Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика, искусственный интеллект и системы управления

КАФЕДРА Системы обработки информации и управления

Методические указания к домашнему заданию по курсу «Технологии разработки программного обеспечения»

Домашнее задание №3 «Применение паттернов проектирования»

Виноградова М.В., Белоусов Е.А., Макрушина В.А.

Под редакцией к.т.н. доц. Виноградовой М.В.

Москва, 2022 г.

ОГЛАВЛЕНИЕ

1. ЗАДАНИЕ 3
1.1. Цель работы 3
1.2. Порядок и время проведения работы 3
1.3. Задание..... 3
1.4. К защите 3
2. ТЕОРЕТИКО-ПРАКТИЧЕСКИЙ МАТЕРИАЛ 5
2.1. Паттерны работы с базой данных 5
2.1.1. Table Data Gateway (Шлюз таблицы данных)..... 5
2.1.2. Row Data Gateway (Шлюз записи данных) 9
2.1.3. Active Record (Активная запись)..... 13
2.1.4. Data Mapper (Преобразователь данных)..... 18
2.2. Паттерны бизнес логики 20
2.2.1. Transaction Script (Сценарий транзакций)..... 20
2.2.2. Domain Model (Модель предметной области) 22
2.2.3. Table Module (Модуль таблицы) 24
2.2.4. Service Layer (Слой служб)..... 26
2.3. Пример..... 28
2.3.1. Задание..... 28
2.3.1. Выделение классов 28
2.3.1. Диаграммы классов и последовательностей..... 32
2.3.1. Программный код..... 35
3. СПИСОК ИСТОЧНИКОВ..... 45

1. ЗАДАНИЕ

Домашнее задание №3 «Применение паттернов проектирования» по курсу «Технологии разработки программного обеспечения».

1.1. Цель работы

- Изучить основные паттерны проектирования, их особенности и область применения;
- Получить практические навыки программирования паттернов;
- Освоить технологию включения паттернов в собственную программу.

1.2. Порядок и время проведения работы

Работа выполняется самостоятельно в часы внеаудиторных занятий. По итогам составляется и защищается отчет в бумажном виде, а также проводится демонстрация работающей программы.

1.3. Задание

1. Разработать программу (основные прецеденты) на основе темы, выданной преподавателем (по варианту).
2. Реализовать в программе паттерны (по варианту) бизнес-логики и работы с БД.
3. Составить набор диаграмм классов и последовательностей, которые демонстрируют структуру и поведение программы.
4. Отдельно составить диаграммы классов и последовательностей для иллюстрации примененных паттернов.

1.4. К защите

Программная реализация:

- Работающая программа, реализующая основные функции системы (по варианту);
- Программа должна содержать реализацию паттернов проектирования (по варианту).

В отчет:

- Диаграмма(ы) классов системы;
- Диаграммы последовательностей для основных функций программы;
- Диаграммы классов и последовательностей для иллюстрации примененных паттернов;
- Исходный код программы (фрагменты, реализующие паттерны и их вызов).

2. ТЕОРЕТИКО-ПРАКТИЧЕСКИЙ МАТЕРИАЛ

2.1. Паттерны работы с базой данных

2.1.1. Table Data Gateway (Шлюз таблицы данных)

Объект, выполняющий роль **шлюза** к базе данных [1].

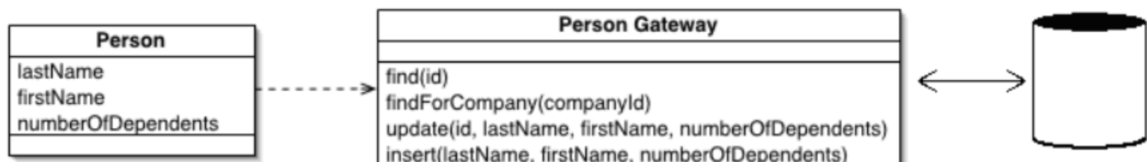

Рисунок 1 – Диаграмма классов Table Data Gateway

Объект паттерна **Table Data Gateway** выступает в качестве **шлюза** между данными в приложении и в базе данных.

Интерфейс **шлюза таблицы данных** обычно содержит методы для доступа к отдельной таблице или представлению (view): несколько методов поиска, предназначенных для извлечения данных, а также методы обновления, вставки, удаления (CRUD) и методы вызова хранимых процедур. Каждый метод передает входные параметры вызову соответствующей команды SQL и выполняет ее в контексте установленного соединения с базой данных. Методы этого паттерна используются другими объектами для взаимодействия с базой данных. Как правило, объект данного паттерна не имеет состояний, поскольку всего лишь передает данные в таблицу и из таблицы. Для каждой таблицы базы данных создается собственный шлюз таблицы данных. Впрочем, в наиболее простых случаях один шлюз может работать с несколькими таблицами.

Примеры кода в данном разделе приведены на языке программирования C# [5].

Метод insert принимает на вход все поля (атрибуты) объекта:

```
class PersonGateway...

public long Insert(String lastName, String firstName,
      long numberOfDependents)
{
      String sql = "INSERT INTO person VALUES (?,?,?,?)";
      long key = GetNextID ();
      IDbCommand comm = new OleDbCommand(sql, DB.Connection);
      comm.Parameters.Add(new OleDbParameter("key", key));
      comm.Parameters.Add(new OleDbParameter("last", lastName));
      comm.Parameters.Add(new OleDbParameter("first", firstName));
      comm.Parameters.Add(new OleDbParameter("numDep",
numberOfDependents)); comm.ExecuteNonQuery();
      return key;
}
```

Метод update принимает на вход атрибуты сущности БД + id:

```
class PersonGateway...

public void Update (long key, String lastname, String firstname,
      long numberOfDependents)
{
      String sql = @" UPDATE person
      SET lastname = ?, firstname = ?, numberOfDependents = ?
      WHERE id = ?";
      IDbCommand comm = new OleDbCommand(sql, DB.Connection) ;
      comm.Parameters.Add(new OleDbParameter("last", lastname));
      comm.Parameters.Add(new OleDbParameter("first", firstname));
      comm.Parameters.Add(new OleDbParameter("numDep",
numberOfDependents));
      comm.Parameters.Add(new OleDbParameter ("key", key));
      comm.ExecuteNonQuery();
}
```

Метод delete принимает на вход часть параметров (id) для корректного определения объекта и его удаления:

```
class PersonGateway...

public void Delete (long key)
{
      String sql = "DELETE FROM person WHERE id = ?";
      IDbCommand comm =new OleDbCommand(sql, DB.Connection);
      comm.Parameters.Add(new OleDbParameter ("key", key));
      comm.ExecuteNonQuery();
}
```

Так как даже простой запрос может вернуть несколько записей, то в качестве возвращаемых значений операции поиска могут быть использованы Record set, Коллекция, Объект, либо признак удачной операции.

```
class PersonGateway...

public IDataReader FindAll
{
      String sql = "select * from person";
      return new OleDbCommand(sql, DB.Connection).ExecuteReader();
}
public IDataReader FindWithLastName(String lastName)
{
      String sql = "SELECT * FROM person WHERE lastname = ?";
      IDbCommand comm = new OleDbCommand(sql, DB.Connection);
      coram.Parameters.Add (new OleDbParameter("lastname", lastName));
      return comm.ExecuteReader();
}
public IDataReader FindWhere(String whereClause)
{
      String sql = String.Format("select * from person where {0}",
whereClause);
      return new OleDbCommand(sql, DB.Connection).ExecuteReader();
}
```

Коротко:

- Объект выступает в качестве шлюза между данными в приложении и в БД
- Объект (Класс) содержит методы для доступа к отдельной таблице или представлению (view): выборка, обновление, вставка, удаление (CRUD) и вызов хранимых процедур.
- Один объект работает сразу со всеми записями в таблице.
- Возможен один шлюз на несколько таблиц
- Может быть Класс со статическими методами

- **Вход методов:** параметры для формирования запроса
  - Для insert - все поля (атрибуты) объекта
  - Для update – атрибуты сущности БД + id
  - Для delete - часть параметров (id) для корректного определения объекта и удаления его
- **Выход методов:**
  - поиск:
    - Record set
    - Коллекция
    - Объект
    - Признак удачной операции

2.1.2. Row Data Gateway (Шлюз записи данных)

Объект, выполняющий роль шлюза к отдельной записи источника данных [1]. Каждой строке таблицы базы данных соответствует свой экземпляр шлюза записи данных.

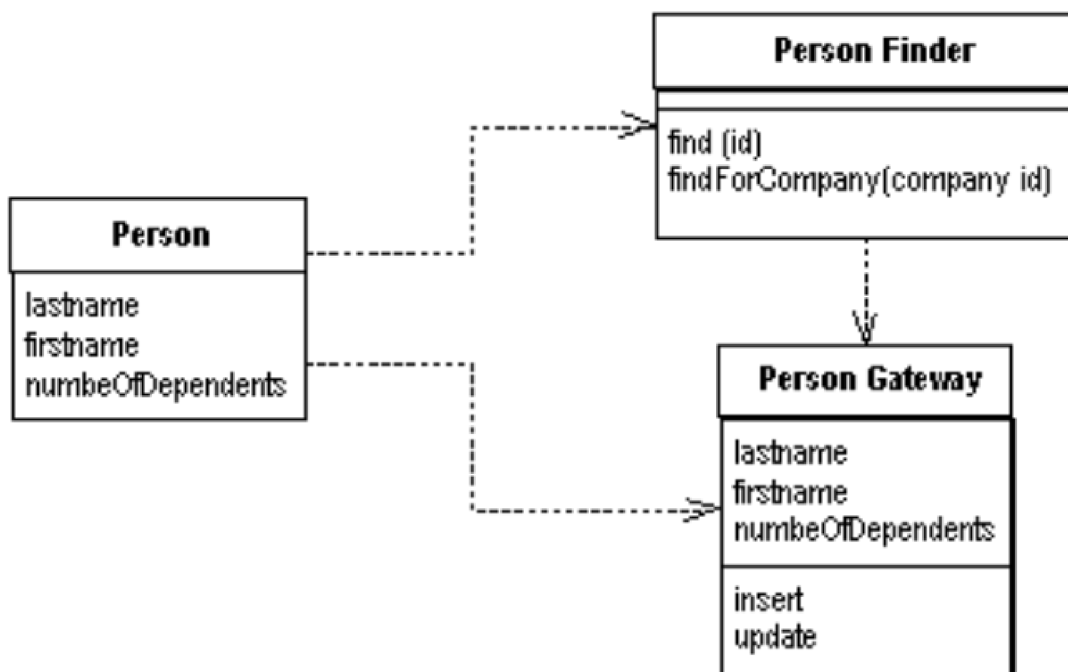

Рисунок 2 – Диаграмма классов Row Data Gateway

Паттерн **шлюз записи данных** предоставляет в распоряжение объекты, которые полностью аналогичны записям базы данных, однако могут быть доступны с помощью обычных механизмов используемого языка программирования. Все детали доступа к базе данных скрыты за интерфейсом.

Шлюз записи данных выступает в роли объекта, полностью повторяющего одну запись, например одну строку в таблице базы данных. Каждому столбцу таблицы соответствует поле записи. Обычно **шлюз записи данных** должен выполнять все возможные преобразования типов источника данных в типы, используемые приложением. Рассматриваемый паттерн содержит все данные о строке, поэтому клиент имеет возможность непосредственного доступа к **шлюзу записи данных**.

При реализации методов поиска можно воспользоваться статическими метода поиска, однако часто имеет смысл создать отдельные объекты поиска, чтобы у каждой таблицы реляционной базы данных был один класс для проведения поиска и один класс шлюза для сохранения результатов этого поиска.

Ниже приведен пример реализации **шлюза записи данных для простой таблицы** сотрудников под именем people на языке Java [6].

```
create table people (
	ID int primary key,
	lastname varchar,
	firstname varchar,
	number_of_dependents int);
```

У данной таблицы есть шлюз PersonGateway. Код этого класса **начинается с описаний** полей данных и функций доступа (accessors).

```
class PersonGateway...

private String lastName;
private String firstName;
private int numberOfDependents;
public String getLastName() { return lastName; }
public void setLastName(String lastName) { this.lastName = lastName; }
public String getFirstName() { return firstName; }
public void setFirstName(String firstName) { this.firstName = firstName; }
public int getNumberOfDependents() { return numberOfDependents; }
public void setNumberOfDependents(int numberOfDependents)
{
	this.numberOfDependents = numberOfDependents;
}
```

Класс шлюза включает в себя собственные методы CRUD, которые берут данные для работы из свойств объекта и работают без параметров.

```
class PersonGateway...

private static final String updateStatementString =
	"UPDATE people " +
	" set lastname = ?, firstname = ?,
number_of_dependents = ? " +
	" where id = ?";
```

```
public void update ()
{
      PreparedStatement updateStatement = null;
      try {
          updateStatement = DB.prepare (updateStatementString);
          updateStatement.setString(1, lastName);
          updateStatement.setString(2, firstName);
          updateStatement.setlnt(3, numberOfDependents);
          updateStatement.setlnt(4, getlDO .intValue());
          updateStatement.execute() ;
      } catch (Exception e) {
          throw new ApplicationException(e) ;
      } finally {DB.cleanup(updateStatement); }
}

private static final String insertStatementString =
      "INSERT INTO people VALUES (?,?,?,?)";
public Long insert()
{
      PreparedStatement insertStatement = null;
      try {
          insertStatement = DB.prepare(insertStatementString);
          setID(findNextDatabaseId());
          insertStatement.setlnt (1, getlDO .intValue());
          insertStatement.setString(2, lastName);
          insertStatement.setString(3, firstName) ;
          insertStatement.setlnt(4, numberOfDependents);
          insertStatement.execute() ;
          Registry.addPerson(this) ;
          return get ID();
      } catch (SQLException e) {
          throw new ApplicationException (e);
      } finally {DB.cleanup(insertStatement); }
}
```

Для извлечения из базы данных записей о сотрудниках применяется специальный Класс PersonFinder. Он используется в сочетании с классом PersonGateway для создания новых объектов шлюза. Принцип функционирования поиска изображен на диаграмме последовательностей (рис. 3).

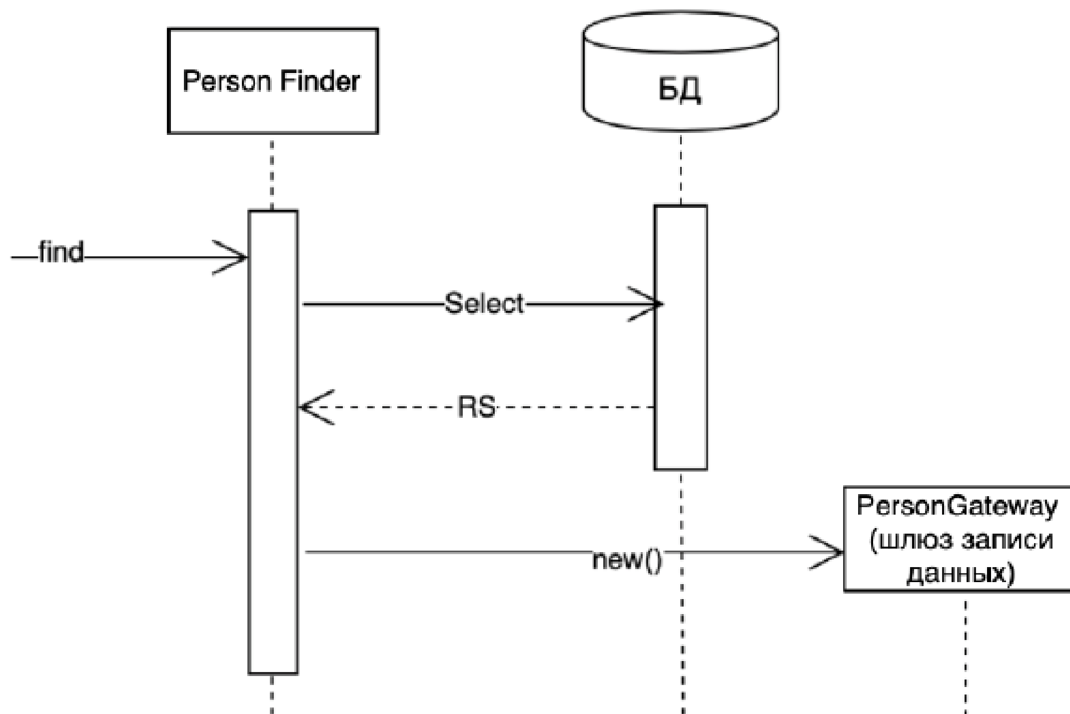

Рисунок 3 – Диаграмма последовательностей для поиска в паттерне ***Row Data Gateway***

Коротко:

- Объект выступает в роли шлюза к отдельной записи в источнике данных. Один экземпляр на одну запись.
- 1 объект - это одна запись из таблицы БД.
- Атрибуты объекта включают все поля записи из БД.
- Методы CRUD работают без параметров.
- Методы CRUD берут данные для работы из свойств объекта.
- Методы поиска статичны, либо прописываются отдельно(class PersonFinder).

2.1.3. Active Record (Активная запись)

Объект, выполняющий роль оболочки для строки таблицы или представления базы данных [1]. Он инкапсулирует доступ к базе данных и добавляет к данным логику домена.

Объект **Active Record** охватывает и данные и поведение. Большая часть его данных является постоянной и должна храниться в базе данных. В паттерне активная запись используется наиболее очевидный подход, при котором логика доступа к данным включается в объект домена. В этом случае все знают, как считывать данные из базы данных и как их записывать в нее.

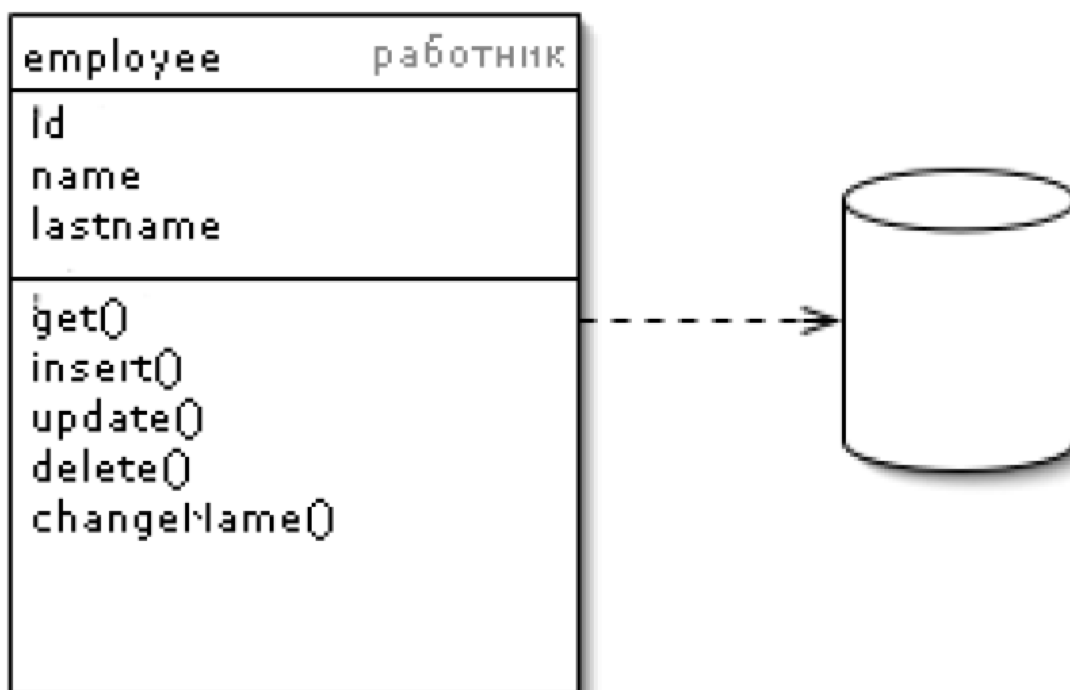

Рисунок 4 – Диаграмма классов Active Record

В основе паттерна **активная запись** лежит модель предметной области, классы которой повторяют структуру записей используемой базы данных. Каждая активная запись отвечает за сохранение и загрузку информации в базу данных, а также за логику домена, применяемую к данным. Это может быть вся бизнес-логика приложения.

Структура данных активной записи должна в точности соответствовать таковой в таблице базы данных: каждое поле объекта должно соответствовать одному столбцу таблицы. Значения полей следует оставлять такими же, какими они были получены в

результате выполнения SQL-команд; никакого преобразования на этом этапе делать не нужно.

Методы извлечения и установки значений полей могут выполнять и другие действия, например преобразование типов SQL в типы, используемые приложением.

Аактивная запись тесно привязана к базе данных, поэтому рекомендуется использовать в этом паттерне статические методы поиска.

Как и другие типовые решения, предназначенные для работы с таблицами, активную запись можно применять не только к таблицам, но и к представлениям или запросам.

Активная запись хорошо подходит для реализации не слишком сложной логики домена, в частности операций создания, считывания, обновления и удаления. Кроме того, она прекрасно справляется с извлечением и проверкой на правильность отдельной записи.

Рассмотрим простой пример, иллюстрирующий основные аспекты применения активной записи. Создадим класс Person на языке Java [6].

```
class Person...

private String lastName;
private String firstName;
private int numberOfDependents;
```

База данных выглядит следующим образом

```
create table people
      ( ID int primary key,
        lastname varchar,
        firstname varchar,
        number_of_dependents int)
```

Для поиска применяется статический метод класса

```
class Person...

private final static String findStatementString =
      "SELECT id, lastname, firstname, number_of_dependents" +
      "FROM people" +
      "WHERE id = ?";
```

```
public static Person find(Long id)
{
	Person result = (Person) Registry.getPerson(id);
	if (result != null) return result;
	PreparedStatement findStatement = null;
	ResultSet rs = null;
	try
	{
	    findStatement = DB.prepare(findStatementString);
	    findStatement.setLong(l, id.longValue());
	    rs = findStatement.executeQuery();
	    rs.next();
	    result = load(rs);
	    return result;
	} catch (SQLException e) {
	    throw new ApplicationException(e);
	} finally { DB.cleanup(findStatement,rs); }
}
```

Методы CRUD работают без параметров и берут данные для работы из свойств объекта.

```
class Person...
private final static String updateStatementString =
	"UPDATE people" +
	"set lastname = ?, firstname = ?, number_of_dependents = ?" +
	"where id = ?";
```

```
public void update()
{
	PreparedStatement updateStatement = null;
	try
	{
	    updateStatement = DB.prepare(updateStatementString);
	    updateStatement.setString(1, lastName);
	    updateStatement.setString(2, firstName);
	    updateStatement.setlnt(3, numberOfDependents);
	    updateStatement.setlnt(4, getID().intValue());
	    updateStatement.execute ();
	} catch (Exception e) {
	    throw new ApplicationException(e);
	} finally { DB.cleanup(updateStatement); }
}
private final static String insertStatementString =
	"INSERT INTO people VALUES (?,?,?,?)";
public Long insert ()
{
	PreparedStatement insertStatement = null;
	try
	{
	    insertStatement = DB.prepare(insertStatementString);
	    setID(findNextDatabaseId());
	    insertStatement.setlnt (1, getID().intValue() );
	    insertStatement.setString(2, lastName);
	    insertStatement.setString(3, firstName);
	    insertStatement.setlnt(4, numberOfDependents);
	    insertStatement.execute();
	    Registry.addPerson(this);
	    return getID();
	} catch (Exception e) {
	    throw new ApplicationException(e);
	} finally { DB.cleanup(insertStatement); }
}
```

Вся бизнес-логика, например определение суммы вычетов при расчете налогов, должна быть реализована непосредственно в классе Person.

```
class Person...

public Money getExemption()
{
      Money baseExemption = Money.dollars(1500);
      Money dependentExemption = Money.dollars(750);
      return baseExemption.add(
          dependentExemption.multiply( this.getNumberOfDependents())
      );
}
```

Коротко:

- Объект, выполняющий роль оболочки для строки таблицы БД или представления. Инкапсулирует доступ к БД и добавляет к данным логику домена.
- Атрибуты объекта включают все поля записи из БД.
- Методы CRUD работают без параметров.
- Методы CRUD берут данные для работы из свойств объекта.
- Содержит бизнес- логику обращения с данными.
- Отличие от шлюза записи – наличие бизнес-логики.

2.1.4. Data Mapper (Преобразователь данных)

Слой преобразователей (Mapper), который осуществляет передачу данных между объектами и базой данных, сохраняя последние независимыми друг от друга и от самого преобразователя [1] [2].

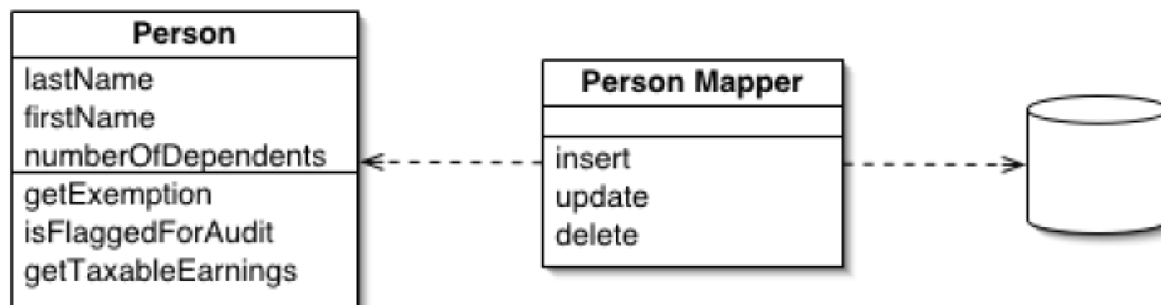

Рисунок 5 – Диаграмма классов Data Mapper

Паттерн **преобразователь данных** представляет собой слой программного обеспечения, который отделяет объекты, расположенные в оперативной памяти, от базы данных.

В функции **преобразователя данных** входит передача данных между объектами и базой данных и изоляция их друг от друга. Благодаря использованию этого типового решения объекты, расположенные в оперативной памяти, могут ничего не «знать» о базе данных. (Схема базы данных тоже не «знает» об объектах, которые ее используют.) А так как преобразователь данных является разновидностью преобразователя, он полностью скрыт от уровня домена.

Основной функцией преобразователя данных является отделение домена от источника данных. При необходимости весь слой **преобразователя данных** может быть заменен, например, чтобы провести тестирование или чтобы использовать один и тот же уровень домена для работы с несколькими базами данных. Обычно **преобразователь данных** применяется для того, чтобы схема базы данных и объектная модель могли изменяться независимо друг от друга. Применение преобразователей данных помогает и в написании кода, поскольку позволяет работать с объектами домена без необходимости понимать принцип хранения соответствующей информации в базе данных.

Рекомендуется создавать по одному **преобразователю** для каждого класса домена или для корневого класса в иерархии доменов.

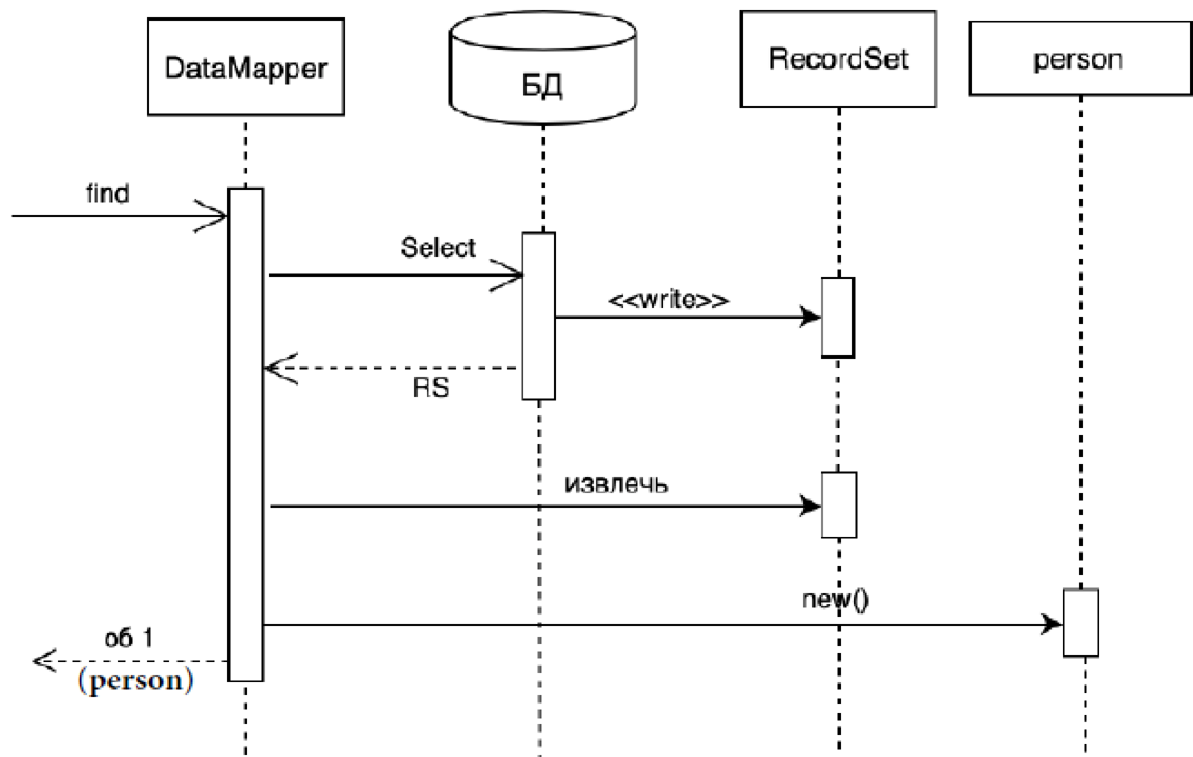

Рисунок 6 – Диаграмма последовательностей для функции поиска паттерна Data Mapper

На рисунке 6 приведена диаграмма последовательностей, демонстрирующая работу поиска при использовании паттерна **Data Mapper**. В этом примере у нас есть классы person и DataMapper. Для загрузки данных в объект person клиент вызывает метод поиска класса DataMapper. Преобразователь обращается к базе данных с запросом, получает в ответ RecordSet (данные в том виде, в котором они хранятся в базе данных). После этого DataMapper преобразует данные из RecordSet в person (тем самым инициализируя его) и возвращает полученный person в качестве результата.

Коротко:

- Слой преобразователей для передачи данных между объектами и таблицами БД, сохраняя их независимо друг от друга и преобразователей.
- Объекты не нуждаются в знании о существовании БД. Они не нуждаются в SQL-коде и в информации о структуре БД.
- Вход методов: объекты бизнес-логики (CUD) и параметры запроса поиска (R)
- Выход методов: объекты, коллекция объектов (поиск)

2.2. Паттерны бизнес логики

2.2.1. Transaction Script (Сценарий транзакций)

Способ организации бизнес-логики по процедурам, каждая из которых обслуживает один запрос, инициируемый слоем представления [1].

Бизнес-приложения могут восприниматься как последовательности транзакций. Каждый акт взаимодействия клиента с сервером описывается определенным фрагментом логики. **Сценарий транзакции** организует логику вычислительного процесса в виде единой процедуры. Каждой транзакции ставится в соответствие собственный **сценарий транзакции**.

При использовании типового решения **сценарий транзакции** логика предметной области распределяется по транзакциям, выполняемым в системе. Если пользователю необходимо заказать номер в гостинице, соответствующая процедура должна предусматривать действия по проверке наличия подходящего номера, вычислению суммы оплаты и фиксации заказа в базе данных.

Способ разнесения кода **сценариев транзакции** по классам связан с разработкой собственного класса для каждого **сценария транзакции**: определяется тип, базовый по отношению ко всем командам, в котором предусматривается некий метод выполнения (run), удовлетворяющий логике **сценария транзакции**. Производные классы переопределяют метод run. Параметры, которыми инициализируются классы сценариев транзакции, передаются в метод run.

Главным достоинством типового решения **сценарий транзакции** является простота. Данный вид организации логики, эффективный с точки зрения восприятия и производительности, характерен для небольших приложений.

По мере усложнения бизнес-логики становится все труднее содержать ее в хорошо структурированном виде. Одна из достойных внимания проблем связана с повторением фрагментов кода.

Пример диаграммы последовательностей паттерна **сценарий транзакции** для класса Служба определения зачтенного дохода приведен на рисунке 7.

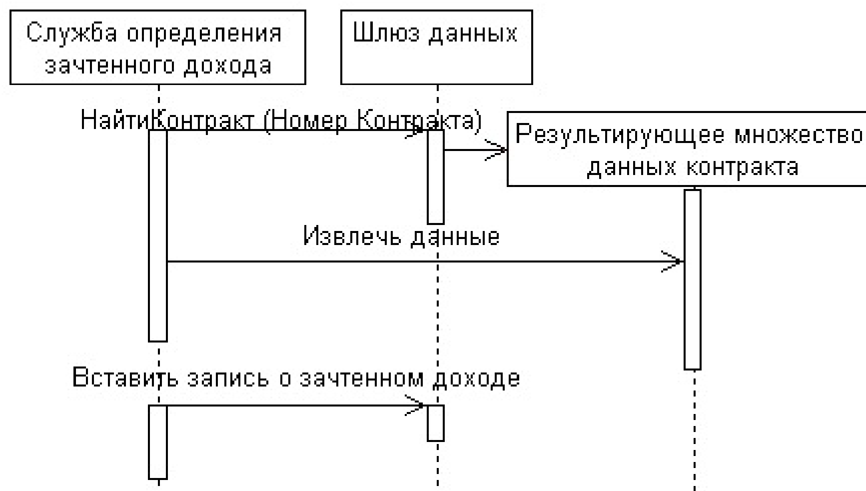

Рисунок 7 – Transaction Script – пример

Коротко:

- Способ организации бизнес-логики по процедурам, каждая из которых обслуживает один запрос, инициируемый слоем представления.
- Сценарий транзакции организует логику вычислительного процесса в виде единой процедуры, которая выполняет все действия, необходимые для выполнения прикладной функции.
- Каждой функции приложения ставится в соответствие собственный сценарий транзакции (общие подзадачи могут быть вынесены в подчиненные процедуры).
- Вся бизнес логика делится на процедуры:
    - 1 класс – одна процедура (соответствует функции приложения).
    - Базовый класс класс с методом run() от которого наследуются другие классы для реализации отдельных функций.

2.2.2. Domain Model (Модель предметной области)

Объектная модель домена, охватывающая поведение (функции) и свойства (данные) [1].

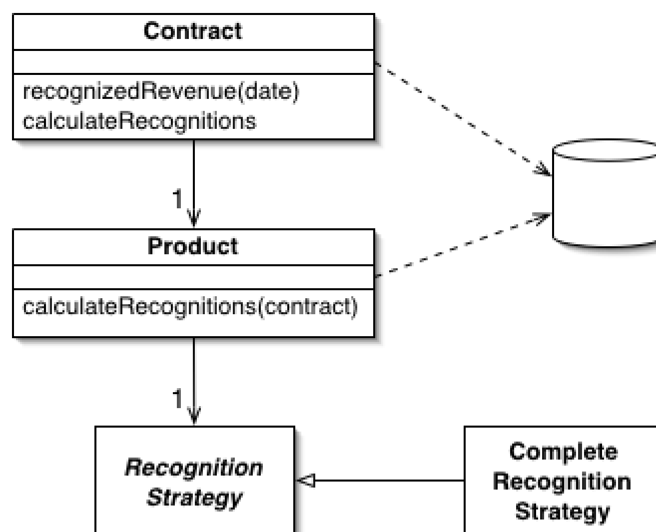

Рисунок 8 – Диаграмма классов Domain Model

Паттерн **Domain Model** предусматривает создание сети взаимосвязанных объектов, каждый из которых представляет некую осмысленную сущность — от строки формы заказа до промышленной корпорации.

Реализация **модели предметной области** означает пополнение приложения целым слоем объектов, описывающих различные стороны определенной области бизнеса. Одни объекты призваны имитировать элементы данных, которыми оперируют в этой области, а другие должны формализовать те или иные бизнес-правила. Функции тесно сочетаются сданными, которыми они манипулируют.

Объектно-ориентированная **модель предметной области** часто напоминает схему соответствующей базы данных, хотя между ними все еще остается множество различий. В модели предметной области смешиваются данные и функции, допускаются многозначные атрибуты, создаются сложные сети ассоциаций и используются связи наследования.

Можно выделить две разновидности **моделей предметной области**. "Простая" во многом походит на схему базы данных и содержит, как правило, по одному объекту домена в расчете на каждую таблицу. "Сложная" модель может отличаться от структуры базы данных и содержать иерархии наследования, стратегии и иные типовые решения, а также сложные сети мелких взаимосвязанных объектов. Сложная модель

более адекватно представляет запутанную бизнес-логику, но труднее поддается отображению в реляционную схему базы данных.

Пример взаимодействий классов паттерна **Domain Model** приведен на рисунке 9.

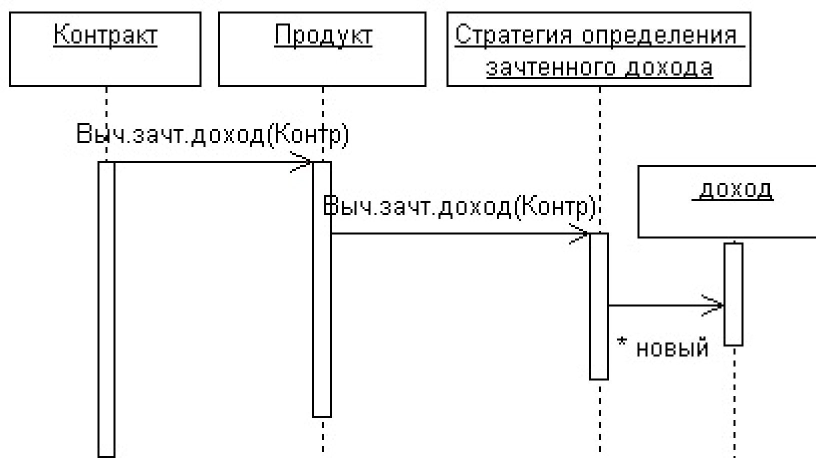

Рисунок 9 – Domain Model диаграмма последовательностей

Коротко:

- Предусматривает создание сети взаимосвязанных объектов, каждый из которых представляет некую осмысленную сущность.
- Означает наполнение приложения слоем объектов, описывающих сущности реального мира и их взаимодействие.
- Одни объекты соответствуют элементам данных, которыми оперируют в этой области, а другие реализуют те или иные бизнес-правила. Функции тесно сочетаются с данными, которыми они манипулируют.
- Объект содержит данные (для одной записи/сущности БД) и бизнес-логику их обработки.
- Объект выполняет обработку только своих данных, для обработки данных другого объекта вызывает методы другого объекта. Выполнение прикладной функции приложения будет последовательностью вызовов методов всех задействованных объектов.
- Один объект хранит данные для одной записи БД.

2.2.3. Table Module (Модуль таблицы)

Объект, охватывающий логику обработки всех записей хранимой или виртуальной таблицы базы данных [1].

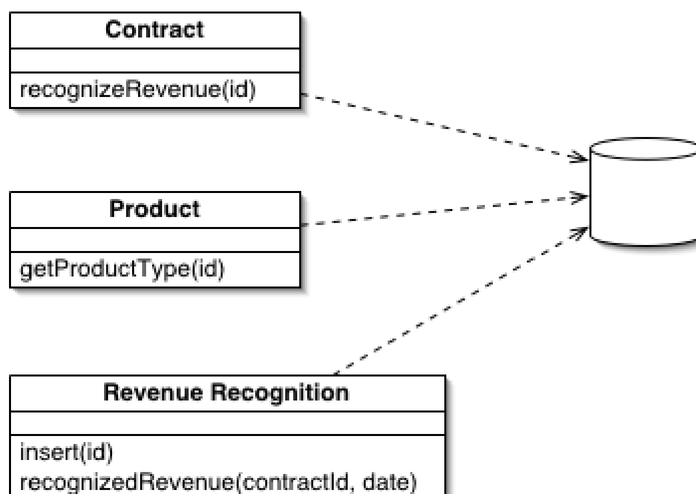

Рисунок 10 – диаграмма классов Table Model

Паттерн **модуль таблицы** предусматривает создание по одному классу на каждую таблицу базы данных, и единственный экземпляр класса содержит всю логику обработки данных таблицы. Основное отличие **модуля таблицы** от **модели предметной области** состоит в том, что если, приложение обслуживает множество заказов, в соответствии с **моделью предметной** области придется сконструировать по одному объекту на каждый заказ, а при использовании **модуля таблицы** понадобится всего один объект, представляющий одновременно все заказы.

Паттерн **модуль таблицы** позволяет сочетать данные и функции для их обработки и в то же время эффективно использовать ресурсы реляционной базы данных. **Модуль таблицы** во многом напоминает обычный объект, но не содержит идентификационного признака объекта.

Каждый класс **модуля таблицы** содержит элемент данных, который соответствует одной из таблиц множества данных. Способность считывать табличную информацию свойственна всем модулям таблицы.

Коротко:

- Предусматривает создание по одному классу на каждую таблицу базы данных, и единственный экземпляр класса содержит всю логику обработки данных таблицы.
- Основное отличие модуля таблицы от модели предметной области состоит в том, что в соответствии с моделью предметной области придется сконструировать по одному объекту на каждую запись БД, а при использовании модуля таблицы понадобится всего один объект, представляющий одновременно записи таблицы БД.
- Содержит методы для обработки данных таблицы (CRUD, типовые запросы), вызова хранимых процедур.
- Возможны статические методы.
- Хранит данные таблиц, информацию о таблице, можно производить переход по связям (запрос к нескольким таблицам).
- Не содержит упоминания об идентификационном признаке объекта.

2.2.4. Service Layer (Слой служб)

Схема определения границ приложения посредством **слоя служб**, который устанавливает множество доступных действий и координирует отклик приложения на каждое действие [1].

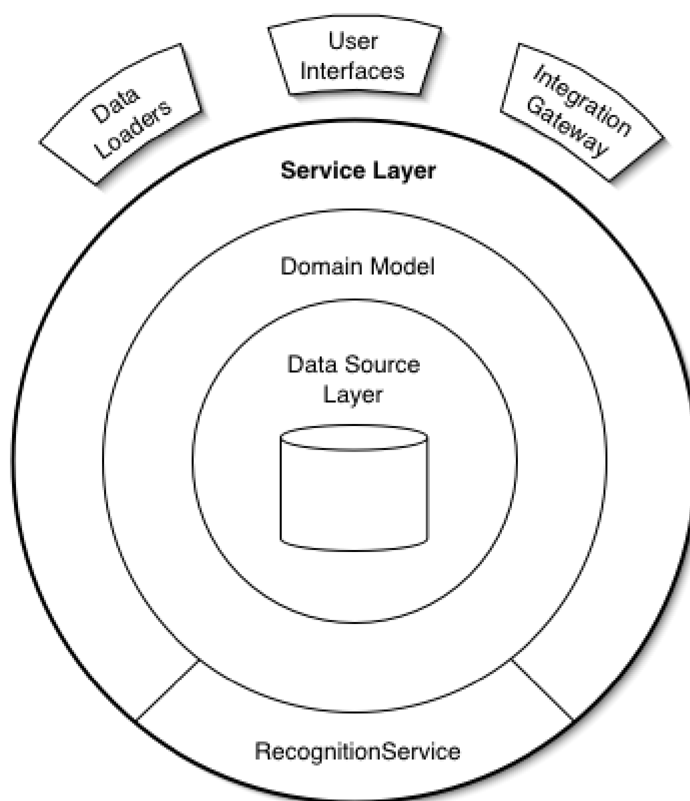

Рисунок 11 – Service Layer

Слой служб определяет границы приложения и множество операций, предоставляемых им для интерфейсных клиентских слоев кода. Он инкапсулирует бизнес-логику приложения, управляет транзакциями и координирует реакции на действия.

Слой служб может быть реализован несколькими способами, удовлетворяющими всем упомянутым выше условиям. Различия проявляются в методах распределения ответственности вне интерфейса слоя служб. Двумя базовыми вариантами реализации слоя служб являются создание интерфейса доступа к домену (domain facade) и конструирование сценария операции (operation script).

При использовании подхода, связанного с интерфейсом доступа к домену, **слой служб** реализуется как набор "тонких" интерфейсов, размещенных "поверх" модели предметной области. В классах, реализующих интерфейсы, нет бизнес-логики — она сосредоточена в контексте модели предметной области. Тонкие интерфейсы

устанавливают границы и определяют множество операций, посредством которых клиентские слои взаимодействуют с приложением, обнаруживая тем самым характерные свойства **слоя служб**.

