С.А. Петров

Техническое обеспечение информационных систем: объектно-ориентированная технология обработки данных с Entity Framework

Учебное пособие для студентов,

обучающихся по направлениям «Прикладная информатика», «Бизнес-информатика»

***Аннотация***

Пособие предназначено для студентов технических специальностей, изучающих информатику и языки программирования.

В пособии рассмотрены технологии ADO.NET, EntityFramework, LINQ to Entities для обработки данных в приложениях, реализованных на платформе .NET. Практическое выполнение базовых операций, в том числе для поиска данных, продемонстрировано на примере реализации прототипа информационной системы.

Оглавление

[1. ПОНЯТИЕ ИНФОРМАЦИОННОЙ СИСТЕМЫ 4](#_Toc497906821)

[1.1. Определение 4](#_Toc497906822)

[1.2. Информационная система «Классный журнал» 4](#_Toc497906823)

[1.3. Контрольные вопросы и задания 5](#_Toc497906824)

[2. ТЕХНОЛОГИЯ ОБРАБОТКИ ДАННЫХ ADO.NET 5](#_Toc497906825)

[2.1. Моделирование предметной области в ADO.NET Entity Data Model 5](#_Toc497906826)

[2.2. Генерация структуры базы данных 8](#_Toc497906827)

[2.3. Контрольные вопросы и задания 13](#_Toc497906828)

[3. ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ ОБРАБОТКА ДАННЫХ С ИСПОЛЬЗОВАНИЕМ ENTITY FRAMEWORK 13](#_Toc497906829)

[3.1. Классы Entity Framework 14](#_Toc497906830)

[3.2. Поддержка операций Create, Read, Update, Delete 15](#_Toc497906831)

[3.3. Контрольные вопросы и задания 23](#_Toc497906832)

[4. ВВЕДЕНИЕ В LINQ 23](#_Toc497906833)

[4.1. Поиск единственной записи – методы Single, SingleOrDefault, First, FirstOrDefault 24](#_Toc497906834)

[4.2. Выборка нескольких значений – методы Where, OrderBy, GroupBy, Join 25](#_Toc497906835)

[4.3. Контрольные вопросы и задания 32](#_Toc497906836)

[Заключение 32](#_Toc497906837)

[СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ 32](#_Toc497906838)

# ПОНЯТИЕ ИНФОРМАЦИОННОЙ СИСТЕМЫ

## Определение

Различают два определения понятия информационной системы (ИС). В широком смысле ИС – это система, предназначенная для хранения, поиска и обработки информации, и соответствующие организационные ресурсы (человеческие, технические, финансовые и т.д.), которые обеспечивают и распространяют информацию [1]. В более узком смысле понятие информационной системы ограничено данными, программным и аппаратным обеспечением.

Задача данного пособия заключается в демонстрации возможностей платформы Microsoft .NET для создания программной части ИС. В ходе изложения будет разработан прототип информационной системы с применением объектно-ориентированных аспектов разработки, небольшой базы данных и интерфейса в консольном окне.

## Информационная система «Классный журнал»

В качестве предметной области для ИС возьмём учебное заведение для получения общего образования – школу.

Рассмотрим следующую абстракцию процесса учёта посещаемости учебных занятий обучающимися. Школьники (или ученики) разделены на учебные классы и обязаны посещать уроки по различным предметам. За проведение уроков по каждому из предметов отвечает отдельный преподаватель. Ограничимся рассмотрением одного учебного класса и разработаем систему, являющуюся аналогом классного журнала. Пусть все уроки учебного класса проводятся в специально отведённой классной комнате (классе). Такой подход практикуется в младшей школе. В назначенное время урока в класс должен явиться преподаватель и все ученики, после чего преподаватель отмечает всех присутствующих.

Для перехода на электронный учёт необходимо обеспечить каждый класс компьютером, но такой вариант учёта посещаемости имеет много преимуществ перед бумажным. В случае обработки данных в информационной системе сокращаются (исключаются) затраты на закупку и хранение бумажных журналов; снижается вероятность утери журнала; повышается скорость получения сводной информации и др. В перспективе система может быть распространена на всё учебное заведение. За счёт этого информация будет более доступной для пользователей различного уровня, сократятся финансовые и трудовые затраты по ведению классных журналов.

Таким образом, ИС должна обеспечить:

1. Хранение и обработку информации о школьных преподавателях: фамилию, имя, отчество, преподаваемый предмет.
2. Хранение и обработку информации об учениках одного школьного класса: фамилию, имя, отчество.
3. Хранение и обработку информации об уроках: тему, дату проведения, преподавателя, список присутствующих.
4. Получение сводных данных по информации, содержащейся в системе.

Первая задача, которую необходимо выполнить, – это разработка схемы данных ИС и создание соответствующей структуры базы данных. Для её решения используем технологию ADO.NET.

## Контрольные вопросы и задания

1. Дайте определение понятию информационной системы в широком и узком смыслах. В чём их отличие?
2. Рассмотрите знакомую предметную область и процессы, происходящие в ней, постройте абстракцию, для которой может быть реализована ИС.

# ТЕХНОЛОГИЯ ОБРАБОТКИ ДАННЫХ ADO.NET

Данные, которыми оперирует ИС, обычно хранятся в базе данных. База данных (БД) – совокупность данных, хранимых в соответствии со схемой данных, манипулирование которыми выполняют в соответствии с правилами средств моделирования данными [2]. Схема данных описывает содержание, структуру и ограничения целостности, которые используются для создания и поддержки БД. Структурой данных является программная единица, позволяющая хранить и обрабатывать данные. За структуру БД отвечает система управления базами данных, через которую осуществляются все взаимодействия с БД. Система управления базами данных (СУБД) – это совокупность языковых и программных средств, предназначенных для создания, ведения и совместного использования БД многими пользователями.

## Моделирование предметной области в ADO.NET Entity Data Model

Разработка ИС начинается с моделирования предметной области и создания схемы БД. Следующим шагом является получение структуры БД в конкретной СУБД. Поэтому современные программные средства, ориентированные на моделирование баз данных, поддерживают и механизмы переноса модели в СУБД.

Microsoft для работы с данными предлагает технологию ADO.NET. В её состав входит конструктор моделей ADO.NET Entity Data Model (EDM), позволяющий визуально смоделировать базу данных, а затем сгенерировать код на языке DDL для создания структуры БД. Язык описания данных DDL (Data Definition Language) является подмножеством языка Transact-SQL (реализация языка запросов к СУБД от компании Microsoft) и отвечает за создание новых объектов в БД. Помимо создания структуры БД, конструктор EDM позволяет сгенерировать объектно-ориентированные классы для работы с таблицами уже существующей базы. В общем случае такая технология называется объектно-реляционное отображение (ORM, Object-Relational Mapping). ADO.NET содержит реализацию технологии ORM, которая называется Entity Framework (EF).

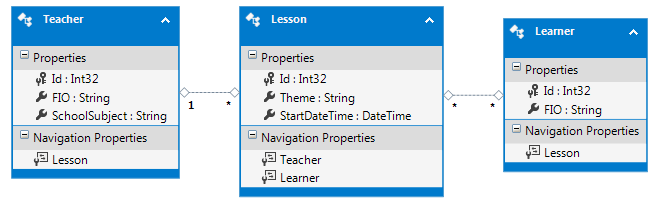
Для разработки ИС нам необходимо спроектировать и создать структуру БД, а затем организовать обработку данных через СУБД с помощью ООП классов нашего приложения.

Для работы над ИС создадим новый проект в MS Visual Studio типа консольное приложение под названием InfoSystem. Чтобы использовать конструктор моделей EDM в проект необходимо добавить новый элемент: «Модель ADO.NET EDM» из раздела «Данные». Мастер создания моделей EDM потребует ввести имя новой модели (назовём её SchoolModel). На вкладке «Выбор содержимого модели» выберем пункт «пустая модель конструктора EF». В проект добавится файл с расширением .EDMX и отобразится пустое окно конструктора EDM.

Создание модели осуществляется с помощью панели элементов («Вид → Панель элементов»). Она позволяет добавлять новые сущности (исходная информация для создания таблиц в БД) и задавать отношения ассоциации и наследования между ними.

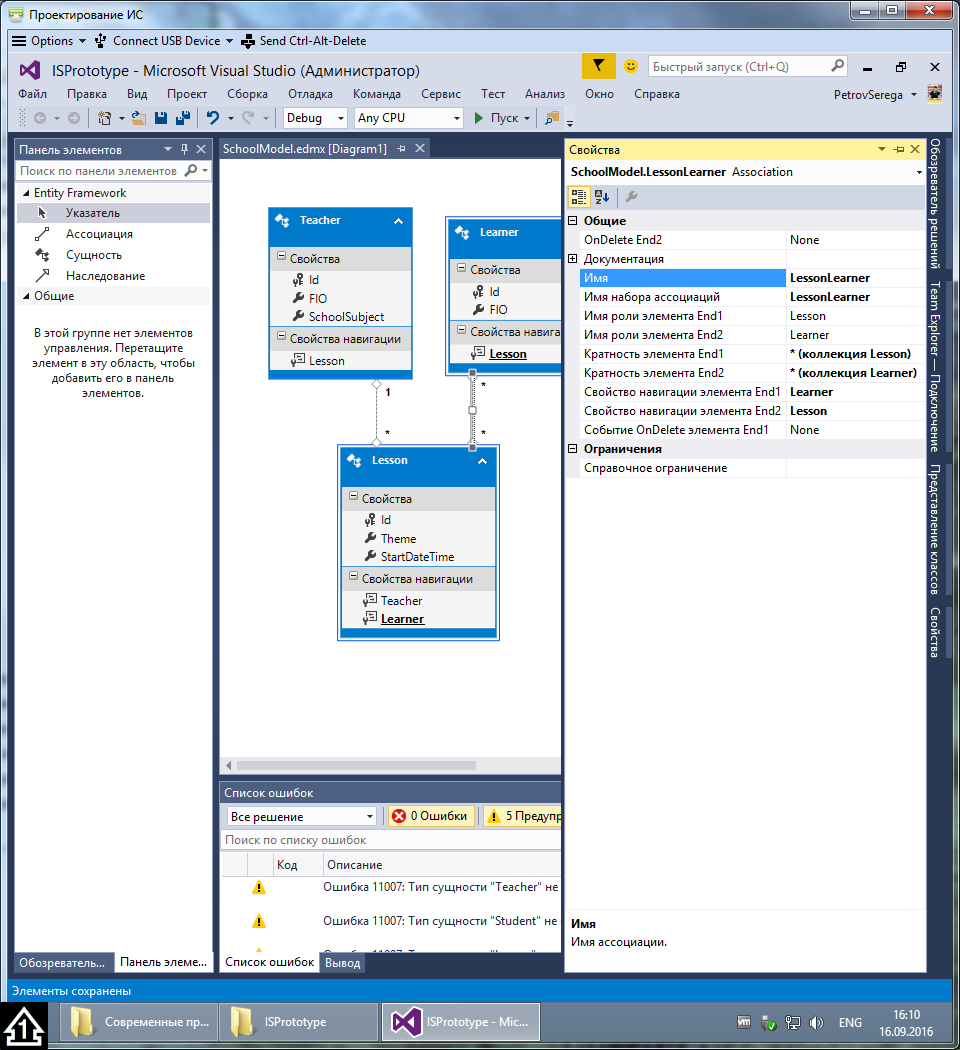
В общем случае, таблицы БД могут не соответствовать «один к одному» классам ООП. Но мы ограничимся достаточно простым случаем, когда каждой таблице БД соответствует один класс ООП, с одним исключением – связь «многие-ко-многим» между учеником и уроком на схеме EDM представляется одной связью, но для её реализации в БД потребуется дополнительная промежуточная таблица.

