Министерство науки и высшего образования РФ
Федеральное государственное автономное
образовательное учреждение высшего образования
«Казанский (Приволжский) федеральный университет»

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Учебно- методическое пособие

«Курсовой проект по Базам данных»

Салимов Фарид Ибрагимович

Хайруллин Альфред Фаридович

Казань 2018

Оглавление

1.База данных (условной) предметной области. 3
2. СУБД Microsoft SQL Server 5
3. Вход в систему 9
4. Создание базы данных. Работа в среде Microsoft SQL Server Management Studio....12
4.1 Создание базы данных....15
5. Создание объектов базы данных 29
5.1 Таблицы 29
5.2 Триггеры и хранимые процедуры. 36
5.3 Работа с курсором 45
5.4 Заполнение таблиц. 48
5.5 Создание представлений. 54
6.Принципы работы с источниками данных в среде MS Visual Studio. 59
7.Разработка клиентского места 61
7.1 Основная экранная форма 63
7.1.1 Динамическое изменение свойств элементов управления. 73
7.1.2 Отображение связи между таблицами «Master – Detal». 77
7.1.3 Использование Lookup – полей. 83
7.1.4 Использование клиентским приложением хранимых процедур и функций. 85
7.2 Создание дополнительных форм клиентского приложения. Построение альтернативной формы....88
8. Запросы 91
8.1 Задача 1 91
8.2 Задача 2. 101
8.3 Задача 3. 105
9. Отчёты....111

1.База данных (условной) предметной области.

Рассмотрим простую учебную базу данных (БД), которая будет в дальнейшем служить иллюстрацией для построения различных демонстрационных примеров. Пусть имеются следующие три таблицы:

Таблица 1. **Товары**

- Код товара Integer
- Наименование товара VarChar(20)
- Единица измерения VarChar(10)
- Цена за единицу Integer
- Количество товара на складе Integer

Таблица 2. **Покупатели**

- Код покупателя Integer
- Наименование покупателя VarChar(30)
- Город VarChar(15)
- Адрес VarChar(20)
- Телефон VarChar(10)

Таблица 3. **Расход товара**

- Код покупки Integer
- Дата покупки Date
- Количество покупки Integer
- Стоимость покупки Integer
- Код товара Integer
- Код покупателя Integer

Первая таблица содержит сведения о продаваемых товарах - наименование, в каких единицах измеряется товар, цена единицы товара и количество товара на складе на момент просмотра таблицы. Для простоты будем предполагать, что цена товара не меняется со временем. Наименование полностью идентифицирует товар. Если один и тот же товар присутствует на складе в разных

единицах измерения, то он должен иметь семантически одинаковое, но синтаксически разное название (например, огурцы баночные, огурцы развесные и пр.). Однако для однозначной идентификации товара используется специальный код товара. Именно код товара будем считать первичным ключом в таблице «Товары».

Таблица 2 содержит данные о покупателях товара. Столбец «город» выделен из столбца «адрес» для дальнейших упражнений по составлению запросов.

Третья таблица предназначена для отслеживания отпуска товаров со склада конкретным покупателям. Каждый день на склад поступают товары. Допустим, что процесс прихода товаров на склад и его учет нас не интересуют (на самом деле, наш пример является составной частью большой задачи). Таблица «Расход товара» является дочерней для таблиц «Товары» и «Покупатели» и находится с ними в связи многие к одному. Поэтому между дочерней таблицей и каждой из ее родительских таблиц должно выполняться требование ссылочной целостности - отпускаемые товары должны присутствовать в таблице «Товары», а покупающие их лица - в таблице «Покупатели». Стоимость покупки определяется произведением цены товара на количество купленного товара.

Связи между таблицами, входящими в базу данных, приведены на рисунке (ключевые поля - подчеркнуты).

2. СУБД Microsoft SQL Server

Для реализации нашей задачи выберем СУБД Microsoft® SQL Server. MS SQL Server представляет собой комплексную платформу доступа к данным, которая характеризуется масштабируемостью, доступностью, безопасностью и управляемостью.

Ядро СУБД SQL Server обеспечивает безопасное и надежное хранение данных в реляционном формате, в формате XML а так же объектов среды CLR Microsoft .NET Framework. В основе решения управления корпоративными данными лежит ядро базы данных SQL Server. Помимо поддержки реляционных баз данных и данных в формате XML, SQL Server также содержит средства анализа, подготовки отчетов, интеграции данных и рассылки уведомлений. Выгодным преимуществом MS SQL Server является тесная его интеграция с Microsoft Visual Studio, Microsoft Office System и целым комплектом средств разработки, включая Business Intelligence Development Studio.

Сервер баз данных Microsoft SQL Server в качестве языка запросов использует версию языка SQL, получившую название Transact-SQL (сокращённо T-SQL). Язык T-SQL является реализацией SQL-92 (стандарт ISO для языка SQL) с множественными расширениями.

SQL был расширен такими дополнительными возможностями как:

- управляющие операторы,
- локальные и глобальные переменные,
- различные дополнительные функции для обработки строк, дат, математики и т. п.,
- поддержка аутентификации Microsoft Windows

SQL Server Management Studio

SQL Server Management Studio — интегрированная среда для доступа, управления, конфигурирования и разработки БД и серверов БД. Эта среда поддерживает графический интерфейс и мощные функции работы со сценариями.

В состав среды SQL Server Management Studio входят следующие средства:

- **Редактор кода** — богатый своими возможностями редактор для написания и редактирования сценариев.
- **Обозреватель объектов,** принадлежащих экземплярам SQL Server.
- **Обозреватель шаблонов** для размещения и написания сценариев и новых шаблонов.
- **Обозреватель решений** для организации и хранения связанных сценариев как части проекта.
- **Окно свойств** для отображения текущих свойств выбранных объектов.

Информация в БД хранится в трех типах файлов:

- Первичные файлы данных – расширение - mdf
- Дополнительные файлы – расширение – ndf
- Файлы журналов транзакций - расширение ldf

Первичные файлы

Первичный файл данных содержит сведения, необходимые для запуска базы данных, и ссылки на другие файлы в базе данных. Первичные файлы также могут содержать данные пользовательских таблиц и индексов. Данные и объекты пользователя могут храниться в данном файле или во вторичном файле данных. В каждой базе данных обязательно имеется один первичный файл данных. Для имени первичного файла данных рекомендуется расширение MDF.

- **Дополнительные (Вторичные) файлы**

Вторичные файлы данных не являются обязательными; это пользовательские файлы, в которых хранятся данные пользователя. Вторичные файлы могут быть использованы для распределения данных на нескольких дисках, в этом случае каждый файл записывается на отдельный диск. Кроме того, если размер базы данных превышает максимальный размер для одного файла ОС Windows, можно использовать вторичные файлы данных, таким образом база данных сможет расти дальше. