

Что то...

И.Н. Блок, ст. преподаватель каф. информатики

9 октября 2018 г.

Оглавление

Список изменений	3
Введение	6
1 Entity Framework	7
1.1 Общие сведения	7
1.2 Описание модели данных (Database first)	8
1.3 Описание модели данных (Code first)	8
1.3.1 Указание связей БД	10
1.3.2 Ограничения целостности	14
1.3.3 Составные ключи	16
1.3.4 Инициализатор базы данных	17
1.4 Общие принципы работы с EF	19
1.4.1 Подключение EF к проекту, настройка подключения	19
1.4.2 Создание и использование контекста базы данных	22
1.4.3 Получение данных	23
1.4.4 Добавление/редактирование данных	24
1.4.5 Удаление данных	26
2 LINQ	28
2.1 Основные типы запросов Linq	28
2.1.1 Общие сведения	28
2.1.2 Where	29
2.1.3 Select	30
2.1.4 SelectMany	30
2.1.5 OrderBy (ThenBy)	31
2.1.6 FirstOrDefault (LastOrDefault)	31
2.1.7 First, Last	31
2.1.8 Union	32
2.1.9 Intersect	32
2.1.10 Except	33
2.1.11 Distinct	33
2.2 LINQ to Entities	34
3 WPF (сделано частично)	35
3.1 Общее понятие биндинга (привязка данных)	36
3.2 Привязка данных с использованием INotifyPropertyChanged	40
3.3 Представления наборов данных (Collection View)	41
3.4 Операции над данными	43
3.4.1 Фильтрация	43

3.4.2	Сортировка	45
3.4.3	Группировка	45
3.5	Создание новых объектов, редактирование(не сделано)	45
3.6	Проверка (валидация) данных	45
3.7	Разное	45
3.7.1	Направления биндинга	45
3.7.2	Настройка отображения элемента (DataTemplate – шаблон данных) для DataGrid, ListBox, ComboBox и т.п.	46
3.7.3	Значения по умолчанию при биндинге. TargetNullValue, FallbackValue	48
3.7.4	Фильтрация, сортировка “на лету”	49
3.7.5	Конвертеры значений при биндинге	49
3.7.6	Мультибиндинг, мультиконвертеры	49
4	MVVM (не сделано)	50
4.1	Общие сведения	50
4.2	Описание модели данных (Model)	51
4.3	Описание представления (View)	51
4.4	Описание представления модели (View-Model)	51
4.5	Добавление, редактирование данных	51
5	Разное	52
5.1	Формирование отчетов Excel	52
5.2	Создание одностраничной навигации	55
5.3	Перехват необработанных исключений	57
5.4	Невошедшее (не сделано)	57
6	Практический пример (не сделано)	58
6.1	Описание задачи	58
6.2	Создание модели с использованием подхода Database First	59
6.3	Создание модели с использованием подхода Code First	67
6.3.1	Описание модели данных	67
6.3.2	Заполнение БД случайными данными	71
6.4	Создание модели с использованием EF Core	71
6.5	Разработка проекта WPF	71
6.5.1	Вывод данных с использованием Collection View	71
6.5.2	Фильтрация данных	81
6.5.3	Вывод связанных данных	81
6.6	Разработка проекта с использованием паттерна MVVM	81

Список изменений

- 14.07.18

- Добавлен раздел(+пример) [5.1](#) по формированию отчетов в Excel;
- Добавлен раздел [2.2](#) "Linq to Entities";
- В раздел [2](#) "Основные типа запросов LINQ" добавлена информация по запросам Union, Intersect, Except, Distinct;
- Для удобства типы запросов Linq вынесены в содержание.

- 20.07.18

- Добавлен раздел [5.2](#) по одностраничной навигации;
- Обновлено описание раздела [3.1](#) "Общее понятие биндинга"
- Добавлен раздел [3.7.3](#) "Значения по умолчанию при биндинге"
- Добавлен раздел [5.4](#) "Невошедшее содержащий ссылки на практические примеры (без теории) некоторых задач, которые могут возникнуть, среди них:
 - * Drag and Drop;

- 29.07.18

- Добавлены пояснения по биндингу, в конце раздела [3.1](#)
- добавлен раздел [3.7.1](#) "Направления биндинга"
- Описан раздел [1.4](#) "Общие принципы работы с EF". В разделе показано, как производить основные операции над данными в EF (выборка, добавление, редактирование, удаление).

- 21.08.18

- В раздел [5.4](#) "Невошедшее" добавлена информация по работе с горячими клавишами в WPF.
- В раздел [5.4](#) "Невошедшее" добавлена информация по одному из способов работы с изображениями, хранимыми в базе данных.

- 22.08.18

- Дописан раздел [6.5.1](#) "Вывод данных с использованием Collection View" и перенесен из главы [3](#) "WPF" в главу [6](#) "Практический пример".
- Добавлен пример ручного создания представления данных из C# в конце раздела [3.3](#) "Представления наборов данных(Collection View)".

- 23.08.18

- Дополнен раздел [3.7.2](#) "Настройка отображения элемента (DataTemplate – шаблон данных)".
- Добавлен раздел [3.4.1](#) в котором рассмотрен вопрос фильтрации данных с использованием представлений данных (CollectionView).

- **25.08.18**

- В раздел [3.4.1](#) добавлен пример с переменным (и по сути неограниченным) числом фильтров.

Определения

UI (англ. User Interface) - интерфейс пользователя;

EF - Entity Framework;

Collection View - представление данных. реализуется через специальный класс....

Введение

Целью данной работы является ознакомление студентов с технологией Entity Framework (EF) для работы с базами данных на примере СУБД Microsoft SQL Server. В рамках ознакомления с EF были рассмотрены 2 подхода EF6: Database First и Code First, а также кроссплатформенная реализация EF: EF Core.

Независимо от выбранного способа работы с БД, манипуляция с данными будет происходить единообразно, с использованием технологии LINQ, которая позволяет обращаться как к обычным коллекциям объектов, так и к объектам баз данных. Использование данного языка запросов показано в разделе 2.

Для проектирования интерфейса использовалась технология WPF, которая в сочетании с EF позволяет значительно сократить время на разработку приложений. Был рассмотрен как «классический» способ разработки, так и с использованием паттерна Model-View – View Model (MVVM), при котором приложение разделяется на 3 слоя – представление (интерфейс пользователя), модель (описание базы данных) и представление модели (классы, описывающие логику работы конкретного представления), что упрощает кодирование и дальнейшую отладку.

Чтобы работа не носила чисто теоретический характер, был рассмотрен пример из раздела 6.1, однако, т.к. работа носит учебный характер, рассматриваемая предметная область была значительно сокращена. Таким образом, работа состоит из двух основных разделов – описание используемых технологий, с небольшими примерами, и раздел, посвященный их практическому применению на примере данной задачи. При составлении теоретической части данного пособия активно использовались материалы сайта metanit.com.

Перед началом работы рекомендуется ознакомиться с основами проектирования баз данных, сделать это можно, например в [3] (Глава 8. Элементы модели "сущность-связь").

Глава 1

Entity Framework

1.1 Общие сведения

Entity Framework представляет собой объектно-ориентированную технологию на базе фреймворка .NET для работы с данными. EF является ORM-инструментом (object-relational mapping - отображения данных на реальные объекты). Entity Framework представляет собой более высокий уровень абстракции, который позволяет абстрагироваться от самой базы данных и работать с данными независимо от типа хранилища. Если на физическом уровне мы оперируем таблицами, индексами, первичными и внешними ключами, то на концептуальном уровне, который нам предлагает Entity Framework, мы уже работаем с объектами некоторых классов.

Центральной концепцией Entity Framework является понятие сущности или entity. Сущность представляет набор данных, ассоциированных с определенным объектом. Поэтому данная технология предполагает работу не с таблицами, а с объектами и их наборами.

Любая сущность, как и любой объект из реального мира, обладает рядом свойств. Например, если сущность описывает человека, то мы можем выделить такие свойства, как имя, фамилия, рост, возраст, вес. Свойства необязательно представляют простые данные типа `int`, но и могут представлять более комплексные структуры данных. И у каждой сущности может быть одно или несколько свойств, которые будут отличать эту сущность от других и будут уникально определять эту сущность. Подобные свойства называют ключами.

При этом сущности могут быть связаны ассоциативной связью один-ко-многим, один-ко-одному и многие-ко-многим, подобно тому, как в реальной базе данных происходит связь через внешние ключи.

Отличительной чертой Entity Framework является использование запросов LINQ для выборки данных из БД. С помощью LINQ мы можем не только извлекать определенные строки, хранящие объекты, из бд, но и получать объекты, связанные различными ассоциативными связями.

Другим ключевым понятием является Entity Data Model. Эта модель сопоставляет классы сущностей с реальными таблицами в БД. Entity Data Model состоит из трех уровней: концептуального, уровень хранилища и уровень сопоставления (маппинга).

На концептуальном уровне происходит определение классов сущностей, используемых в приложении.

Уровень хранилища определяет таблицы, столбцы, отношения между таблицами и типы данных, с которыми сопоставляется используемая база данных.

Уровень сопоставления (маппинга) служит посредником между предыдущими двумя, определяя сопоставление между свойствами класса сущности и столбцами таблиц. Таким образом, мы можем через классы, определенные в приложении, взаимодействовать с таблицами из базы данных.

Чтобы показать преимущества работы с EF ниже показан пример, в котором одно и то же действие выполняется с помощью «классического» ADO.NET (левая колонка) и с использованием EF (правая). Как видно, при использовании EF снижается количество кода, повышается его читаемость.

```
var query = @"SELECT * FROM Students WHERE ID = @ID";
using (var connection = new SqlConnection())
{
    connection.ConnectionString =
        @"Data Source=.\SQLEXPRESS;Initial Catalog=Test_DB;" +
        "Integrated Security=True";
    connection.Open();
    using (var command = new SqlCommand(query, connection))
    {
        command.Parameters.AddWithValue("ID", id);
        using (var reader = command.ExecuteReader())
        {
            var student = new Student
            {
                Name = reader.
                    GetString((reader.GetOrdinal("Surname")))
            };
            return student;
        }
    }
}

public static Student LoadStudent(Guid id)
{
    using (var context = new StudentContext())
    {
        return context
            .Students
            .FirstOrDefault(x => x.Id == id);
    }
}
```

Рис. 1.1: Сравнение использования ADO.Net и EF на примере простой выборки данных

Entity Framework предполагает три возможных способа взаимодействия с базой данных:

- Database first: На момент начала разработки уже есть база данных. Entity Framework создает набор классов, которые отражают модель этой БД;
- Model first: сначала разработчик создает модель данных, по которой Entity Framework создает реальную базу данных на сервере;
- Code first: разработчик создает класс модели данных, а также описания сущностей, которые будут храниться в бд, а затем Entity Framework по этой модели генерирует базу данных и ее таблицы.

Далее будут рассмотрены подходы Database first и CodeFirst.

1.2 Описание модели данных (Database first)

Подход Database First предполагает наличие базы данных к моменту начала разработки приложения. В этом случае требуется спроецировать существующие таблицы БД и отношения в соответствующие классы, отражающие предметную область.

Чтобы Entity Framework мог получить доступ к базе данных, в системе, помимо самой базы данных, должен быть установлен соответствующий провайдер. По умолчанию Visual Studio поддерживает соответствующую инфраструктуру для СУБД MS SQL Server. Для остальных СУБД, например, MySQL, Oracle и других необходима доустановка соответствующих провайдеров.

Чтобы использовать подход DatabaseFirst выполните действия, описанные в разделе [6.2](#).

1.3 Описание модели данных (Code first)

Подход Code First позволяет описать модель базы данных, используя ОО язык, например C#. В этом случае Entity Framework в соответствии с некоторыми принятыми соглашениями

сгенерирует БД по заданной строке подключения. При этом, следует отметить, что по умолчанию БД создается в виде локального файла SQL Server. Чтобы исправить ситуацию, нужно отредактировать строку подключения в App.Config (для Entity Framework Core строка подключения будет задаваться программно).

Чтобы использовать Code First, нужно выполнить шаги 1-6 из раздела 6.3.1. После выполнения этих действий будет сгенерирован файл примерно следующего содержания:

```
public class UniversityModel : DbContext
{
    public UniversityModel()
        : base("name=UniversityModel")
    {
    }
}
```

Это прототип нашей базы данных, класс UniversityModel будет содержать описания всех используемых таблиц БД.

Чтобы добавить в базу данных новую таблицу, ее нужно описать в виде класса с требуемым набором полей. Типы данных для полей выбираются из стандартных типов C#, EF сам выполнит преобразование в соответствующие типы заданной БД (например String C# будет конвертирован в nvarchar(MAX) в SQL Server'e). Опишем, например, сущность Студент, выделив в качестве полей следующий набор данных: Имя, Фамилия (FirstName, LastName, оба - строковый тип данных, string), дата рождения (BirthDate, тип - DateTime):

```
public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }
}
```

EF требует, чтобы каждая описанная сущность имела **идентификатор** - ключевое поле, которое позволяет однозначно различать между собой объекты. В большинстве случаев в качестве такого поля выступает числовое поле-счетчик, значение которого увеличивается для каждой вновь добавленной записи. Чтобы создать такое поле в EF, нужно добавить в класс свойство с именем Id, или **ИмяСущностиId**, например, StudentId для сущности Student. Добавим идентификатор для описанного выше класса Student:

```
public class Student
{
    //задаем ключевое поле:
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }
}
```

В случае, если тип данных ключа не целочисленный, либо имя должно отличаться от Id, следует использовать атрибут Key (см. раздел 1.3.2), в случае составных ключей, должен использоваться Fluent API (об этом в разделе 1.3.3).

Когда класс, описывающий какую то сущность был создан, он должен быть включен в контекст, для этого необходимо объявить набор данных (DbSet) объектов этого типа в классе контекста, например:

```

public class UniversityModel : DbContext
{
    public UniversityModel()
    : base("name=UniversityModel")
    {
    }
    //добавляем в контекст набор данных для хранения студентов
    //теперь созданная БД будет содержать таблицу Students с описанными выше полями:
    public virtual DbSet<Student> Students { get; set; }
}

```

Подобным образом должны быть описаны все сущности проектируемой БД.

1.3.1 Указание связей БД

После того, как описаны все таблицы, необходимо их связать между собой. Для указания внешних ключей нужно следовать некоторым соглашениям при написании кода. Рассмотрим следующие варианты связей: один-ко-многим, многие-ко-многим, один-к-одному (необязательная связь).

Далее будут показаны способы реализации основных типов связей и сгенерированная диаграмма (на примере Microsoft Sql Server):

- **Один-ко-многим** на примере связи сущностей «группа – студент»: в одной группе может учиться несколько студентов, но каждый студент закреплен только за одной конкретной группой.

Так же, как и для обычных сущностей, необходимо объявить ключевое поле в обеих таблицах. Кроме того, для студента нужно указать связь с группой, при этом сделать это с помощью двух полей – создав поле для хранения внешнего ключа `GroupId`, а также прописав свойство навигации `Group`, по которому в дальнейшем мы сможем получить ссылку на группу, в которой обучается студент.

В классе группы, в свою очередь, для организации доступа к студентам этой группы необходимо объявить коллекцию студентов. EF позволит загрузить данные этой группы автоматически, используя свойство `Students`.

```

public class UniversityModel : DbContext
{
    public UniversityModel()
    : base("name=UniversityModel")
    {
    }
    public virtual DbSet<Student> Students { get; set; }
    public virtual DbSet<Group> Groups { get; set; }
}
public class Student
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public int GroupId { get; set; }
    public virtual Group Group { get; set; }
}
public class Group

```

```

{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Year { get; set; }
    public virtual ICollection<Student> Students { get; set; }
    public Group()
    {
        Students = new List<Student>();
    }
}

```

Данный код приведет к генерации следующей БД:

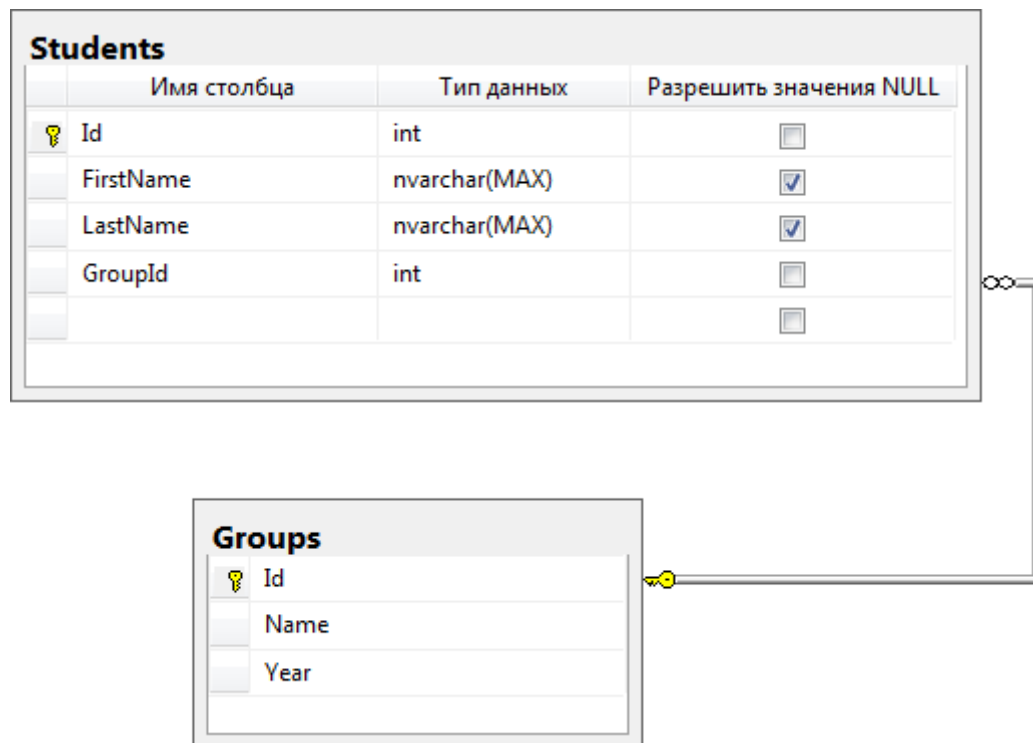


Рис. 1.2: Сгенерированная EF база данных для связи один-много

В случае связи Студент-группа эта связь является обязательной. Если нужно сделать необязательную связь, соответствующий внешний ключ (в данном случае поле GroupId в классе студент) должен быть объявлен как Nullable поле. Для вышеприведенного примера это будет выглядеть следующим образом:

```

public class Student
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public int? GroupId { get; set; }
    public virtual Group Group { get; set; }
}

```

- **Многие-ко-многим** на примере «группа – дисциплина(учебный предмет)»: одна и та же дисциплина может читаться разным группам, одной группе могут преподаваться несколько дисциплин.

Так же, как и в примере выше, для указания множественной связи, нужно использовать ICollection, только в данном случае в обоих классах:

```
public class UniversityModel : DbContext
{
    public UniversityModel()
        : base("name=UniversityModel")
    {
    }
    public virtual DbSet<Discipline> Disciplines { get; set; }
    public virtual DbSet<Group> Groups { get; set; }
}
public class Discipline
{
    public int Id { get; set; }
    //наименование дисциплины
    public string Name { get; set; }
    //ссылка на группы, которым читается данная дисциплина:
    public virtual ICollection<Group> Groups { get; set; }
    public Discipline()
    {
        Groups = new List<Group>();
    }
}
public class Group
{
    public int Id { get; set; }
    //название группы:
    public string Name { get; set; }
    public int Year { get; set; }
    //ссылка на дисциплины, которые читаются данной группой:
    public virtual ICollection<Discipline> Disciplines { get; set; }
    public Group()
    {
        Students = new List<Student>();
    }
}
```

Это приведет к генерации следующей БД:

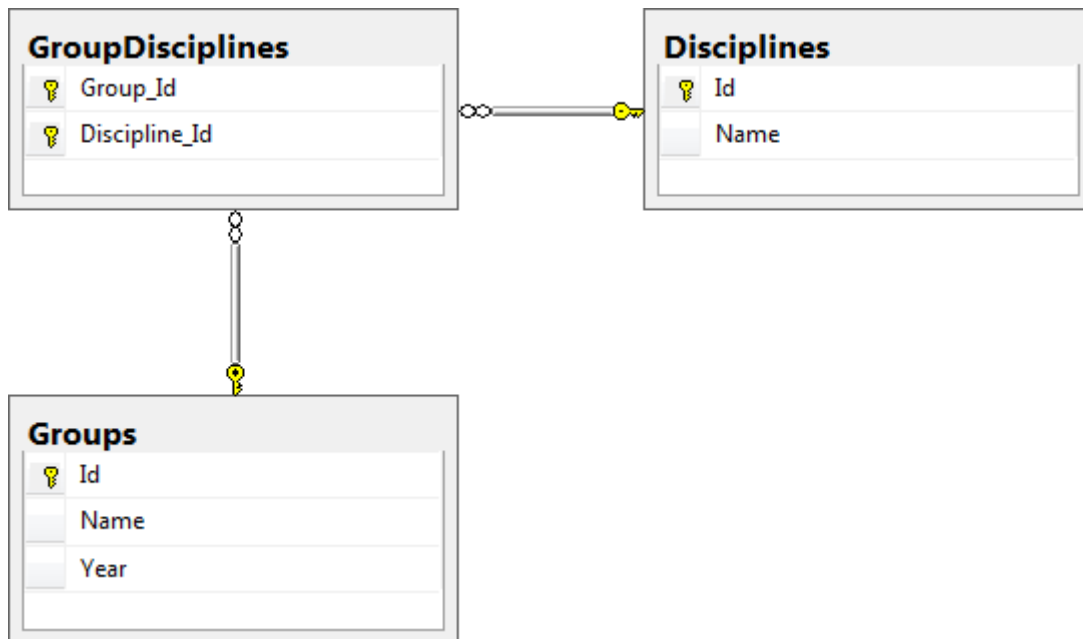


Рис. 1.3: Сгенерированная EF база данных для связи много-много

Как видно, третья таблица, в которую раскрывается связь много-много, была создана автоматически, вручную, если она не содержит дополнительных полей, ее описывать не нужно!

- **Один-к-одному** на примере связи «Абитуриент-Документ об образовании».

Для того, чтобы Entity Framework создал связь один-к-одному, обе сущности должны содержать навигационные свойства друг к другу, а зависимая сущность (Документ), в дополнении к этому, должна содержать свойство для хранения внешнего ключа к главной (абитуриент) сущности. При этом, их первичные ключи должны совпадать, т.к. каждому документу будет сопоставлен ровно один абитуриент, поэтому, имеет смысл сделать первичный ключ документа одновременно внешним ключом к таблице студент. Проиллюстрируем это на примере:

```

public class UniversityModel : DbContext
{
    public UniversityModel() :
        base("name=UniversityModel")
    {
    }
    public virtual DbSet<Enrollee> Enrollees { get; set; }
    public virtual DbSet<EnrolleeDocument> Documents { get; set; }
}
//абитуриент
public class Enrollee
{
    public int Id { get; set; }
    //поля, описывающие абитуриента:
    public string LastName { get; set; }
    public string FirstName { get; set; }
    //свойство для перехода к документу, может быть нулевым(!):
    public virtual EnrolleeDocument Document { get; set; }
}
//документ об образовании
public class EnrolleeDocument

```

```

{
    //используем атрибут ForeignKey для ручного задания внешнего ключа
    //в скобках указываем к какой таблице ставим связь (Абитуриент-Enrollee)
    //это же поле будет одновременно выступать и первичным ключом
    [ForeignKey("Enrollee")]
    public int Id { get; set; }
    //поля, описывающие документ:
    public string Type { get; set; }
    public string DocumentNum { get; set; }
    //свойство для перехода к абитуриенту, которому
    //принадлежит этот документ
    public virtual Enrollee Enrollee { get; set; }
}

```

Это создаст следующую БД (рисунок 1.4):

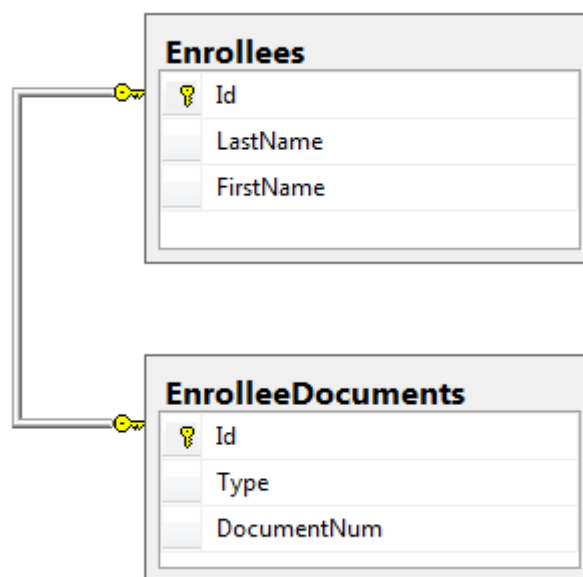


Рис. 1.4: Сгенерированная EF база данных для связи один-к-одному

1.3.2 Ограничения целостности

Кроме вышеперечисленного, EF позволяет задавать ограничения над столбцами таблиц, при нарушении которых будут генерироваться исключения. Это возможно благодаря аннотациям, которые задаются в виде атрибута соответствующего поля. Рассмотрим наиболее используемые (приведен список для EF, для EF Core набор атрибутов может отличаться).

Название	Описание	Пример использования
Key	Позволяет создавать первичные ключи с именем, отличающимся от Id, ИмяСущностиId	<p>Переопределим первичный ключ как StudentNum:</p> <pre> public class Student { [Key] public int StudentNum { get;set; } public string FirstName { get; set; } public string LastName { get;set; } } </pre>

Required	Указывает, что поле не может быть нулевым. Применимо к ссылочным типам (таким как строки), т.к. типы-значения (числа) по умолчанию не могут иметь null значения (для этого они должны быть объявлены как Nullable)	Сделаем обязательными поля имя и фамилия сущности студент: <pre>public class Student { public int Id { get;set; } [Required] public string FirstName { get; set; } [Required] public string LastName { get;set; } }</pre>
? (не атрибут, ставится после указания типа данных при объявлении переменной)	Указывает, что поле может быть нулевым. Применимо к типам-значениям (таким как DateTime, int, double,...), т.к. для остальных типов (таких как строки) это действует по умолчанию.	Сделаем необязательной оценку студента при сдаче экзамена (т.к. студент может не явиться на экзамен или быть недопущенным)*: <pre>public class StudentExam { public int StudentId { get;set; } public int ExamId { get;set; } public DateTime ExamDate { get;set; } //оценка: public int? Assessment { get;set; } }</pre> <p><i>*в StudentExam не задано ключевое поле, т.к. в данном случае это StudentId, ExamId и ExamDate. Составные ключи определяются в классе контекста, подробнее см. раздел 1.3.3</i></p>
MaxLength и MinLength	Максимальное и минимальное количество символов в строке	Зададим ограничение на максимальную длину для имени и фамилии: <pre>public class Student { public int Id { get;set; } [Required, MaxLength(50)] public string FirstName { get; set; } [Required, MaxLength(50)] public string LastName { get;set; } }</pre>
NotMapped	Свойство, помеченное данным атрибутом не будет сопоставляться со столбцом БД	Поле AverageRate, которое является расчетной характеристикой и не должно храниться в БД: <pre>public class Student { public int Id { get;set; } //средняя оценка студента: [NotMapped] public double AverageRate { get;set; } }</pre>

Range (min,max)	Задаёт ограничение для числовых полей	<p>Для класса экзамен, оценка может находиться в диапазоне от 2 до 5*:</p> <pre>public class StudentExam { public int StudentId { get;set; } public int ExamId { get;set; } public DateTime ExamDate { get;set; } //оценка может принимать значения от 2 до 5: [Range(2,5)] public int Assessment { get;set; } }</pre> <p><i>*в StudentExam не задано ключевое поле, т.к. в данном случае это StudentId, ExamId и ExamDate. Составные ключи определяются в классе контекста, подробнее см. раздел 1.3.3</i></p>
С другими атрибутами можно ознакомиться по ссылке: https://metanit.com/sharp/entityframework/6.3.php		

1.3.3 Составные ключи

Для задания составных ключей потребуется использование Fluent API. Fluent API по большому счету представляет набор методов, которые определяют сопоставление между классами и их свойствами и таблицами и их столбцами. Entity Framework использует метод HasKey из Fluent API для обозначения свойств, которые являются идентификаторами и которые будут сопоставлены с первичным ключом в базе данных. Как правило, функционал Fluent API задействуется при переопределении метода OnModelCreating, покажем это на примере:

```
class Student
{
    public int Id { get; set; }
    //....остальные поля....
}
//преподаваемая дисциплина
class Exam
{
    public int Id { get; set; }
    //....остальные поля....
}
class StudentExam
{
    //следующие 3 поля сделаем идентифицирующими:
    public int StudentId { get; set; }
    public int ExamId { get; set; }
    public DateTime ExamDate { get; set; }
    public int Assesment { get; set; }
}
public class UniversityModel : DbContext
{
    public UniversityModel() : base("name=UniversityModel") { }
    //переопределяем метод OnModelCreating для создания составных ключей:
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //для сущности StudentExam определяем идентификатор как объект
        //анонимного класса поле StudentNumber:
```

```

        modelBuilder.Entity<StudentExam>().HasKey(se =>
            new { se.StudentId, se.ExamId, se.ExamDate } );
        base.OnModelCreating(modelBuilder);
    }
    public virtual DbSet<Student> Students { get; set; }
    public virtual DbSet<Exam> Exams { get; set; }
}

```

На выходе имеем следующую таблицу:

StudentExams	
🔑	StudentId
🔑	ExamId
🔑	ExamDate
	Assessment

Рис. 1.5: Сгенерированная EF таблица с составным первичным ключом

1.3.4 Инициализатор базы данных

Часто бывает нужно, чтобы вновь созданная база данных содержала какие то начальные данные. EF предоставляет для этого специальный механизм инициализации.

Для инициализации мы можем использовать один из классов инициализаторов:

- **CreateDatabaseIfNotExists**: инициализатор, используемый по умолчанию. Он не удаляет автоматически базу данных и данные, а в случае изменения структуры моделей и контекста данных генерирует исключение;
- **DropCreateDatabaseIfModelChanges**: данный инициализатор проверяет на соответствие моделям определения таблиц в базе данных. И если модели не соответствуют определению таблиц, то база данных пересоздается;
- **DropCreateDatabaseAlways**: этот инициализатор будет всегда пересоздавать базу данных.

Покажем использование инициализатора. Допустим, дан контекст данных, определенный следующим образом:

```

public class UniverContext : DbContext
{
    public UniverContext():
        base( "name=UniverContext" )
    {
    }
    public DbSet<Group> Groups { get; set; }
    public DbSet<Student> Students { get; set; }
}
public class Student
{
    public int Id {get;set;}
    public string FullName {get;set;}
    public string BirthDate {get;set;}
}

```

```

    public int GroupId {get;set;}
    public virtual Group Group {get;set;}
}
public class Group
{
    public Group()
    {
        Students = new List<Student>();
    }
    public int Id {get;set;}
    public string Name {get;set;}
    public virtual ICollection <Student> Students {get;set;}
}

```

Определим инициализатор, пересоздающий базу данных каждый раз при изменении модели(DropCreateDatabaseIfModelChanges):

```

//Класс инициализатор должен наследоваться от DropCreateDatabaseIfModelChanges с
//указанием типа контекста, для которого выполняется инициализация
class UniverDbInitializer : DropCreateDatabaseIfModelChanges<UniverContext>
{
    //Метод, вызываемый при создании БД и выполняющий наполнение начальными
    // значениями. Параметром здесь выступает объект контекста БД, для
    // которого выполняется инициализация
    protected override void Seed(UniverContext db)
    {
        //создаем двух студентов:
        Student s1 = new Student
        {
            FullName = "Иванов Иван",
            BirthDate = new DateTime(2000,05,05)
        };
        Student s2 = new Student
        {
            FullName = "Сидоров Сидр",
            BirthDate = new DateTime(2001,02,02)
        };
        //создаем группу:
        Group g = new Group { Name = "АСМ-11" };
        //добавляем студентов в группу:
        g.Students.Add( s1 );
        g.Students.Add( s2 );
        //а можно сразу создать группу и заполнить ее студентами:
        Group g2 = new Group
        {
            Name = "АСМ-12",
            Students = new List<Student>
            {
                new Student
                {
                    FullName = "Петя",
                    BirthDate = new DateTime(2001,1,1)
                },
                new Student
                {
                    FullName = "Оля",
                    BirthDate = new DateTime(2001,2,2)
                }
            }
        }
    }
}

```

```

    };
    //после того,как данные сгенерированы, их можно добавить в контекст:
    db.Groups.Add(g);
    db.Groups.Add(g2);
    //не забываем сохранить(!):
    db.SaveChanges();
}
}

```

Обратите внимание, что в вышеприведенном примере в контекст добавляются только сведения по группам, без студентов. Т.к. группы содержат в себе ссылки на студентов, они будут добавлены автоматически.

Чтобы инициализатор сработал, надо его вызвать. Один из способов вызова инициализатора предполагает вызов его в статическом конструкторе класса контекста. Изменим класс `UniverContext` следующим образом:

```

class UniverContext : DbContext
{
    static UniverContext()
    {
        Database.SetInitializer<UniverContext>(new UniverDbInitializer());
    }
    public UniverContext():
    base( "name=UniverContext" )
    {
    }
    public DbSet<Group> Groups { get; set; }
    public DbSet<Student> Students { get; set; }
}

```

В заключении стоит отметить, что предложенная здесь модель инициализации является лишь примером, не обязательно добавлять все данные в одном методе, правильнее разбить это на несколько методов, а сами данные, в зависимости от задачи генерировать или случайным образом (если работаем с тестовой версией проекта), или из внешних источников (например файл).

1.4 Общие принципы работы с EF

1.4.1 Подключение EF к проекту, настройка подключения

Первое что необходимо сделать - подключить библиотеку Entity Framework к проекту. Это происходит автоматически при выполнении шагов, описанных в разделах [6.2](#) (если у вас уже готова база данных) и [6.3.1](#) (если вы предпочитаете описать модель вручную в виде набора классов).

При использовании подхода Database First будет использована база данных, указанная в строке подключения, для Code First так же будет использована БД, указанная в строке подключения, однако, если такой БД не будет существовать, EF попытается ее создать.

Когда EF подключен, вы можете внести изменения в строку подключения, чтобы настроить параметры подключения к базе данных.

Строка подключения

Строки подключения располагаются в файле `App.Config` в разделе `"connectionStrings"`. Строка подключения указывает программе, где фактически располагается база данных, с которой

вы работаете, а также какие параметры необходимо использовать при работе с этой БД. Внешний вид файла App.Config показан на рисунке 1.6. Видно, что в данном примере программа располагает двумя базами данных, названными UniversityModel и SuperBdshka.

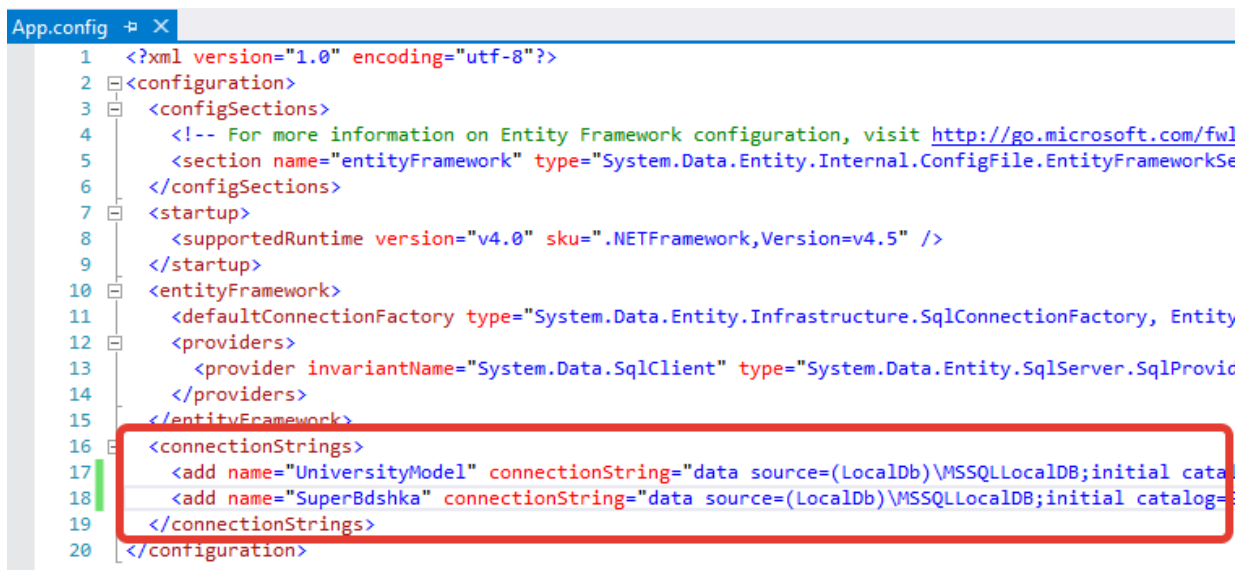


Рис. 1.6: Расположение строк подключения в файле App.Config

Далее будут пояснены основные компоненты строки подключения на примере базы данных Microsoft Sql Server. Сразу стоит отметить, что работа с SQL Server'ом возможна в двух вариантах:

- Экземпляр БД изолирован и доступ к нему идет через запущенную службу SQL Server'a. При этом БД может располагаться удаленно, возможна совместная работа нескольких пользователей с одной БД;
- Экземпляр БД расположен локально и представлен в виде отдельно лежащего файла в формате .mdf. Этот файл можно разместить в папке с программой, тогда при копировании программы БД будет всегда с ней. Однако, этот вариант не подразумевает совместной работы, одновременно работать с БД может только один пользователь. Вариант удаленного доступа возможен только в случае, если файл базы данных расположен в папке, к которой предоставлен общий доступ.

Рассмотрим первый случай. В этом случае в разделе connectionStings будет запись вида:

```
<add name="testEntities" connectionString="metadata=res://*/Model1.csdl|res://*/Model1.ssdl|res://*/Model1.msl;
provider=System.Data.SqlClient;
provider connection string="
data source=Server-PC\SQLEXPRESS;
initial catalog=test;
integrated security=True;
MultipleActiveResultSets=True;
App=EntityFramework";"
providerName="System.Data.EntityClient" />
```

либо:

```
<add name="UniversityModel" connectionString=
"data source=Server-PC\SQLEXPRESS;
```

```
initial catalog=test;
integrated security=True;
MultipleActiveResultSets=True;
App=EntityFramework"
providerName="System.Data.SqlClient" />
```

Это две эквивалентные строки подключения, указывающие на одну базу данных. Первая создана при использовании подхода DatabaseFist, вторая - CodeFirst. Для удобства восприятия записи разбиты на несколько строк, в App.Config это однострочная запись. К редактированию этого раздела настроек нужно относиться осторожно, т.к. даже небольшая опечатка может сделать ваше приложение неработающим, поэтому рассмотрим только основные элементы:

- provider - провайдер базы данных, отвечает за то, какой драйвер базы данных будет использоваться, в данном случае значение System.Data.SqlClient указывает на использование Microsoft Sql Server'a;
- data source - указание, где расположен Sql Server. Состоит из двух частей - указание машины, на которой запущен сервер и имени экземпляра сервера БД, разделенных символом '\ '.

В данном примере "Server-PC"- имя рабочей станции, на которой запущен сервер. Может быть представлено в виде символьного имени, в виде ip-адреса (например 192.168.0.121). Если сервер запущен на той же машине, где выполняется программа, может быть задано в виде точки, т.е. data source выглядел бы как .\SQLEXPRESS.

"SQLEXPRESS"- имя экземпляра сервера (этот параметр задается при установке СУБД, в случае использования бесплатной редакции SQL Server'a, скорее всего имя будет, как и в текущем примере - SQLEXPRESS);

- initial catalog - имя используемой базы данных (в данном примере test);
- integrated security - задает режим аутентификации. Значение true означает использование проверки подлинности Windows.

Если база данных будет расположена удаленно, возможно использование доступа по логину и паролю, в этом случае необходимо добавить еще несколько параметров: persist security info = false; user id=имя_пользователя; password=пароль_пользователя. Здесь следует учитывать, что проверка подлинности Windows имеет приоритет над именами входа SQL Server. Если указать значение Integrated Security=true, а также ввести имя пользователя и пароль, то имя пользователя и пароль не будут учитываться и будет применяться проверка подлинности Windows.

Незашифрованные строки подключения предоставляют некоторую угрозу безопасности вашего приложения. О способах защиты строки подключения и в общем файла конфигурации см. [11].

Теперь рассмотрим внешний вид строки подключения при использовании локального файла базы данных Microsoft SQL Server. Данный тип базы данных выбирается по умолчанию при использовании подхода CodeFirst.

```
<add name="SuperBdshka" connectionString=
"data source=(LocalDb)\MSSQLLocalDB;
initial catalog=myDataBase;
integrated security=True;
MultipleActiveResultSets=True;
App=EntityFramework"
providerName="System.Data.SqlClient" />
```

Здесь параметр *data source* указывает, что будет использоваться локальный экземпляр SQL Server'a, *initial catalog* - так же, как и в прошлом случае имя базы данных. По умолчанию базы данных, соответствующие такой строке подключения должны располагаться в домашнем каталоге текущего пользователя. Чтобы изменить ситуацию, можно заменить параметр *initial catalog* на другой - *attachDbFileName*, который позволяет указать точное расположение файла БД:

```
attachDbFileName=c:\temp\myDataBase.mdf;
```

Чтобы расположить файл БД в каталоге выполняемого приложения, нужно указать параметр `|DataDirectory|`, например:

```
attachDbFileName=|DataDirectory|myDataBase.mdf
```

Однако, в некоторых случаях, этот параметр может быть не определен, в этом случае, вы получите ошибку, показанную на рисунке 1.7

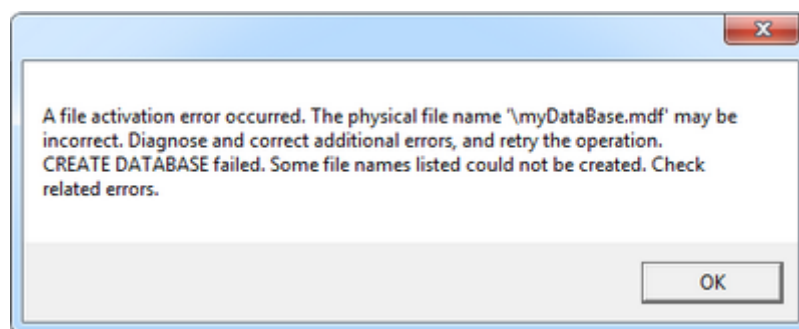


Рис. 1.7: Возможная ошибка при использовании параметра `DataDirectory`

Чтобы исправить ситуацию, необходимо вручную указать значение этого параметра. Сделать это можно в конструкторе класса приложения (по умолчанию расположен в файле `App.xaml.cs`). Ниже показано, как указать в качестве *DataDirectory* путь к текущему каталогу приложения:

```
/// <summary>
/// Логика взаимодействия для App.xaml
/// </summary>
public partial class App : Application
{
    public App()
    {
        //вместо Environment.CurrentDirectory можно использовать и абсолютные пути,
        //например можно было указать @"c:\temp\"
        AppDomain.CurrentDomain.SetData("DataDirectory", Environment.CurrentDirectory );
    }
}
```

Более подробно с этими и другими параметрами строк подключения можно ознакомиться в [10].

1.4.2 Создание и использование контекста базы данных

Все примеры в этом разделе и оставшихся разделах этой главы основаны на модели данных, описанной в разделе 6.1.

Когда подключение сконфигурировано, можно начинать работу с базой данных. Чтобы произвести какие-то действия, сначала необходимо создать объект контекста базы данных. Как узнать имя контекста при использовании DatabaseFirst показано на рисунке 6.2 в разделе 6.2. Про контекст в CodeFirst написано в разделе 1.3.

Допустим, класс контекста называется UniversityModel, тогда, чтобы создать его и начать работать с данными, достаточно написать:

```
using (UniversityModel model = new UniversityModel())
{
    //какие-то операции с данными
}
```

Также вы можете создать и использовать несколько контекстов данных для одной базы данных:

```
UniversityModel context1 = new UniversityModel();
UniversityModel context2 = new UniversityModel();
//работаем с данными...

//не забываем уничтожить context1, context2, когда они не нужны:
context1.Dispose();
context2.Dispose();
```

Однако, следует учитывать, что это два несвязанных контекста данных, изменения данных в одном контексте не отразятся на данных другого контекста. Чтобы поместить объект в контекст (или переместить из одного в другой), нужно использовать метод Attach соответствующего набора данных, но, в один момент времени объект может принадлежать только одному контексту. Поэтому, по возможности, избегайте работы с одной базой данных через разные контексты.

Если вы планируете использовать базу данных на протяжении всего приложения, можно создать единый глобальный контекст для всего приложения. Для этого необходимо объявить его внутри класса приложения (по умолчанию расположен в файле App.xaml.cs). Это можно сделать следующим образом:

```
public partial class App : Application
{
    public static UniversityModel DatabaseContext { get; } = new UniversityModel();
}
```

Теперь везде, где мы хотим обратиться к контексту базы данных, мы должны писать `App.DatabaseContext`, например `App.DatabaseContext.Students` даст нам ссылку на “таблицу” студентов.

Примечание: В данном случае использована статическая переменная, которая инициализируется в момент загрузки приложения. В “боевых” условиях нужно осторожнее относиться к такого рода действиям. Т.к., если в момент инициализации базы данных возникнет исключение, оно не будет обработано, сборка не будет загружена и приложение “упадет”.

1.4.3 Получение данных

В качестве контекста будет использоваться `App.DatabaseContext` из примера выше. Ниже показаны основные сценарии выборки данных:

- **Однократная выборка всех данных** - выборка всех записей из таблицы на текущий момент времени. Данные, добавляемые в контекст после этой выборки никак не отражаются на полученном результате.

```
//получаем студентов как массив
var studentsArray = App.DatabaseContext.Students.ToArray();
//получаем список студентов:
var studentsList = App.DatabaseContext.Students.ToList();
//выполняем с ними какие-то операции, например, выводим на консоль:
foreach( var student in studentsArray)
{
    Console.WriteLine( student.LastName + " " + student.FirstName);
}
```

- **Однократная выборка с фильтрацией, преобразованиями данных** - для этих целей необходимо использовать методы LINQ, описанные в разделе 2, например:

```
//получаем имена студентов с фамилией "Иванов", имеющих среднюю оценку > 4
//и преобразуем результат в массив
var studentsArray = App.DatabaseContext.Students
    .Where( s => s.LastName == "Иванов") //фильтр по фамилии
    .Where( s => s.StudentExams.Avg( se => se.Assessment) > 4) //по оценке
    .Select( s => s.FirstName) //из всего студента берем только имя
    .ToArray(); //преобразуем результат в массив
//полученный studentsArray имеет тип string[] - массив строк
```

- **Выборка с отслеживанием состояния контекста** - для этих целей необходимо использовать свойство Local требуемого набора данных. Это коллекция данных, автоматически уведомляющая о своем изменении. Далее приведен пример работы с данным типом выборки.

```
//предварительно необходимо загрузить данные, используя метод Load()
App.DatabaseContext.Students.Load();
//после этого можно начинать работать:
foreach( var student in App.DatabaseContext.Students.Local){
    Console.WriteLine( student.LastName + " " + student.FirstName);
}
```

1.4.4 Добавление/редактирование данных

Добавление новых объектов осуществляется через вызовы методов Add, AddRange (для добавления одного и нескольких объектов) к соответствующему набору данных. Покажем это на примере:

```
Group g = new Group
{
    Name = "ACM-11", //название группы
    Year = 2011, //год набора { get; set; }
};
//добавляем созданную группу в контекст:
App.DatabaseContext.Groups.Add( g );
//теперь создадим несколько студентов:
var students = new[]
{
```

```

new Student()
{
    FirstName = "Сидор",
    LastName = "Сидоров",
    Group = g //в качестве группы указываем созданную выше
},
new Student()
{
    FirstName = "Петр",
    LastName = "Иванов",
    Group = g
},
new Student()
{
    FirstName = "Николай",
    LastName = "Николаев",
    Group = g,
    BirthDay = new DateTime(2000,10,10)
}
};
//и добавим всех сразу через вызов AddRange:
App.DatabaseContext.Students.AddRange( students );
//чтобы эти изменения попали в базу данных, необходимо сохранить изменения
//для этого вызываем метод SaveChanges:
App.DatabaseContext.SaveChanges();

```

Как видно, вся работа с данными идет на уровне объектов и ссылок на них. EF избавляет от необходимости ручного проставления значений внешних ключей при работе со связанными данными. Так, в примере выше, чтобы указать, что студент обучается в группе АСМ-11, достаточно инициализировать свойство навигации Group ссылкой на нужную группу, задачу по определению и проставлению первичных и внешних ключей берет на себя EF. Также, EF избавляет от необходимости преобразований данных, используются стандартные типы .net.

Пример выше можно немного сократить - дело в том, что если у сущности есть новые связанные записи, то при вызове SaveChanges они автоматически добавятся в контекст. Например, у класса Group есть ссылка на множество студентов, обучающихся в этой группе. Поэтому добавить студентов можно сразу в группу:

```

Group g = new Group
{
    Name = "АСМ-11", //название группы
    Year = 2011, //год набора{ get; set; }
    Students = new [] //добавляем студентов при создании группы
    {
        new Student()
        {
            FirstName = "Сидор",
            LastName = "Сидоров"
        },
        new Student()
        {
            FirstName = "Петр",
            LastName = "Иванов"
        },
        new Student()
        {
            FirstName = "Николай",
            LastName = "Николаев",

```

```

        BirthDay = new DateTime(2000,10,10)
    }
};
//добавляем созданную группу в контекст:
App.DatabaseContext.Groups.Add( g );
//сохраняем изменения:
App.DatabaseContext.SaveChanges();

```

Несмотря на то, что в контекст мы добавили только группу, в базу данных также попадут и три студента, причем каждый из них получит `GroupId` той группы, в которую они добавлены, хотя это нигде явным образом не указано.

Редактируются объекты в EF простым изменением полей соответствующего объекта класса. Например, изменим имя и группу студента:

```

//для опытов возьмем первого студента:
Student s = App.DatabaseContext.Students.FirstOrDefault();
//если в Students нет ни одного студента, s будет равно null, редактировать будет некого
if( s!= null )
{
    //изменим группу на "ИФ-71", для этого сначала попробуем найти ее в БД:
    Group group = App.DatabaseContext.Students.FirstOrDefault( g => g.Name == "ИФ-71");
    //присваиваем найденную группу студенту
    s.Group = group;
    //и меняем имя:
    s.FirstName = "Евгений";
    //чтобы изменения попали в базу данных, сохраняем:
    App.DatabaseContext.SaveChanges();
}

```

В дальнейшем, при рассмотрении работы с WPF будет рассмотрен механизм привязки данных, который значительно упрощает редактирование, избавляя от ручного копирования данных.

1.4.5 Удаление данных

Удаление объектов осуществляется через вызовы методов `Remove`, `RemoveRange` (для удаления одного и нескольких объектов) к соответствующему набору данных. Покажем это на примере:

```

//удалим студента с фамилией Иванов
//метод получения ссылки на объект может варьироваться
//например, это м.б. выделенная запись на форме
//в данном примере - через LINQ запрос
Student s = App.DatabaseContext.Students.FirstOrDefault( s => s.LastName == "Иванов");
//удаляем:
App.DatabaseContext.Students.Remove( s );

//теперь удалим всех студентов с именем "Петр".
//т.к. студентов может быть много, для получения набора данных используем метод Where:
Student deletedStudents = App.DatabaseContext.Students.Where( s => s.FirstName == "Петр");
App.DatabaseContext.Students.RemoveRange( deletedStudents );

```

Если удаляемый объект является "родительским" для других объектов, то вышеприведенный код может выдать ошибку. Например, если бы мы удаляли группу, с которой были бы связаны студенты. В этом случае возможны следующие варианты решения проблемы:

- Настройка каскадного удаления. В этом случае при удалении "родительского" объекта все дочерние записи будут удалены автоматически. При использовании подхода DatabaseFirst настраивается на уровне базы данных, при использовании CodeFirst - через FluentApi при методе OnModelCreating контекста БД.
- Ручное удаление через Remove всех связанных записей, например:

```
//удалим группу "АСМ-910" и всех связанных с ней студентов
//сначала получаем ссылку на удаляемую группу:
var group = App.DatabaseContext.Group.FirstOrDefault( g => g.Name == "АСМ-910" );
if( group != null )
{
    //теперь ссылку на студентов, обучающихся в этой группе:
    var students = group.Students;
    //удаляем студентов:
    App.DatabaseContext.Students.RemoveRange(students);
    //а теперь группу:
    App.DatabaseContext.Group.Remove(group);
    //сохраняем изменения:
    App.DatabaseContext.SaveChanges();
}
```

Если у студентов также есть связанные записи, эту процедуру необходимо повторить для каждого удаляемого студента.

- Удаление с помощью SQL запроса:

```
var group = App.DatabaseContext.Group.FirstOrDefault( g => g.Name == "АСМ-910" );
if( group != null )
{
    //удаляем всех студентов этой группы:
    App.DatabaseContext.Database.ExecuteSqlCommand(
        "delete from Students where GroupId = " + group.Id.ToString());
    //либо можно использовать альтернативную форму с использованием параметра:
    App.DatabaseContext.Database.ExecuteSqlCommand(
        "delete from Students where GroupId = @GroupId",
        new SqlParameter("@GroupId", group.Id ));
    //после чего удаляем саму группу:
    App.DatabaseContext.Database.ExecuteSqlCommand(
        "delete from Groups where Id = @GroupId",
        new SqlParameter("@GroupId", group.Id ));
}
```

Глава 2

LINQ

2.1 Основные типы запросов Linq

2.1.1 Общие сведения

LINQ (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных. В качестве источника данных может выступать объект, реализующий интерфейс `IEnumerable` (например, стандартные коллекции, массивы), набор данных `DataSet`, документ XML. Но вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход для выборки данных.

В качестве запросов могут выступать запросы на фильтрацию данных (`where`), преобразование/проекцию (`select`), группировку (`group`) и т.д.

LINQ представлен в двух вариантах: SQL подобный язык запросов вида `from .. in .. where .. select`, а также специальные методы расширения, которые определены в интерфейсе `IEnumerable`. Остановимся на втором методе, т.к. он является более гибким и не требует освоения нового языка запросов.

Всего насчитывается около 40 методов расширения, рассмотрим некоторые из них:

- `Where` - фильтрация, отбор данных;
- `Select` - проекция, преобразование исходных данных в нужную форму;
- `SelectMany` - выборка, преобразование и объединение множеств объектов;
- `OrderBy` (`ThenBy`) - сортировка;
- `FirstOrDefault`(`First`, `Last`, `LastOrDefault`) - выбор первого объекта, удовлетворяющего условиям поиска;
- `Single` (`SingleOrDefault`) - выбор единственного элемента, удовлетворяющего условиям поиска. Если элементов больше одного, генерируется исключение, в остальное поведение похоже на `First` (`FirstOrDefault`).
- `Union`, `Intersect`, `Except` - операции над множествами: объединение, пересечение, исключение;
- `Distinct` - исключение повторяющихся значений из исходного набора, получение набора уникальных значений;

Большинство методов расширения принимают в качестве параметра делегат (ссылка на функцию), управляющий работой данного метода. Формат делегата определяется назначением метода расширения, так, например, метод расширения Where будет принимать функции, возвращающие bool и принимающие в качестве параметра один объект коллекции, например:

```
bool FilterStudent( Student student )
{
    //выбираем всех студентов с фамилией Иванов:
    if( student.LastName == "Иванов")
        return true;
    return false;
}
```

Однако, писать для каждого используемого метода расширения отдельную процедуру может быть неудобно, поэтому, часто для сокращения объема кода используют лямбда-выражения. В этом случае вызов метода расширения может быть представлен следующим образом:

```
наборОбъектов.Метод( o => результат );
```

Здесь *o* - один из элементов коллекции, рассматриваемый в данный момент - выражение, преобразующее *o* в результат заданного типа. Например, для примера выше:

```
наборОбъектов.Where( s => s.LastName == "Иванов" );
```

Если выражения для получения результата не может быть представлено одним оператором, можно использовать альтернативный синтаксис:

```
//то же самое, но в фигурных скобках и с оператором return:
наборОбъектов.Where( s =>
{
    if( s.LastName == "Иванов")
        return true;
    return false;
});
```

Далее будут рассмотрены примеры использования основных методов расширения LINQ. Кроме того рекомендуется ознакомиться с упражнениями из [2].

2.1.2 Where

Выборка множества объектов, удовлетворяющих заданному условию. В качестве параметра принимает функцию вида:

```
bool Filter ( ТипОбъектаКоллекции o );

//выборка всех студентов с фамилией Сидоров и возрастом меньше 23 лет
var result=students.Where( s => s.LastName == "Сидоров" && Age < 23 ).ToList();
```

то же, с использованием обычной функции для фильтрации:

```
bool StudenFilter( Student s)
{
    return s.LastName == "Сидоров" && Age < 23;
}
//пример использования:
var result = students.Where( StudenFilter ).ToList();
```

Важно: при работе с Entity Framework LINQ пытается преобразовать выражения, переданные в методы расширения в эквивалентные SQL запросы, поэтому, не весь код будет допустимым, т.к. не каждая операция имеет эквивалент в SQL. В этом случае можно использовать несколько последовательных фильтров: `students.Where(отбор1).Where(отбор2).ToList();` Первый запрос к базе данных получает набор объектов из базы данных, которые могут быть дофильтрованы локально.

Пример запроса Where см. в "Ресурсы\Примеры типовых проектов\LinqExamples"

2.1.3 Select

Преобразование объектов одной коллекции в множество объектов другого типа. Количество элементов при этом совпадает. В качестве параметра принимает функцию вида:

```
TResult Transform( ТипОбъектаКоллекции o );
```

где TResult - какой-то тип данных

Например, из списка студентов требуется выбрать только их фамилии:

```
//тип результата List<string>:  
var result = students.Select ( s => s.LastName ).ToList();
```

или:

```
//выборка множества объектов анонимного класса:  
var result = students.Select ( s => new {s.LastName, s.FirstName} ).ToList();
```

или наоборот, дано множество строк вида "Фамилия; Имя" и на основе этих строк нужно построить множество объектов типа Student:

```
//читаем данные из файла  
var strings = File.ReadAllLines("students.csv");  
var result = string.Select( s=>  
{  
    //разбиваем исходную строку на подстроки по разделителю ';' ;'  
    var subStrings = s.Split(';');  
    //получаем массив строк  
    //в нулевом элементе хранится фамилия, в первом - имя  
    //с учетом этого создаем объект класса Student:  
    return new Student()  
    {  
        LastName = subStrings[0],  
        FirstName = subStrings[1]  
    };  
}).ToList();    //тип результата List<Student>
```

Пример запроса Select см. в "Ресурсы\Примеры типовых проектов\LinqExamples"

2.1.4 SelectMany

Выборка и объединение одной коллекций какого-то объекта. Например, даны классы группа и студент, описанные следующим образом:

```

public class Group
{
    public int Id {get;set;}
    public string Name {get;set;}
    public int Kurs {get;set;} //курс, на котором учится группа
    public virtual ICollection<Student> Students {get;set;} //студенты группы
}
public class Student
{
    public int Id {get;set;}
    public string LastName {get;set;}
    public string FirstName {get;set;}

    public int GroupId {get;set;}
    public virtual Group Group {get;set;}
}

```

В классе "группа" есть ссылка на студентов, обучающихся в данной группе. Тогда, например, запрос на выбор всех студентов третьего курса будет иметь вид:

```

//тип результата List<Student>
var result = groups.Where( g => g.Kurs == 3 ).SelectMany( g => g.Students ).ToList();

```

Пример запроса SelectMany см. в "Ресурсы\Примеры типовых проектов\LinqExamples"

2.1.5 OrderBy (ThenBy)

Вызов данных функций выполняет сортировку. Первым употребляется OrderBy, если нужна досортировка данные по другим полям - последовательно применяются вызовы ThenBy:

```

//на выходе получим отсортированное множество студентов
//сначала по названию группы, потом по фамилии, потом по имени:
var result = students.
    OrderBy( s => Group.Name ).
    ThenBy( s => s.LastName ).
    ThenBy( s => s.FirstName ).
    ToList();

```

2.1.6 FirstOrDefault (LastOrDefault)

FirstOrDefault - то же, что Where, но возвращает первый и единственный объект, удовлетворяющий условию. Если таких объектов не найдено, возвращает null. Может использоваться, например, при аутентификации:

```

var user = users.FirstOrDefault( u => u.Login == "Vasja" && u.Password == "12345" );
if( user != null ){
    /*выполнить действия для успешной аутентификации*/
}

```

LastOrDefault - аналогично FirstOrDefault, но возвращает последний элемент, удовлетворяющий условию.

2.1.7 First, Last

Выполняют те же действия, что и FirstOrDefault, LastOrDefault, но если элементов не найдено, генерируется исключение

2.1.8 Union

Объединение двух множеств данных в одно. Это может потребоваться, когда мы работаем с разными источниками данных, либо с разными подмножествами одного источника.

Одним из сценариев использования union'a может стать формирование выпадающего списка для фильтрации каких-либо данных. Например, мы хотим вывести студентов, с возможностью выбора группы в которой студент обучается, а также с возможностью выбора студентов всех групп. В этом случае мы должны выбрать список всех групп их базы данных, а также добавить "виртуальную" сущность "все группы" в результирующую коллекцию:

```
//здесь Database - имя контекста EF
//выбираем все записи из таблицы Groups и сортируем их по полю Name
//результат записываем в массив
var groups = Database.Groups.OrderBy(g => g.Name).ToArray();
//формируем массив из одного элемента, содержащий виртуальную группу
//с названием "Все группы" и заведомо несуществующим Id = -1
var allGroups = new [] { new Group{ Name = "Все группы", Id = -1 } };
//производим объединение множеств:
var resultList = allGroups.Union( groups );
//теперь осталось загрузить этот набор данных в соответствующий
//ComboBox, ListBox, DataGrid,... используя свойство ItemsSource
//cbGroups - имя ComboBox'a, содержащего список групп
cbGroups.ItemsSource = resultList;
```

То же самое можно сделать и одним запросом:

```
cbGroups.ItemsSource = new[] { new Group { Name = "Все группы", Id = -1 } }
    .Union( Database.Groups.OrderBy(g => g.Name).ToArray() );
```

Стоит обратить внимание, что Union выполняет объединение множеств последовательно, а также оставляет дубликаты, поэтому, даны массивы `var a = new[] {1, 2, 3}; var b = new[] {3, 7, 8}; var c = new[] {4, 5}` и к этим массивам применить операцию Union: `a.Union(b).Union(c);` получим следующую последовательность чисел: 1, 2, 3, 3, 7, 8, 4, 5.

2.1.9 Intersect

Операция пересечения множеств, выбор только тех элементов, которые присутствуют в обоих множествах.

Например, у нас стоит задача отобрать студентов, имеющих "особые" достижения в обучении для начисления стипендии имени УУУУУ. "Особыми" достижениями будем считать средний балл выше 4.5 + наличие научных публикаций.

За основу возьмем модель из раздела 6.1, дополнив ее сущностью *Papers* - публикации, а также, добавив ссылку на эту сущность в класс *Student*. Будем предполагать, что они связаны отношением один-ко-многим (один студент может опубликовать несколько статей, одна статья может иметь одного автора).

```
///
```

```

        public virtual Student Student { get; set; }
    }

    public class Student
    {
        //список публикаций студента:
        public virtual ICollection<Paper> Papers { get; set; }
        //...остальные поля класса студент...
    }

```

Имея такую модель данных, чтобы выбрать студентов, удовлетворяющих условию, определенному выше, мы должны сначала выбрать студентов, имеющих оценку выше 4.5, затем студентов, имеющих публикации и в качестве результата показать тех, кто присутствует в обоих множествах:

```

//выбираем всех студентов с оценкой выше 4.5 (Average возвращает среднее значение):
var goodStudents = Database.Students.Where(s => s.StudentExams.Average(se => se.Assessment) >= 4.5);
//выбираем студентов, имеющих публикации:
var studentsWithPapers = Database.Students.Where(s => s.Papers.Count != 0);
//выбираем студентов, присутствующих в обоих списках
var superStudents = goodStudents.Intersect(studentsWithPapers);

```

Пример запроса Intersect см. в "Ресурсы\Примеры типовых проектов\LinqIntersect"

2.1.10 Except

Разность множеств. Если даны два массива `var a = new[] {1, 2, 3, 4, 10, 20}; var b = new[] {0, 3, 7, 10, 20}`, то выполнение операции `a.Except(b)` даст множество `{1, 2, 4}`.

Решим задачу, определенную выше, используя Except. Для этого из множества студентов, имеющих хорошие оценки нужно вычесть тех, кто НЕ имеет публикаций:

```

//выбираем всех студентов с оценкой выше 4.5 (Average возвращает среднее значение):
var goodStudents = Database.Students.
    Where(s => s.StudentExams.Average(se => se.Assessment) >= 4.5);
//выбираем студентов, НЕ(!) имеющих публикации:
var studentsWithPapers = Database.Students.Where(s => s.Papers.Count == 0);
//выбираем студентов, присутствующих в первом, но отсутствующих во втором списке
var superStudents = goodStudents.Except(studentsWithPapers);

```

2.1.11 Distinct

Выборка уникальных значений. Применение этой функции к множеству `{1, 3, 4, 1, 2, 4}` оставит в нем `{1, 3, 4, 2}`, т.е. все дубликаты будут устранены.

Выберем всех студентов, сдавших хоть один экзамен на отлично:

```

var students = Database.StudentExams
    .Where(se => se.Assessment == 5)
    .Select( se => se.Student)
    .ToArray()

```

Если мы посмотрим результаты подобного запроса, то увидим, что некоторые имена повторяются. Это произошло по причине того, что один студент может сдать на отлично несколько предметов (см. рисунок 2.1).

27	Сергеев Сидор	Математика
27	Сергеев Сидор	Организация ЭВМ
28	Петриков Евгений	Программирование
28	Петриков Евгений	Организация ЭВМ
28	Петриков Евгений	БЖД
28	Петриков Евгений	Операционные системы
28	Петриков Евгений	Русский язык

Рис. 2.1: Пример дублирования данных

Чтобы это исключить, нужно использовать `Distinct`:

```
var students = Database.StudentExams
    .Where(se => se.Assessment == 5)
    .Select( se => se.Student)
    .Distinct()
    .ToArray()
```

Это исключит дублирование данных и приведет к результату, показанному на рисунке 2.2

18	Серегов Петр
19	Сергеев Петр
20	Петрикан Кирилл
21	Сидоров Евгений
22	Петрикан Никитос
23	Иванов Сергей
24	Сергеев Кирилл
25	Сергеев Петр
26	Петров Петр
27	Сергеев Сидор
28	Петриков Евгений
29	Серегов Кирилл

Рис. 2.2: Результат применения оператора `Distinct`

2.2 LINQ to Entities

При использовании LINQ запроса к источнику данных Entity Framework, он автоматически преобразуется в эквивалентный SQL запрос. Однако, т.к. LINQ спроектирован как универсальный язык запросов и позволяет использовать все возможности языка C#, не любой LINQ запрос может быть преобразован в SQL. Подробнее со списком поддерживаемых LINQ операторов можно ознакомиться в [5, 6]. Также будет полезным ознакомиться со списком функций .NET, которые имеют отображение в функции SQL, сделать это можно, например, в [7].

Кроме функций, общих для всех баз данных, можно также использовать специфические для конкретной БД, например, класс `SqlFunctions` предоставляет набор функций .NET, которые имеют прямое отображение на функции Microsoft Sql Server'a. Со списком функций можно ознакомиться в [8].

В крайних случаях можно сделать выборку из базы данных, используя "SQL совместимые операторы", а затем перейти к Linq to Objects, который поддерживает все возможности LINQ. Т.к. в данном случае данные будут обрабатываться на клиенте, нет нужды использовать LINQ to Entities, поэтому данный подход работает.

Глава 3

WPF (сделано частично)

Технология WPF (Windows Presentation Foundation) представляет собой подсистему для построения графических интерфейсов.

Если при создании традиционных приложений на основе WinForms за отрисовку элементов управления и графики отвечали такие части ОС Windows, как User32 и GDI+, то приложения WPF основаны на DirectX. В этом состоит ключевая особенность рендеринга графики в WPF: используя WPF, значительная часть работы по отрисовке графики, как простейших кнопочек, так и сложных 3D-моделей, ложиться на графический процессор на видеокарте, что также позволяет воспользоваться аппаратным ускорением графики.

Одной из важных особенностей является использование языка декларативной разметки интерфейса XAML, основанного на XML: вы можете создавать интерфейс, используя или декларативное объявление интерфейса, или код на управляемых языках C# и VB.NET, либо совмещать и то, и другое.

Преимущества WPF

- Возможность декларативного определения графического интерфейса с помощью специального языка разметки XAML, основанном на XML и представляющем альтернативу программному созданию графики и элементов управления, а также возможность комбинировать XAML и C#/VB.NET;
- Независимость от разрешения экрана: поскольку в WPF все элементы измеряются в независимых от устройства единицах, приложения на WPF легко масштабируются под разные экраны с разным разрешением;
- Новые возможности, которых сложно было достичь в WinForms, например, создание трехмерных моделей, привязка данных, использование таких элементов, как стили, шаблоны, темы и др.;
- Хорошее взаимодействие с WinForms, благодаря чему, например, в приложениях WPF можно использовать традиционные элементы управления из WinForms;
- Богатые возможности по созданию различных приложений: это и мультимедиа, и двухмерная и трехмерная графика, и богатый набор встроенных элементов управления, а также возможность самим создавать новые элементы, создание анимаций, привязка данных, стили, шаблоны, темы и многое другое;
- Аппаратное ускорение графики - вне зависимости от того, работаете ли вы с 2D или 3D, графикой или текстом, все компоненты приложения транслируются в объекты, понятные DirectX, и затем визуализируются с помощью процессора на видеокарте, что повышает производительность, делает графику более плавной;

- Создание приложений под множество ОС семейства Windows - от Windows XP до Windows 10.

В тоже время WPF имеет определенные ограничения. Стоит учитывать, что по сравнению с приложениями на Windows Forms объем программ на WPF и потребление ими памяти в процессе работы в среднем несколько выше. Но это компенсируется более широкими графическими возможностями и повышенной производительностью при отрисовке графики.

С основами WPF можно ознакомиться по ссылке: <https://metanit.com/sharp/wpf/> Для начала работы рекомендуется ознакомиться с основами компоновки и основными элементами управления.

Далее будет дан обзор основных механизмов, которые понадобятся для решения задачи, описанной в разделе 6.1.

3.1 Общее понятие биндинга (привязка данных)

Привязка данных WPF предоставляет простой путь для представления и отображения данных. Элементы управления могут быть непосредственно связаны с данными, взятыми из разных источников данных. Привязка данных подразумевает под собой процесс установления соединения между пользовательским интерфейсом и бизнес логикой. Если биндинг правильно настроен, данные предоставляют правильные уведомления, тогда, как только поле данных изменяет свое значение, это изменение автоматически отображается на всех связанных элементах управления. Привязка данных включает и обратную связь - если внешнее представление данных в элементе управления изменилось, базовые данные также автоматически обновятся, чтобы соответствовать этим изменениям.

Также, стоит отметить, что механизм привязки WPF поддерживает широкий спектр свойств (здесь и свойства, отвечающие за визуальное оформление, и контентные свойства, и даже возможность привязки команд - механизм, представляющий собой альтернативу событиям); предоставляет возможность гибкой настройки интерфейса пользователя для отображения данных, возможность создавать собственные шаблоны отображения сложных данных; дает возможность использовать разные источники данных для привязки, это может быть любой CLR объект, обладающий свойствами, включая ADO.NET объекты или объекты, ассоциированные с Web-сервисами, XML данными. Немаловажным является отделение бизнес-логики приложения от пользовательского интерфейса - в идеале, человек, разрабатывающий интерфейс может быть даже не знаком с языками программирования, достаточно согласовать имена объектов привязки [9].

Допустим, мы создаем форму для просмотра/редактирования какого-то объекта. Для решения этой задачи рассмотрим 2 подхода: с ручной загрузкой и выгрузкой данных и с использованием механизма привязки данных (биндинга).

Начнем с ручной загрузки данных. Для определенности, работать с объектом класса Student, описание которого представлено ниже:

```
class Student
{
    public string FirstName {get;set;}
    public string LastName {get;set;}
    //дата рождения:
    public DateTime BirthDay {get;set;}
}
```

Опишем форму на языке Xaml, позволяющую вывести эти данные:

```

<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="auto"/>
    <RowDefinition Height="auto"/>
    <RowDefinition Height="auto"/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width=".5*"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
  <TextBlock Text="Имя" Grid.Column="0" Grid.Row="0"/>
  <TextBlock Text="Фамилия" Grid.Column="0" Grid.Row="1"/>
  <TextBlock Text="Дата рождения" Grid.Column="0" Grid.Row="2"/>
  <TextBox x:Name="tbFirstName" Grid.Column="1" Grid.Row="0"/>
  <TextBox x:Name="tbLastName" Grid.Column="1" Grid.Row="1"/>
  <TextBox x:Name="tbBirthDate" Grid.Column="1" Grid.Row="2"/>
</Grid>

```

Как видно, всем элементам управления, с которыми нам предстоит работа, необходимо дать имя, в нашем случае это tbFirstName, tbLastName, tbBirthDate.

Для того, чтобы отобразить данные класса студент нужно выполнить следующие действия:

```

Student s = new Student {/*каким то образом заполняем поля класса*/};
//берем значения полей из объекта s и помещаем их в соответствующие
//элементы управления:
tbFirstName.Text = s.FirstName;
tbLastName.Text = s.LastName;
//т.к. дата рождения в памяти представлена не строкой, нужно выполнить преобразование
//вызвав метод ToString. У типа DateTime метод ToString имеет перегруженную версию,
//позволяя вывести дату в нужном нам формате (день, месяц, год):
tbBirthDate.Text = s.BirthDate.ToString("dd.MM.yyyy");

```

Загрузка данных с формы обратно в объект класса будет похожей:

```

//s - объект класса Student, см. код выше
s.FirstName = tbFirstName.Text;
s.LastName = tbLastName.Text;
s.BirthDay = DateTime.Parse(tbBirthDate.Text);

```

Получившийся код простой, однотипный, но его может быть достаточно много - для каждого поля класса необходимо прописать загрузку и выгрузку. К тому же, данный механизм подразумевает ручную обработку ошибок. WPF позволяет автоматизировать этот процесс с помощью механизма привязки (биндинга).

Привязка подразумевает взаимодействие двух объектов: источника и приемника. Объект-приемник создает привязку к определенному свойству объекта-источника. В случае модификации объекта-источника, свойство объекта-приемника также будет модифицировано. Например, в качестве объекта приемника может выступать TextBox со свойством Text, в качестве объекта-источника данных - объект типа Student. Это избавляет программиста от ручного считывания и записи данных, а также некоторых преобразований и проверок.

Биндинг задается из XAML разметки с использованием ключевого слова Binding для того свойства, которое мы хотим связать, после чего указывается свойство источника данных, с которым происходит связывание, например:

```
<TextBox Text = "{Binding Path=FirstName}"/>
```

Данный код свяжет значение свойства FirstName некоторого объекта с полем для ввода. Источник данных (объект, содержащий свойство FirstName), задается через свойство DataContext данного или родительского элемента управления.

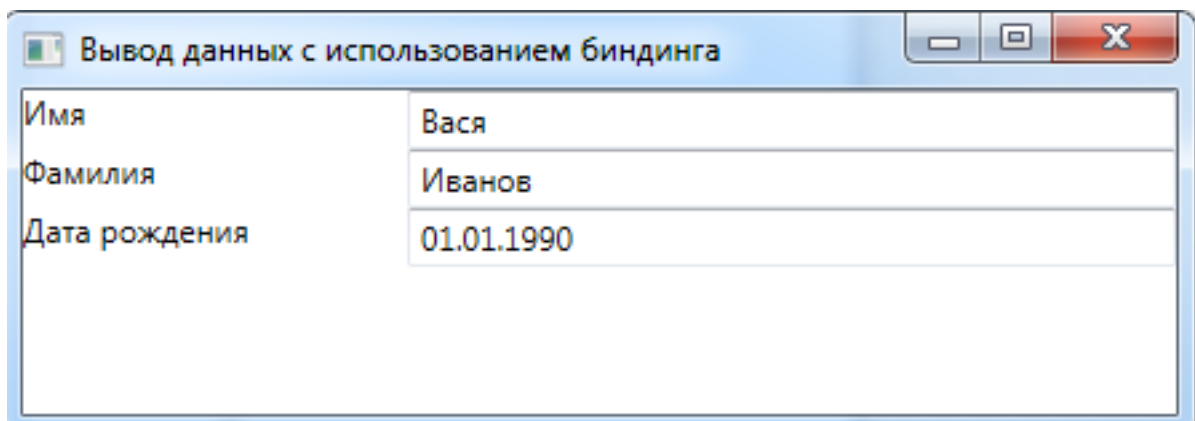
Решим задачу вывода данных, описанную выше с использованием биндинга. Для этого потребуется модифицировать разметку, чтобы указать привязки полей:

```
<Grid x:Name = "gridUserInfo">
  <Grid.RowDefinitions>
    <RowDefinition Height="auto"/>
    <RowDefinition Height="auto"/>
    <RowDefinition Height="auto"/>
    <RowDefinition Height="*/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width=".5*"/>
    <ColumnDefinition Width="*/>
  </Grid.ColumnDefinitions>
  <TextBlock Text="Имя" Grid.Column="0" Grid.Row="0"/>
  <TextBlock Text="Фамилия" Grid.Column="0" Grid.Row="1"/>
  <TextBlock Text="Дата рождения" Grid.Column="0" Grid.Row="2"/>
  <TextBox Text = "{Binding Path=FirstName}" Grid.Column="1" Grid.Row="0"/>
  <TextBox Text = "{Binding Path=LastName}" Grid.Column="1" Grid.Row="1"/>
  <TextBox Text = "{Binding Path=BirthDay, StringFormat=dd.MM.yyyy}" Grid.Column="1"
    Grid.Row="2"/>
  <ListBox ItemsSource = "{Binding Path=Papers}" Grid.Column="0" Grid.Row="3"
    Grid.ColumnSpan="2" Margin="5"/>
</Grid>
```

Это заставит форму "искать" поля FirstName, LastName, BirthDay в объекте DataContext. Осталось задать его в конструкторе формы, либо в любом другом месте, где необходимо загрузить данные:

```
Student s = new Student {/*каким то образом заполняем поля класса*/};
//gridUserInfo - имя Grid'а, в котором расположены поля ввода
gridUserInfo.DataContext = s;
```

При запуске формы мы увидим, что данные в поля ввода были выгружены автоматически:



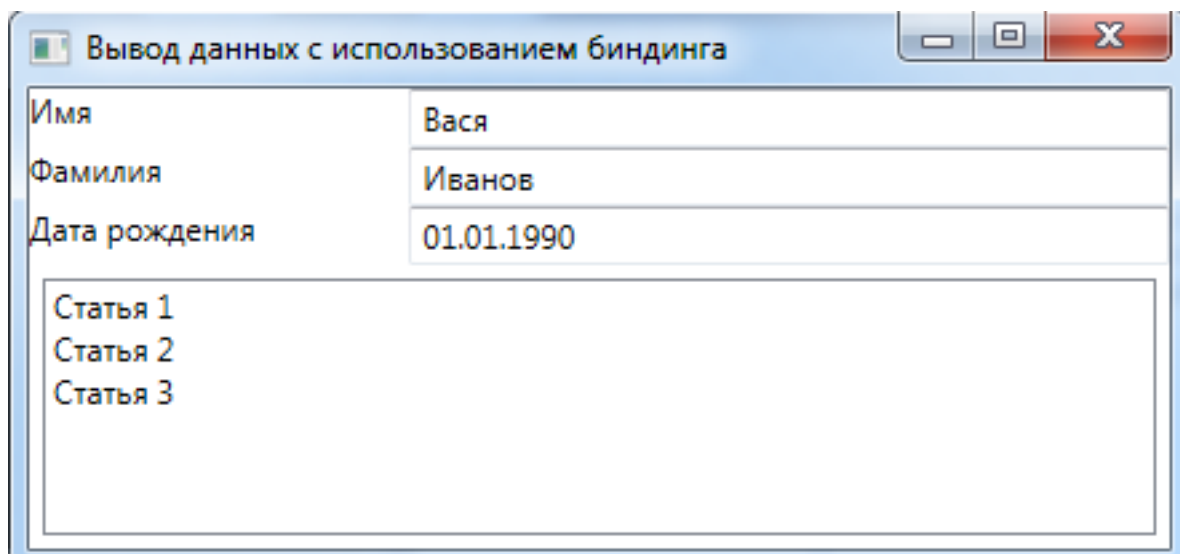
Имя	Вася
Фамилия	Иванов
Дата рождения	01.01.1990

Считывание значений с полей ввода в объект класса будет происходить автоматически при каждом изменении поля ввода, ничего дополнительно прописывать не нужно!

Для привязки множества объектов - коллекций, нужно использовать свойство `ItemsSource` элемента управления (`ListBox`, `ComboBox`, `DataGrid`,...), к которому привязывать объект класса, реализующий интерфейс `IEnumerable`. Модифицируем вышеприведенный пример для отображения списка `Papers` класса `UserInfo`, для этого, в разметка добавим `ListBox` и зададим привязку к свойству `Papers` (список публикаций):

```
<ListBox ItemsSource = "{Binding Path=Papers}"/>
```

После запуска получим форму, показанную ниже:



В данном примере `Papers` - список строк, поэтому механизм привязки понимает, что нужно выводить. Бывают ситуации, когда необходимо отобразить список сложных объектов, состоящих из нескольких полей. В этом случае, необходимо описать, как должен быть представлен, например, изображения следует выводить в `Image`, текст только для чтения в `TextBlock`'и, для редактирования - в `TextBox`'ы и т.д., про настройку отображения списков сложных объектов, см. раздел 3.7.2.

На рисунке 3.1 показана графическая схема биндинга. Как видно, в качестве контекста данных формы (свойство `DataContext`) установлен объект класса `Student`. `TextBox`'ы для вывода фамилии и имени привязаны к свойствам `LastName` и `FirstName` соответственно, которые берутся из объекта, на который указывает `DataContext`, т.е. в данном случае это поля объекта `Student`. Для вывода списка публикаций использован `DataGrid`, в котором для указания источника данных задействовано свойство `ItemsSource`, которое инициализировано свойством `Papers`. Стоит обратить внимание, что каждая строка таблицы получает свой собственный контекст данных, который ссылается на конкретный элемент коллекции `Papers`, поэтому, при поиска полей для связывания будут учитываться только поля объектов `Paper`, объект `Student` здесь уже недоступен.

Примеры данного раздела см. в "Ресурсы\Примеры типовых проектов\WpfBindingExample". Пример вывода данных без использования механизма привязки в проекте `NoBinding`, пример простого биндинга - проект `BindingExample`, пример биндинга с выводом и просмотром сложных объектов (на примере класса `Paper`, показанного на рисунке 3.1) - проект `BindingExample2`.

Как правило, для стандартных типов данных привязка является однонаправленной, т.е. данные будут передаваться с формы в объект класса, однако, если объект изменится извне, эти изменения не отобразятся на форме. Как это исправить, будет показано далее.

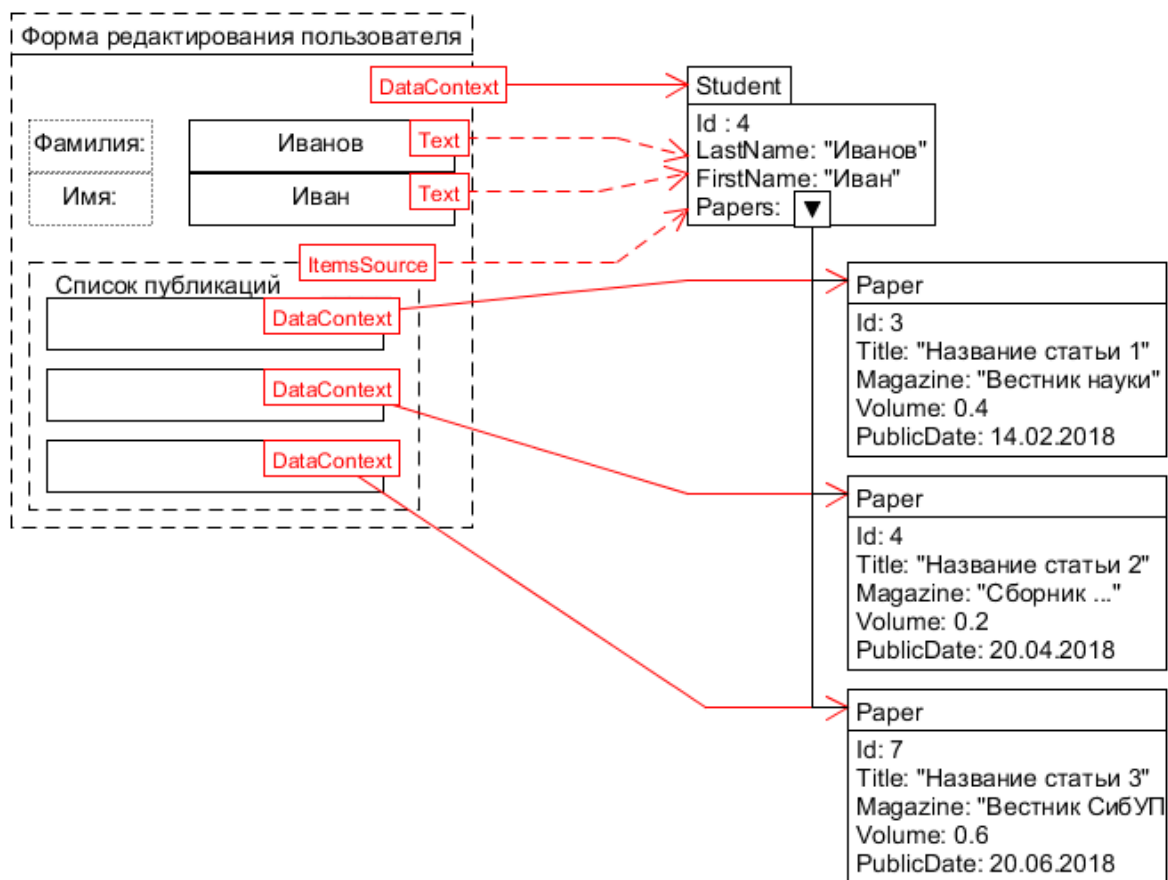


Рис. 3.1: Схема биндинга WPF

Примечание: В примере выше для указания источника данных было использовано свойство `DataContext`. Использование `DataContext`'а родительского элемента полезно, если вам необходимо забиндить несколько свойств одного объекта-источника данных (например фамилию, имя, отчество, дату рождения человека). Однако, иногда более подходящим является вариант указания отдельных источников для каждой привязки, для этих целей необходимо использовать свойство `Source`. Либо же вы можете создать привязку относительно другого элемента управления, в этом случае необходимо использовать свойства `ElementName` для указания имени элемента управления, и `RelativeSource` для указания источника привязки. В случаях, когда необходимо создать привязку и в качестве источника. Более подробную информацию можно получить в [9]

3.2 Привязка данных с использованием `INotifyPropertyChanged`

Чтобы изменить ситуацию, класс, содержащий данные, должен реализовать интерфейс `INotifyPropertyChanged`, либо привязываемые свойства должны быть реализованы как `DependencyProperty`. Более подробно см. <https://metanit.com/sharp/WPF/11.2.php> и <https://metanit.com/sharp/WPF/13.php>.

Рассмотрим первый вариант для укороченной версии класса `Student`, показанного выше:

```
//чтобы объект класса мог уведомить элемент управления о
//своем изменении, он должен быть наследником от INotifyPropertyChanged
```

```

//единственным членом этого интерфейса является событие:
//event PropertyChangedEventHandler PropertyChanged;
class Student : INotifyPropertyChanged
{
    //для хранения имени создаем приватную переменную:
    private string _firstName = String.Empty;
    //и переписывает свойство
    public string FirstName
    {
        //getter возвращает значение переменной
        get{ return _firstName; }
        set
        {
            //сеттер инициализирует переменную
            _firstName = value;
            //и генерирует событие об этом изменении
            OnPropertyChanged();
        }
    }
    //событие для уведомления об изменении значения свойств объекта
    public event PropertyChangedEventHandler PropertyChanged;
    //для упрощения работы с событием, создадим метод:
    void OnPropertyChanged( [CallerMemberName] string prop="")
    {
        if( PropertyChanged!=null) {
            PropertyChanged(this, new PropertyChangedEventArgs(prop));
        }
    }
}

```

По сути, каждый раз при изменении свойства генерируется событие, в роли подписчика этого события является элемент управления.

Реализация данного интерфейса поможет только при работе с одиночными объектами. Если же требуется отслеживать изменение содержимого коллекций, то они должны быть представлены как `ObservableCollection`, либо `CollectionViewSource`.

Подробнее с работой с этим интерфейсом вы познакомитесь в разделе 4 при реализации класса View-Model'и.

3.3 Представления наборов данных (Collection View)

В разделе 3.1 было показано, как с помощью механизма привязки отобразит данные какого-то объекта, включая коллекции. Здесь будет рассмотрен чуть более сложный, но более универсальный метод отображения коллекций.

В WPF в качестве источника данных для привязки вы можете использовать любую коллекцию, реализующую интерфейс `IEnumerable`. Однако, для того, чтобы отображать изменения коллекции, такие как добавление и удаление элементов, на UI, коллекция должна реализовать интерфейс `INotifyCollectionChanged`. Данный интерфейс предоставляет событие, которое должно возникать всякий раз, когда меняется базовая коллекция.

Для этих целей WPF предоставляет класс `ObservableCollection<T>`, который является встроенной реализацией коллекции данных, которая предоставляет интерфейс `INotifyCollectionChanged`. Обратите внимание, что для полной поддержки передачи данных в процесс биндинга, необходимо, чтобы каждый объект коллекции также реализовал интерфейс `INotifyPropertyChanged`. Это необходимо, чтобы не только изменение содержимого коллекции

отображалось на UI, но и изменение отдельных полей элементов коллекции также было синхронизировано с UI.

Как только вы связали ваш `ItemsControl` (любой элемент управления, предназначенный для вывода множества данных, например `ListBox`, `ComboBox`, `DataGrid`) с коллекцией данных, вы, возможно, захотите сортировать, фильтровать или группировать эти данные. Для того, чтобы сделать это, рекомендуется использовать специальные классы-представления коллекций (`Collection Views`), которые реализуют интерфейс `ICollectionView`.

`Collection View` - это дополнительный слой над источником данных, который позволяет выполнять сортировки, фильтрации, запросы группировки без изменения базовой коллекции. Кроме того, `Collection View` содержат указатель на текущий (выбранный) элемент коллекции. Если базовая коллекция реализует интерфейс `INotifyCollectionChanged`, то ее изменения также будут отображаться на ее представлении.

Пример работы с `CollectionView` показан в разделе 6.5.1. Там показано автоматическое генерация как приемника данных (имеется ввиду таблица на форме пользователя), так и источник данных - объекта `CollectionViewSource`. Однако, C# позволяет проделывать те же манипуляции, задавая источник данных вручную. Допустим, на форме размещен `DataGrid` с названием `dgStudents`, куда требуется вывести список студентов из контекста базы данных `App.DatabaseContext` (что такое контекст БД и как его использовать, см. раздел 1.4.2). Далее показаны несколько способов получения доступа к представлению данных. Каждый из них в большинстве случаев дают примерно равные возможности.

1. Использование `CollectionViewSource`. Данный способ предлагается Visual Studio при использовании объектов из вкладки источников данных (пример показан в разделе 6.5.1). Как правило применяется для задания источника данных из XAML разметки или в случае, если для одного набора данных нужно создать несколько представлений.

```
//объявляем переменную источник данных
CollectionViewSource studentsViewSource;
//инициализируем
studentViewSource = new CollectionViewSource();
//заполняем данными, в данном случае просто загружаем всех студентов
studentViewSource.Source = App.DatabaseContext.Students.ToList();
//и указываем в качестве источника данных для таблицы dgStudents представление данных:
dgStudents.ItemsSource = studentViewSource.View;
```

Как видно, несмотря на то, что работаем мы с объектом `CollectionViewSource`, в качестве источника данных для таблицы мы используем свойство `View` данного объекта, которое является представителем класса `CollectionView`.

2. Использование `CollectionView` по умолчанию. На самом деле, если вы не используете представление данных, WPF это делает за вас - для каждого источника данных, при использовании свойства `ItemsSource` создается объект типа `CollectionView`, который принимает набор данных и потом сам выступает источником данных.

Получить доступ к представлению данных можно с помощью вызова статического метода `GetDefaultView` класса `CollectionViewSource`:

```
//делаем выборку студентов
var students = App.DatabaseContext.Students.ToList();
//получаем представление данных по умолчанию для этого набора:
ICollectionView studentsView = CollectionViewSource.GetDefaultView(students);
dgStudents.ItemsSource = studentView;
```

Однако, использование `CollectionView` описанным способом имеет одну особенность, которую необходимо учитывать при разработке. Этот способ для одного и того же набора данных всегда создают одно и то же представление, поэтому, если вам нужно использовать разные фильтрации/сортировки/группировки над данными, то рассмотрите возможность использования `CollectionViewSource`.

3.4 Операции над данными

Во всех подразделах для определенности будем работать с объектом типа `ICollectionView`. Если вы работаете с `CollectionViewSource` его можно получить через свойство `View` данного класса (см. раздел 6.5.2), здесь же будем получать его через вызов метода `GetDefaultView` (см. пример выше).

3.4.1 Фильтрация

Для отбора данных, отвечающих некоторому критерию мы можем использовать LINQ запрос, либо, если данные уже отобраны и необходимо их дофильтровать на стороне приложения, можно использовать свойство `Filter` представления данных.

LINQ запросы рассмотрены ранее в разделе 2, поэтому здесь затрагиваться не будут. Если вы хотите отфильтровать с помощью LINQ запроса, просто создайте представление данных для результата запроса и установите его как источник данных для соответствующего элемента управления.

У объекта `CollectionView` есть свойство `Filter`, представляющее из себя делегат для ссылки на метод, возвращающий значение типа `bool` и принимающий параметр `object`. Данная функция вызывается для каждого элемента коллекции в тех случаях, когда необходимо обновить внешний вид коллекции, в качестве параметра у нее выступает объект, анализируемый в данный момент. В зависимости от реализуемой логики функция должна вернуть или `true`, если объект принимается, или `false`, если объект отбрасывается и не должен быть отображен.

Создадим представление и установим для него функцию фильтрации:

```
//настраиваем загрузку данных в обработчике события Loaded окна
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    //загружаем данные
    var students = App.DatabaseContext.Students.ToArray();
    var studentsView = CollectionViewSource.GetDefaultView(students);
    studentsView.Filter = FilterFunction;
}
//функция фильтрации
bool FilterFunction(object item)
{
    //item является объектом Student, преобразуем:
    var s = item as Student;
    //отбираем всех студентов с фамилией Петров
    if (s.LastName == "Петров") {
        return true;
    }
    return false;
}
```

Здесь показана простейшая функция, которая никак не зависит от внешних данных. В других условиях, вместо "Петров" вероятно, было бы значение из `TextBox`'а.

Фильтр может быть установлен только один, поэтому, если необходимо проверить объект на несколько условий, просто создайте функции проверки для каждого критерия, после чего вызовите их в функции фильтрации. Например, если бы было нужно выбрать всех Петровых из группы с Id = 1, то код фильтрации мог бы выглядеть следующим образом:

```
bool FilterFunction(object item)
{
    //item является объектом Student, преобразуем:
    var s = item as Student;
    //принимает объект только если он удовлетворяет обоим критериям
    if( FilterByLastName(s) && FilterByGroup(s)) {
        return true;
    }
    return false;
}
//фильтр по фамилии
bool FilterByLastName(Student s)
{
    if (s.LastName == "Петров") {
        return true;
    }
    return false;
}
//фильтр по группе
bool FilterByGroup(Student s)
{
    if (s.GroupId == 1) {
        return true;
    }
    return false;
}
```

В более сложных случаях, когда количество фильтров варьируется, можно собрать их в некоторую коллекцию, и потом "прогонять" проверяемый элемент по всем фильтрам этой коллекции:

```
//коллекция фильтров
ObservableCollection<Predicate<Student>> filters =
    new ObservableCollection<Predicate<Student>>();

//далее приведены 2 варианта фильтра, учитывающие коллекцию filters
//оба метода по сути делают одно и то же, различается только форма записи.

private bool StudentFilter(object o)
{
    var s = o as Student;
    //перебираем список фильтров
    for(int i = 0; i < filters.Count; i++) {
        //если хотя бы один из них вернул false, элемент должен быть
        // отброшен, поэтому возвращаем false
        if (filters[i](s) == false) {
            return false;
        }
    }
    return true;
}
```

```
private bool StudentFilter2(object o)
{
    var s = o as Student;
    //если хотя бы один из фильтров вернул false, элемент должен быть отброшен,
    //поэтому возвращаем false
    if (filters.Any(f => f(s) == false)) {
        return false;
    }
    return true;
}
```

Примеры данного раздела см. в "Ресурсы\Примеры типовых проектов\WpfCollectionViewFilter". Проекты SimpleFilter, SimpleMultiFilter, SimpleMultiFilter2.

Фильтрацию на основе взаимодействия с пользователем (вводимый текст, выбор элементов в ComboBox'ах) см. в разделе [6.5.2](#).

3.4.2 Сортировка

3.4.3 Группировка

3.5 Создание новых объектов, редактирование(не сделано)

3.6 Проверка (валидация) данных

ValidationRule, интерфейс IDataErrorInfo...

3.7 Разное

3.7.1 Направления биндинга

В WPF вы можете управлять направления работы биндинга, т.е. есть способ ограничить влияние источника или приемника данных на привязку. На рисунке [3.2](#) показаны возможные направления потоков данных при биндинге [\[9\]](#).

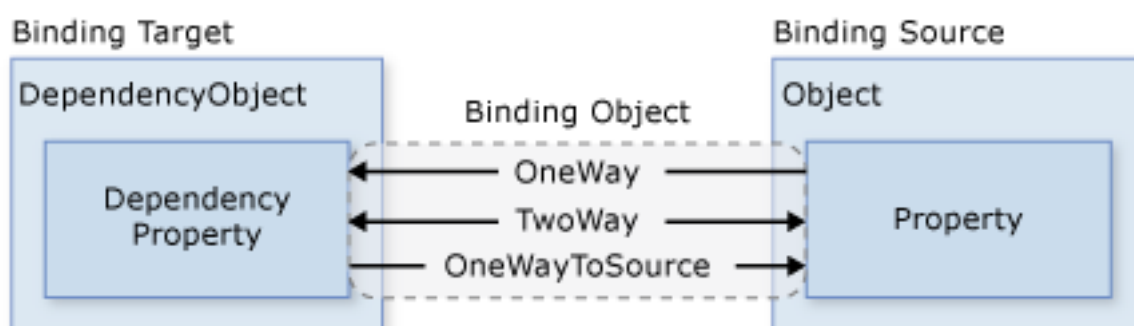


Рис. 3.2: Направления биндинга

Далее приведены краткие пояснения:

- **OneWay** - режим биндинга, при котором изменения свойства источника данных автоматически обновляют соответствующее свойство приемника, но изменения приемника не отражаются на содержимом источника. Этот тип биндинга соответствует элементам управления, доступным только для чтения. Если нет необходимости отслеживать изменения

целевого объекта, это поможет сократить накладные расходы по сравнению с двусторонним биндингом (TwoWay).

- TwoWay - двусторонний биндинг означает, что изменения источника и приемника влияют друг на друга и приводят к автоматическим изменениям связанных свойств.
- OneWayToSource - биндинг, обратный по смыслу OneToWay; свойство источника данных обновляется в случае изменения свойство объекта - приемника.
- OneTime - режим биндинга, при котором объект приемник получает свое значение один раз - при первой загрузке данных. Последующие изменения свойств источника и приемника не оказывают влияния друг на друга.

Для того, чтобы установить режим работы биндинга нужно в XAML разметке установить свойство Mode:

```
<!--устанавливаем однократную загрузку имени в TextBox-->
<TextBox Text = "{Binding Path=FirstName, Mode = OneTime}"/>
```

***Примечание:** Обратите внимание, чтобы изменения объекта источника были видны (для OneWay и TwoWay режимов), источник должен реализовать соответствующий механизм оповещения, такой как INotifyPropertyChanged (см. раздел 3.2).*

3.7.2 Настройка отображения элемента (DataTemplate – шаблон данных) для DataGrid, ListBox, ComboBox и т.п.

Для перечисленных элементов управления можно определить шаблон отображения данных. Это бывает необходимо в случаях, когда недостаточно стандартного текстового отображения элемента, и необходимо отобразить дополнительные данные, например несколько текстовых полей, изображение, и т.п. для каждого элемента.

Шаблон отображения задается в XAML разметке. Рассмотрим, каким способом можно определить шаблон данных, если требуется вывести поля FirstName, LastName, некоторого объекта а также кнопку редактирования:

```
<DataTemplate>
  <StackPanel Orientation="Vertical">
    <TextBlock Text="{Binding FirstName}"/>
    <TextBlock Text="{Binding LastName}"/>
    <Button Content="Edit"/>
  </StackPanel>
</DataTemplate>
```

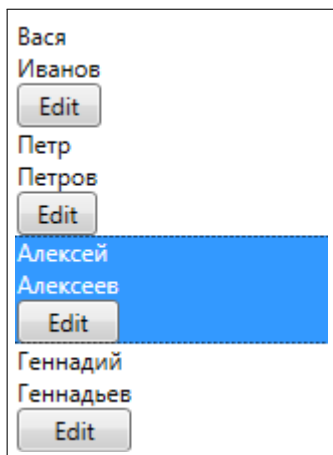
Как видно, шаблон данных DataTemplate представляет из себя обычный код разметки, использующий привязку к полям заданного объекта.

В таблице ниже приведены способы определения шаблона данных в разных элементах управления:

Таблица 3.1: Примеры определения шаблонов отображения списков

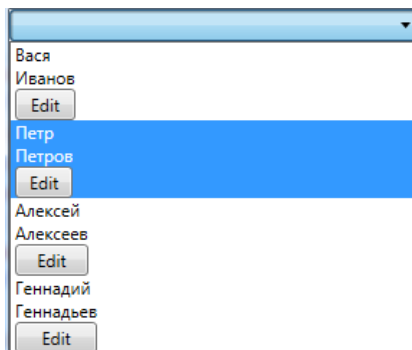
Элемент управления, внешний вид после применения шаблона	Код разметки для задания шаблона отображения
--	--

ListBox - простой список



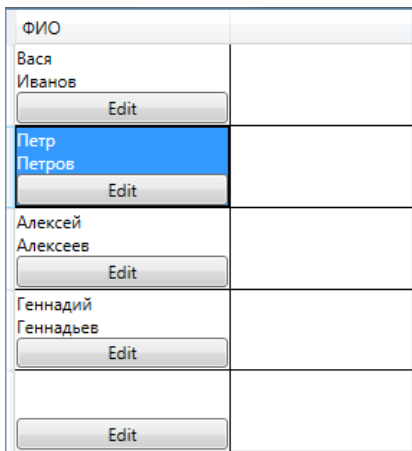
```
<ListBox x:Name="lbStudents">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Vertical">
        <TextBlock Text="{Binding FirstName}"/>
        <TextBlock Text="{Binding LastName}"/>
        <Button Content="Edit"/>
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

ComboBox - выпадающий список



```
<ComboBox x:Name="cbStudents">
  <ComboBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Vertical">
        <TextBlock Text="{Binding FirstName}"/>
        <TextBlock Text="{Binding LastName}"/>
        <Button Content="Edit"/>
      </StackPanel>
    </DataTemplate>
  </ComboBox.ItemTemplate>
</ComboBox>
```

DataGrid - таблица



```
<DataGrid x:Name="dgStudents" AutoGenerateColumns="False">
  <DataGrid.Columns>
    <DataGridTemplateColumn Header="Заголовк столбца">
      <DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
          <StackPanel Orientation="Vertical">
            <TextBlock Text="{Binding FirstName}"/>
            <TextBlock Text="{Binding LastName}"/>
            <Button Content="Edit"/>
          </StackPanel>
        </DataTemplate>
      </DataGridTemplateColumn.CellTemplate>
    </DataGridTemplateColumn>
  </DataGrid.Columns>
</DataGrid>
```

Если вам нужно использовать один шаблон для нескольких элементов управления (например на разных формах), вы можете определить их в ресурсах приложения. Это также позволяет определить шаблон по умолчанию для класса, который будет использоваться везде, где используются объекты этого класса. Для этого необходимо определить `DataTemplate` в ресурсах приложения. Для этого в файле `App.xaml` добавьте определение шаблона между тегами `<Application.Resources>` и `</Application.Resources>`. Чтобы в последующем мы могли

использовать данный шаблон мы должны задать ему имя через свойство **x:Key**, либо указать тип данных, для которого шаблон будет применяться через свойство **DataType**

```
<Application.Resources>
  <!--задаем шаблон для показа студента в списках, имя шаблона StudentItemTemplate-->
  <DataTemplate x:Key="StudentListDataTemplate">
    <StackPanel Orientation="Vertical">
      <TextBlock Text="{Binding FirstName}"/>
      <TextBlock Text="{Binding LastName}"/>
      <Button Content="Edit"/>
    </StackPanel>
  </DataTemplate>
  <!--можно задать шаблон по умолчанию для типа-->
  <!--он будет использован везде, где не переопределен шаблон данных-->
  <!--приставка local: перед указанием класса Student использована для
  указания того факта, что используется пользовательский класс, определенный в этой сборке-->
  <DataTemplate DataType="{x:Type local:Student}">
    <StackPanel Orientation="Horizontal">
      <TextBlock Text="{Binding LastName}" Margin="4"/>
      <TextBlock Text="{Binding FirstName}" Margin="4"/>
    </StackPanel>
  </DataTemplate>
</Application.Resources>
```

Далее, чтобы использовать шаблон в разметке достаточно указать его имя:

```
<!--задаем шаблон для отображения студентов для ListBox'a-->
<ListBox x:Name="lbStudents" ItemTemplate="{StaticResource StudentItemTemplate}"/>
```

Кроме того, можно задать шаблон данных для отображения одного элемента. Это применимо для элементов управления, имеющих свойство **Content**. Одним из наиболее применимых для этой задачи является **ContentControl**. Шаблон данных в этом случае определяется аналогично, в самом **ContentControl** для выбора шаблона используется свойство **ContentTemplate** (значение задается аналогично **ItemTemplate** из примера выше), если вы хотите выбрать шаблон, заданный по имени, либо не указывается ничего, если хотите использовать шаблон по умолчанию. Данные для **ContentControl**'а нужно загружать в свойство **Content**.

Примеры данного раздела см. в "Ресурсы\Примеры типовых проектов\WpfDataTemplateExample".

3.7.3 Значения по умолчанию при биндинге. **TargetNullValue**, **FallbackValue**

В некоторых случаях поля объекта, к которому происходит привязка могут быть не заполнены, либо, какие-то данные могли быть загружены с ошибкой. В этом случае мы можем задать некоторое значение по умолчанию. Например, наша система требует учета гражданства для всех обучающихся. В этом случае в класс "Студент" будет добавлено поле для хранения национальности:

```
class Student
{
    public int Id {get;set;}
    [Required]
    public string FirstName {get;set;}
    [Required]
```

```

    public string LastName {get;set;}
    ///<summary>национальность</summary>
    public string Nationality {get;set;}
}

```

Как видно, поля FirstName, LastName сделаны обязательными для заполнения с помощью атрибута Required. Для поля Nationality этот атрибут не указан, т.к. разработчик предположил, что, если национальность не указана, то следует считать, что это гражданин РФ.

В этом случае, в XAML разметке, в местах, где необходимо вывести гражданство следует указать TargetNullValue:

```

<TextBlock x:Name="tbNationality" Text="{Binding Nationality, TargetNullValue=Россия}"/>

```

***Примечание:** эту задачу можно было решить и по другому, например, написав новое свойство, которое возвращало бы значение по умолчанию.*

Кроме текстовых полей, значения по умолчанию могут получать и любые другие объекты, например, изображения. Если вы разрабатываете систему, ведущую учет сотрудников, и для каждого сотрудника может быть загружена фотография, следует рассмотреть использование этого механизма. В случае если у человека отсутствует фотография (либо файл изображения в настоящее время недоступен) может быть загружено какое-то изображение-заместитель. Пример см. “наРесурсы\Примеры типовых проектов\NotIncludedEmployeePhoto”

Также, это свойство может быть использовано для создания текстов-заместителей (т.н. Placeholder’ов) при редактировании объектов. В этом случае, если какое то поле данных не имеет своего значения, в качестве TargetNullValue можно задать текст следующего содержания: "Введите фамилию"или просто "Фамилия". См. пример “Ресурсы\Примеры типовых проектов\WpfBindingDefaultValues”

3.7.4 Фильтрация, сортировка “на лету”

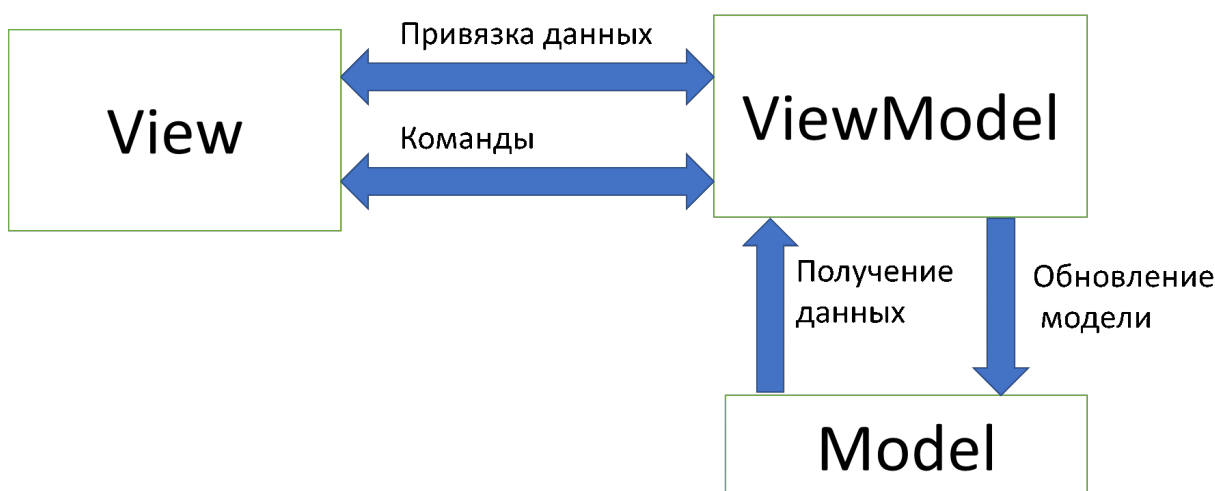
3.7.5 Конвертеры значений при биндинге

3.7.6 Мультибиндинг, мультиконвертеры

Глава 4

MVVM (не сделано)

4.1 Общие сведения



Model

- Модель описывает используемые в приложении данные. Модели могут содержать логику, непосредственно связанную этими данными, например, логику валидации свойств модели. В то же время модель не должна содержать никакой логики, связанной с отображением данных и взаимодействием с визуальными элементами управления.
- В объектно-ориентированных языках программирования модель данных представлена совокупностью взаимосвязанных объектов некоторых классов.
- Как правило, большинство приложений требуют хранения обрабатываемых данных, результатов вычислений. Для этих целей обычно используют БД.
- БД хранят данные и генерируют результаты запросов табличном виде.
- Возникает задача связки хранимых данных и объектной модели программы.

View

- Определяет визуальный интерфейс, через который пользователь взаимодействует с приложением. Применительно к WPF представление - это код в xaml, который определяет интерфейс в виде кнопок, текстовых полей и прочих визуальных элементов.

- В идеале, вся основная логика приложения выносится в компонент ViewModel, behind код представления должен оставаться чистым.
- Представление не обрабатывает события за редким исключением, а выполняет действия в основном посредством команд.

ViewModel

- Представляет собой отдельный класс, в котором содержится вся логика построения графического интерфейса и ссылка на модель, поэтому он выступает в качестве модели для представления.
- ViewModel или модель представления связывает модель и представление через механизм привязки данных, при этом не важно, как выглядит представление, какие элементы управления выбраны для отображения того или иного свойства. Если в модели изменяются значения свойств, при реализации моделью интерфейса INotifyPropertyChanged автоматически идет изменение отображаемых данных в представлении, хотя напрямую модель и представление не связаны.
- ViewModel также содержит логику по получению данных из модели, которые потом передаются в представление. И также ViewModel определяет логику по обновлению данных в модели.
- Элементы представления, то есть визуальные компоненты типа кнопок, не используют события, представление взаимодействует с ViewModel посредством команд.
- Например, пользователь хочет сохранить введенные в текстовое поле данные. Он нажимает на кнопку и тем самым отправляет команду во ViewModel. А ViewModel уже получает переданные данные и в соответствии с ними обновляет модель.
- Итогом применения паттерна MVVM является функциональное разделение приложения на три компонента, которые проще разрабатывать и тестировать, а также в дальнейшем модифицировать и поддерживать.

4.2 Описание модели данных (Model)

4.3 Описание представления (View)

4.4 Описание представления модели (View-Model)

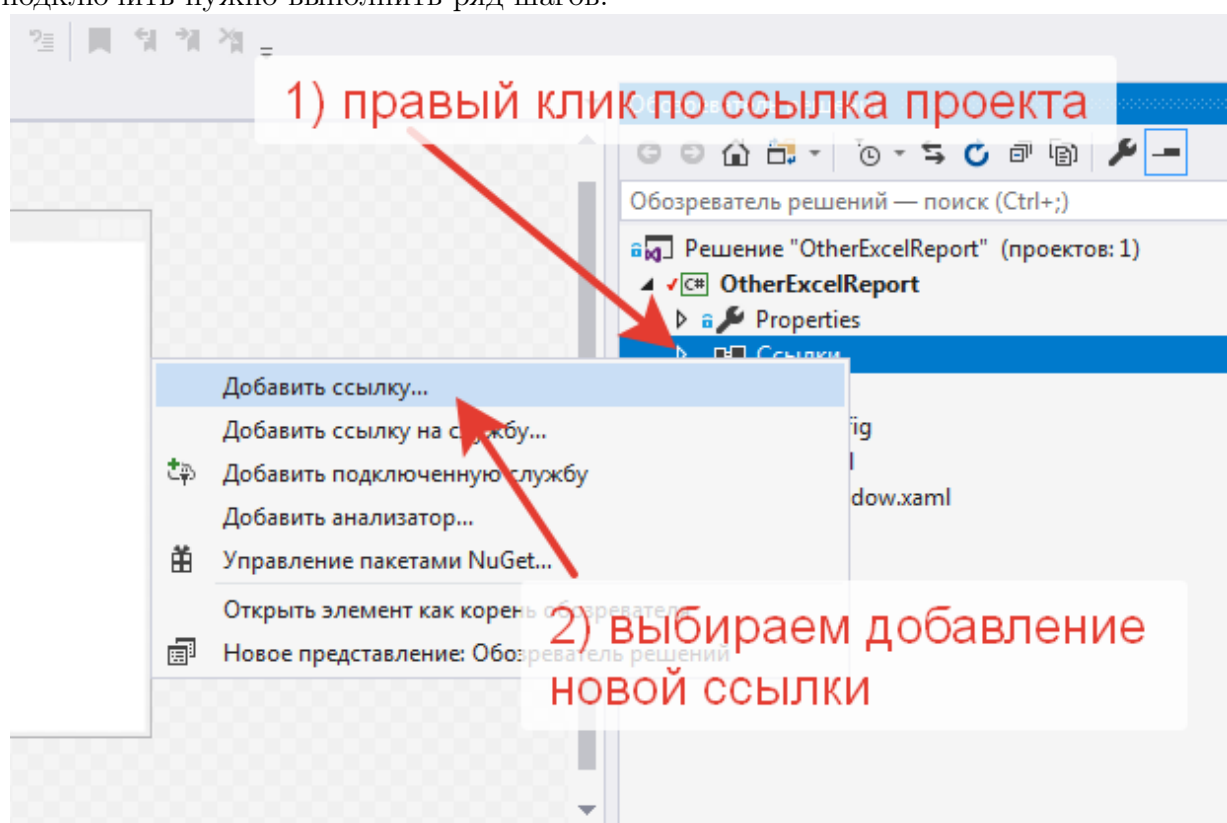
4.5 Добавление, редактирование данных

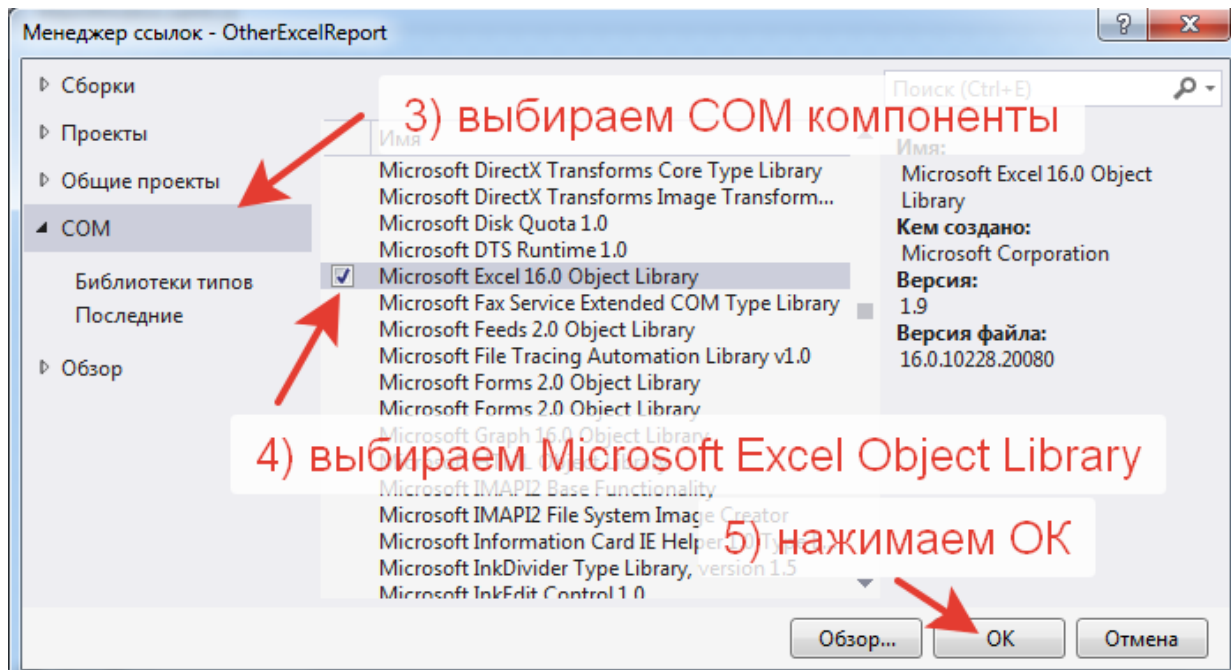
Глава 5

Разное

5.1 Формирование отчетов Excel

Для создания отчетов будем использовать библиотеку Microsoft Excel Object Library. Чтобы ее подключить нужно выполнить ряд шагов:





Версия библиотеки (в данном примере 16) будет зависеть от того, какая версия офиса установлена.

Теперь, чтобы начать использовать библиотеку в своем проекте, достаточно ее подключить, прописав в области using'ов следующую строку:

```
using Excel = Microsoft.Office.Interop.Excel;
```

Далее покажем создание минимального документа, включающего наиболее часто встречающиеся действия:

```
//указываем, что будем использовать библиотеку Microsoft Excel Object Library
//под псевдонимом Excel:
using Excel = Microsoft.Office.Interop.Excel;

//формируем документ, например, по нажатию на кнопку:
private void MakeReport_Click(object sender, RoutedEventArgs e)
{
    // Создаём экземпляр Excel
    Excel.Application excelApp = new Excel.Application();
    // В запущенном экземпляре Excel добавляем рабочую книгу Excel
    Excel.Workbook workBook = excelApp.Workbooks.Add();
    // Получаем первый лист книги Excel
    Excel.Worksheet workSheet = (Excel.Worksheet)workBook.Worksheets.get_Item(1);
    //далее работа сводится к заданию значений ячеек
    //обратиться к ячейке мы можем двумя способами:
    //1) workSheet.Cells[номер_строки, номер_столбца],
    //   например, следующий код запишет "Привет" в ячейку B1:
    //   workSheet.Cells[1, 2] = "Привет";
    //2) другой способ - через явное задание диапазона:
    //   workSheet.Range["B1"] = "Привет";
    //   используя Range можно получать не только одну ячейку, но и выделять
    //   для обработки сразу несколько: workSheet.Range["B2", "B5"]

    //формируем заголовок, для этого объединяем верхние ячейки:
    Excel.Range header = workSheet.Range["A1", "I1"];
    header.Merge();
    //и записываем туда какой то текст:
    header.Value = "Бесполезный документ";
}
```

```

//задаем выравнивание по центру и жирный шрифт:
header.HorizontalAlignment = Excel.Constants.xlCenter;
header.Font.Bold = true;

//покажем использование формул, для этого вторую строку заполним
//случайными числами, а на третьей посчитаем среднее значение
Random rnd = new Random();
for (int j = 1; j < 10; j++) {
    workSheet.Cells[2, j] = rnd.Next(1,10);
}
Excel.Range avgCell = workSheet.Cells[3, 1];
avgCell.Formula = "=CP3HA4(A2:I2)";
avgCell.FormulaHidden = false;
//оформим тонкую границу вокруг каждого из случайных чисел:
for (int j = 1; j < 10; j++) {
    Excel.Range cell = workSheet.Cells[2, j];
    cell.BorderAround2();
}
//а весь документ возьмем в жирную рамку:
Excel.Range docRange = workSheet.Range["A1", "I3"];
docRange.BorderAround2(Weight : Excel.XlBorderWeight.xlThick);

//построим график типа "столбчатая диаграмма"
Excel.ChartObjects chartObjs = (Excel.ChartObjects)workSheet.ChartObjects();
//рассчитаем координаты создаваемого графика
//график будет располагаться в диапазоне 4-10 строки,1-9 столбцы
//берем заданный диапазон
Excel.Range chartRange = workSheet.Range["A4", "I10"];
//и размещаем график по координатам этого диапазона:
Excel.ChartObject chartObj = chartObjs.Add( chartRange.Left, chartRange.Top,
    chartRange.Width, chartRange.Height);
Excel.Chart xlChart = chartObj.Chart;
// Устанавливаем тип диаграммы
xlChart.ChartType = Excel.XlChartType.xlColumnClustered;
//указываем источник данных:
xlChart.SetSourceData(workSheet.Range["A2", "I2"]);
// Открываем созданный excel-файл
excelApp.Visible = true;
//или, например, можно сразу отправить на печать:
//workSheet.PrintOutEx();
}

```

Выполнение данного кода приведет к генерации документа, показанного на рисунке 5.1.



Рис. 5.1: Сгенерированный документ Excel

Пример работы с Excel см. в "Ресурсы\Примеры типовых проектов\OtherExcelReport".

5.2 Создание одностраничной навигации

WPF поддерживает переходы в стиле веб-браузера. В страничных приложениях содержимое страниц встраивается в специальный каркас, в который загружается какой то контент. При этом имеется поддержка переходов, включая журнал навигации.

Страничную навигацию в WPF можно осуществить с помощью элемента управления Frame, либо с помощью NavigationWindow, который должен быть установлен как базовый класс для окна. Оба этих компонента в качестве содержимого позволяют загружать в себя произвольные страницы - объекты WPF типа Page. Рассмотрим первый вариант (размещение страниц внутри фрейма). Для этого создадим приложение, содержащее активную центральную область, возможность перехода назад, как показано на рисунке ниже:

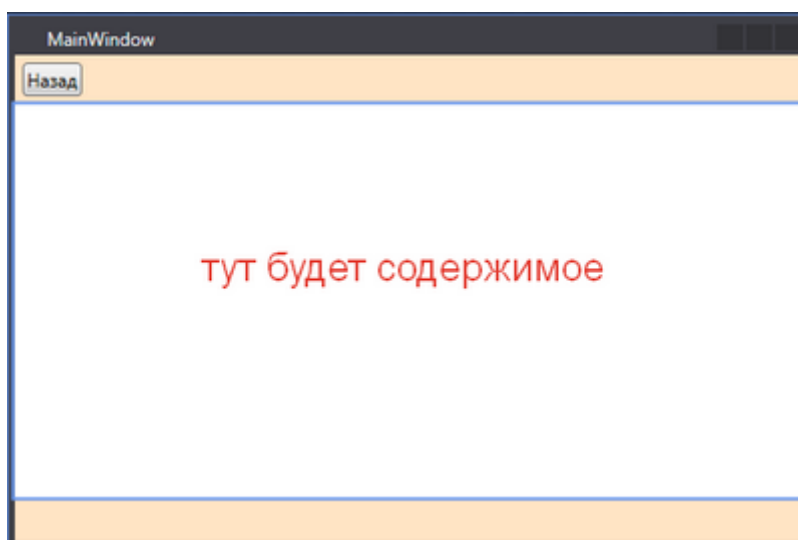


Рис. 5.2: Внешний вид проектируемого интерфейса

Пример разметки для такого интерфейса приведен ниже:

```
<DockPanel>
  <!--Наверху создаем стек панель для размещения кнопки "Назад"-->
```



```

<StackPanel DockPanel.Dock="Top" Background="Bisque">
    <Button Content="Назад" Margin="5" HorizontalAlignment="Left" Click="Button_Click"/>
</StackPanel>
<!--Панель внизу окна. Пока не используется,
но этим мы резервируем пространство на будущее-->
<StackPanel DockPanel.Dock="Bottom" Background="Bisque" Height="30"/>
<!--Фрейм для загрузки контента. Т.к. с ним будет идти
работа из C#, даем ему имя-->
<Frame x:Name="ContentFrame" NavigationUIVisibility="Hidden" Source="MenuPage.xaml"/>
</DockPanel>

```

Для фрейма необходимо установить свойство `ShowsNavigationUI="False"` иначе будет показан стандартный интерфейс навигации, который может не подойти вашему приложению.

Добавим страницу в проект (Правый клик по проекту -> Добавить -> Окно), которая будет содержать, например, меню нашего приложения, назовем ее `MenuPage.xaml`. Внешний вид страницы представлен на рисунке 5.3.

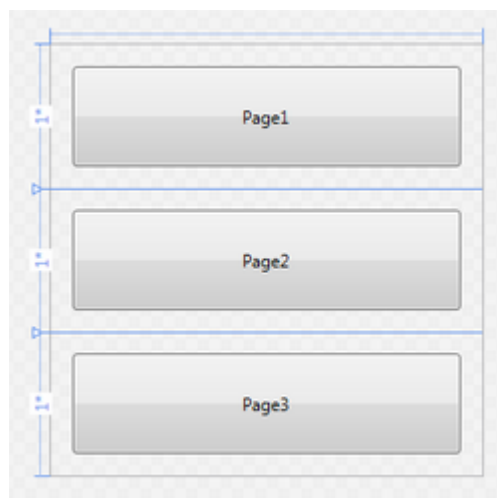


Рис. 5.3: Главное меню проектируемого интерфейса

Чтобы добиться такого вида, нужно в режиме разметки прописать следующий код:

```

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Content="Page1" Grid.Row="0" Margin="15"/>
    <Button Content="Page2" Grid.Row="1" Margin="15"/>
    <Button Content="Page3" Grid.Row="2" Margin="15"/>
</Grid>

```

Для фрейма на главной форме установим свойство `Source` равным `MenuPage.xaml` (расширение `.xaml` нужно указывать!), это сделает данное меню страницей по умолчанию.

Создадим контентные страницы `Page1`, `Page2`, `Page3`, на которые будет осуществляться переход с `MenuPage`.

Для смены активного содержимого страницы необходимо работать со свойством `NavigationService` класса `Frame`. Так, для перехода на `Page1` в обработчике кнопки следует прописать следующий код:

```
NavigationService.Navigate(new Page1());
```

В данном случае код прописывается внутри страницы, которая содержит ссылку на `NavigationService`, поэтому ссылку на родительский фрейм указывать не нужно. Для кнопки «Назад», которая располагается на главной форме, нужно дополнительно указать фрейм, для которого мы производим действие:

```
ContentFrame.NavigationService.GoBack();
```

Пример работы выложен в папке “Ресурсы\Примеры типовых проектов\OtherSinglePageNavigation”.

***Примечание:** Для этих же целей можно использовать класс `NavigationWindow` как базовый для стартового окна. Поведение данного типа сходно с `Frame`, т.е. включает в себя `NavigationService` и позволяет устанавливать в качестве своего содержимого некоторые страницы*

5.3 Перехват необработанных исключений

Рассказать про `UnhandledException`.

5.4 Невошедшее (не сделано)

Ряд задач не был рассмотрен, т.к. в данном пособии ставились другие цели, однако, некоторые из них могут быть полезны. Поэтому, следующие задачи, снабженные комментариями размещены в папке “Ресурсы\Примеры типовых проектов”:

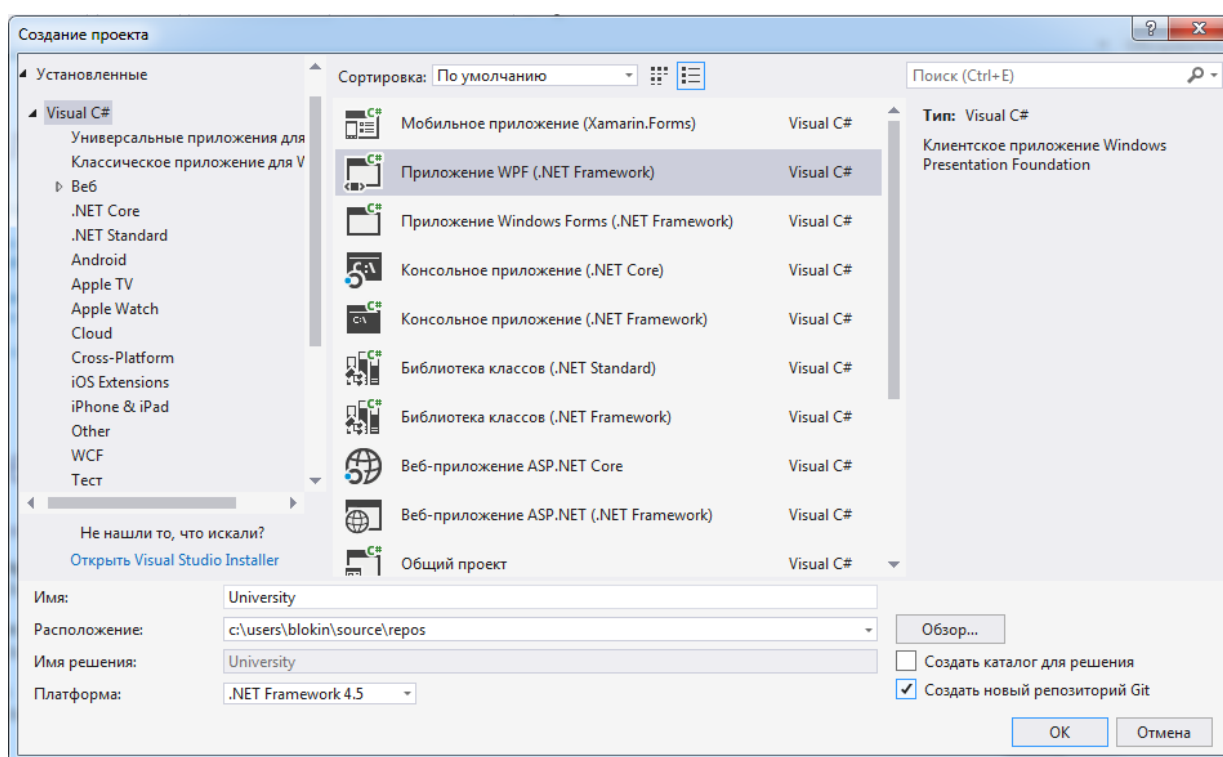
- Drag and drop - механизм, позволяющий перемещать объекты между компонентами формы с помощью мыши. Чтобы ознакомиться с возможной реализацией см. проект `OtherDragAndDrop`;
- Работа с изображениями, хранимыми в базе данных. Рассмотрены способ при котором в БД хранится расположение картинки, а само изображение лежит отдельно. Показан способ загрузки изображения для просмотра, добавление новых, замены существующих, загрузки изображения по умолчанию, если в БД отсутствует нужное значение. См. проект `OtherEmployeePhoto`. Для полного понимания проекта рекомендуется ознакомиться с разделом 3.2. Данный проект можно улучшить, если при загрузке формировать миниатюры и на предпросмотр выводить не отмасштабированные оригинальные изображения, а миниатюры;
- Привязка горячих клавиш - см. проект `OtherKeyBindings`. Все действия производятся в файле разметки `MainWindow.xaml` (при желании часть кода можно сделать общим для всех форм, вынеся его в `App.xaml`).

Глава 6

Практический пример (не сделано)

Все нижеприведенные действия были выполнены в Visual Studio 2017 с установленными компонентами "Разработка классических приложений .NET "Хранение и обработка данных".

Для начала работы создадим проект типа "Приложение WPF":



6.1 Описание задачи

База данных должна содержать сведения о следующих объектах:

Студент - фамилия, имя, отчество, адрес, дата рождения, группа, семестровая успеваемость (курс, семестр, предмет, оценка)

Необходимо разработать функционал для ведения учета студентов. Разрабатываемая программа должна позволять выполнять следующие действия:

- вести учет студентов;
- создать различные выборки по студентам (фильтрация по имени, группе, успеваемость) с возможностью сортировки результата;

- предоставлять возможность выгрузки вышеприведенных отчетов в excel файл.

Для описания предметной области будет использоваться модель, показанная на рисунке 6.1. Стоит отметить, что данная модель является урезанной, например, если при разработке полноценной системы, следовало бы указать связь между сущностями группа и экзамен, т.к. экзамены планируются в соответствии с некоторым учебным планом на группу, но это повлекло бы за собой реализацию дочерних сущностей и в итоге пример стал бы более громоздким и сложным для восприятия.

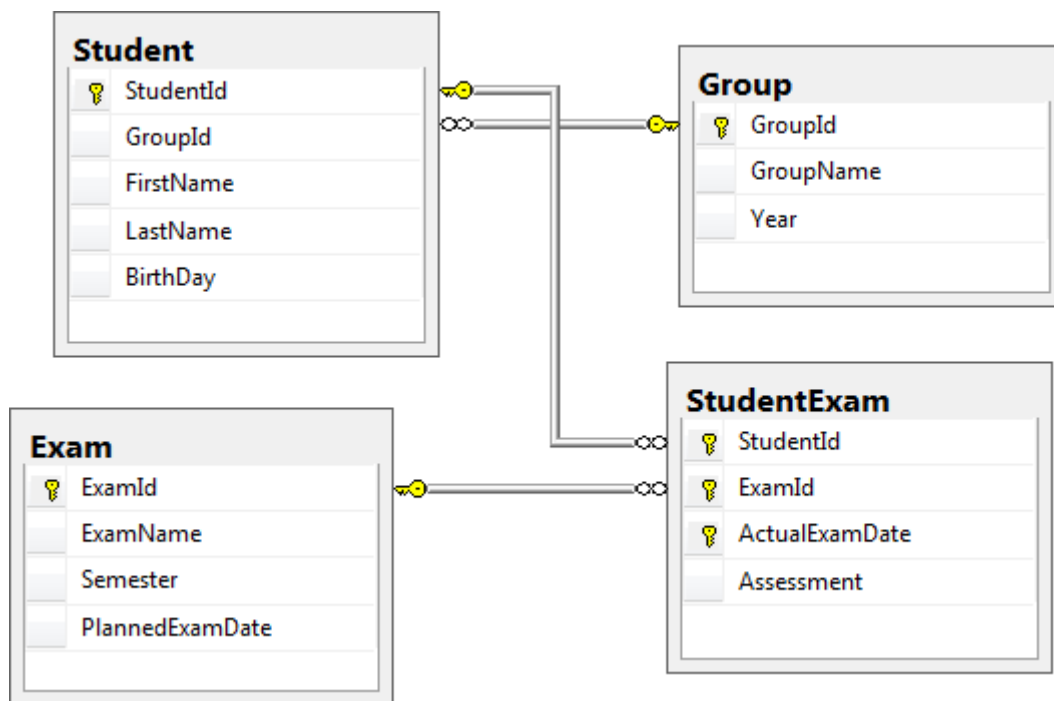
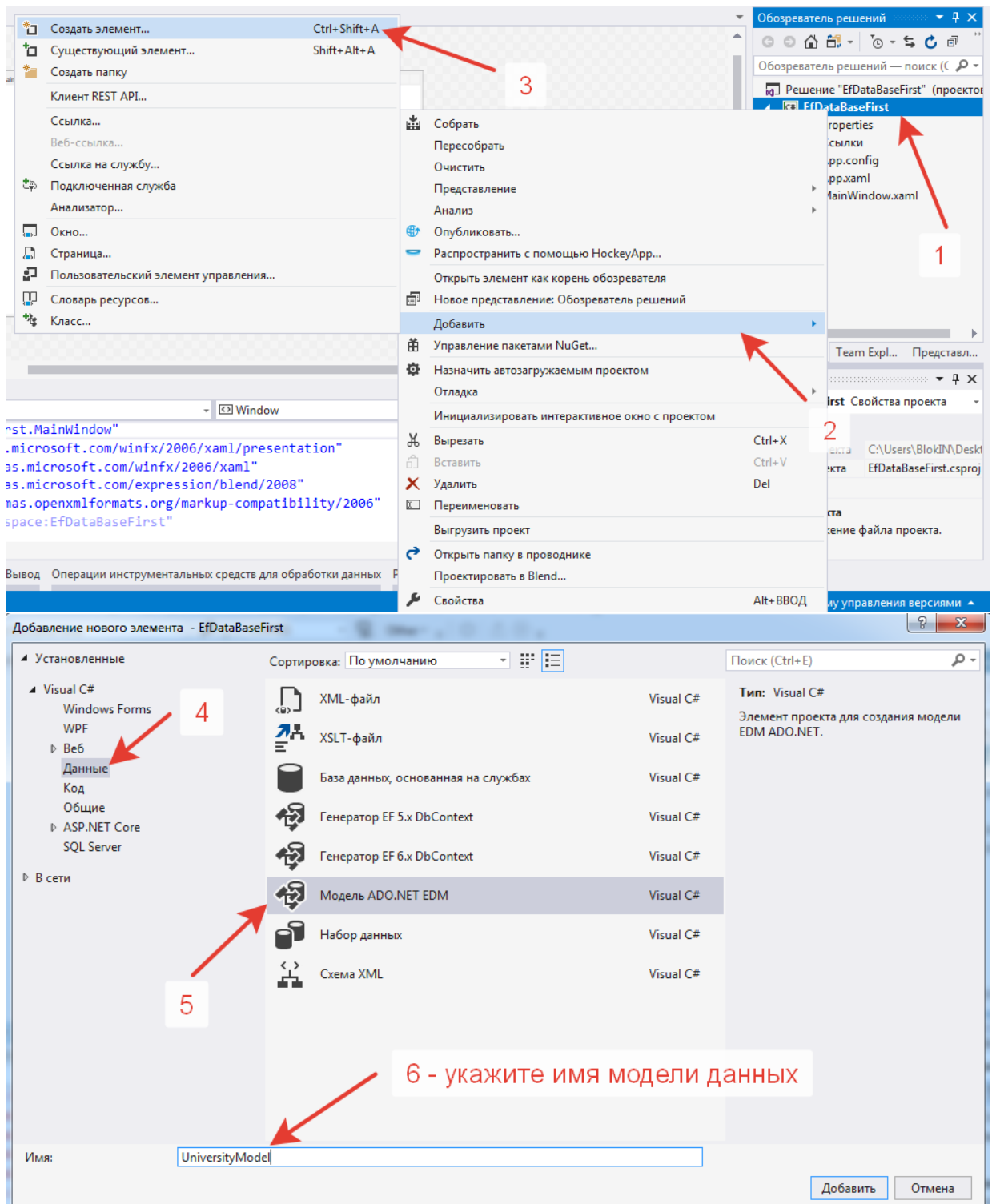
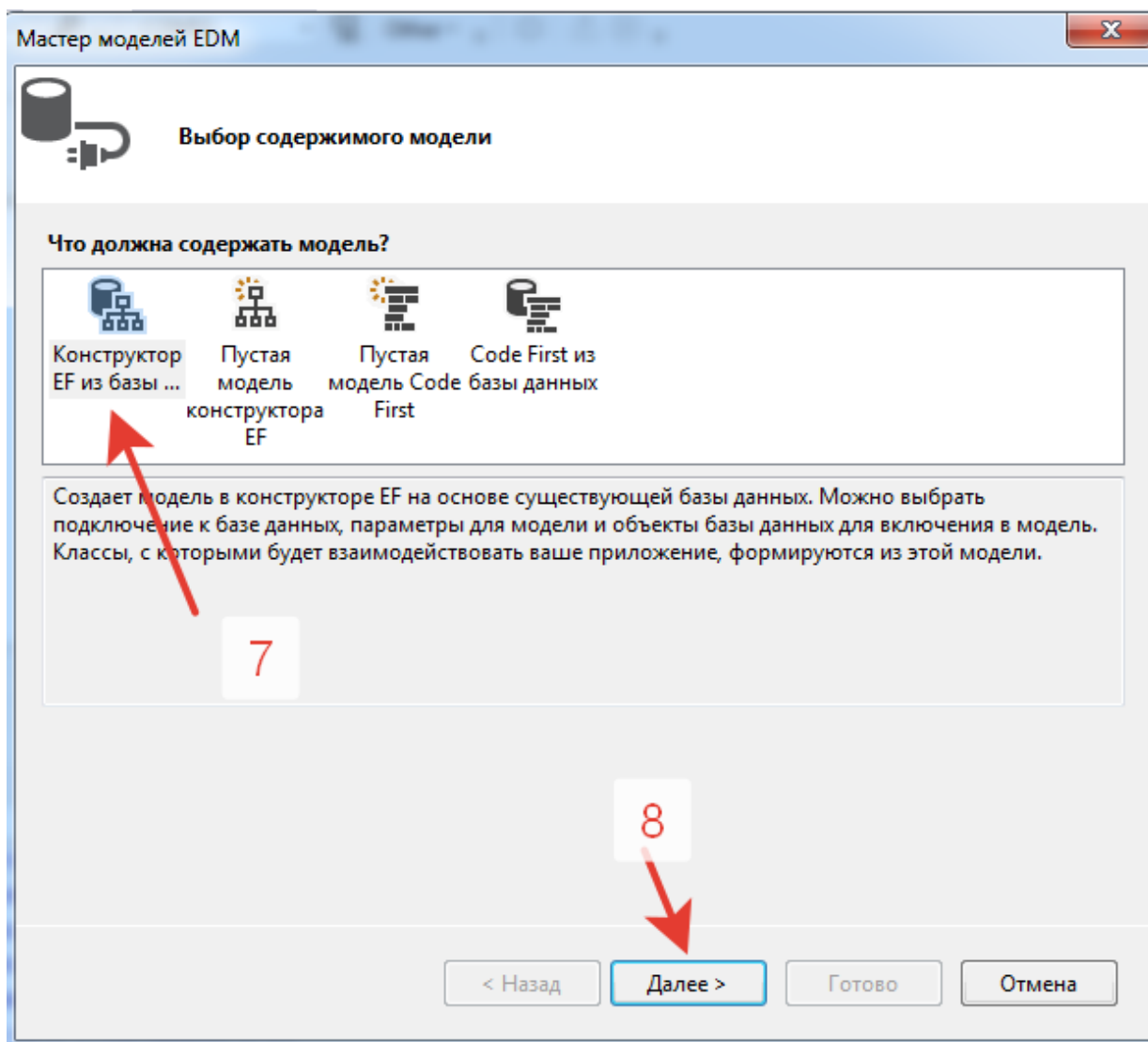


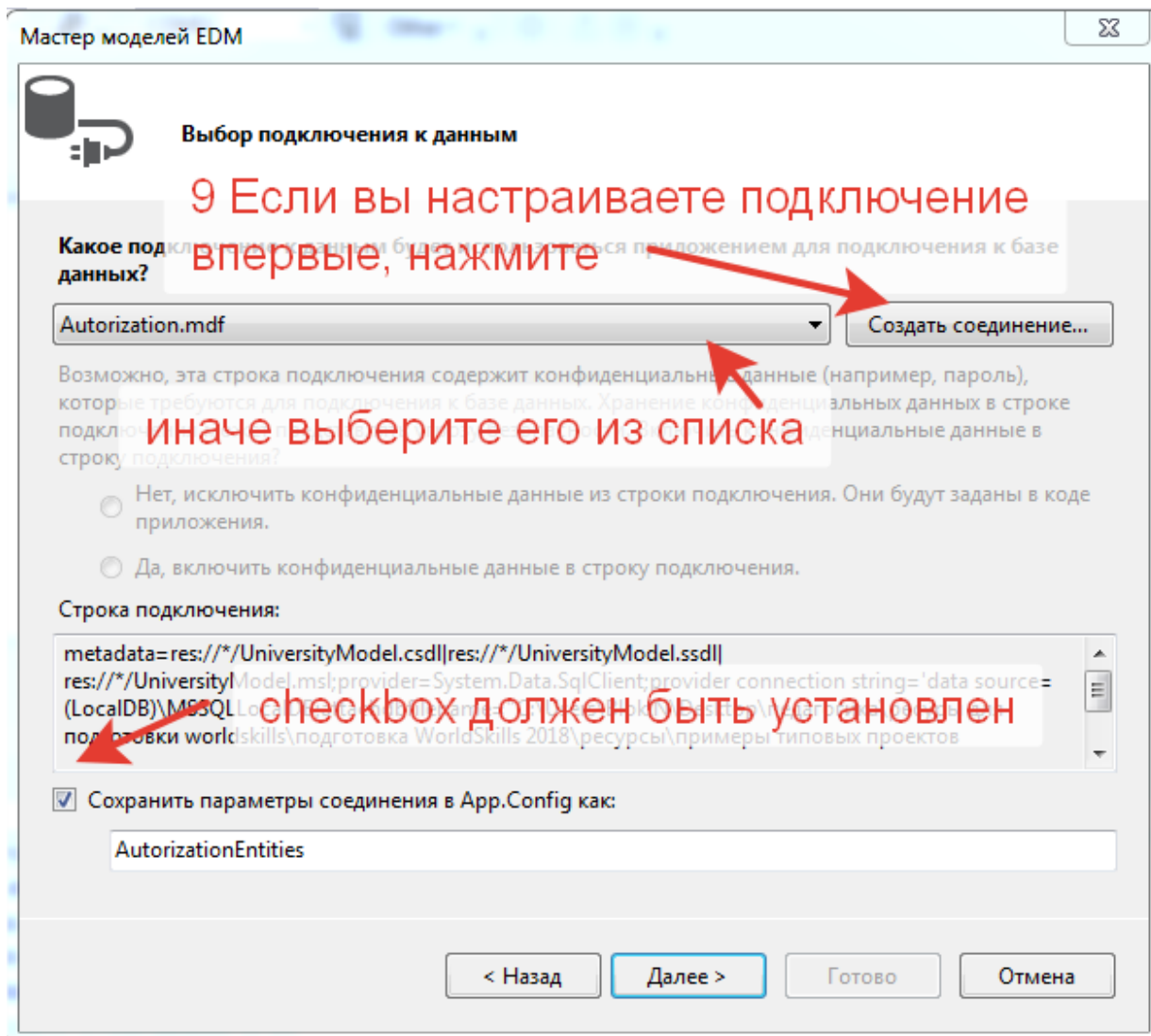
Рис. 6.1: Модель предметной области

6.2 Создание модели с использованием подхода Database First

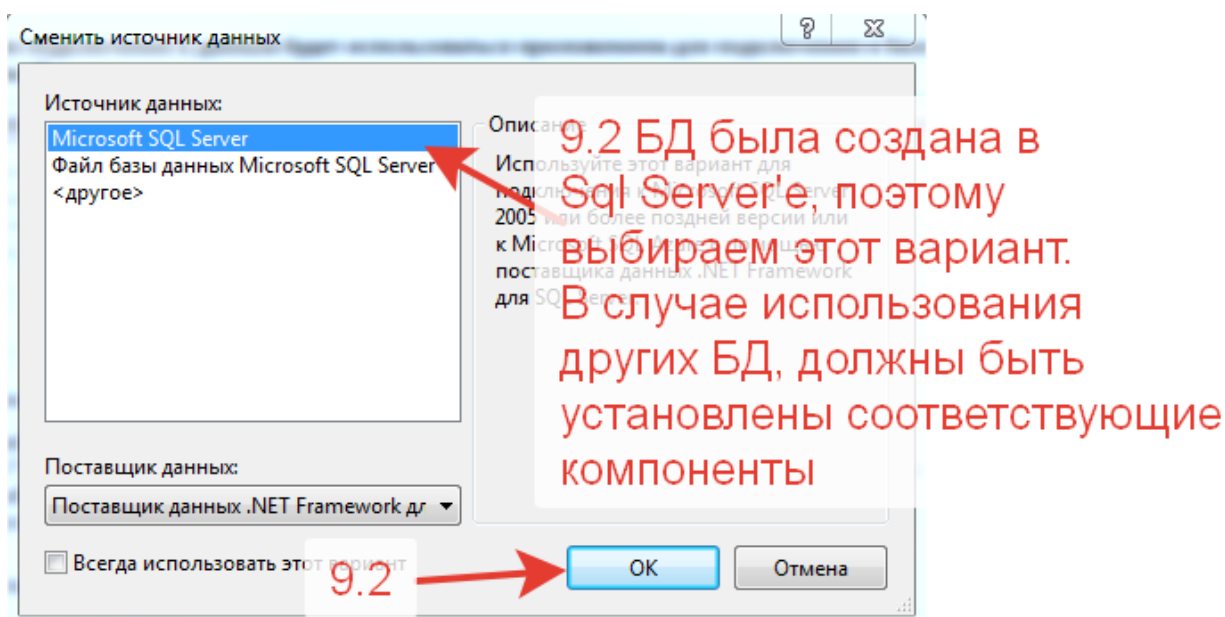
Чтобы использовать данный подход, требуется выполнить следующие действия:







Покажем создание соединения для базы, описанной в разделе 6.1.



Вид следующего окна зависит от конкретной СУБД, для MS SQL Server'a окно настроек выглядит следующим образом:

Свойства подключения

Введите данные для подключения. Если вы не знаете, как ввести данные, нажмите кнопку "Изменить", чтобы выбрать другой источник данных и (или) поставщик.

Источник данных: Microsoft SQL Server (SqlClient) Изменить...

Имя сервера: BLOKIN-ПК\SQLEXPRESS Обновить

Вход на сервер

Проверка подлинности: Проверка подлинности Windows

Имя пользователя:

Пароль:

Подключение к базе данных

☒ Выберите или введите имя базы данных: University

☐ Прикрепить файл базы данных: Обзор...

Логическое имя:

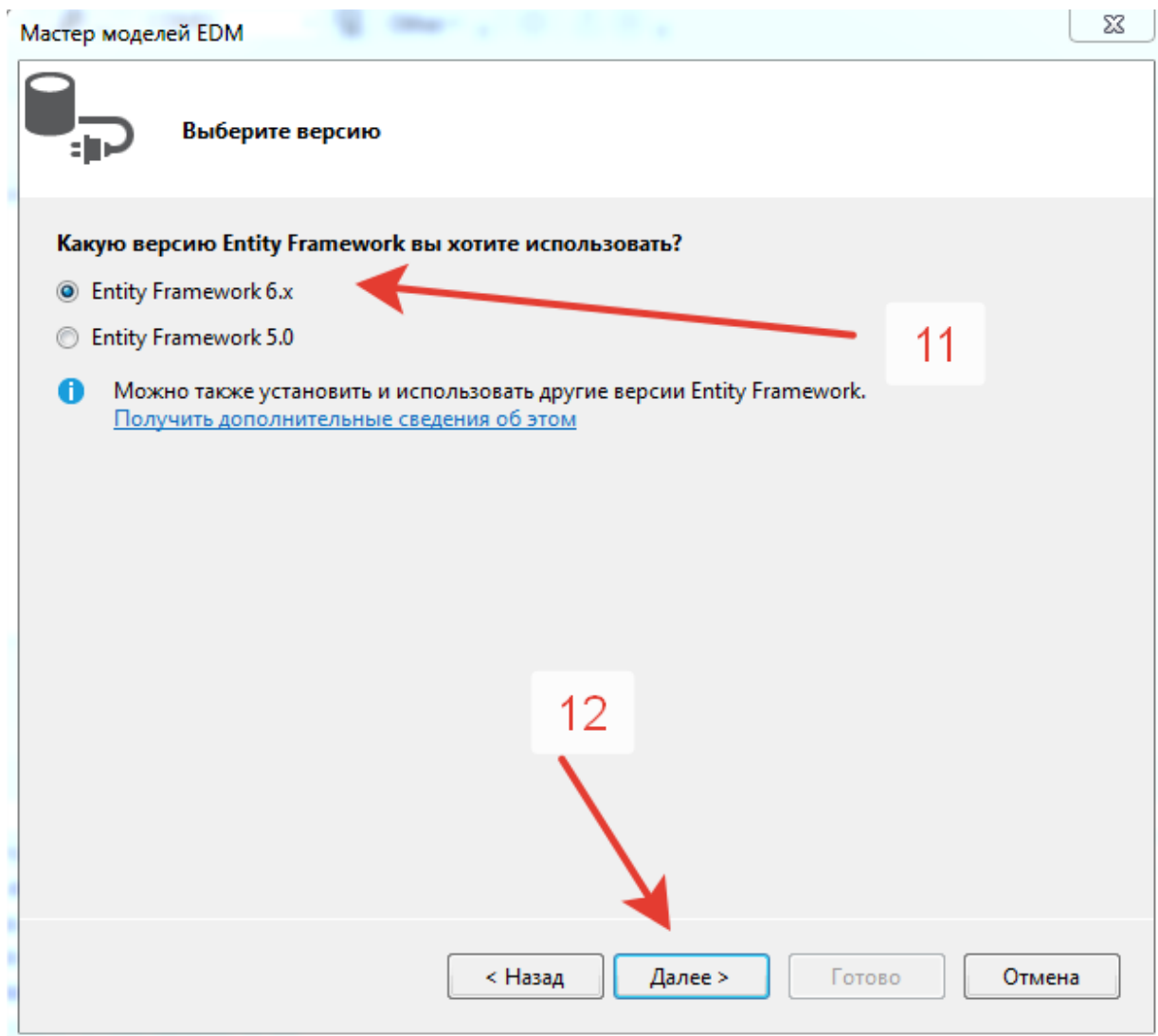
Дополнительно...

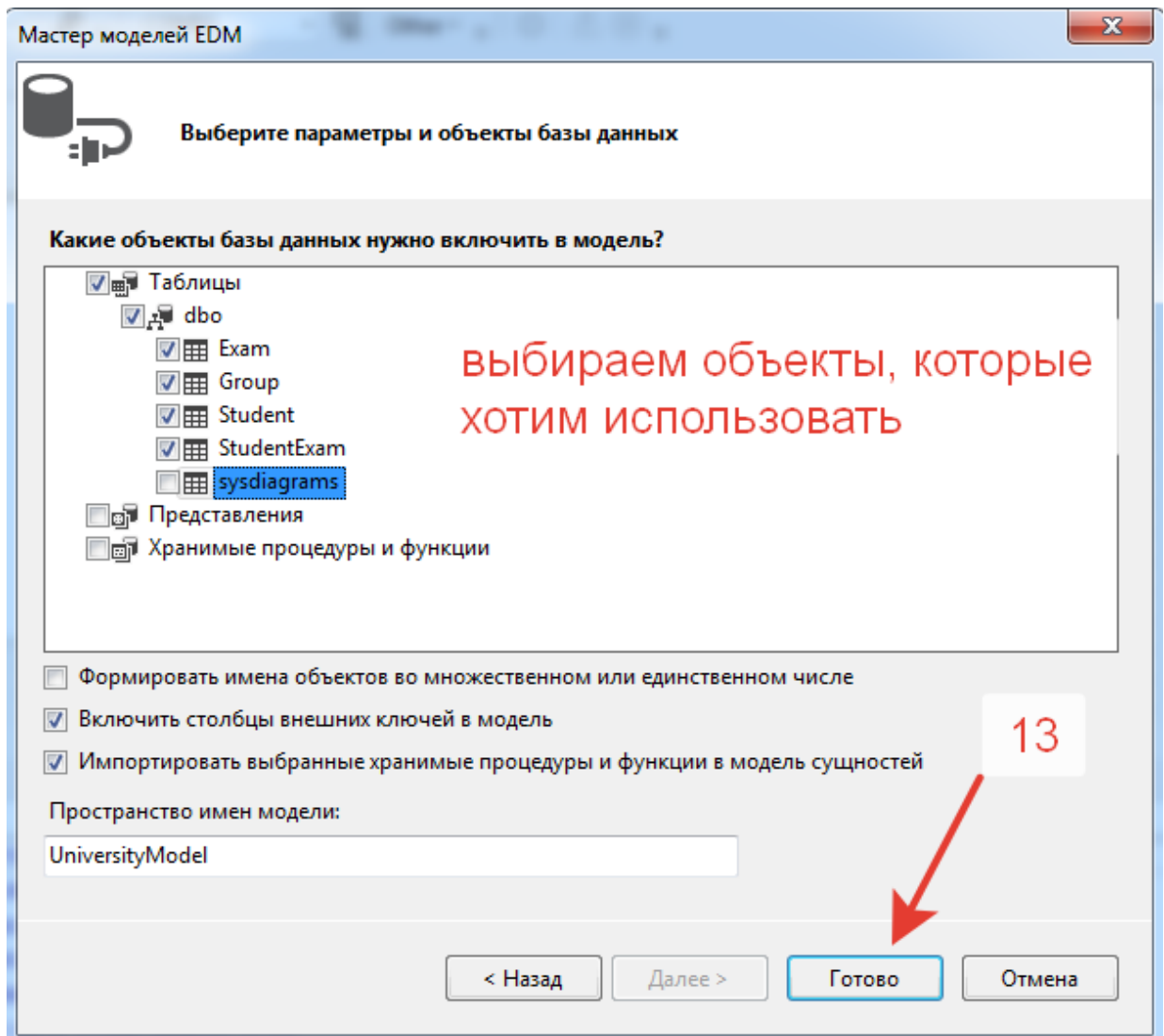
Проверить подключение 9.3 → OK Отмена

путь к экземпляру сервера
МОЖНО ВЗЯТЬ ИЗ ОКНА СОЕДИНЕНИЯ
SSMS

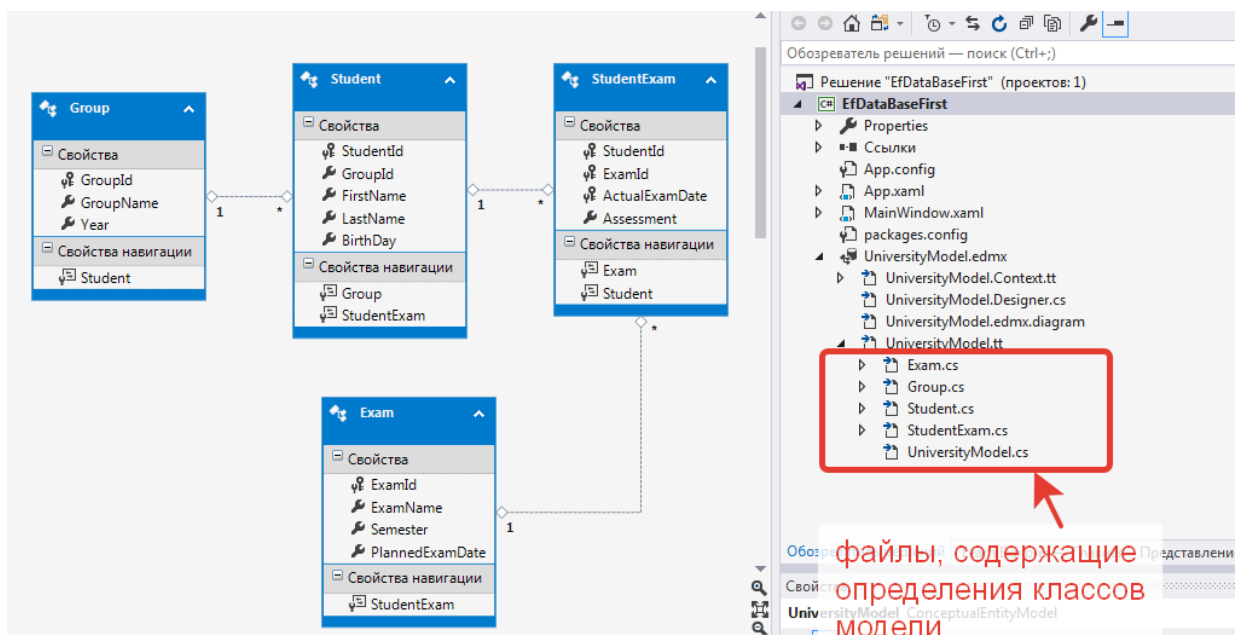
если БД на локальной машине - "Проверка
подлинности Windows"

имя БД, которую хотим
использовать





Если все было сделано верно, будет создана модель, описывающая классы и отношения между ними:



Также надо отметить, что в файле App.config появилось определение подключения:

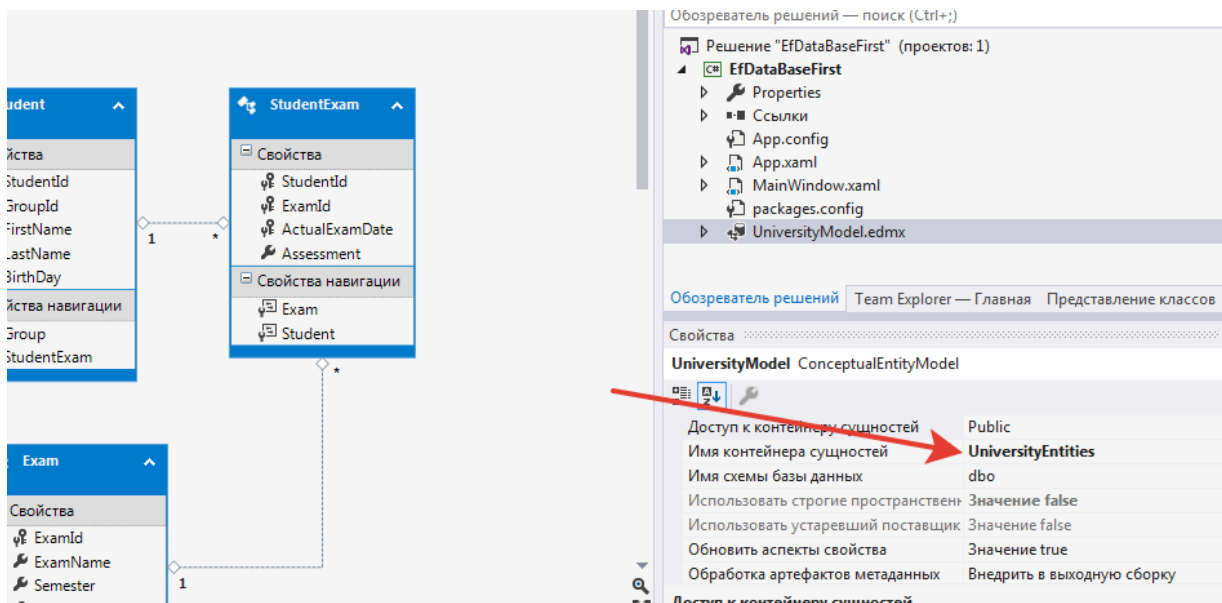


Рис. 6.2: Определение имени контекста данных из свойств модели

```
<connectionStrings>
<add name="UniversityEntities" connectionString="metadata=res://*/UniversityModel.csdl|
res://*/UniversityModel.ssdl|res://*/UniversityModel.msl;
provider=System.Data.SqlClient;provider connection string="
data source=BLOKIN-ПК\SQLEXPRESS;initial catalog=University;integrated security=True;
MultipleActiveResultSets=True;App=EntityFramework";"
providerName="System.Data.EntityClient" />
</connectionStrings>
```

Для работы с созданной моделью, нужно знать имя контекста данных. Оно отображается, например, в свойствах модели, как показано на рисунке

Теперь, чтобы выполнить какую-то работу с БД, например, добавить студента, нужно будет сделать следующие действия:

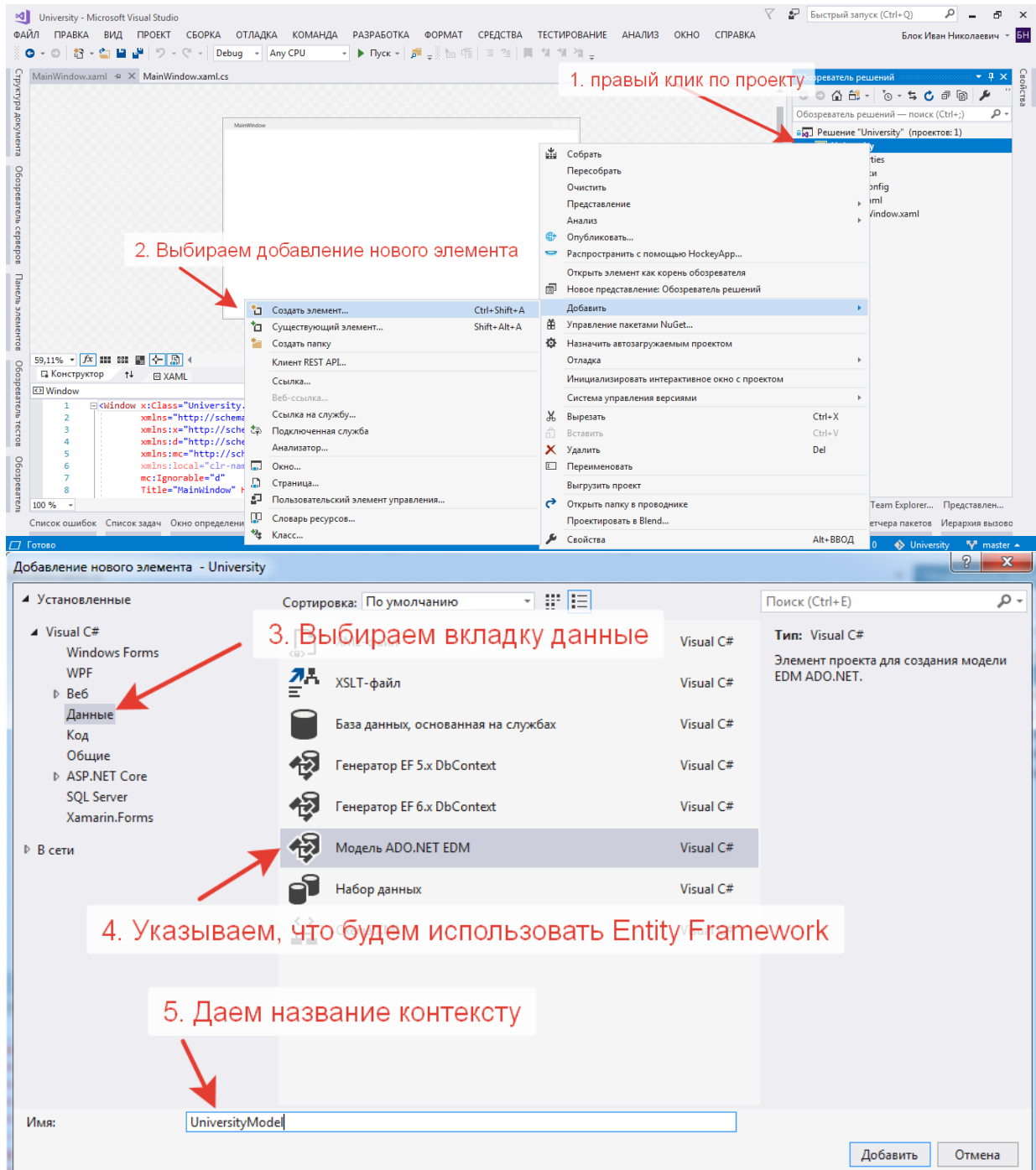
```
using( UniversityEntities entities = new UniversityEntities())
{
    //создаем студента, заполняем его поля
    Student s = new Student() { FirstName = "Сидор", LastName = "Петров" };
    //добавляем созданного студента в коллекцию Student, которая, по сути,
    //отражает соответствующую таблицу БД
    entities.Student.Add(s);
    //отправляем запрос на сохранение данных в БД. Если присутствуют нарушения
    //ограничений целостности, в этом месте будет сгенерировано исключение
    entities.SaveChanges();
}
```

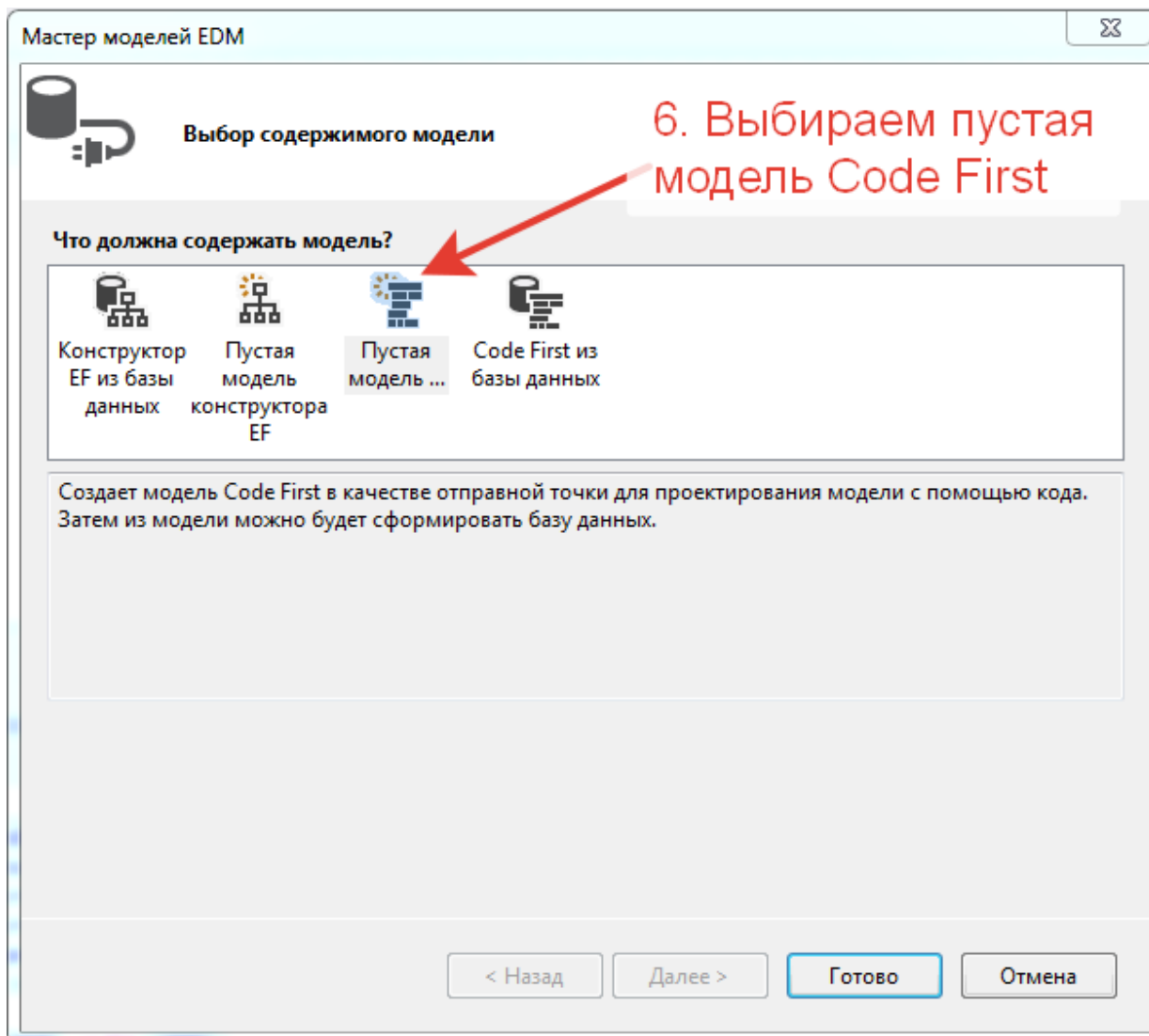
Отдельно стоит упомянуть «Свойства навигации». Это свойство, которое позволяет получить доступ к связанным сущностям, используя обычную ссылку, без дополнительных вызовов операций загрузки данных. Так, например, для примера выше, если мы хотим получить или задать группу, мы будем использовать свойство Group, если список сданных экзаменов – StudentExam. При этом стоит учитывать кратность связи, Group – ссылка на группу студента, в единственном экземпляре, StudentExam – ссылка на коллекцию, множество объектов.

6.3 Создание модели с использованием подхода Code First

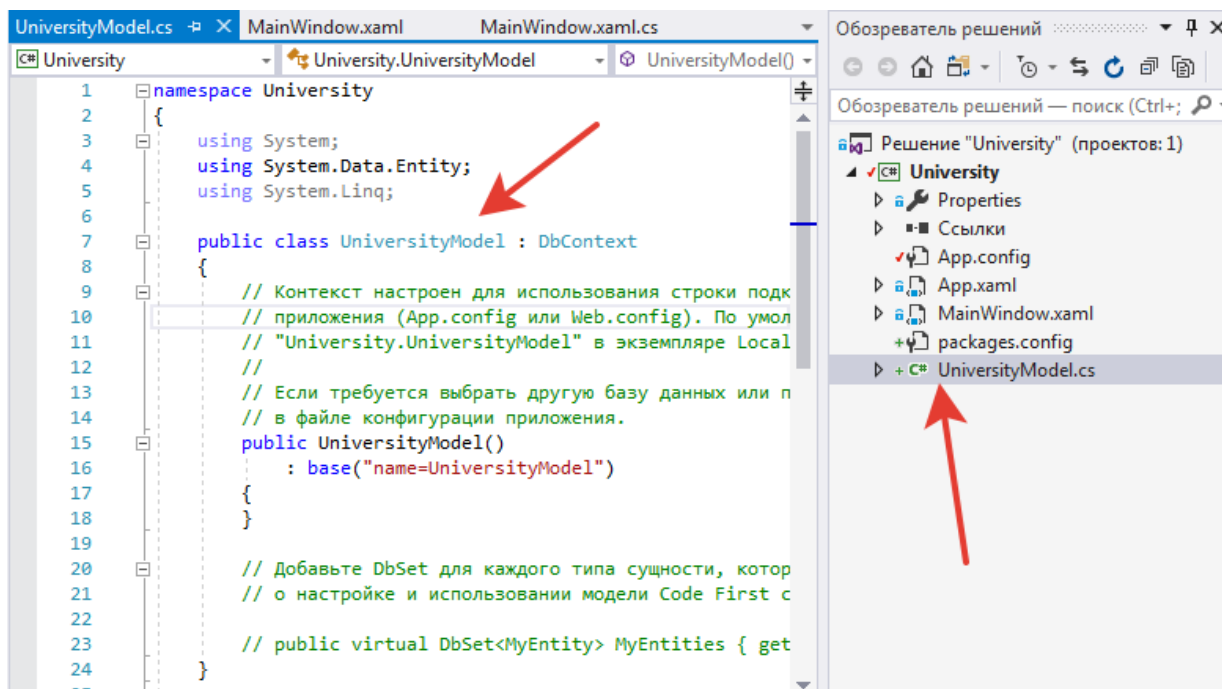
6.3.1 Описание модели данных

Создадим класс контекста базы данных, используя стандартный диалог Visual Studio:





Если все было сделано верно, будет создан новый класс, содержащий описание будущего контекста нашей БД:



Создадим прототипы требуемых классов и добавим их в класс контекста:

```
//описываем классы сущностей:
public class Student
{
}
public class Group
{
}
public class Exam
{
}
public class StudentExam
{
}
//-----
//добавляем их в контекст (класс, созданный на пред. этапе)
public class UniversityModel : DbContext
{
    public UniversityModel()
        : base("name=UniversityModel")
    {
    }
    public virtual DbSet<Student> Students { get; set; }
    public virtual DbSet<Group> Groups { get; set; }
    public virtual DbSet<Exam> Exams { get; set; }
    public virtual DbSet<StudentExam> StudentExams { get; set; }
}
```

Наполним созданные классы. При описании полей следует учитывать как требования постановки задачи к структуре объекта, так и требования, накладываемые EntityFramework. Например, исходя из постановки задачи, для класса "Студент" следует хранить Имя (строковый тип - string), Фамилию(string), Дату рождения (DateTime). Плюс к этому, EF требует, чтобы у каждой сущности был ключ, в простых случаях это должно быть поле типа int с названием Id. Кроме того, сущность "Студент" связана с другими сущностями - с группой и оценками студента. Студент всегда учится в одной группе и может иметь множество оценок, поэтому, необходимо реализовать свойства навигации (см. раздел 1.3.1). С учетом вышесказанного, описание класса Студент станет следующим:

```
public class Student
{
    //поле для хранения идентификатора:
    public int Id { get; set; }
    public string FirstName { get; set; } //имя
    public string LastName { get; set; } //фамилия
    //знак вопроса означает допустимость пустого значения для даты рождения:
    public DateTime ? BirthDate { get; set; }

    //сданных экзаменов у студента может быть много
    //поэтому реализуем как ICollection:
    public virtual ICollection<StudentExam> StudentExams { get; set; }
    //согласно принятым в EF соглашениям, для указания внешнего ключа
    //создается 2 поля, одно ключевое - типа int с именем,
    //формируемым как ИмяСущностиId и второго поля, по
    //которому мы можем получить доступ непосредственно к объекту
    //Имя поля навигации совпадает с именем класса
    public int GroupId { get; set; }
```

```

public Group Group { get; set; }

public Student()
{
    //изначально инициализируем список экзаменов пустым списком:
    StudentExams = new List<StudentExam>();
}
}

```

Данного описания достаточно для генерации таблицы Student в базе данных, однако, для нашей задачи требуется наложить ряд ограничений. Ограничим длину имени и фамилии 50-ю символами, а также сделаем их обязательными для заполнения, для этого пометим их соответствующими атрибутами (см. раздел [1.3.2](#)):

```

public class Student
{
    //Required для указания недопустимости пустых значения
    //MaxLength для задания длины строчковых полей
    [Required, MaxLength(50)]
    public string FirstName { get; set; }
    [Required, MaxLength(50)]
    public string LastName { get; set; }
    //.....остальной код.....
}

```

Остальные классы описываются аналогично. Единственное, на что стоит обратить внимание - на класс StudentExam. В данном случае мы имеем дело с составным ключом (StudentId, ExamId, ExamDate). Для указания составного необходимо переопределить метод OnModelCreating в классе контекста (в нашем случае это UniversityModel) и в теле метода указать дополнительные ограничения модели, как показано ниже:

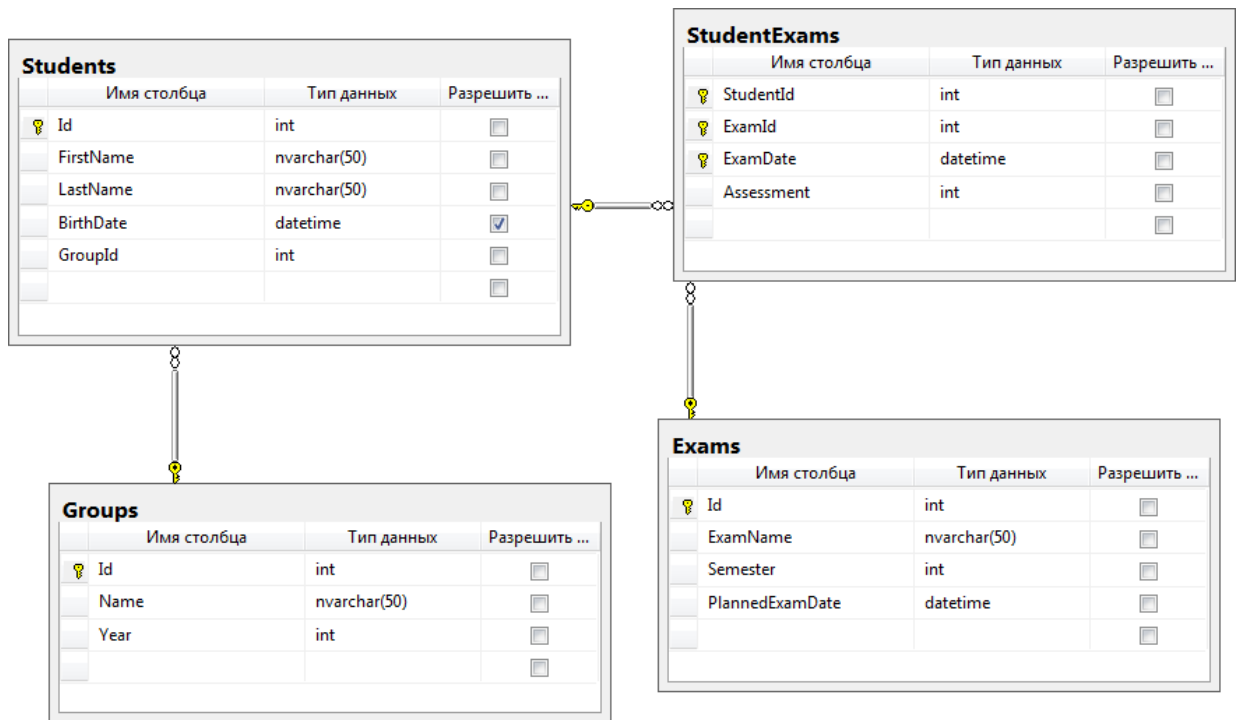
```

public class UniversityModel : DbContext
{
    public UniversityModel()
    : base("name=UniversityModel")
    {
    }
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //для сущности StudentExam задаем ключ как объект
        //анонимного класса с полями StudentId, ExamId, ExamDate
        modelBuilder.Entity<StudentExam>()
            .HasKey(se => new { se.StudentId, se.ExamId, se.ExamDate });
        base.OnModelCreating(modelBuilder);
    }
    public virtual DbSet<Exam> Exams { get; set; }
    public virtual DbSet<Student> Students { get; set; }
    public virtual DbSet<Group> Groups { get; set; }
    public virtual DbSet<StudentExam> StudentExams { get; set; }
}

```

Исходные коды для данной модели можно найти в "Ресурсы\UniversityModel.cs"

При первом запуске программы, EF генерирует БД в соответствии с описанными выше классами:



6.3.2 Заполнение БД случайными данными

Для тестовой версии нашего проекта мы хотим, чтобы в нем изначально были какие то данные. Т.к. вносить все данные вручную утомительно, реализуем заполнение БД случайными данными. Далее будет описан процесс создания инициализатора, реализующего данный сценарий.

Написать, что можно сделать ExecuteSQL для заполнения БД из скрипта.

6.4 Создание модели с использованием EF Core

Когда-нибудь и здесь будет полезный текст....

6.5 Разработка проекта WPF

6.5.1 Вывод данных с использованием Collection View

Создавать CollectionView можно как из C# кода, так и из XAML разметки. Здесь будет показан второй способ, чтобы не загромождать code-behind класса формы. К тому же Visual Studio позволяет автоматически генерировать большую часть кода, связанного с использованием Colleciton View при использовании "классического" WPF. Первый способ будет показан в разделе 4, где CollectionView's будут использоваться внутри ViewModel'ей.

После того, как модель данных создана (что это и как это сделать см. раздел 1), мы можем использовать специальный инструмент Visual Studio, позволяющий автоматизировать разработку интерфейса и вывод данных. Чтобы его использовать откройте окно "Источники данных" в Visual Studio. Сделать это можно в главном меню: Вид->Другие окна->Источники данных (рисунок 6.3).

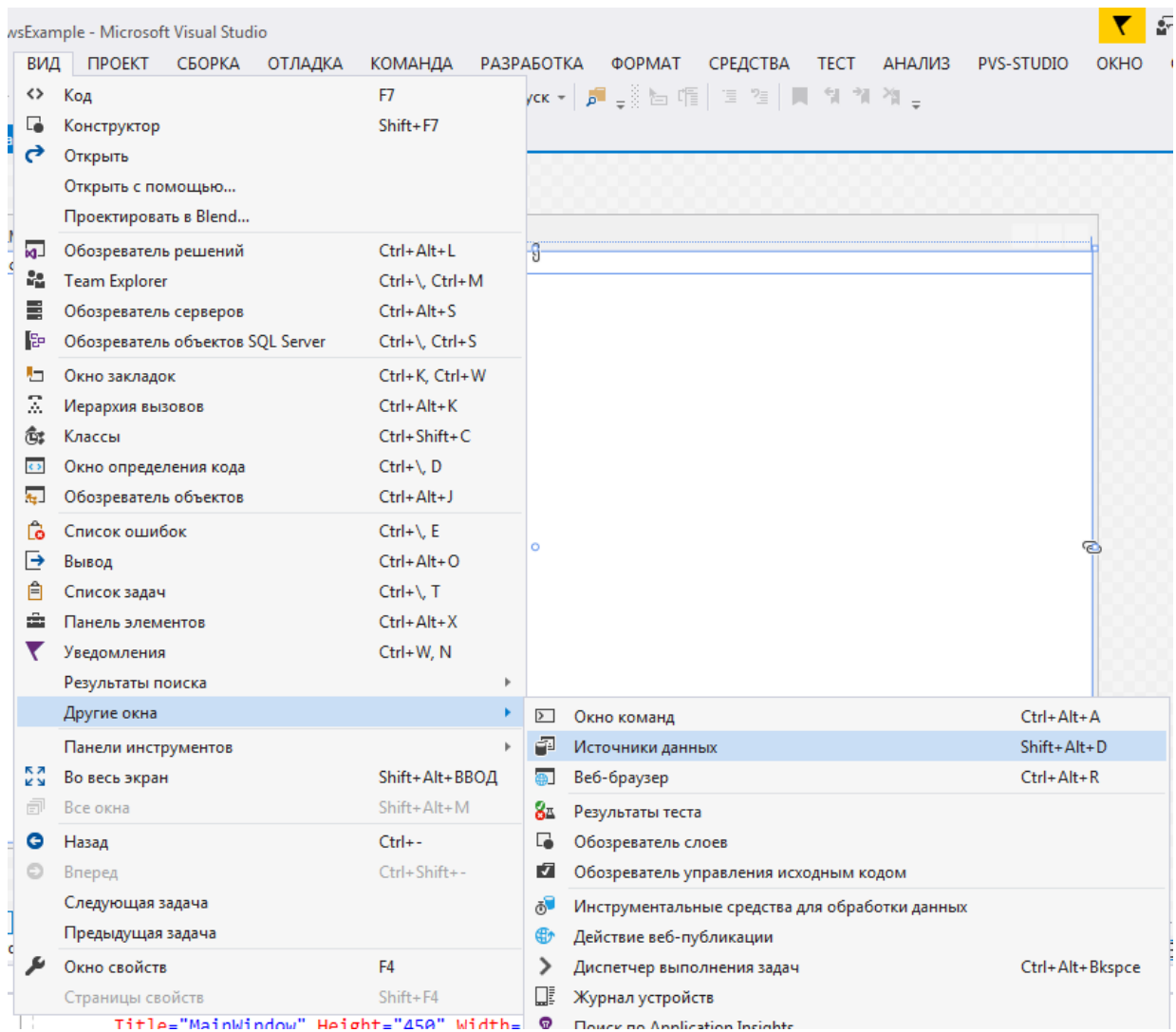
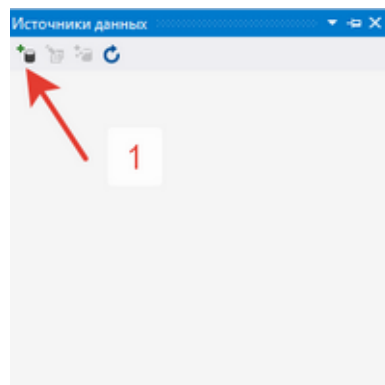
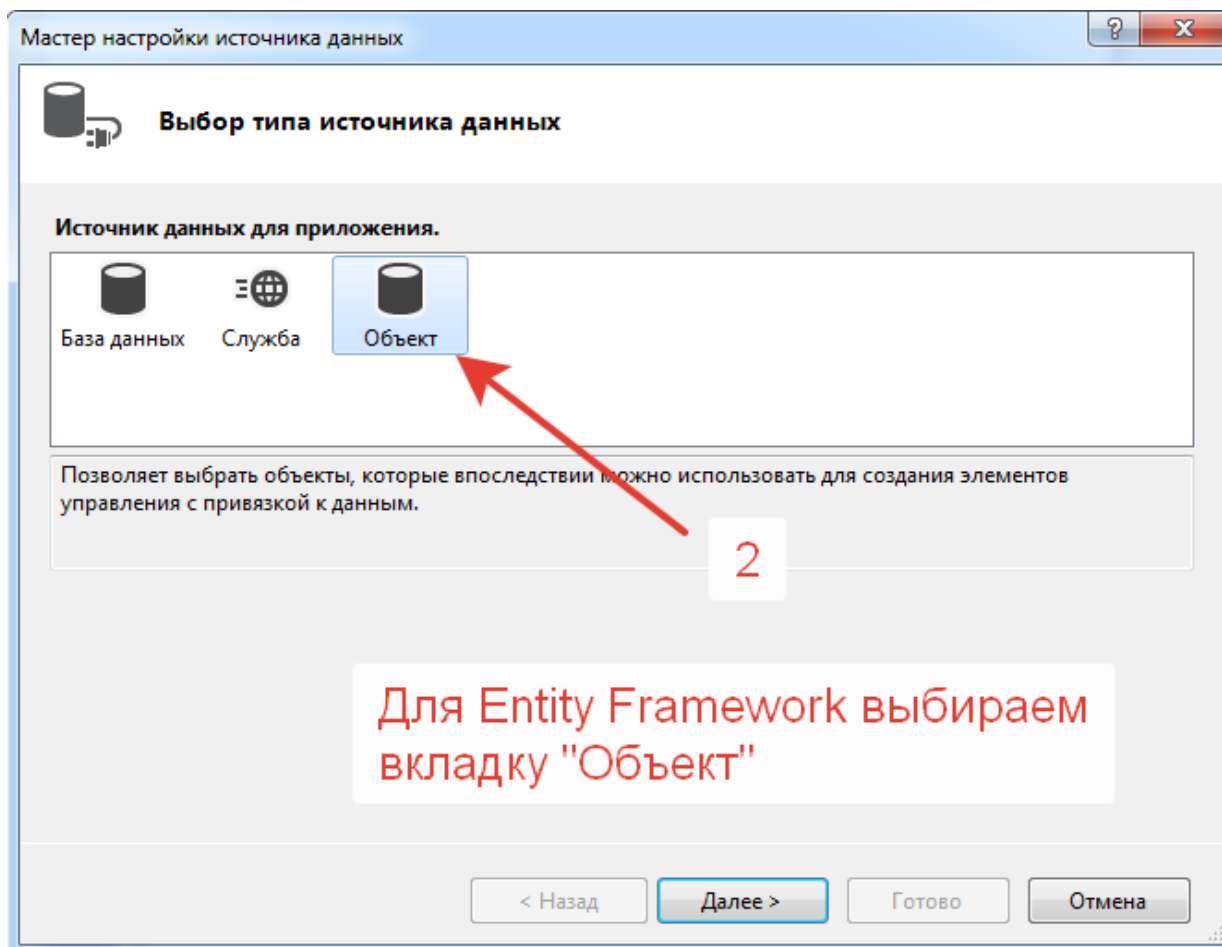
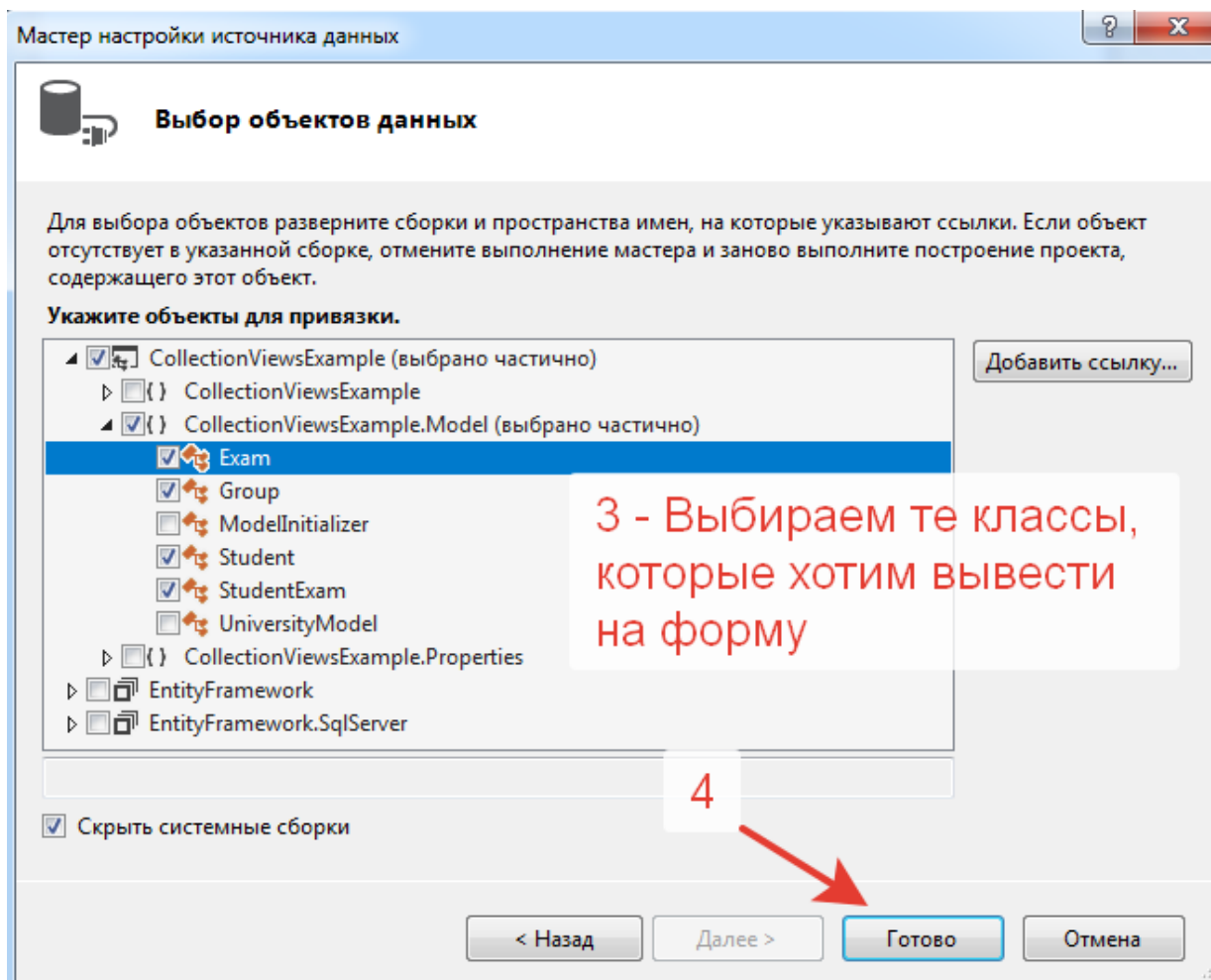


Рис. 6.3: Путь к окну "Источники данных"

Теперь необходимо добавить классы нашей модели данных, для этого необходимо проделать следующие шаги:



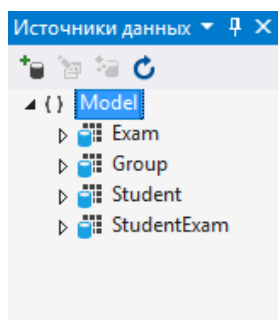




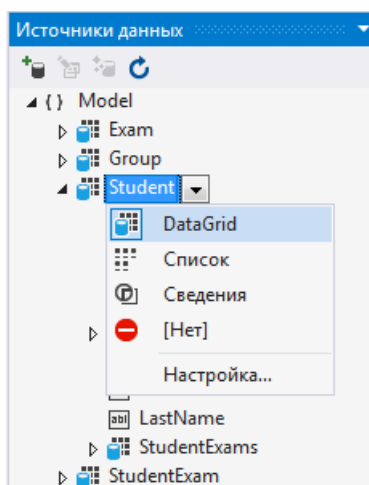
Если все было сделано верно, выбранные классы отобразятся на вкладке источников данных (рисунок 6.4a). По умолчанию для каждого класса доступно 3 представления (рисунок 6.4b):

- DataGrid - вывод списка сущностей в виде таблицы;
- Список - вывод списка сущностей в элемент управления ListView (аналог ListBox с колонками);
- Сведения - вывод подробностей одного элемента данного типа.

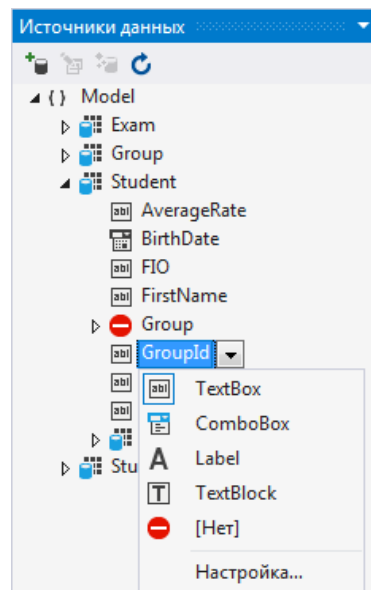
Кроме того, каждое поле может быть представлено разными элементами управления (рисунок 6.4c). Тип элемента управления определяется типом данных поля класса. По умолчанию для каждого типа уже заданы некоторые сопоставления, например строковые данные (string) будут отображены в TextBox, строковые данные только для чтения в TextBlock. Существует возможность выбрать другой элемент управления для отображения данных, в этом случае при использовании DataGrid'а или ListView для этого поля будет создан шаблон данных (см. 3.7.2). Также можно выключить отображение поля, выбрав вариант "Нет" в соответствующем диалоге (рисунок 6.4c).



(а) Список источников данных



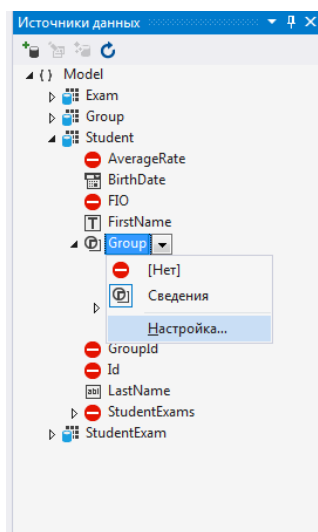
(б) Выбор представления объекта



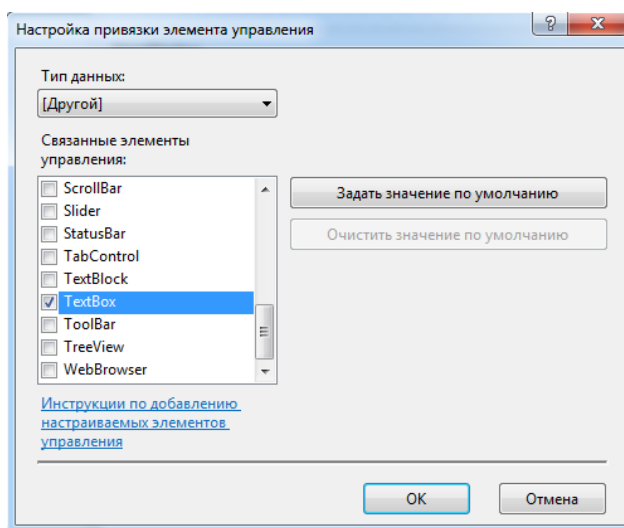
(с) Выбор элемента управления для отображения поля данных

Рис. 6.4: Настройка источников данных

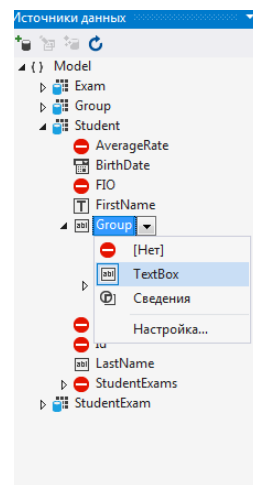
Настроим вывод студентов в таблицу. Для этого выберем вариант `DataGrid`, а также оставим только нужные поля: `LastName`, `FirstName`, `BirthDate`, название группы, отключив остальные поля. Чтобы отобразить название группы, необходимо выбрать в какой элемент его выводить. Настроим вывод в `TextBox`. Для этого кликните "Настройка..." в меню сопоставления поля `Group` (рис. 6.5a), в открывшемся окне выберите вариант `TextBox` (рис. 6.5b), затем, повторно открыв меню, выберите `TextBox` (рис. 6.5c).



(а)



(б)



(с)

Рис. 6.5: Настройка представления для отображения группы, в которой учится студент при табличном просмотре

Аналогично изменим тип представления для `BirthDate` на `TextBlock`.

Теперь перетащим объект `Student` из источников данных на форму, открытую в режиме дизайнера, получим результат, показанный на рисунке 6.6.

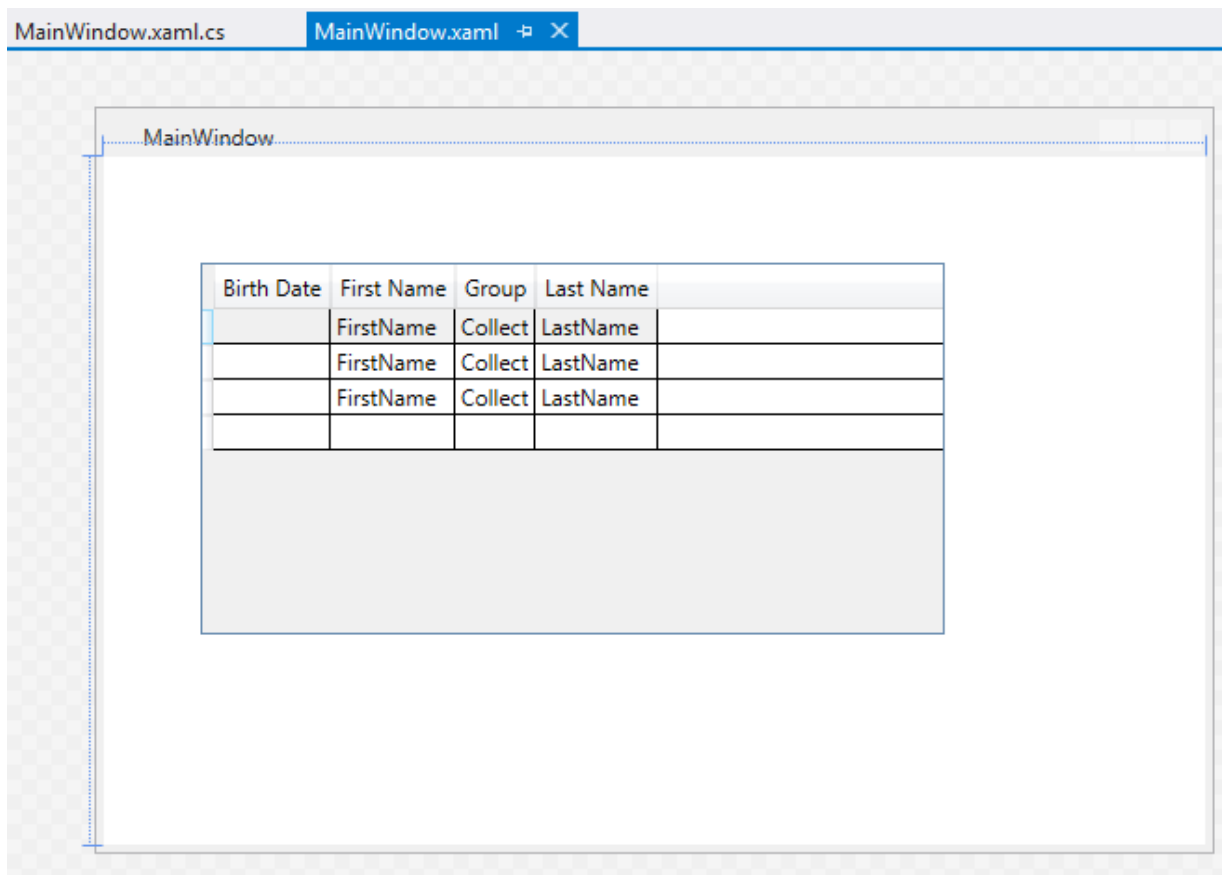


Рис. 6.6: Автосгенерированная форма отображения студентов

Рассмотрим код, сгенерированный Visual Studio. Для этого откроем XAML файл, соответствующий форме:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:Model="clr-namespace:CollectionViewExample.Model"
  x:Class="CollectionViewExample.MainWindow"
  mc:Ignorable="d"
  Title="MainWindow" Height="295" Width="600" Loaded="Window_Loaded">
<Window.Resources>
  <CollectionViewSource x:Key="studentViewSource" d:DesignSource="{d:DesignInstance
    {x:Type Model:Student}, CreateList=True}"/>
</Window.Resources>
<Grid DataContext="{StaticResource studentViewSource}">
  <DataGrid x:Name="studentDataGrid" AutoGenerateColumns="False"
    EnableRowVirtualization="True" ItemsSource="{Binding}" Margin="46,17,146,48"
    RowDetailsVisibilityMode="VisibleWhenSelected">
  <DataGrid.Columns>
    <DataGridTemplateColumn x:Name="birthDateColumn" Header="Birth Date"
      Width="SizeToHeader">
      <DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
          <TextBlock Text="{Binding BirthDate}"/>
        </DataTemplate>
      </DataGridTemplateColumn.CellTemplate>
    </DataGridTemplateColumn>
  </DataGrid.Columns>
</Grid>
```

```

<DataGridTextColumn x:Name="firstNameColumn" Binding="{Binding FirstName}"
    Header="First Name" Width="SizeToHeader"/>
<DataGridTextColumn x:Name="groupColumn" Binding="{Binding Group}"
    Header="Group" Width="SizeToHeader"/>
<DataGridTextColumn x:Name="lastNameColumn" Binding="{Binding LastName}"
    Header="Last Name" Width="SizeToHeader"/>
</DataGrid.Columns>
</DataGrid>
</Grid>
</Window>

```

Заголовок разметки может у вас может отличаться, в остальном же должен быть похожий код. Выделим основные пункты, на которые стоит обратить внимание:

- Для формы был создан обработчик события Loaded. Данное событие возникает, когда форма размещена, отрисована и готова к взаимодействию. По умолчанию обработчик называется Window_Loaded. В дальнейшем, используя это событие будет производиться загрузка данных;
- Добавлены локальные ресурсы окна (блок <Window.Resources>), а именно создан объект типа CollectionViewSource, который является наследником CollectionView для применения в XAML разметке. На данный момент он не содержит данных, они будут загружены позже, однако, при необходимости, вы всегда можете создать привязку свойства Source к нужному источнику данных.

Для CollectionViewSource важно запомнить значение свойства x:Key, с помощью которого в дальнейшем сможем найти данный объект;

- В качестве контекста данных установлен определенный выше CollectionViewSource:

```
DataContext="{StaticResource studentViewSource}"
```

Т.к. контекст данных у объекта может быть только один, данное свойство устанавливается только при добавлении первого объекта из вкладки "источники данных". Все последующие связываются напрямую.

- Добавлен DataGrid - таблица для отображения данных, в которой источник данных - свойство ItemsSource установлен как ItemsSource="{Binding}". Указание ключевого слова **Binding** без каких либо параметров указывает на привязку к контексту данных. Контекстом данных в данном случае является studentViewSource. Если бы контекст данных установлен был не был, или был инициализирован другим объектом, то получить тот же результат можно было бы задав значение источника данных как:

```
ItemsSource="{Binding Source={StaticResource studentViewSource}}"
```

Подобную строку вы увидите для всех добавляемых объектов, начиная со второго.

Произведем некоторую доработку сгенерированного кода. Для этого переставим столбцы DataGrid'a. Для этого достаточно поменять порядок следования тегов DataGridTemplateColumn, DataGridTextColumn в секции <DataGrid.Columns>. Сейчас определение столбцов выглядит следующим образом (приведена только секция, отвечающая за столбцы):

```

<DataGrid.Columns>
  <DataGridTemplateColumn x:Name="birthDateColumn" Header="Birth Date" Width="SizeToHeader">
    <DataGridTemplateColumn.CellTemplate>
      <DataTemplate>
        <TextBlock Text="{Binding BirthDate}"/>
      </DataTemplate>
    </DataGridTemplateColumn.CellTemplate>
  </DataGridTemplateColumn>
  <DataGridTextColumn x:Name="firstNameColumn" Binding="{Binding FirstName}" Header="First Name"
    Width="SizeToHeader"/>
  <DataGridTextColumn x:Name="groupColumn" Binding="{Binding Group}" Header="Group"
    Width="SizeToHeader"/>
  <DataGridTextColumn x:Name="lastNameColumn" Binding="{Binding LastName}" Header="Last Name"
    Width="SizeToHeader"/>
</DataGrid.Columns>

```

Это дает столбцы в следующем порядке: Birth Date, FirstName, Group, Last Name. Изменим порядок на Group, Last Name, First Name, Birth Date, для этого изменим разметку:

```

<DataGrid.Columns>
  <DataGridTextColumn x:Name="groupColumn" Binding="{Binding Group}" Header="Group"
    Width="SizeToHeader"/>
  <DataGridTextColumn x:Name="lastNameColumn" Binding="{Binding LastName}" Header="Last Name"
    Width="SizeToHeader"/>
  <DataGridTextColumn x:Name="firstNameColumn" Binding="{Binding FirstName}" Header="First Name"
    Width="SizeToHeader"/>
  <DataGridTemplateColumn x:Name="birthDateColumn" Header="Birth Date" Width="SizeToHeader">
    <DataGridTemplateColumn.CellTemplate>
      <DataTemplate>
        <TextBlock Text="{Binding BirthDate}"/>
      </DataTemplate>
    </DataGridTemplateColumn.CellTemplate>
  </DataGridTemplateColumn>
</DataGrid.Columns>

```

Отредактируем заголовки столбцов таблицы. Для этого требуется изменить свойство Header соответствующего столбца. Кроме того, необходимо настроить биндинг (что это, см. 3.1). Так, например, столбец для отображения группы ссылается на поле Group, однако поле Group - это свойство навигации, которое содержит ссылку на весь объект типа Group. Нам же необходимо взять только название, поэтому конкретизируем, указав Group.Name. Также модифицируем выражение привязки для даты рождения. В данном случае путь привязки менять не нужно, однако, если оставим как есть, то будет выведено значение, содержащее время (часы, минуты, секунды и даже миллисекунды!). Поэтому добавим в выражение биндинга строку форматирования результата StringFormat=dd.MM.yy. По желанию можно настроить ширину столбцов и другие параметры отображения. Далее приведен возможный вариант разметки столбцов, отвечающий перечисленным выше действиям:

```

<DataGrid.Columns>
  <DataGridTextColumn x:Name="groupColumn" Binding="{Binding Group.Name}" Header="Группа"
    Width="SizeToHeader"/>
  <DataGridTextColumn x:Name="lastNameColumn" Binding="{Binding LastName}" Header="Фамилия"
    Width="*/>
  <DataGridTextColumn x:Name="firstNameColumn" Binding="{Binding FirstName}" Header="Имя"
    Width="*/>

```

```

<DataGridTemplateColumn x:Name="birthDateColumn" Header="Дата рождения" Width="SizeToHeader">
    <DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding BirthDate, StringFormat=dd.MM.yy}"/>
        </DataTemplate>
    </DataGridTemplateColumn.CellTemplate>
</DataGridTemplateColumn>
</DataGrid.Columns>

```

***Примечание:** В рассмотренном примере не затрагивался вопрос форматирования. Размещение, выравнивание элементов остается на вашей совести.*

На этом с разметкой можно закончить. Осталось загрузить данные. Будем предполагать, что контекст базы данных объявлен глобально и размещен в классе App, как показано в разделе 1.4.2. Перейдем в code-behind класса формы (файл с расширением .xaml.cs). В нем нас интересует обработчик события Loaded. Если в точности повторить описанные выше шаги, то он будет иметь следующий вид:

```

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    System.Windows.Data.CollectionViewSource studentViewSource =
        ((System.Windows.Data.CollectionViewSource)(this.FindResource("studentViewSource")));
    // Загрузите данные, установив свойство CollectionViewSource.Source:
    // studentViewSource.Source = [универсальный источник данных]
}

```

Как видно, здесь Visual Studio также сгенерировала значительную часть кода. Чтобы заставить пример работать осталось раскомментировать последнюю строку и указать выводимые данные, в нашем случае это объекты из App.DatabaseContext.Students (App.DatabaseContext - контекст БД):

```

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    System.Windows.Data.CollectionViewSource studentViewSource =
        ((System.Windows.Data.CollectionViewSource)(this.FindResource("studentViewSource")));
    // Загружаем студентов в список
    var students = App.DatabaseContext.Students.ToList();
    //и устанавливаем данный список как источник данных для CollectionViewSource
    studentViewSource.Source = students;
}

```

Запустим приложение, как видно, все имеющиеся студенты были загружены (рисунок 6.7).

Group	Last Name	Имя	Birth Date
Иф-51	Петров	Алексей	01.01.00
Иф-51	Петриков	Кирилл	01.01.99
Иф-51	Сидоров	Алексей	01.01.99
Иф-51	Петриков	Кирилл	01.01.00
Иф-51	Петрикан	Петр	01.01.96
Иф-51	Серегов	Петр	01.01.98
Иф-51	Серегов	Сергей	01.01.98
Иф-51	Серегов	Кирилл	01.01.96
Иф-51	Петрикан	Евгений	01.01.00
Иф-51	Иванов	Никитос	01.01.96
Иф-51	Петрикан	Евгений	01.01.00

Рис. 6.7: Загруженные с помощью CollectionView данные

При необходимости, на данном этапе могут быть использованы LINQ запросы для преобразования данных, например, мы могли бы отсортировать выводимых студентов, или выбрать студентов, обучающихся только в одной группе:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    System.Windows.Data.CollectionViewSource studentViewSource =
        ((System.Windows.Data.CollectionViewSource)(this.FindResource("studentViewSource")));
    // Загружаем студентов в список, предварительно фильтруя по группе и сортируя фамилии, имени
    var students = App.DatabaseContext.Students
        .Where ( g => g.Group.Id == 4 )//выбираем студентов, у которых id группы = 4
        .OrderBy( s=> s.LastName )      //сортируем по фамилии
        .ThenBy ( s=> s.FirstName )     //досортировываем по имени
        .ToList ();                    //результат сохраняем в список
    //устанавливаем данный список как источник данных для CollectionViewSource
    studentViewSource.Source = students;
}
```

В некоторых случаях загрузка данных может выполняться достаточно долго. В это время пользователю кажется, что программа "зависает". Это связано с тем, что сложные операции выполняются в том же потоке, в котором происходит отрисовка интерфейса пользователя, поэтому, пока операция не завершится, интерфейс не может быть обновлен, взаимодействие с пользователем также невозможно. В некоторых случаях период этого зависания может быть ощутимым, поэтому, ресурсоемкие операции рекомендуется выносить в отдельный поток. Для задач загрузки данных в этом случае можно использовать либо вызов Task.Run для запуска потока из пула потоков, либо перегруженные версии методов с приставкой Async, означающим асинхронность выполнения операции. Покажем оба способа. Далее приведен код с пояснениями производимых действий:

```
//помечаем метод модификатором async для возможности использования оператора await
private async void Window_Loaded(object sender, RoutedEventArgs e)
{
    System.Windows.Data.CollectionViewSource studentViewSource =
        ((System.Windows.Data.CollectionViewSource)(this.FindResource("studentViewSource")));
```

```

/* Task.Run запускает поток на выполнение и возвращает объект типа Task<>,
 * где содержится результат выполнения операции (в данном случае это
 * загрузка студентов: App.DatabaseContext.Students.ToList().
 * достаточно простым способом получить результат является использование
 * оператора await. */
var students = await Task.Run(() => App.DatabaseContext.Students.ToList());
/* родительский поток во время загрузки свободен и возвращается к выполнению
 * функции только после срабатывания оператора await, т.к. завершения дочернего потока.*/
//выводим сообщение:
MessageBox.Show("Студенты загружены!!!");
//устанавливаем источник данных:
studentViewSource.Source = students;
}

```

Теперь покажем, как сделать то же самое с использованием ToListAsync. Для его использования необходимо предварительно подключить пространство имен System.Data.Entity (В начале файла прописать using System.Data.Entity;).

```

private async void Window_Loaded(object sender, RoutedEventArgs e)
{
    System.Windows.Data.CollectionViewSource studentViewSource =
        ((System.Windows.Data.CollectionViewSource)(this.FindResource("studentViewSource")));
    var students = await App.DatabaseContext.Students.ToListAsync();
    MessageBox.Show("Студенты загружены!!!");
    studentViewSource.Source = students;
}

```

***Примечание:** Объяснение механизма async..await выходит за пределы данной работы, поэтому, для полного понимания примера рекомендуется ознакомиться с этим самостоятельно*

Данный **пример** можно найти в "Ресурсы\Примеры типовых проектов\WpfExample проект CollectionView.

6.5.2 Фильтрация данных

Доработаем приложение, добавив возможность выбирать не все данные, а только часть, соответствующую запросу пользователю. Добавим возможность выводить студентов только выбранного курса, выбранной группы, а также производить поиск по фамилии. Для фильтрации у нас есть 2 возможности - обновлять данные из базы данных, с помощью LINQ запроса с использованием выражения where и фильтровать уже выбранные данные, используя специальное свойство Filter класса CollectionView. Это свойство представляет из себя ссылку на метод, вызываемый каждый раз при необходимости обновить представление данных. Это позволяет отобразить ту часть данных, которая будет отображена. Подробнее см. в разделе [3.4.1](#).

Здесь будем использовать оба способа. При выборе группы будем использовать новый запрос к базе данных и заполнение CollectionViewSource новыми данными. Это связано с потенциально большим объемом выборки всех студентов. Фильтр CollectionView будем использовать для отбора групп при выборе курса и для фильтрации студентов по фамилии.

Разместим на форме соответствующие элементы управления так, чтобы форма приобрела следующий вид, показанный на рисунке ... (стрелками обозначены имена элементов управления для дальнейшего взаимодействия с ними в коде):

Данный **пример** можно найти в "Ресурсы\Примеры типовых проектов\WpfExample проект CollectionFiltering.

6.5.3 Вывод связанных данных

6.6 Разработка проекта с использованием паттерна MVVM

Литература

1. *Learn Entity Framework Core. Your guide to using the latest version of Microsoft's Object Relational Mapper*, URL: www.learnentityframeworkcore.com
2. *Ulearn.me. Курс по LINQ запросам*, URL: ulearn.me/Course/linq/
3. *ВВЕДЕНИЕ В СИСТЕМЫ УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ*, URL: <http://citforum.ru/database/dblearn/>
4. *Канонические функции*, URL: <https://docs.microsoft.com/ru-ru/dotnet/framework/data/adonet/ef/language-reference/canonical-functions>
5. *Поддерживаемые и неподдерживаемые методы LINQ (LINQ to Entities)*, URL: <https://docs.microsoft.com/ru-ru/dotnet/framework/data/adonet/ef/language-reference/supported-and-unsupported-linq-methods-linq-to-entities>
6. *Стандартные операторы в запросах LINQ to Entities*, URL: <https://docs.microsoft.com/ru-ru/dotnet/framework/data/adonet/ef/language-reference/standard-query-operators-in-linq-to-entities-queries>
7. *Сопоставление методов CLR с каноническими функциями*, URL: <https://docs.microsoft.com/ru-ru/dotnet/framework/data/adonet/ef/language-reference/clr-method-to-canonical-function-mapping>
8. *SqlFunctions Class*, URL: <https://docs.microsoft.com/ru-ru/dotnet/api/system.data.objects.sqlclient>
9. *Data Binding Overview*, URL: <https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-overview>
10. *Синтаксис строки подключения*, URL: <https://docs.microsoft.com/ru-ru/dotnet/framework/data/adonet/connection-string-syntax>
11. *Строки подключения и файлы конфигурации*, URL: <https://docs.microsoft.com/ru-ru/dotnet/framework/data/adonet/connection-strings-and-configuration-files>