Для моделирования БД нам потребуются элементы «Сущность» и «Ассоциация». Сущности будут соответствовать таблицам в БД, а на основании ассоциаций будут созданы внешние ключи и ограничения таблиц. Трижды перетащим элемент Сущность в конструктор EDM и зададим названия: Teacher (преподаватель), Learner (ученик), Lesson (урок). Затем, используя элемент «Ассоциация», свяжем сущности согласно нашему представлению предметной области.



**Рис. 2.1. Схема БД в конструкторе EDM**

По нашей задумке, преподаватель связан с уроком ассоциацией «один-ко-многим» (один преподаватель провёл много уроков), а ученик с уроком – «многие-ко-многим» (один ученик посетил много уроков и много учеников посетили один урок). По умолчанию при создании ассоциации конструктор EDM добавляет связь «один-ко-многим». Поэтому кратность ассоциации между учеником и уроком надо изменить. Для этого необходимо выделить связь и открыть окно «Свойства» («Вид → Окно Свойств», рис. 2.2). Для изменения кратности связи предусмотрены пункты: «Кратность элемента End1» и «Кратность элемента End2».



**Рис. 2.2. Редактирование связи типа ассоциация в конструкторе EDM**

Помимо связей модель EDM позволяет задать свойства сущностей (будущие колонки БД), добавить которые можно из контекстного меню сущности («Контекстное меню → Добавить новый → Скалярное свойство»). Для простоты в нашей системе фамилия, имя и отчество ученика будут храниться в одном свойстве под названием FIO. Сущность преподаватель будет содержать аналогичное свойство FIO плюс SchoolSubject для хранения преподаваемого предмета. Урок опишем с помощью двух полей: тема занятия (Theme) и дата со временем проведения (StartDateTime).

Для редактирования свойств сущностей, как и для редактирования отношений, служит окно «Свойства». При добавлении новых свойств в модель EDM им автоматически присваивается тип String и максимально поддерживаемая СУБД длина. Изменить тип свойства можно в разделе «Общие», а максимальную длину – в разделе «Аспекты». Ограничим длину всех свойств строкового типа 100 символами, а тип свойства StartDateTime изменим на тип DateTime.

Конструктор EDM позволяет разработать модель для генерации структуры БД, получить модель существующей базы или на основании ООП классов построить модель и сгенерировать структуру БД. Мы воспользуемся возможностью генерации структуры, но перед этим нам потребуется создать БД и определиться, какая СУБД будет обеспечивать её функционирование.

## Генерация структуры базы данных

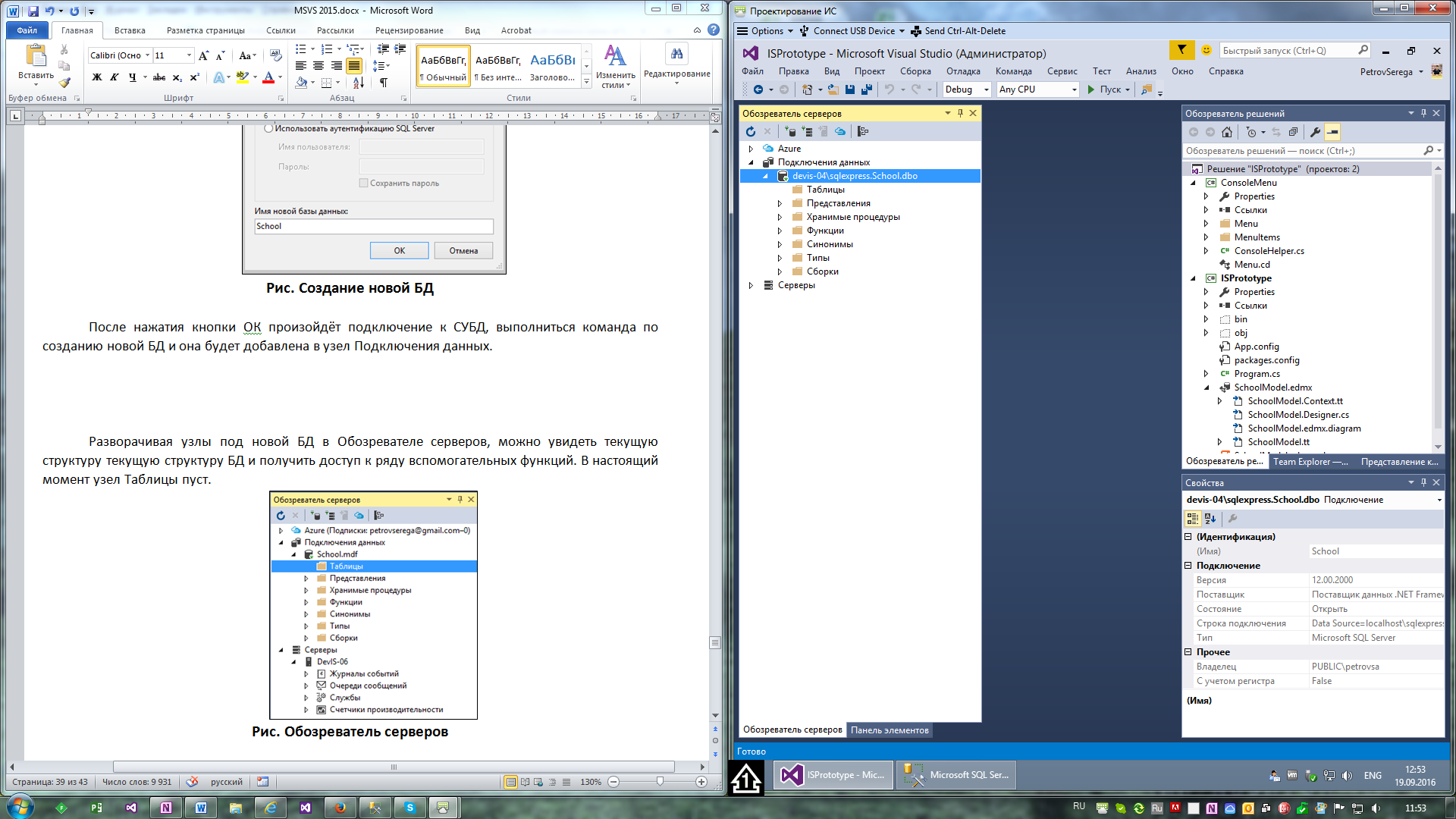
В сфере систем управления базами данных Microsoft предоставляет решение Microsoft SQL Server. Это программная платформа для работы с базами данных, обладающая огромным количеством возможностей, среди которых поддержка транзакционных механизмов, аудит событий сервера, полнотекстовый поиск, службы интеграции и преобразования данных, механизмы репликации, службы анализа данных, платформа для построения отчётов и многое другое. Как и Visual Studio, СУБД SQL Server поставляется в различных редакциях с различным функционалом и ограничениями. Бесплатная версия называется SQL Express, скачать её можно с официального сайта: <https://www.microsoft.com/en-us/cloud-platform/sql-server> [3].

После установки будут запущены службы операционной системы, обеспечивающие работу с СУБД. Для просмотра объектов, имеющихся в СУБД SQL Server, используется «Обозреватель серверов» («Вид→ Обозреватель серверов»). Он отображает подключения данных (отдельные файлы БД или СУБД, предлагающие доступ к БД), подключённые сервера (например, для быстрого просмотра журнала событий) и подключения к «облачным» службам (например, для просмотра данных, хранящихся на серверах Microsoft). Есть и другие программы для «общения» с SQL Server. Наиболее популярная – SQL Server Management Studio (SSMS). Она требует отдельной установки (может быть выполнена при установке SQL Express), поэтому остановимся на интегрированном в Visual Studio обозревателе серверов.

Для добавления нового подключения к БД необходимо в контекстном меню узла «Подключение данных» выбрать пункт «Добавить подключение», но, так как в нашем случае база ещё не создана, воспользуемся опцией «Создать новую базу данных SQL Server».

Откроется окно, в котором необходимо указать имя сервера СУБД, параметры входа на сервер и имя новой БД. SQL Server находится на локальном компьютере, поэтому имя сервера СУБД будет начинаться с имени локального компьютера. Вместо конкретного имени удобно указать localhost или «.». Далее идёт имя экземпляра SQL Server. Для редакции SQL Express именем по умолчанию является SQLExpress. Для подключения к СУБД будем использовать аутентификацию Windows, а в качестве имени новой БД укажем School.

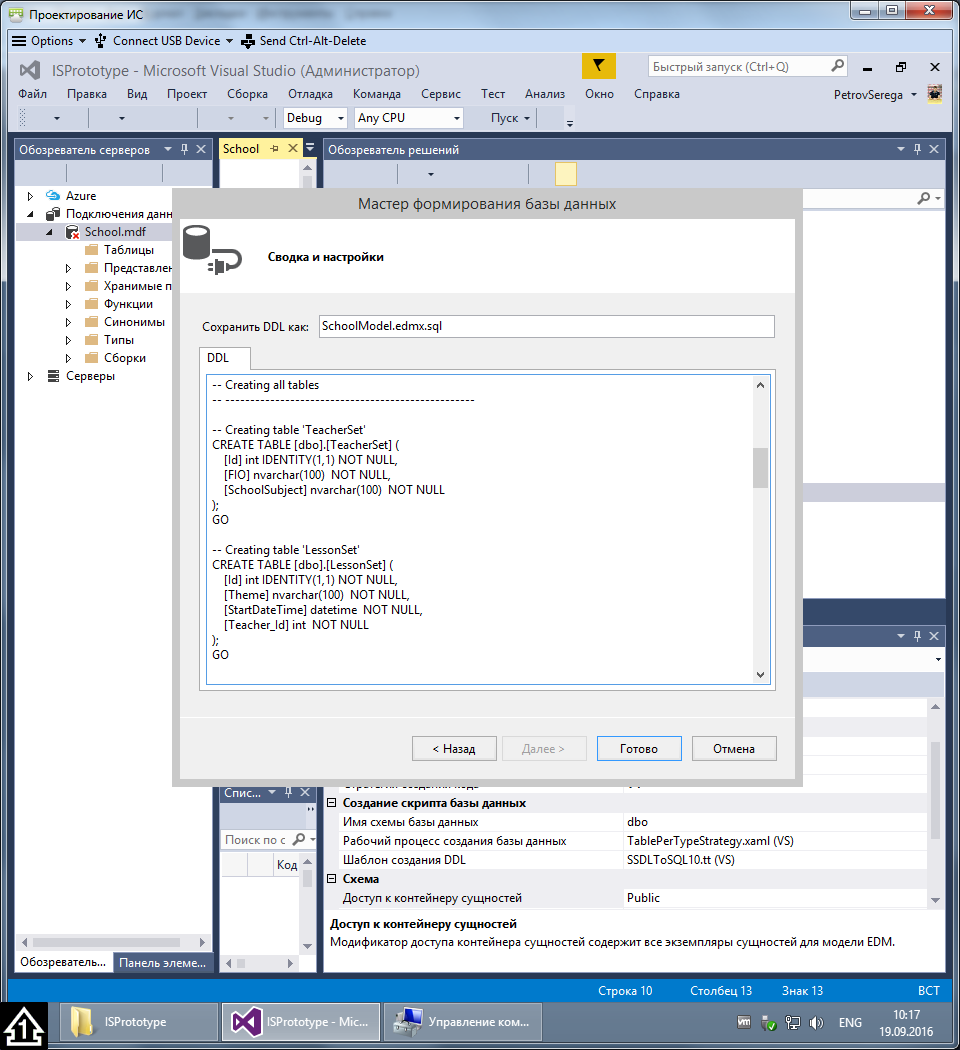
После нажатия на кнопку «ОК» произойдёт подключение к СУБД, выполнятся команды создания БД, и соответствующая новая запись отобразится в узле «Подключения данных». Разворачивая узлы под новой БД в обозревателе серверов, можно увидеть текущую структуру БД и получить доступ к ряду вспомогательных функций. В настоящий момент узел «Таблицы» пуст (рис. 2.3).