Создавая сценарий операции, реализуется слой служб как множество более "толстых" классов, которые непосредственно воплощают в себе логику приложения, но за бизнес логикой обращаются к классам домена. Операции, предоставляемые клиентам слоя служб, реализуются в виде сценариев, создаваемых группами в контексте классов, каждый из которых определяет некоторый фрагмент соответствующей логики.

Операции, которые должны быть размещены в **слое служб**, определяются нуждами клиентов слоя служб, наиболее важной из которых обычно является пользовательский интерфейс. Поэтому в качестве отправной точки для определения набора операций **слоя служб** должны рассматриваться варианты использования и пользовательский интерфейс приложения. Обычно варианты использования приложений составляют операции CRUD над объектами домена.

Преимуществом использования **слоя служб** является возможность определения набора общих операций, доступных для применения многими категориями клиентов, и координация откликов приложения на выполнение каждой операции. Поэтому рекомендуется использовать паттерн **Service Layer** если в приложении есть более одной категории клиентов.

Коротко:

- Определяет границу между приложением и слоем сервисов, который образует набор доступных операций и управляет ответом приложения в каждой операции.
- Определяет для приложения границу и набор допустимых операций с точки зрения взаимодействующих с ним клиентских модулей.
- Он инкапсулирует бизнес-логику приложения, управляя транзакциями и управляя ответами в реализации этих операций.

2.3. Пример

2.3.1. Задание

Рассмотрим реализацию паттернов в приложении «Бронирование билетов в театр». В примере будет использоваться язык программирования Python [3] и фреймворк Django [4]. Вариант задания:

- Паттерн бизнес логики – **Domain Model**;
- Паттерн работы с БД – **Table Data Gateway**.

2.3.1. Выделение классов

2.3.1.1. Выделение классов Table Data Gateway

Для работы с паттерном **Table Data Gateway** для каждой таблицы базы данных необходимо создать класс шлюза таблицы. В базе данных разрабатываемого приложения имеется три таблицы: **teatr_ticket**, **teatr_concert** и **teatr_show**. В соответствии с выбранным паттерном создаем для каждой таблице по классу шлюзу таблицы. Таким образом, выделяем классы **TicketGateway**, **ConcertGateway** и **ShowGateway**.

Класс **ShowGateway** – спектакль, который играют в театре. Атрибуты: название, адрес и театр (класс шлюза данных для работы с БД).

Методы:

- deleteShowID – удаление записи о билете в БД (открывает соединение с БД, удаляет запись, сохраняет данные и закрывает соединение).
- getAllShows - получение всех записей из БД (открывает соединение с БД, получает все данные и закрывает соединение).
- filterShow(name) – получение записей с необходимым названием (открывает соединение с БД, получает данные и закрывает соединение).

Класс **ConcertGateway** - представление, конкретное время и дата, когда можно посмотреть определенный спектакль (класс шлюза данных для работы с БД)

Методы:

- getConcertOnShow – выборка всех записей представлений на определенный концерт (show) (открывает соединение с БД получает записи и закрывает соединение).
- deleteConcert – удаление представления (открывает соединение с БД, удаляет запись, сохраняет данные и закрывает соединение).
- getAll() – получение всех представлений (открывает соединение с БД, получает все данные и закрывает соединение).
- insert(self, Row, Cost) – добавление нового представления (открывает соединение с БД, добавляет запись, сохраняет и закрывает соединение).

Класс **TicketGateway** – представляет собой билет на представление (класс шлюза данных для работы с БД)

Методы:

- make – создание билета (открывает соединение с БД, добавляет запись, сохраняет и закрывает соединение).
- getTicket(id) – получение данных определенного билета (открывает соединение с БД, получает запись и закрывает соединение).
- getTicketsOnCons(id, user1) – получение данных билетов на определенное проедставление (открывает соединение с БД получает записи и закрывает соединение).
- delete – удаление билета (открывает соединение с БД, удаляет запись, сохраняет данные и закрывает соединение).

2.3.1.2. Выделение классов Domain model

Поиск классов в соответствии с паттерном **Domain Model** в общем случае является нетривиальной задачей. Необходимо составить объектную модель предметной области и создавать классы в соответствии с ней. Предметной областью приложения по бронированию билетов является «Бронирование билетов в театр». Уже в самой формулировке предметной области фигурирует существительное «**билет**», поэтому его следует выделить в отдельный класс **Ticket**.

Клиент покупает билет, когда хочет посетить некое мероприятие (или **шоу**), которое проходит с некоторой периодичностью. Выделим его в отдельный класс **Show**.

Однако, в конечном счете, билет покупается на **концерт** – конкретное время и дата проведения шоу. Выделим данную сущность в отдельный класс **Concert**.

Таким образом, мы выделили три класса модели предметной области: **Ticket**, **Show** и **Concert**.

Класс **Concert** - представление, конкретное время и дата, когда можно посмотреть определенный спектакль (класс доменной модели)

Методы:

- MakeTickets – создание множества билетов на представление .
- getConcertOnShow(show) – выборка всех представлений на определенный концерт (show) (вызов метода deleteConcert класса работы с БД ConcertGateway).
- delete(self, user1) – удаление представления (вызов метода deleteConcert класса работы с БД ConcertGateway).
- getAll() – получение всех представлений (вызов метода getAll класса работы с БД ConcertGateway).
- update(self) – обновление нового представления (вызов метода update класса работы с БД ConcertGateway).
- insert(self, Row, Cost) – добавление нового представления (вызов метода insert класса работы с БД ConcertGateway).

Класс **Ticket** – представляет собой билет на представление (класс доменной модели)

Методы:

- make – создание билета (вызов метода make класса работы с БД TicketGateway)
- getTicket(id) – получение данных определенного билета (вызов метода getTicket класса работы с БД TicketGateway)
- getTicketsOnCons(id, user1) – получение данных билетов на определенное проедставление (вызов метода getTicketsOnCons класса работы с БД TicketGateway)
- delete – удаление билета (вызов метода delete класса работы с БД TicketGateway)

Класс **Show** – спектакль который играют в театре. Атрибуты: название, адрес и театр. (класс доменной модели)

Методы:

- delete – удаление объекта класса (вызов метода deleteShowID класса работы с БД ShowGateway).
- getAllShows - получение всех объектов (вызов метода getAllShows класса работы с БД ShowGateway).
- filterShow(name) – получение объектов с необходимым названием (вызов метода filterShow класса работы с БД ShowGateway).

Стоит отметить, что в рамках паттерна Domain Model возможен дальнейший поиск сущностей с последующим выделением этих сущностей в классы модели предметной области, но в рамках данного примера остановимся на уже выделенных классах.

2.3.1.3. Прочие классы программы

Класс **ShowsList** – страница входа, по нажатию кнопки перенаправляет на страницы добавления концерта (**AddConcert**), изменения спектакля (**ChangeShow**), просмотра концертов на выбранный спектакль (**ConsertListShow**).

Класс **AddConcert** – добавление нового концерта вызывает форму (GetConcertForm) – форма добавления, непосредственно из метода save() этой формы происходит вызов класса **Concert** доменной модели.

2.3.1. Диаграммы классов и последовательностей

Диаграмма классов для прецедента Создания билетов на концерт приведена на рисунке 12.

