "\n" +
            "\tdate: " + this.date1 + "\n" +
            "\ttotalcostwithtax: " + this.totalcostwithtax + "\n" +
            "\ttotalcostwithouttax: " + this.totalcostwithouttax + "\n" +
            "\tisbycard: " + this.isbycard + "\n" +
            "\tdiscount: " + this.discount + "\n" +
            "\temployeecode: " + this.employeecode + "\n" +
            "\tshopcode: " + this.shopcode + "\n" +
            "}";
    }
}

@Table(db = "shopdb", schema = "shopschema", name = "card_check_int")
public class Card_check_int extends Entity {
    @PK
    @FK(table = "check", referencedColumn = "checkid")
    @Column(name = "checkid", nullable = false)
    private Integer checkid;

    @FO(table = "check")
    private Check check;

    @PK
    @Column(name = "cardid", nullable = false)
    private Integer cardid;
```

```
    @Column(name = "purchases", nullable = false)
    private ArrayList<Integer> purchases;

    public Integer getCheckid() {
        return this.checkid;
    }

    public void setCheckid(Integer checkid) {
        this.checkid = checkid;
    }

    public Integer getCardid() {
        return this.cardid;
    }

    public void setCardid(Integer cardid) {
        this.cardid = cardid;
    }

    public ArrayList<Integer> getPurchases() {
        return this.purchases;
    }

    public void setPurchases(ArrayList<Integer> purchases) {
        this.purchases = purchases;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (this.getClass() != obj.getClass())
            return false;

        Card_check_int other = (Card_check_int) obj;
        return Objects.equals(this.checkid, other.checkid) &&
Objects.equals(this.cardid, other.cardid);
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.checkid, this.cardid);
    }

    @Override
    public String toString() {
        return "Card_check_int {\n" +
            "\tcheckid: " + this.checkid + "\n" +
            "\tcardid: " + this.cardid + "\n" +
            "\tpurchases: " + this.purchases + "\n" +
            "}";
    }
}
```

```
@Routines(db = "shopdb", schema = "shopschema")
public class Item_type_code_checkFunction extends FunctionsExecutor {
   public Item_type_code_checkFunction(Connection connection) {
      super(connection);
   }

   public Boolean item_type_code_check(Integer id, String sex, String
type, String model) {
      return (Boolean) super.executeScalarFunction(this.getClass(),
String.format("item_type_code_check(%s, %s, %s, %s)", id, sex, type,
model));
   }
}
```

Листинг А.2. Результат работы компилятора.