**Рис. 2.3. Обозреватель серверов**

Для генерации структуры БД необходимо в контекстном меню EDM модели выбрать пункт «Сформировать базу данных на основе модели». Откроется мастер формирования базы данных, который предложит выбрать подключение для соединения с БД и сохранить строку подключения в файле настроек приложения app.config. Если автоматически подключение к базе School предложено не будет, необходимо нажать «Создать соединение», выбрать источник базы данных «Microsoft SQL Server», указать имя сервера и имя БД.

На следующем шаге предлагается выбрать версию Entity Framework, которая будет обеспечивать генерацию ООП классов для связи с таблицами БД. Выберем версию 6.x. В результате работы мастера сгенерируется DDL код, отвечающий за создание структуры БД, которая соответствует исходной EDM модели (рис. 2.4).

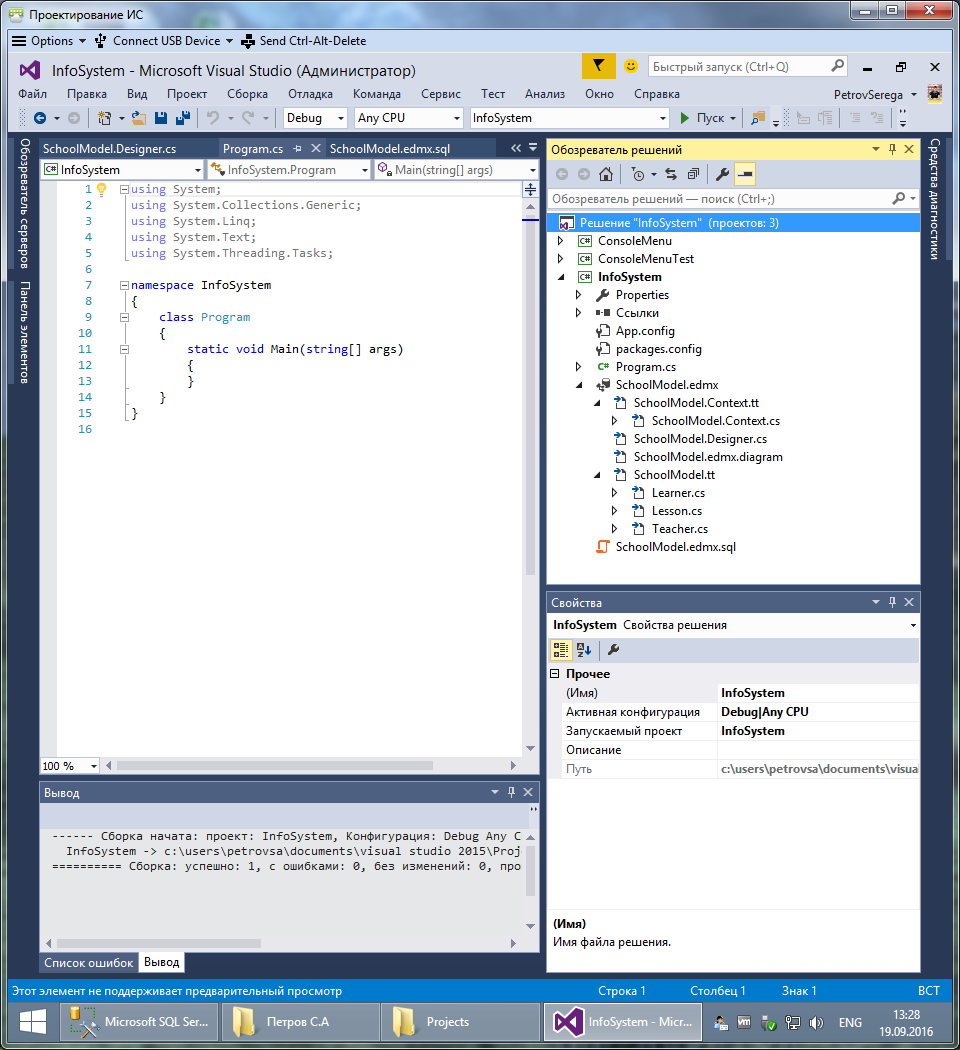


**Рис. 2.4. Результат работы мастера формирования базы данных**

Другой результат вызова мастера формирования базы данных – это добавление в проект файлов-шаблонов Text Template Transformation Toolkit (T4). Их использует Entity Framework и подробнее они будут рассмотрены в следующем разделе. Сейчас важно, что к этим шаблонам относится предупреждение безопасности, которые может отобразиться после закрытия мастера. Оно сообщает о потенциальной угрозе запуска шаблона текста, так как шаблон мог быть получен из ненадёжного источника. Шаблоны, связанные с генерацией классов Entity Framework, мы считаем надёжными, поэтому соглашаемся на запуск генерации кода. Чтобы окно с предупреждением больше не показывалось, следует отметить опции «Больше не выводить это сообщение».

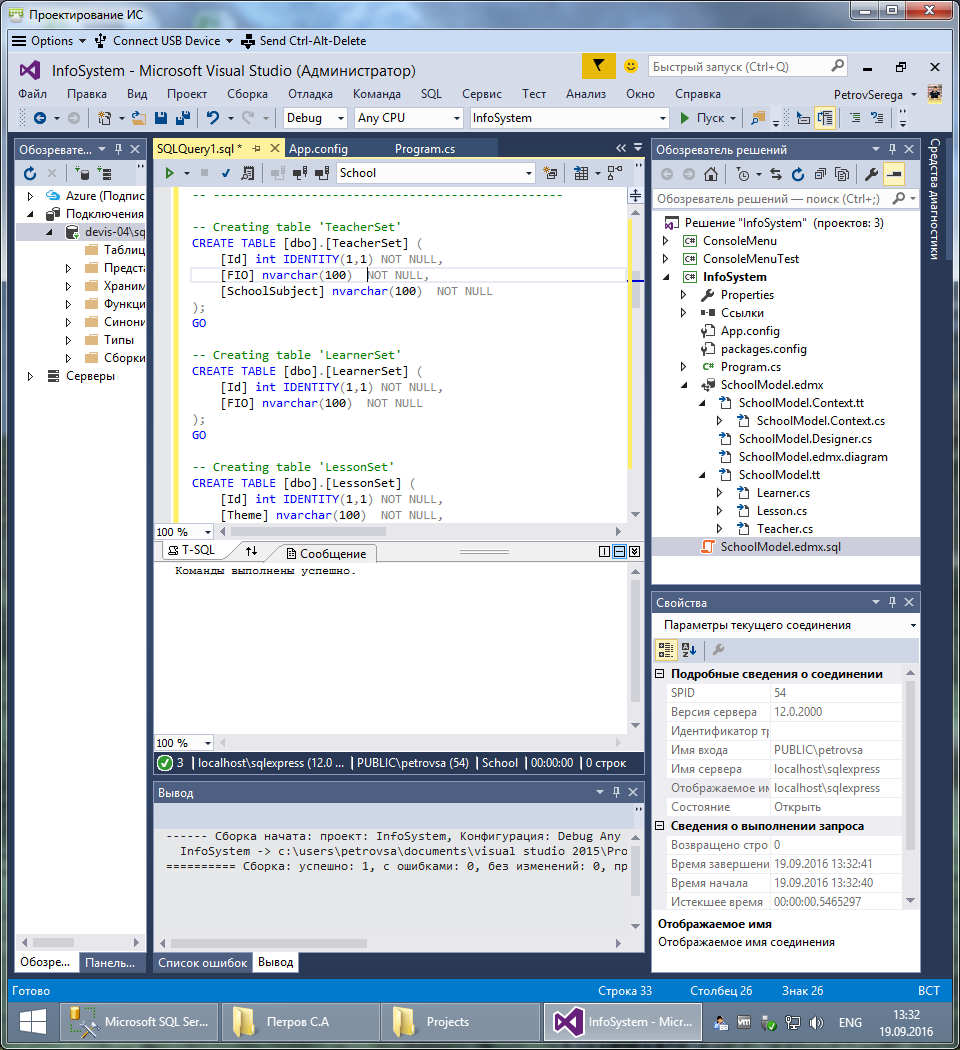
На рисунке 2.5 представлена текущая структура проекта, в которой присутствуют:

* Файл базы данных School.mdf;
* Файл, содержащий DDL-код для создания структуры БД, – SchoolModel.edmx.sql;
* Шаблоны T4 – файлы SchoolModel.Context.tt и SchoolModel.tt;
* Файлы SchoolModel.Context.cs, Learner.cs, Lesson.cs, Teacher.cs, содержащие сгенерированные Entity Framework классы для взаимодействия с СУБД.



**Рис. 2.5. Текущая структура проекта**

Используем сгенерированный DDL код для создания структуры БД. В обозревателе серверов в контекстном меню для базы School выберем пункт «Новый запрос», перенесём в открывшееся окно содержимое файла SchoolModel.edmx.sql и нажмём на кнопку «Выполнить» в левом верхнем углу. Должно отобразиться сообщение «Команды успешно выполнены» (рис. 2.6).

  
**Рис. 2.6. Выполнение DDL кода**

Если после этого в обозревателе серверов обновить узел «Таблицы» базы данных School, то можно увидеть 4 созданные таблицы для хранения информации об учениках, преподавателях, уроках и связи между уроками и учениками. Заметим, что в случае если в БД уже имеются таблицы или другие объекты, которые необходимо создать заново, EDM генерирует код для их удаления, чтобы исключить ошибки при создании. При удалении могут пропасть важные данные, так что перед запуском автоматически сгенерированного кода его рекомендуется изучить.

Кроме этого, полученный с помощью конструктора EDM код может быть изменён вручную для указания более тонких настроек создания структуры БД. Добавим туда несколько выражений на языке манипулирования данными DML (Data Manipulation Language) для добавления начальной информации и выполним их:

-- Разрешаем указание идентификаторов при вставке строк оператором insert

SET IDENTITY\_INSERT [dbo].[TeacherSet] ON

-- Вставляем записи в таблицу Преподаватели

INSERT INTO [dbo].[TeacherSet](Id, FIO, SchoolSubject)

VALUES (1, 'Петров Сергей Андреевич', 'Информатика')

, (2, 'Копытов Иван Григорьевич', 'Английский язык')

, (3, 'Малолетов Андрей Сергеевич', 'Экономика')

, (4, 'Истомин Дмитрий Сергеевич', 'Информатика')

-- Снимаем разрешение на вставку идентификаторов

SET IDENTITY\_INSERT [dbo].[TeacherSet] OFF

-- Добавление учеников

SET IDENTITY\_INSERT [dbo].[LearnerSet] ON

INSERT INTO [dbo].[LearnerSet](Id, FIO)

VALUES (1, 'Кузнецов Иван Александрович')

, (2, 'Макарова Виктория Сергеевна')

, (3, 'Павлов Иван Юрьевич')

, (4, 'Сидоров Максим Валерьевич')

, (5, 'Маркова Ирина Витальевна')

SET IDENTITY\_INSERT [dbo].[LearnerSet] OFF

-- Добавление занятий

SET IDENTITY\_INSERT [dbo].[LessonSet] ON

INSERT INTO [dbo].[LessonSet] ([Id], [Theme], [StartDateTime], [Teacher\_Id])

VALUES (1, 'Введение в информатику', '2016-09-02 09:20:00', 1)

, (2, 'Школа', '2016-09-02 10:10:00', 2)

, (3, 'Как работает компьютер', '2016-09-09 09:20:00', 1)

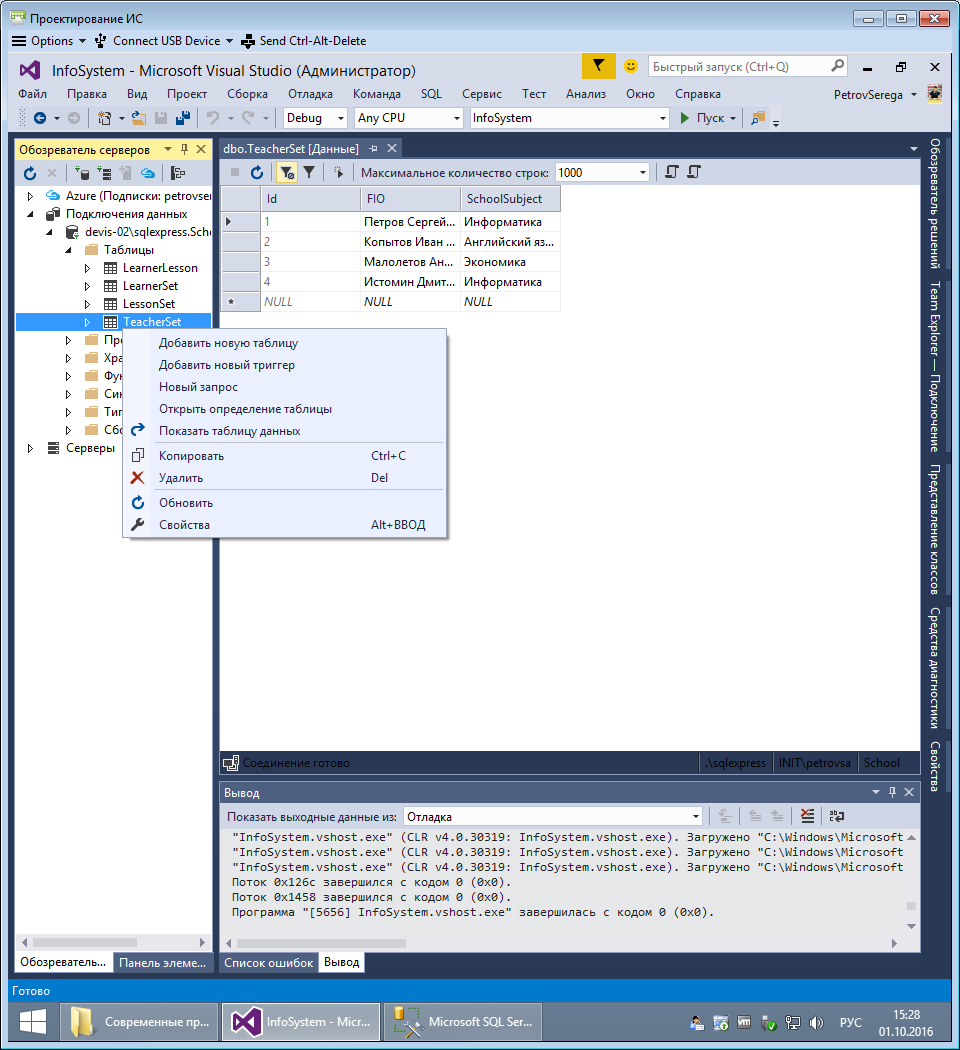
SET IDENTITY\_INSERT [dbo].[LessonSet] OFF

-- Добавление связей ученик-занятие

INSERT INTO [dbo].[LearnerLesson] ([Lesson\_Id], [Learner\_Id])

VALUES (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (3, 2), (3, 3), (3, 4), (3, 5)

Если из контекстного меню для таблицы вызвать команду «Показать таблицу данных», можно увидеть данные, которые она содержит (рис. 2.7). Например, таблица преподаватели содержит 3 записи:



**Рис. 2.7. Просмотр данных таблицы**

Окно «Данные» поддерживает ручное добавление записей, но рекомендуется добавлять новые записи с помощью команд, так как они могут быть сохранены в файл и повторно выполнены для другой БД с такой же структурой. Чтобы сгенерировать код добавления записей для текущей таблицы нажмите на иконку «скрипт» справа от выпадающего списка «Максимальное количество строк».

Теперь, когда мы создали структуру БД и добавили начальные данные, перейдём к реализации обработки данных в информационной системе.

## Контрольные вопросы и задания

1. С помощью конструктора EDM разработайте схему данных для предметной области, рассмотренной в рамках задания первого раздела.
2. С помощью платформы EntityFramework подготовьте скрипт и используйте его для создания структуры данных в MS SQL Server.
3. Наполните таблицы БД начальными данными.

# ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ ОБРАБОТКА ДАННЫХ С ИСПОЛЬЗОВАНИЕМ ENTITY FRAMEWORK

Entity Framework (EF) – технология для работы с реляционной базой данных с помощью объектов, входящих в состав платформы .NET. EF является промежуточным слоем между записями в БД и их обработкой в приложении. Подход с использованием промежуточного слоя называется объектно-реляционное отображение (ORM, Object-Relational Mapping). EF – реализация этого подхода компанией Microsoft.

Под объектно-ориентированной обработкой данных будем понимать изменения в БД посредством классов объектно-реляционного отображения.

Объектно-реляционное отображение позволяет повысить гибкость решения, так как различные схемы БД могут быть сопоставлены одинаковым классам и наоборот. В простых случаях при изменении схемы БД не потребуется перерабатывать всё приложение, необходимо поправить лишь сопоставление полей. Для программистов использование объектов C#, соответствующих таблицам БД, более удобно, чем составление DML запросов. Помимо удобства для работы с записями БД технология EF позволяет задействовать кэширование данных. При правильном использовании это может дать существенный прирост производительности при минимальных затратах со стороны разработчика.

## Классы Entity Framework

Для связи с БД Entity Framework использует шаблоны Text Template Transformation Toolkit (T4) для генерации специализированных классов, с которыми удобно работать разработчику на языке программирования C#. Шаблоны T4 используют собственный синтаксис и семантику для описания формата, согласно которому будет сгенерирован исходный код. Файлы шаблонов имеются расширение .tt.

При вызове мастера формирования базы данных в проект автоматически добавляются файлы шаблонов EntityFramework (в нашем случае это SchoolModel.Context.tt и SchoolModel.tt).

В случае, когда мастер не вызвался, например, если сначала планируется подготовить программный код для работы с базой и лишь потом создать её, шаблоны T4 для EntityFramework могут быть включены вручную. Для этого стоит добавить в проект элемент под названием EF 6.x DbContext Generator. После этого в обозревателе решений под файлом модели EF отобразятся добавленные файлы шаблонов.

Для нашей ИС классы EF для работы с БД были сгенерированы в предыдущем разделе с помощью конструктора Entity Data Model.

Основные классы, используемые EntityFramework, – DbContext и DbSet. Шаблоны кодогенерации работают таким образом, что для всех сущностей генерируются классы наследники DbSet, а потомок класса DbContext обеспечивает подключение к базе данных и предоставляет базовые методы для работы с ней (конфигурация подключения, выполнение запросов, отслеживание изменений, сохранение изменений и другие). Наследники DbContext получают базовый функционал и расширяют его за счёт добавления полей, предоставляющих доступ к таблицам конкретной БД, описанных в модели.

В нашем случае дочерним для DbContext является класс SchoolModelContainer, представленый в файле SchoolModel.Context.cs. Класс определяет конструктор для создания подключения к БД с использованием строки подключения SchoolModelContainer из файла app.config и три коллекции DbSet для каждой из таблиц: преподаватель, ученик, урок. Декларация DbSet параметризуется классом, описывающим столбцы конкретной таблицы.

public virtual DbSet<Teacher> TeacherSet { get; set; }

public virtual DbSet<Learner> LearnerSet { get; set; }

public virtual DbSet<Lesson> LessonSet { get; set; }

Ключевые слова get и set, применённые выше, можно назвать «синтаксическим сахаром» в языке C#. Они называются аксессорами и служат для быстрого описания закрытых полей, доступ к которым будет осуществляться через специальные методы. Аксессоры get и set позволяют задать такие методы в более компактной форме. Переменные класса, за которыми следуют get и/или set, называются свойствами. Мы будем работать с ними как с обычными полями.

Каждый их классов Teacher, Learner и Lesson представлен в отдельном файле. Они содержат свойства, соответствующие колонкам таблиц БД, и коллекции, ссылающиеся на связанные таблицы по внешнему ключу. Например, класс Урок выглядит следующим образом:

public partial class Lesson {

// Уникальный идентификатор записи

public int Id { get; set; }

// Тема занятия

public string Theme { get; set; }

// Дата занятия

public DateTime StartDateTime { get; set; }

// Ссылка на объект учитель

public virtual Teacher Teacher { get; set; }

// Коллекция ссылок на объекты ученик

public virtual ICollection<Learner> Learner { get; set; } }

Аналогичным образом выглядят и остальные классы, сгенерированные EF для работы с базой School. Воспользуемся ими для выполнения базовых операций обработки таблиц БД.

## Поддержка операций Create, Read, Update, Delete

Коллекции DbSet используются для реализации базовых операций обработки данных CRUD (create, read, update, delete). Реализуем операции CRUD для каждой из таблиц в отдельном классе: CRUDTeacher – преподаватели, CRUDLearner – ученики, CRUDLesson – уроки. Реализация этих классов схожа, поэтому рассмотрим пример одного класса CRUDTeacher. Затем рассмотрим реализацию операции Create для урока, как наиболее сложный случай.

Заранее стоит отметить, что представленный исходный код содержит ряд упрощений, которые могут ивести к дальнейшим ошибкам в программе. Например, мы уверены, что когда просим ввести пользователя число, он вводит именно число, а не произвольные символы; когда просим указать идентификатор записи, то пользователь укажет идентификатор существующей в БД записи, а не произвольный. Разрешение таких ситуаций остаётся читателям на самостоятельный разбор.