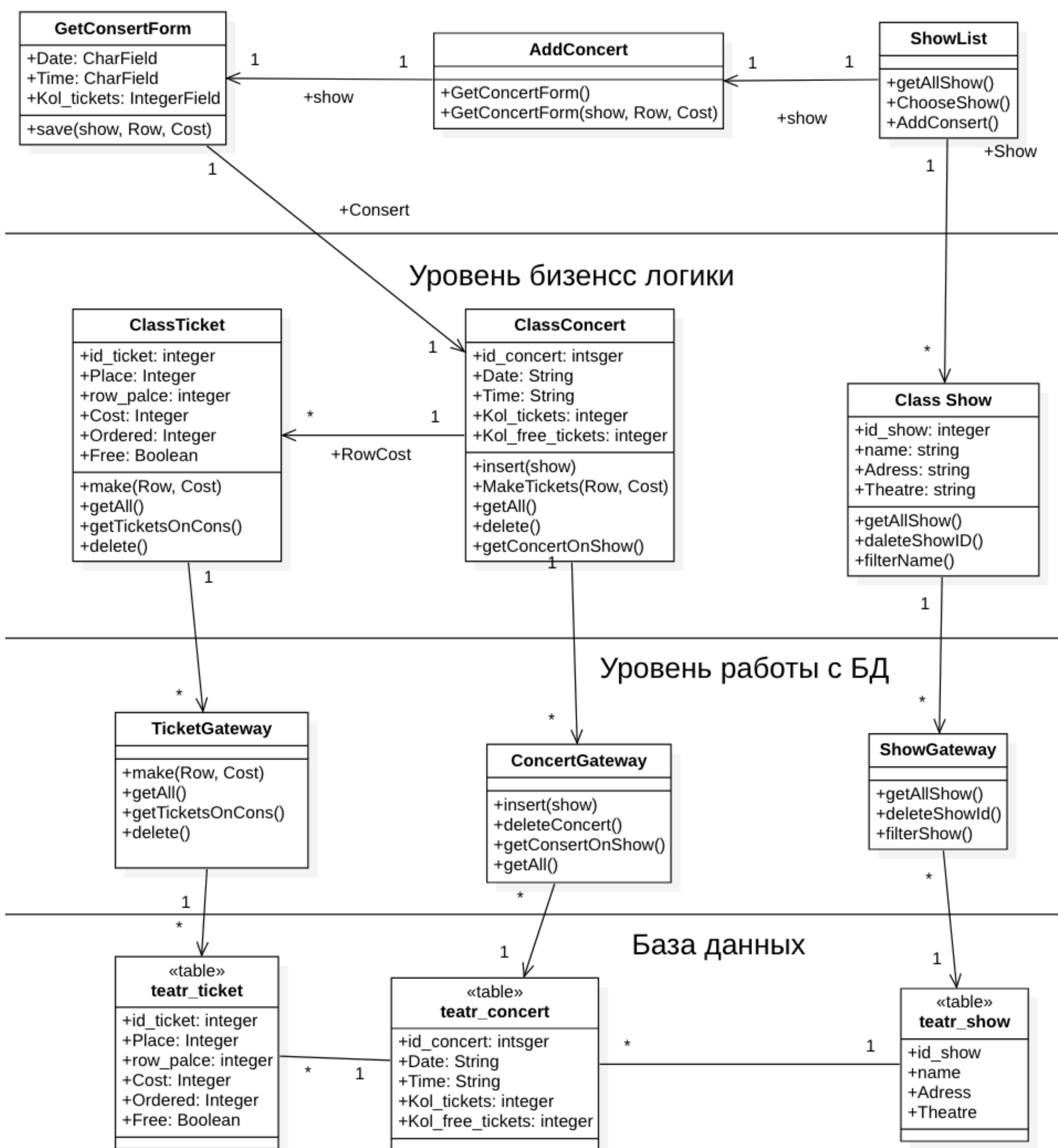

Рисунок 12 – Диаграмма классов для прецедента Создание билетов на концерт

Диаграмма последовательностей для прецедента Создания билетов на концерт приведена на рисунке 13.

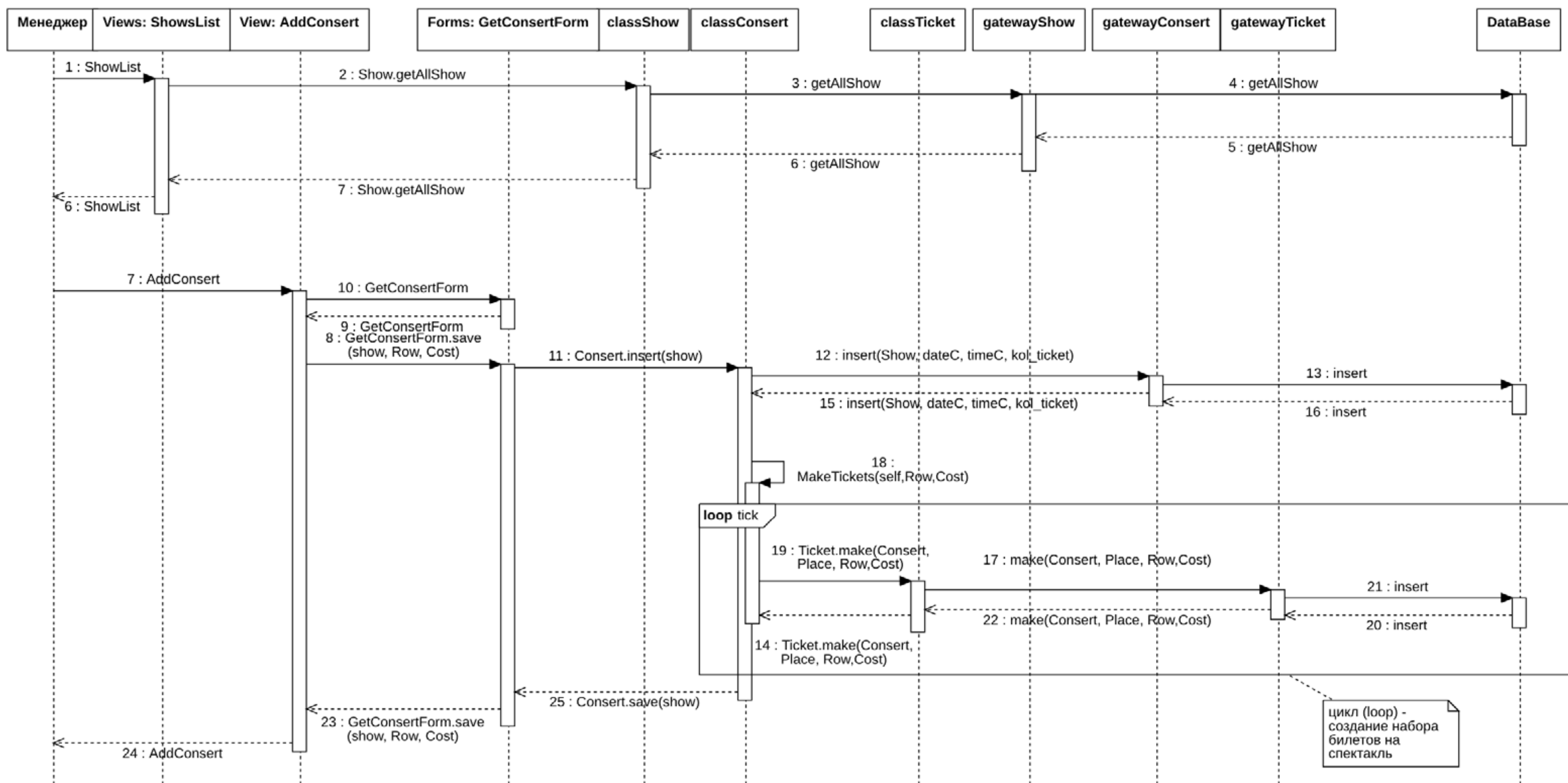

Рисунок 13 – Диаграмма последовательности для прецедента Создание билетов на концерт

2.3.1. Программный код

Начальная страница входа – **ShowList** все последующие формы и представления вызываются с этой страницы.

```
def ShowsList(request):
      shows = Show.getAllShows() # Реализация Table Data Gateway
      if request.method == 'POST':
          if '_search' in request.POST:
              Name = request.POST['Name']
              if Name:
                  shows = Show.filterShow(Name)
      if request.method == 'GET':
          if request.GET.get('del'):
              id = request.GET.get('del')
              s = Show.delete(id)
              shows = Show.getAllShows()
              return render(request, 'LookShow.html', locals())
          if request.GET.get('chng'):
              id = request.GET.get('chng')
              return redirect('ChangeShow', id=id)
          if request.GET.get('add'):
              id = request.GET.get('add')
              return redirect('AddConcert', id=id)
          if request.GET.get('look'):
              id = request.GET.get('look')
              return redirect('ConsertListShow', id_show=id)
      return render(request, 'LookShow.html', locals())
```

Добавление нового концерта **AddConcert** – представление, **GetConcertForm** – форма добавления, непосредственно из метода save() этой формы происходит вызов класса **Concert** доменной модели.

```
def AddConcert(request, id):
	shows = Show.objects.get(id_show=id)
	if request.method == 'POST':
	    form = GetConcertForm(request.POST,request.FILES)
	    if form.is_valid():
	        Row = request.POST['Row']
	        Cost = request.POST['Cost']
	        consert = form.save(shows, Row, Cost) # Вызов функции,
	                                 # которая реализует Domain Model
	        return render(request, '../templates/AddShowDone.html',
{'form': form})
	else:
	    form = GetConcertForm()
	return render(request, '../templates/AddConcert.html', locals())

class GetConcertForm(forms.Form):
	Date = forms.CharField(label='дата', max_length=255)
	Time = forms.CharField(label='время', max_length=255 )
	Kol_tickets = forms.IntegerField(label='количество билетов')

	def clean(self):
	    . . . . . . . . .
	    return self.cleaned_data

	def save(self, show, Row, Cost):
	    concert = Concert()
	    concert.Date = self.cleaned_data['Date']
	    concert.Time = self.cleaned_data['Time']
	    concert.Kol_tickets = self.cleaned_data['Kol_tickets']
	    concert.Kol_free_tickets = self.cleaned_data['Kol_tickets']
	    concert.Show = show
	    concert.insert(Row,Cost) # вызов функции, которая реализует
	                             # Domain Model
	    return concert
```

2.3.1.1. Реализация классов Domain model

Класс доменной модели **Show** в этой предметной области представляет собой спектакль, который играют в театре. Атрибуты: название, адрес и театр.

```
class Show(models.Model):
      id_show = models.AutoField(primary_key=True)
      name = models.CharField(max_length=255, verbose_name='Назавание')
      adress = models.CharField(max_length=255, verbose_name='Адрес')
      Theatre = models.CharField(max_length=255, verbose_name='Театр')

      def __str__(self):
          return str(self.id_show)

      def delete(id):
          conserts = Consert.objects.filter(Show = id)
          for i in conserts:
              i.deleteConsert()
          ShowGateway.deleteShowID(id)

      def getAllShows():
          return ShowGateway.getAllShows()

      def filterShow(name):
          return ShowGateway.filterShow(name)
```