Итак, для вывода всех преподавателей из БД необходимо создать экземпляр класса SchoolModelContainer и обратиться к коллекции TeacherSet. Программный код программы, осуществляющий данные действия, выглядит следующим образом:

static void Main(string[] args) {

// Создание объекта для доступа к БД

SchoolModelContainer db = new SchoolModelContainer();

// Перебор записей из таблицы TeacherSet

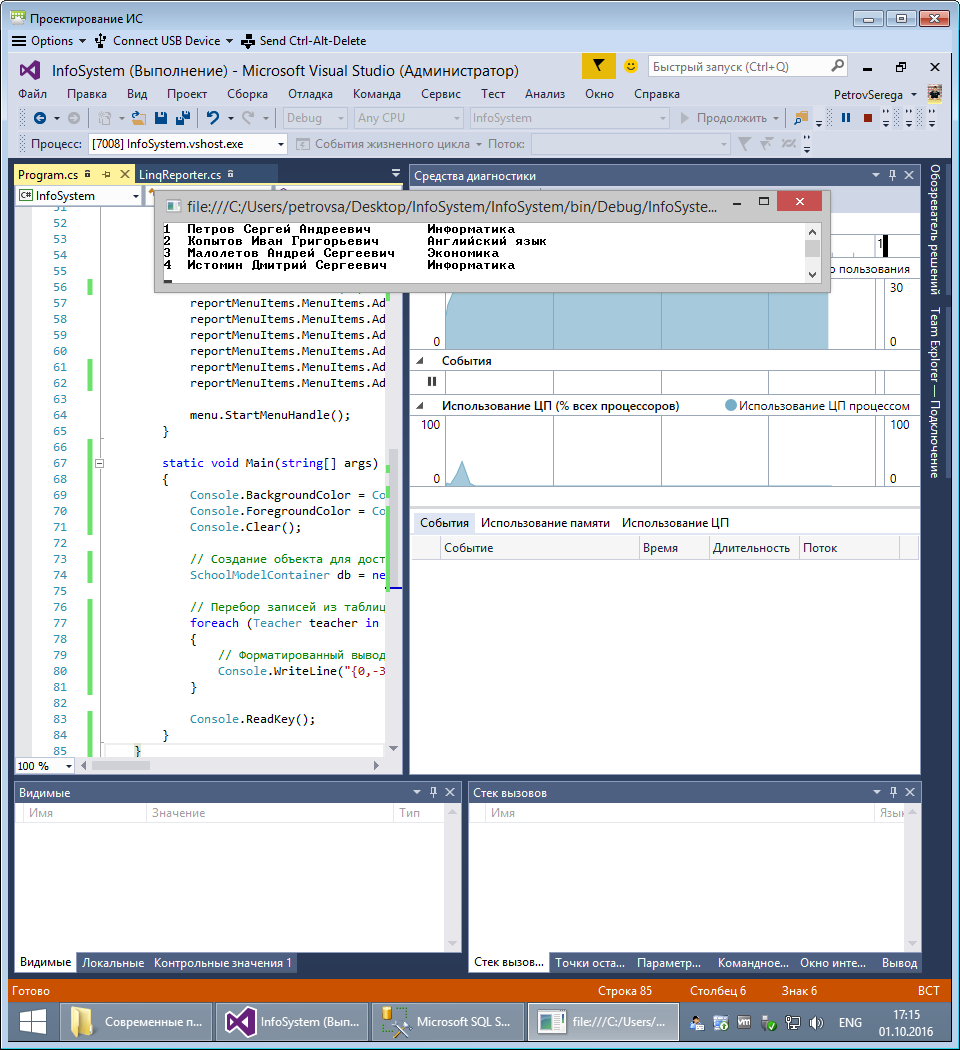
foreach (Teacher teacher in db.TeacherSet) {

// Форматированный вывод колонок каждой записи

Console.WriteLine("{0,-3}{1,-30}{2}", teacher.Id, teacher.FIO, teacher.SchoolSubject); }

Console.ReadKey();}

Результатом работы будет список преподавателей, добавленный в БД в предыдущем разделе:



**Рис. 3.1. Чтение данных из БД**

Перенесём код вывода преподавателей в класс CRUDTeacher. Объект этого класса будет содержать ссылку на объект класса SchoolModelContainer и использовать его для работы с БД. По логике нашей программы для её выполнения достаточно единственного экземпляра класса CRUDTeacher. Поэтому в описании класса добавим статическое поле, указывающее на единственный экземпляр этого класса:

public class CRUDTeacher {

// Ссылка на единственный экземпляр класса CRUDTeacher

public static CRUDTeacher Instance = null;

private SchoolModelContainer \_schoolDB = null;

public CRUDTeacher(SchoolModelContainer schoolDB) {

this.\_schoolDB = schoolDB; } }

Такой подход является частичной реализацией паттерна программирования «Одиночка». Подробно он разобран в [4]. Добавим в класс CRUDTeacher методы Read и Create:

// Создание новой записи

public void Create() {

Teacher newTeacher = new Teacher();

Console.Write("Введите ФИО нового преподавателя ");

newTeacher.FIO = Console.ReadLine();

Console.Write("Введите специализацию нового преподавателя ");

newTeacher.SchoolSubject = Console.ReadLine();

// Добавляем нового преподавателя в коллекцию DbSet

this.\_schoolDB.TeacherSet.Add(newTeacher);

// Переносим изменения из программного объекта БД в реальную БД

this.\_schoolDB.SaveChanges();}

// Чтение записей из БД

public void Read(){

Console.WriteLine("Список преподавателей:");

foreach (Teacher teacher in this.\_schoolDB.TeacherSet)

{

Console.WriteLine("{0,-3}{1,-30}{2}", teacher.Id, teacher.FIO, teacher.SchoolSubject);

} }

Метод Create позволяет добавить новую запись в БД. Для этого мы создали новый объект Teacher, попросили пользователя задать значения полей, добавили новый объект в DbSet и вызвали метод SaveChanges из класса DbContext для сохранения изменений в БД.

В момент сохранения, как и при любом другом обращении к объекту, подразумевающем связь с БД, библиотека Entity Framework формирует соответствующий SQL запрос и передаёт его СУБД. Получив результат запроса, Entity Framework отображает его и содержащиеся в нём объекты на DbContext, используя данные EDM модели. Детально принципы ORM библиотеки Entity Framework описаны в [5].

Чтобы реализовать методы обновления и удаления, пользователю необходимо выбрать запись, для которой будет выполнена команда. Для однозначности выбора будем использовать уникальное на уровне таблицы свойство Id. Оно было добавлено по умолчанию конструктором EDM и присутствует в каждой из таблиц. Метод Delete выглядит следующим образом:

public void Delete() {

Console.Write("Введите ИД преподавателя, которого необходимо удалить ");

// Получение строки, введённой пользователем

string userInput = Console.ReadLine();

// Преобразование строки к целочисленному типу данных

int teacherID = int.Parse(userInput);

// Поиск преподавателя по ИД

Teacher teacher = this.\_schoolDB.TeacherSet.Single(i => i.Id == teacherID);

// Удаление объекта из коллекции

this.\_schoolDB.TeacherSet.Remove(teacher);

this.\_schoolDB.SaveChanges(); }

Метод Update получается за счёт соединения части исходного кода для указания значений свойств объекта из метода Create и части для получения объекта по Id из метода Delete:

public void Update() {

Console.Write("Введите ИД преподавателя, которого необходимо обновить ");

int teacherID = int.Parse(Console.ReadLine());

Teacher teacher = this.\_schoolDB.TeacherSet.Single(i => i.Id == teacherID);

Console.Write("Введите новое ФИО преподавателя ");

teacher.FIO = Console.ReadLine();

Console.Write("Введите новую специализацию преподавателя");

teacher.SchoolSubject = Console.ReadLine();

this.\_schoolDB.SaveChanges(); }

В реальных системах необходимо предусмотреть возможность редактирования значения одного поля и не требовать от пользователя повторного ввода значений полей, которые не следует изменять. Это можно реализовать, предложив пользователю выбрать поле для изменения или применив специальный символ для указания того факта, что программе необходимо пропустить текущее поле при обновлении и перейти к следующему.

У нас получился класс CRUDTeacher, предоставляющий основные методы для работы с таблицей Teacher. Большинство методов CRUD для классов CRUDLearner и CRUDLesson реализуются аналогичным образом.

Остановимся на рассмотрении метода создания уроков из класса CRUDLearner, так как он является центральным для нашей системы. Его отличие от ранее реализованного класса заключается в необходимости указать преподавателя, проводившего занятие, и учеников, посетивших его. При запросе идентификатора преподавателя напомним пользователю, какие в базе присутствуют преподаватели. Для этого выведем их список с помощью метода Read класса CRUDTeacher. Аналогично поступим при запросе идентификаторов учеников.

public void Create(){

Lesson newLesson = new Lesson();

Console.Write("Введите тему занятия ");

newLesson.Theme = Console.ReadLine();

Console.Write("Введите дату занятия в формате дд.мм.гг чч:мм ");

string userInput = Console.ReadLine();

newLesson.StartDateTime = DateTime.Parse(userInput);

Console.Write("Укажите ИД преподавателя, проводившего занятие");

CRUDTeacher.Instance.Read();

int teacherID = int.Parse(Console.ReadLine());

newLesson.Teacher = this.\_schoolDB.TeacherSet.Single(i => i.Id == teacherID);

Console.Write("Укажите ИД учеников, посетивших занятие");

CRUDLearner.Instance.Read();

// Разбиваем строку, введённую пользователем, на массив строк с помощью символа-разделителя: пробела

string[] strLearnerIDs = Console.ReadLine().Split(new[] { ' ' });

// Для каждого идентификатора находим ученика и добавляем его к занятию

foreach(string strLearnerID in strLearnerIDs)

{

int learnerID = int.Parse(strLearnerID);

Learner learner = this.\_schoolDB.LearnerSet.Single(i => i.Id == learnerID);

newLesson.Learner.Add(learner);

}

this.\_schoolDB.LessonSet.Add(newLesson);

this.\_schoolDB.SaveChanges();

}

В данном примере для работы со связными таблицами мы использовали обращение к полям, сгенерированным EDM конструктором. Это поле класса Lesson под названием Teacher, указывающее на запись из связанной таблицы преподавателей, и поле Learner, указывающее на множество связанных записей таблицы учеников через промежуточную таблицу LearnerLesson. Работа с этими вспомогательными полями упрощает разработку, так как для обращения к связанным записям нет необходимости делать запрос к другим DbSet-ам. Все данные можно получить через имеющийся объект. Конечно, сам запрос к базе происходит, но исходный код на языке C# получается короче и понятнее.

Например, для вывода учеников, посетивших занятие, не нужно обращаться сначала к таблице LearnerLesson, а затем к LearnerSet. Все данные получаются через поля классов Entity Framework: объект Lesson → коллекция Learner.

public void Read() {

Console.WriteLine("Список всех занятий:");

foreach (Lesson lesson in this.\_schoolDB.LessonSet) {

Console.WriteLine("{0,-3}{1,-20}{2,-25}{3}", lesson.Id, lesson.StartDateTime, lesson.Theme, lesson.Teacher.FIO);

// Если на урок никто не пришел, выводить нечего, поэтому перейдём к следующему занятию

if (lesson.Learner.Count == 0)

continue;

Console.WriteLine("Ученики, посетившие занятие:");

foreach (Learner learner in lesson.Learner) {

Console.WriteLine("{0,-3}{1,-30}", learner.Id, learner.FIO);

} } }

Теперь, когда реализованы классы для работы с таблицами БД, необходимо организовать доступ пользователя к этим возможностям. Воспользуемся для этой цели библиотекой классов, упрощающей создание консольного меню, реализация которой описана в другом пособии автора [6]. После добавления в проект соответствующей сборки, можно реализовать собственные элементы меню через потомков класса MenuItem. Для этого необходимо создать класс-наследник MenuItem и определить метод Action, который выполняется при выборе пользователем соответствующего пункта меню. Конструктор этого класса будет принимать два аргумента – заголовок, отображаемый пользователю, и текстовую строку (например, “1”), которую пользователь должен ввести для запуска метода Action. Для удобства навигации разложим элементы меню, относящиеся к каждой из таблиц в различные папки. Реализация всех элементов меню однотипна. Например, файл CreateMenuItem.cs для добавления нового преподавателя (папка TeacherMenu) выглядит следующем образом:

using ConsoleMenu;

using InfoSystem.CRUD;

using System;

namespace InfoSystem.TeacherMenu {

// Наследование возможностей базового элемента меню – MenuItem

public class CreateMenuItem : MenuItem

{

// Конструктор

public CreateMenuItem(string title, string activationComand) : base(title, activationComand) {}

// Переопределение метода Action

public override void Action() {

// Вызов метода для добавления нового преподавателя

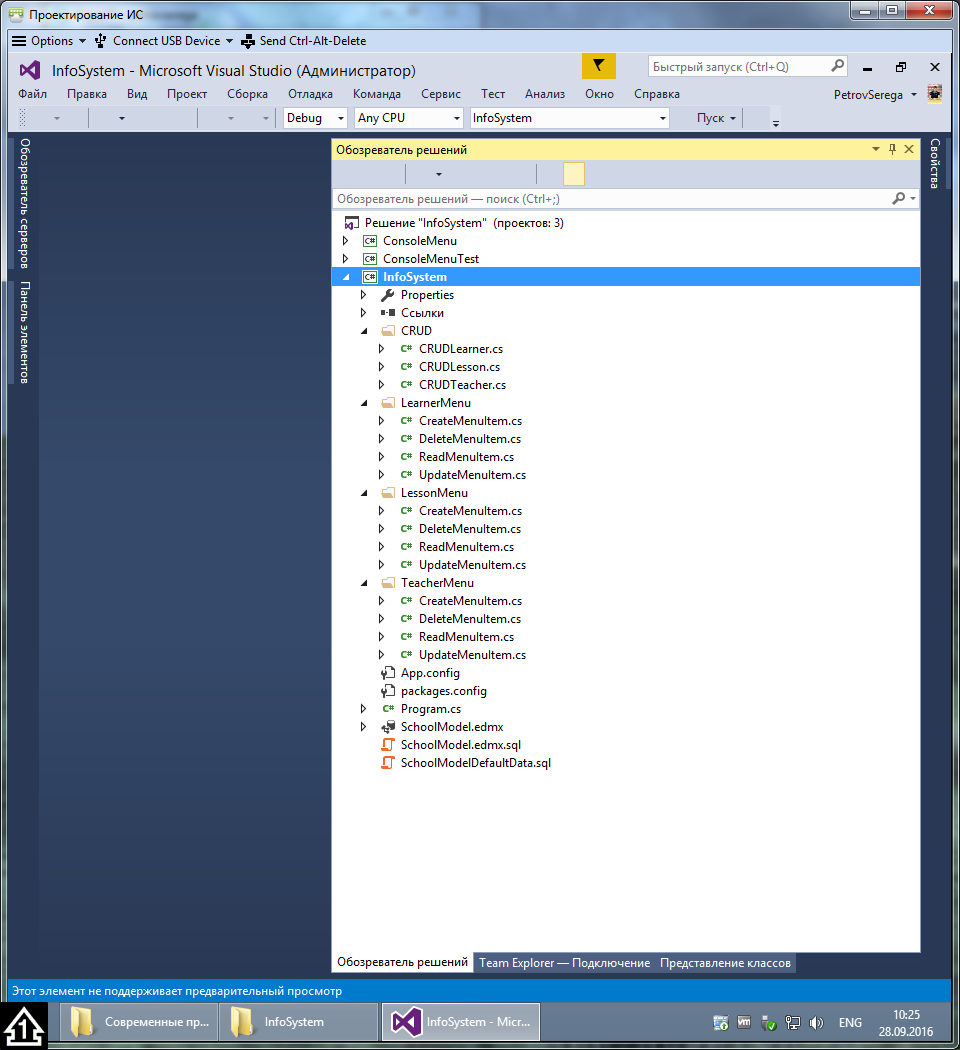
CRUDTeacher.Instance.Create();

Console.WriteLine("Нажмите любую клавишу для продолжения...");

Console.ReadKey();

} } }

Все остальные элементы меню для разных таблиц и для разных действий отличаются лишь пространством имён (TeacherMenu, LearnerMenu или LessonMenu), названием класса (CreateMenuItem, ReadMenuItem, UpdateMenuItem или DeleteMenuItem), классом для работы с БД (CRUDTeacher, CRUDLearner или CRUDLesson) и вызываемым методом (Create, Read, Update или Delete). Получившаяся структура проекта выглядит следующим образом:



**Рис. 3.2. Структура проекта InfoSystem**

Итоговый вариант метода Main для запуска консольного интерфейса разрабатываемой ИС для работы с преподавателями следующий:

static void Main(string[] args) {

SchoolModelContainer schoolDB = new SchoolModelContainer();

// Создание экземпляра класса, предоставляющего доступ к методам CRUD, для таблицы преподавателей

CRUDTeacher.Instance = new CRUDTeacher(schoolDB);

Menu menu = new Menu();

// Наполнение меню для работы с таблицей преподавателей

HierarhicalMenuItem teachersMenuItems = new HierarhicalMenuItem("Преподаватели", "1");

menu.MenuItems.Add(teachersMenuItems);

teachersMenuItems.MenuItems.Add(new TeacherMenu.CreateMenuItem("Добавить нового преподавателя", "1"));

teachersMenuItems.MenuItems.Add(new TeacherMenu.ReadMenuItem("Вывести список всех преподавателей", "2"));

teachersMenuItems.MenuItems.Add(new TeacherMenu.UpdateMenuItem("Обновить преподавателя", "3"));

teachersMenuItems.MenuItems.Add(new TeacherMenu.DeleteMenuItem("Удалить преподавателя", "4"));

// Наполнение меню другими операциями

//…

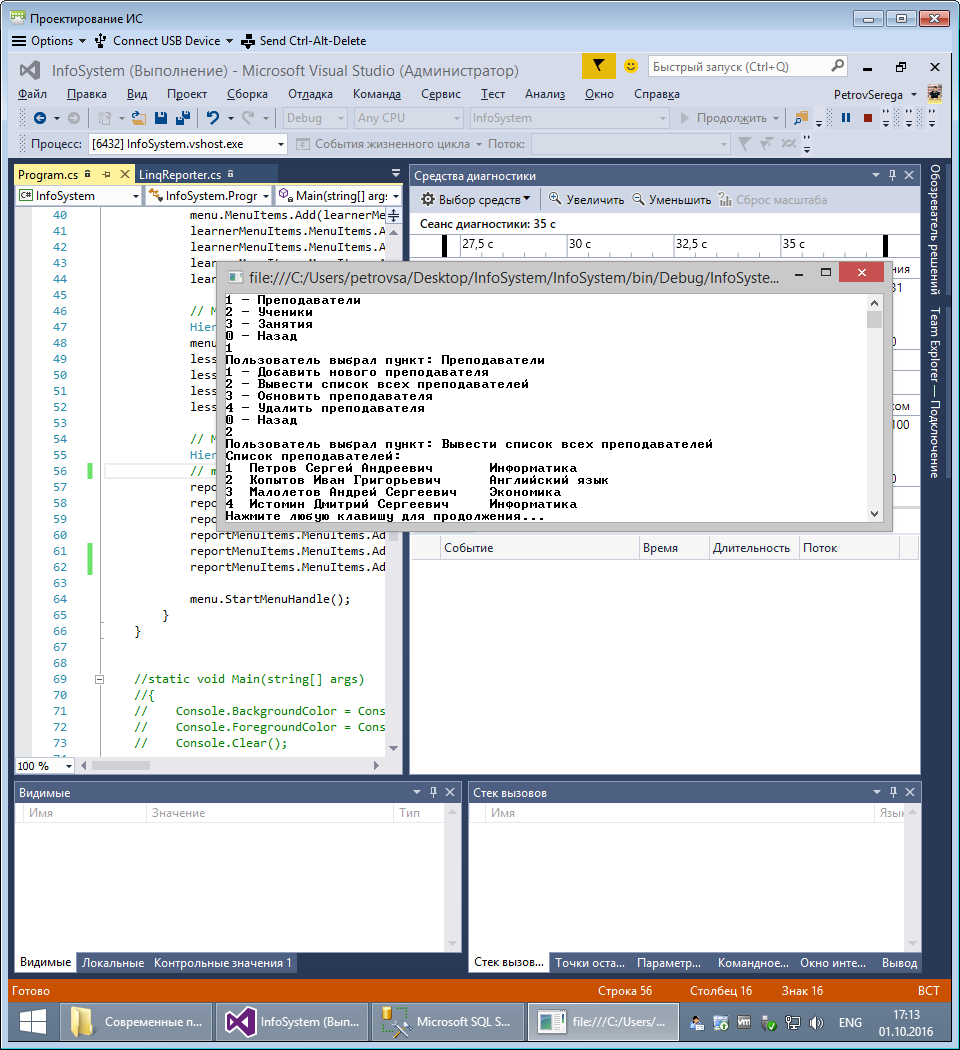
// «Запуск» меню

menu.StartMenuHandle();

}

После запуска проекта отобразятся верхние уровни меню, соответствующие таблицам в БД. Для изменения информации в ИС пользователю необходимо выбрать таблицу (первый уровень меню), которую он хочет изменить, и один из вариантов изменения (второй уровень меню). Для навигации предусмотрены элементы меню «Назад», позволяющие «подняться» на уровень вверх, или, если текущий уровень меню самый верхний, закончить работу с системой.

Получился простой для понимания, но, в то же время, функциональный интерфейс работы с системой. Чтобы получить список преподавателей, представленных в системе, необходимо выбрать пункт «Преподаватели», нажав 1, а затем пункт «Вывести список всех преподавателей», нажав 2 (рис. 3.3).



**Рис. 3.3. Пример работы InfoSystem**

В текущей реализации по мере работы с системой могут возникать ошибки при удалении записей из-за связи с другими таблицами. Например, согласно нашей модели, занятие обязательно должно быть проведено преподавателем. То есть запись, соответствующая занятию, должна ссылаться на существующего преподавателя через поле Teacher\_Id, поэтому удаление преподавателя, для которого заведены занятия, вызовет ошибку. Для решения следует предусмотреть удаление связанных занятий, либо запрет на удаление преподавателей, имеющих проведённые занятия. Кроме этого, любое действие, которое не может быть отменено (а удаление является именно таким), должно быть дополнительно подтверждено пользователем, иначе это может привести к потере важных данных. Доработку и развитие системы, в том числе для решения обозначенных проблем, оставим читателю для самостоятельной работы.