Класс доменной модели **Concert** в этой предметной области представляет собой представление, конкретное время и дата, когда можно посмотреть определенный спектакль.

```
class Concert(models.Model):
      id_concert = models.AutoField(primary_key=True)
      Show = models.ForeignKey(Show, on_delete=models.CASCADE,
related_name="Show" )
      Date = models.CharField(max_length=255, verbose_name='дата' )
      Time = models.CharField(max_length=255, verbose_name='время' )
      Kol_tickets = models.IntegerField(verbose_name='количество билетов' )
      Kol_free_tickets = models.IntegerField(verbose_name='количество
свободных билетов')
```

```
    def __str__(self):
        return str(self.id_concert)

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def MakeTickets(self, Row, Cost):
        Place = int(self.Kol_free_tickets) / int(Row)
        for i in range(1,int(Row)+1):
            for j in range(1,int(Place)+1):
                ticket = Ticket()
                ticket = ticket.make(self, j, i, Cost)
        return self

    def delete(self, user1):
        tickets = Ticket.getTicketsOnCons(self.Concert, user1.is_staff)
        for i in tickets:
            i.delete()
        return ConcertGateway.deleteConcert(self.id)

    def getConcertOnShow(show):
        return ConcertGateway.getConcertOnShow(show.id)

    def getAll():
        return ConcertGateway.getAll()

    def update(self):
        return ConcertGateway.update(cons)

    def insert(self, Row, Cost):
        ConsertGateway.insert(self)
        self.MakeTickets(Row,Cost)
        return self
```

Класс доменной модели **Ticket** в этой предметной области представляет собой билет на определенное представление.

```
class Ticket(models.Model):
	id_ticket = models.AutoField(primary_key=True)
	Concert = models.ForeignKey(Concert, on_delete=models.CASCADE, related_name="Concert")
	Place = models.CharField(max_length=100, verbose_name='Место')
	row_place = models.CharField(max_length=100, verbose_name='Ряд')
	Cost = models.IntegerField(verbose_name='Цена билета')
	Ordered= models.IntegerField(verbose_name='номер заказа', default=0)
	Free = models.BooleanField(verbose_name='Свободно/Занято', default=True)

	def __str__(self):
	    return str(self.id_ticket)

	def __init__(self, *args, **kwargs):
	    super().__init__(*args, **kwargs)

	def make(self, concert, Place, Row, Cost):
	    TicketGateway.make(self, concert, Place, Row, Cost)

	def getTicket(id):
	    return TicketGateway.getTicket(id)

	def getTicketsOnCons(id, user1):
	    return TicketGateway.getTicketsOnCons(id, user1.is_staff)

	def delete(self):
	    return TicketGateway.delete(self.id)
```

2.3.1.2. Реализация классов Data Gateway

Далее представлены шлюзы таблиц для каждого класса доменной модели. Класс **ShowGateway** является шлюзом таблицы данных для таблицы teatr_show.

```
class ShowGateway:
      def getAllShows():
          conn = psycopg2.connect(dbname='DZ', user='name', password =
'pass')
          cursor = conn.cursor()
          s = 'teatr_show'
          # Выполняем запрос.
          cursor.execute("SELECT * FROM %s" %s)
          shows = cursor.fetchall()
          # Закрываем подключение.
          cursor.close()
          conn.close()
          return shows

      def deleteShowID(id):
          conn = psycopg2.connect(dbname='DZ', user='name', password =
'pass')
          cursor = conn.cursor()
          s = 'teatr_show'
          i = id
          sqlString = "DELETE FROM " + s + " WHERE id_show="+str(i)
          cursor.execute(sqlString)
          conn.commit()
          return 0

      def filterShow(name):
          conn = psycopg2.connect(dbname='DZ', user='name',password='pass')
          cursor = conn.cursor()
          s = "teatr_show"
          i = name
          sqlString = "Select * From " + s + " Where name="+str(i)
          cursor.execute(sqlString)
          show = cursor.fetchall()
          cursor.close()
          conn.close()
          return show
```

Класс **ConcertGateway** – шлюз таблицы данных для таблицы teatr_concert.

```
class ConcertGateway:
      def insert(Show, dateC, timeC, kol_ticket):
          conn = psycopg2.connect(dbname='DZ', user='name',password='pass')
          cursor = conn.cursor()
          s = 'teatr_concert'
          sqlString = "INSERT INTO " + s + " VALUES ("+str(Show)+ "," +
str(dateC)+","+ str(timeC) +","+ str(kol_tiket)+","+ str(kol_free_tiket)
+")"
          cursor.execute(sqlString)
          conn.commit()
          return 0

      def getConcertOnShow(id):
          conn = psycopg2.connect(dbname='DZ', user='name', password =
'pass')
          cursor = conn.cursor()
          s = "teatr_concert"
          i = id
          sqlString = "Select * From " + s + " Where Show="+str(i)
          cursor.execute(sqlString)
          concert = cursor.fetchall()
          cursor.close()
          conn.close()
          return concert

      def getAll():
          conn = psycopg2.connect(dbname='DZ', user='name',password='pass')
          cursor = conn.cursor()
          s = 'teatr_concert'
          # Выполняем запрос.
          cursor.execute("SELECT * FROM %s" %s)
          concert = cursor.fetchall()
          # Закрываем подключение.
          cursor.close()
          conn.close()
          return concert
      def deleteConcert(id):
```

```
        conn = psycopg2.connect(dbname='DZ', user='name', password =
'pass')
        cursor = conn.cursor()
        s = 'teatr_concert'
        i = id
        sqlString = "DELETE FROM " + s + " WHERE id_concert="+str(i)
        cursor.execute(sqlString)
        conn.commit()
        return 0
```

Класс **TicketGateway** представляет собой шлюз таблицы данных для таблицы teatr_ticket.

```
class TicketGateway:
    def make(ticket, concert, Place, Row, Cost):
        conn = psycopg2.connect(dbname='DZ', user='name',
password='pass')
        cursor = conn.cursor()
        s = 'teatr_ticket'
        sqlString = "INSERT INTO " + s + " VALUES ("+str(concert)+ ", " +
str(place)+", "+ str(row) +", "+ str(cost) +")"
        cursor.execute(sqlString)
        conn.commit()
        return 0

    def getTicket(id):
        conn = psycopg2.connect(dbname='DZ', user='name',password='pass')
        cursor = conn.cursor()
        s = 'teatr_ticket'
        # Выполняем запрос.
        sqlString = "SELECT * FROM" + s + "WHERE id="+ str(id)
        cursor.execute(sqlString)
        ticket = cursor.fetchall()
        # Закрываем подключение.
        cursor.close()
        conn.close()
        return ticket
```

```
      def getTicketsOnCons(id, staff):
          cons = Concert.objects.get(id_concert = id)
          if staff == False:
              conn = psycopg2.connect(dbname='DZ', user='name', password =
'pass')
              cursor = conn.cursor()
              s = 'teatr_ticket'
              # Выполняем запрос.
              sqlString = "SELECT * FROM" + s + "WHERE Concert="+ str(id) +
"and Free = True"
              cursor.execute(sqlString)
              ticket = cursor.fetchall()
              # Закрываем подключение.
              cursor.close()
              conn.close()
          elif staff == True:
              conn = psycopg2.connect(dbname='DZ', user='name', password =
'pass')
              cursor = conn.cursor()
              s = 'teatr_ticket'
              # Выполняем запрос.
              sqlString = "SELECT * FROM" + s + "WHERE Concert="+ str(id)
cursor.execute(sqlString)
              ticket = cursor.fetchall()
              # Закрываем подключение.
              cursor.close()
              conn.close()
      return ticket
```

```
      def delete(id):
          conn = psycopg2.connect(dbname='DZ', user='name', password =
'pass')
          cursor = conn.cursor()
          s = 'teatr_ticket'
          i = id
          sqlString = "DELETE FROM " + s + " WHERE id_ticket="+str(i)
          cursor.execute(sqlString)
          conn.commit()
          return 0
```

3. СПИСОК ИСТОЧНИКОВ

1. Фаулер М. Архитектура корпоративных приложений. - М.:Изд.дом Вильямс. - 2008г.
2. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. — СПб.: Питер, 2018. — 352 с.: ил. — (Серия «Библиотека программиста»)
3. Лутц, Марк. Изучаем Python, 5-е изд.: Пер. с англ. — СПб.: ООО “Диалектика”, 2019. — 832 с. : ил. — Парад, тит. англ.
4. Django documentation. — Текст : электронный // djangoproject.com : [сайт]. — URL: https://docs.djangoproject.com/en/4.1/ (дата обращения: 26.06.2022).
5. Мюллер, Джон Пол, Семпф, Билл, Сфер, Чак. C# для чайников.: Пер. с англ. - СПб.: ООО "Диалектика", 2019. - 608 с. : ил. - Парал. тит. англ.
6. Хорстманн, Кей С. Java. Библиотека профессионала, том 1. Основы. 10-е и зд.: Пер. с англ. — М .: ООО "И.Д. Вильямс", 2016. — 864 с . : ил. — Парал. тит. англ.