В этом разделе описана реализация операций для обработки записей в БД и приведён пример использования разработанного ранее консольного меню для доступа к новому функционалу. Перейдём к реализации запросов к ИС и для этого познакомимся с библиотекой LINQ.

## Контрольные вопросы и задания

1. С помощью сгенерированных EntityFramework классов наследников DbContext и DbSet, реализуйте операции создания, чтения, обновления и удаления для одной из таблиц.
2. Проверьте корректность вводимых пользователем данных с помощью метода TryParse.
3. Для отношения «один ко многим» реализуйте удаление связных записей при удалении экземпляра со стороны «один».

# ВВЕДЕНИЕ В LINQ

Language Integrated Query (LINQ) – технология, позволяющая выполнять запросы к источникам данных, используя языки программирования платформы .NET Framework и синтаксис, похожий на язык запросов SQL. Впервые LINQ появился в Visual Studio 2008 и, постепенно развиваясь, поддерживал всё новые источники данных. На текущий момент, в зависимости от вида источника данных, выделяют несколько разновидностей LINQ:

* LINQ to Objects: запросы к коллекциям;
* LINQ to Entities: запросы к БД через Entity Framework;
* LINQ to Sql: запросы к MS SQL Server;
* LINQ to XML: запросы к XML файлам;
* LINQ to DataSet: запросы к объекту DataSet (не путать с DbSet, используемый EF);
* Parallel LINQ (PLINQ): выполнение параллельных запросов.

Синтаксис и семантика всех разновидностей достаточно схожи, но есть некоторая специфика в технической реализации получения данных. Нас будет интересовать технология LINQ to Entities, которая дополняет библиотеку Entity Framework и, используя EDM-отображение, позволяет выполнить запросы к MS SQL. Далее, говоря о LINQ, будем подразумевать разновидность LINQ to Entities.

## Поиск единственной записи – методы Single, SingleOrDefault, First, FirstOrDefault

Ранее мы уже использовали синтаксис LINQ для получения записи по её идентификатору:

Teacher teacher = this.\_schoolDB.TeacherSet.Single(i => i.Id == teacherID);

В данном примере TeacherSet определяет источник данных, то есть экземпляр класса DbSet, teacher – переменную, которая будет указывать на результат, а метод Single – запрос для получения данных.

Запись «i => i.Id == teacherID» называется лямбда-выражением и используется для лаконичного описания локальных функций, которые могут быть переданы в качестве аргументов. Лямбда-выражение разделяется на две части – до и после лямбда-оператора "=>". С левой стороны указываются входные параметры, а с правой – сама функция. Аргументом метода Single является лямбда-выражение для проверки соответствия текущего элемента коллекции искомому элементу по каким-либо признакам. Переменная i ссылается на проверяемого в данный момент преподавателя. Функция в правой части имеет логический тип возвращаемого значения, который равен true, если идентификатор преподавателя i соответствует значению переменной teacherID, и false в противном случае.

В случае если ни один из элементов коллекции не удовлетворяет критериям поиска, то произойдёт исключительная ситуация и программа завершится ошибкой. Схожий метод, не вызывающий исключения – SingleOrDefault – вернёт null, если подходящий объект не найден.

Другая возможная проблема заключается в том, что метод Single проверяет единственность записи, подходящей по критериям выбора. В нашем случае был использован выбор по идентификатору, уникальность которого контролируется на уровне СУБД, поэтому ошибки не будет. Но если использовать более мягкие критерии, например, превышение порогового значения, то результатом может быть ошибка, возникающая из-за наличия нескольких записей, подходящих по критериям. Для выхода из этой ситуации можно использовать методы First или FirstOrDefault, возвращающие первый элемент, соответствующий условиям.

Мы уже познакомились с четырьмя методами LINQ, возвращающими один единственный элемент.

Ещё два метода, возвращающие единственное значение, – это методы Any и All. Результат их работы true (правда) или false (ложь). Метод All проверяет, что каждый элемент выборки соответствует условию, а метод Any – что, хотя бы один.

Для организации LINQ-запросов к нашей БД создадим класс LinqReporter, принимающий в качестве аргумента конструктора экземпляр SchoolModelContainer для работы с БД. Логика его работы соответствует классам CRUD, а каждый из методов оформляется в элемент меню аналогично методам изменения данных в таблицах.

Определим, существует ли в системе хотя бы один преподаватель, который ведёт информатику. Для этого используем метод Any:

public void CheckComputerScienceTeacher()

{

bool checkResult = this.\_schoolDB.TeacherSet.Any(i => i.SchoolSubject == "Информатика");

Console.WriteLine("Результат проверки: {0}", checkResult);

}

Метод All используем для проверки корректности дат проведения занятий – ни одно занятие не может быть проведено в период летних каникул с июня по август:

public void CheckNoLessonInSummerHolidays(){

// Месяц проведения каждого занятия должно быть больше 8-го и меньше 6-го

bool checkResult = this.\_schoolDB.LessonSet

.All(i => i.StartDateTime.Month > 8

|| i.StartDateTime.Month < 6);

Console.WriteLine("Результат проверки: {0}", checkResult);

}

Какой будет результат работы этих функций, если коллекция окажется пустой? Для метода Any ответ логичный – false, а вот для метода All – не совсем. Согласно официальной документации, в случае вызова метода All для пустой коллекции, результатом её работы будет true.

## Выборка нескольких значений – методы Where, OrderBy, GroupBy, Join

Для фильтрации коллекции используется метод where. В отличие от ранее рассмотренных методов, он возвращает не единственное значение, а коллекцию объектов, удовлетворяющих условию.

Например, запрос для получения всех учеников по имени «Иван» выглядит следующим образом:

public void SelectLearnersByName(){

IQueryable<Learner> learners = this.\_schoolDB.LearnerSet.Where(i => i.FIO.Contains("Иван"));

foreach (Learner learner in learners)

Console.WriteLine("{0} {1}", learner.Id, learner.FIO);

}

Работа с результатом запроса выглядит обычным образом – мы перебираем элементы коллекции и выводим информацию об отдельных экземплярах. Корректней сказать, что данный запрос возвращает всех учеников, у которых значение поля FIO содержит слово «Иван», так как в результат попадут все «Ивановы», «Ивановичи» и т.д. Для проверки одного имени необходимо отвести для него отдельную колонку или договориться о формате разделения фамилии, имени и отчества в поле FIO.

Результатом метода Where является экземпляр, поддерживающий интерфейс IQueryable. Этот интерфейс представляет собой источник данных, к которому может быть выполнен запрос.

В примере, приведённом выше, запрос выполняется лишь в строке с ключевым словом foreach, то есть при первом обращении к результирующей коллекции. Это можно проверить следующим образом:

public void SelectLearnersByName2()

{

IQueryable<Learner> learners = this.\_schoolDB.LearnerSet.Where(i => i.FIO.Contains("Иван"));

// Изменяем коллекцию, к которой выполняется запрос, после создания запроса

this.\_schoolDB.LearnerSet.Add(new Learner() { FIO = "Новый Иван" });

this.\_schoolDB.SaveChanges();

foreach (Learner learner in learners)

Console.WriteLine("{0} {1}", learner.Id, learner.FIO);

}

После создания запроса к БД добавляются новые элементы таблицы. При выводе результатов запроса видно, что в них попал и новый элемент, то есть запрос был выполнен после изменения таблицы. Таким образом, важно понимать, что по умолчанию запрос LINQ выполняется только в момент первого обращения к результирующей коллекции. Чтобы заставить IQueryable выполнить запрос и получить его результат, необходимо применить один из методов ToArray или ToList:

public void SelectLearnersByName3()

{

Learner[] learners = this.\_schoolDB.LearnerSet.Where(i => i.FIO.Contains("Иван")).ToArray();

// Изменяем коллекцию, к которой выполняется запрос, после создания запроса

this.\_schoolDB.LearnerSet.Add(new Learner() { FIO = "Новый Иван" });

this.\_schoolDB.SaveChanges();

foreach (Learner learner in learners)

Console.WriteLine("{0} {1}", learner.Id, learner.FIO);

}

В этом случае в результат попадут записи только из начальной коллекции объектов, так как вызов ToArray создаст новый массив учеников, а метод Add будет изменять исходную коллекцию.

Если нет необходимости выполнять запрос сразу, то после создания экземпляра, реализующего интерфейс IQueryable, его можно изменять. Например, добавлять новые условия выбора или сортировку. Вот так можно сформировать запрос для получения всех преподавателей информатики, отсортированных по значению поля FIO:

public void SelectComputerScienceTeacherOrderByFIO()

{

IQueryable<Teacher> teachers = this.\_schoolDB.TeacherSet.Where(i => i.SchoolSubject == "Информатика");

// Изменяем исходный запрос

teachers = teachers.OrderBy(i => i.FIO);

Teacher[] queryResult = teachers.ToArray();

foreach (Teacher teacher in queryResult)

Console.WriteLine("{0} {1}", teacher.Id, teacher.FIO);

}

Методы OrderBy и OrderByDescending позволяют отсортировать исходную коллекцию в порядке возрастания или убывания значения правой части лямба-выражения.

В библиотеке LINQ реализован подход, который называется текучий интерфейс (fluent interface). Он позволяет вызвать цепочку методов для одного объекта. Для этого необходимо, чтобы каждый метод возвращал тип, соответствующий типу исходного объекта. В случае LINQ, объектом является источник данных, а типом – IQueryable. Запись вышеуказанного запроса можно сократить следующим образом:

IQueryable<Teacher> teachers = this.\_schoolDB.TeacherSet

.Where(i => i.SchoolSubject == "Информатика")

.OrderBy(i => i.FIO);

Способ записи LINQ-запросов с использованием методов называется Method Syntax (синтаксис методов). Вместе с реализацией текучего интерфейса они представляют собой мощное средство для выборки данных с помощью библиотеки LINQ. Однако не всегда сформированный таким образом запрос понятен и лёгок для понимания. Поэтому существует и другой способ записи запросов. Он использует ряд ключевых слов, которые были специально добавлены в язык C#. Запрос для получения отсортированного списка преподавателей информатики можно переписать следующим образом:

IQueryable<Teacher> teachers = (from i in this.\_schoolDB.TeacherSet

where i.SchoolSubject == "Информатика"

orderby i.FIO

select i);

Такой способ записи называется Query Syntax (синтаксис запросов) и он максимально напоминает запросы на языке SQL. Сложно сказать какой из подходов записи LINQ-запросов лучше. Необходимо ориентироваться на собственные предпочтения и стандарты, принятые в организации. Часто бывает удобно совмещать оба подхода в одном запросе.

Помимо простых запросов для выборки данных, удовлетворяющим заданным критериям, существуют и более сложные – для группировки и вычисления сводных значений по группе. Для примера посчитаем количество преподавателей по каждой дисциплине:

public void SelectTeacherCountForSubject()

{

IGrouping<string, Teacher>[] group = this.\_schoolDB.TeacherSet

// Группируем преподавателей по школьному предмету

.GroupBy(t => t.SchoolSubject)

// Сортируем в порядке убывания кол-ва преподавателей

.OrderByDescending(i => i.Count())

.ToArray();

foreach (IGrouping<string, Teacher> groupItem in group)

Console.WriteLine("{0} - {1}", groupItem.Key, groupItem.Count());

}

В данном случае результатом запроса будет массив записей, каждая из которых описывает группу: ключ группы – это название предмета, а значение – преподаватель, ведущий предмет. Запрос записан с использованием Method Syntax, но может быть переписан и через Query Syntax. Результат верен, но можно обратить внимание, что программе требуется дважды считать количество занятий для преподавателя. Первый раз для сортировки, а второй – для вывода на экран.

Перед тем как поправить эту ситуацию, необходимо познакомиться с ключевым словом var и анонимными типами.

Раньше при объявлении переменных мы всегда явно указывали их тип (explicitly type). Например, int или IGrouping<string, Teacher>[]. Эта информация использовалась компилятором при сборке программы. Но компилятор, в большинстве случаев, сам в состоянии определить тип переменной. Для этого он использует выражение из правой части. Чтобы задать тип переменной неявно (implicitly type), необходимо вместо типа переменной указать ключевое слово var:

// Неявно задаём тип переменной group

var group = this.\_schoolDB.TeacherSet

.GroupBy(t => t.SchoolSubject)

.OrderByDescending(i => i.Count())

.ToArray();

// Неявно задаём тип переменной groupItem

foreach (var groupItem in group)

Console.WriteLine("{0}-{1}",groupItem.Key, groupItem.Count());

В обоих случаях тип переменных указан неявно и вычислен компилятором автоматически. Замена типа на ключевое слово var сокращает размер исходного кода программы, но иногда он становится сложнее для понимания. Поэтому решение о способе указания типа остаётся на усмотрение разработчика.

Однако есть ситуации, когда без var не обойтись. Они возникают, когда тип (класс) переменной определяется прямо в момент объявления переменной, то есть без предварительного описания класса. В результате такого объявления получается анонимный тип.

Например, можно объявить переменную анонимного типа, имеющую два поля: название школьного предмета и количество преподавателей:

// Объявление анонимного типа

var schoolSubjectWithTeacherCount = new { SchoolSubject = "Астрономия", TeacherCount = 0 };

// Обращение к полю анонимного типа

Console.WriteLine(schoolSubjectWithTeacherCount.SchoolSubject);

При объявлении переменной анонимного типа необходимо указывать её тип как var. В нашем запросе логично использовать анонимный тип, так как вполне вероятно, что больше в программе нигде такое сочетание полей (класс) не потребуется. Поля анонимного типа доступны только для чтения и их значения задаются прямо в момент объявления переменной. Обращаться к значению поля анонимного типа можно так же, как и к полю любого класса.

Используем анонимный тип, чтобы при запросе количества преподавателей по каждому из предметов метод Count вызывался единственный раз:

// Неявно задаём тип переменной group

var group = this.\_schoolDB.TeacherSet

.GroupBy(t => t.SchoolSubject)

.Select(t => new { SchoolSubject = t.Key, TeacherCount = t.Count() })

.OrderByDescending(i => i.TeacherCount)

.ToArray();

// Неявно задаём тип переменной groupItem

foreach (var groupItem in group)

Console.WriteLine("{0} - {1}", groupItem.SchoolSubject, groupItem.TeacherCount);

Анонимные типы часто используются в контексте LINQ-запросов для указания результата запроса, так как для каждого результата описывать отдельный класс слишком трудозатратно.

Последний пример, который мы рассмотрим, связан с объединением нескольких таблиц.

Предположим, что задача следующая – необходимо вывести на экран даты всех встреч каждого из преподавателей с каждым из учеников. Если бы мы писали запрос на языке SQL, нам потребовалось объединить четыре таблицы с помощью оператора JOIN. С Entity Framework задачу можно решить, использую сгенерированные поля для доступа к связанным коллекциям. Но для примера составим запрос с применением ключевого слова join:

public void SelectAllMeetings()

{

var meetings = (from teacher in this.\_schoolDB.TeacherSet

// Присоединяем таблицу уроков по внешнему ключу

join lesson in this.\_schoolDB.LessonSet on teacher.Id equals lesson.Teacher.Id

// Присоединяем таблицу учеников

from learner in this.\_schoolDB.LearnerSet

// Накладываем ограничение на соединение с таблицей учеников – ученики должны быть связаны с уроком

where lesson.Learner.Contains(learner)

// Формируем строки результата запроса

select new { Teacher = teacher.FIO, DateTime = lesson.StartDateTime, Learner = learner.FIO }).ToArray();

foreach (var meeting in meetings)

Console.WriteLine("{0} - {1} - {2}", meeting.DateTime, meeting.Teacher, meeting.Learner);

}

Запрос начинается с обращения к таблице преподавателей. Затем с помощью внешнего ключа присоединяется таблица занятий. После ключевого слова on указывается идентификатор записи преподаватель, а после equals – внешний ключ, соответствующей идентификатору преподавателя для записи занятия. Таблица с учениками присоединяется не по внешнему ключу, так как она имеет связь с занятием «многие-ко-многим». Для добавления строк занятий к запросу добавляется дополнительная таблица с помощью оператора from и налагается условие, указывающее, что занятия должны присоединяться к ученику, только если он связан с этим занятием. Наконец, формируется результат, каждая строка которого описывается на основании объединения трёх таблиц и включает ФИО преподавателя, дату занятия и ФИО ученика.

Запрос можно реализовать и меньшими усилиями, задействовав связанные коллекции, сгенерированные EntityFramework:

var meetings = (from teacher in this.\_schoolDB.TeacherSet

from lesson in teacher.Lesson

from learner in lesson.Learner

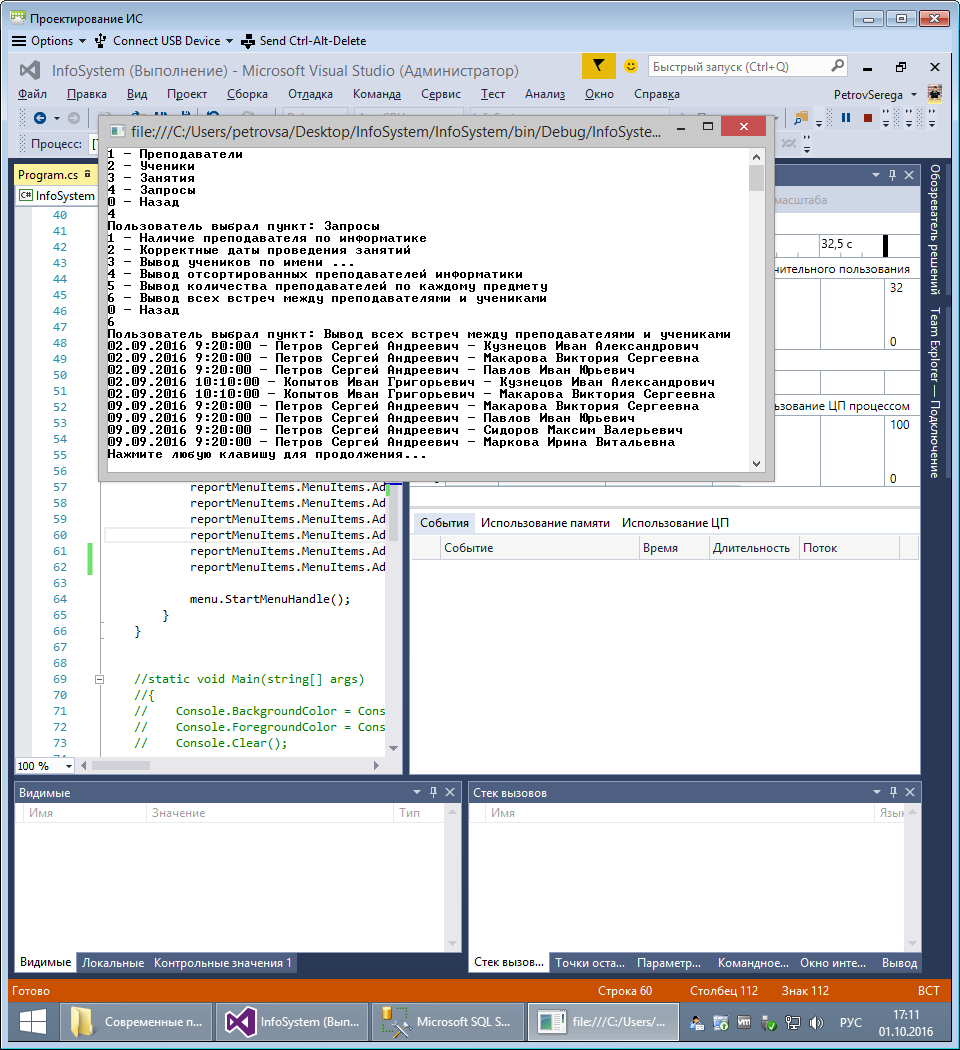
select new { Teacher = teacher.FIO, DateTime = lesson.StartDateTime, Learner = learner.FIO }).ToArray();

Результат обоих запросов одинаковый.

LINQ-запросы основываются на многих современных возможностях языка C# и во многом соединяют их. Например, неявную типизацию и анонимные типы. Тема является очень обширной, а в этом разделе хотелось продемонстрировать несколько практических задач и их решений. Для более глубокого погружения в библиотеку LINQ, развития навыков написания запросов и применения возможностей, не рассмотренных в данном пособии, можно использовать материалы [2,7].

Для включения в ИС всех рассмотренных LINQ-запросов каждый из них должен быть оформлен в отдельный элемент меню (MenuItem), а в само меню добавлен новый иерархический раздел «Запросы». Аналогичные действия были рассмотрены при реализации CRUD операций, поэтому здесь опустим их описание.

В качестве завершения раздела приведём экранный снимок ИС при доступе пользователя к последнему реализованному нами запросу – получение данных обо всех личных встречах (рис. 4.1).



**Рис. 4.1. Вывод всех встреч преподавателей и учеников**

## Контрольные вопросы и задания

1. Придумайте и реализуйте по одному запросу на каждый из методов: SingleOrDefault, FirstOrDefault, Any, All, Where, OrderBy, GroupBy, Join.

# Заключение

Разработанное решение может служить прототипом для более функциональной информационной системы. Представленный подход является гибким и легко адаптируется под другую предметную область. В ходе реализации использованы технологии ADO.NET, Entity Data Model, EntityFramework, система управления базами данных MS SQL Server и язык запросов LINQ.

# СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Стандарт ISO/IEC 2382:2015.
2. Дейт К. Дж. Введение в системы баз данных – 8-е изд. – М.: Вильямс, 2008. – 1328 с. – ISBN 978-5-8459-0788-2.
3. Загрузка СУБД MS SQL Express: <https://www.microsoft.com/en-us/cloud-platform/sql-server> .
4. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования = Design Patterns: Elements of Reusable Object-Oriented Software. - СПб: [«Питер»](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D1%82%D0%B5%D1%80_%28%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE%29), 2007. – С. 366. ISBN 5-272-00355-1.
5. Julia Lerman. Programming Entity Framework, Second Edition. - Published by O’Reilly Media, 2010, - С. 914. ISBN: 978-0-596-80726-9.
6. Петров, С.А. Техническое обеспечение информационных систем: введение в разработку на Visual Studio C#: учеб. пособие / С.А. Петров. – М.: Издательство МЭИ, 2017. – 40 с. ISBN 978-5-7046-1839-3
7. LINQ. Карманный справочник: Пер. С англ. / Дж. Албахари, Б. Албахари. – СПб.: БХВ-Петербург, 2009. – 240 с.: ил. ISBN 978-5-9775-0317-4.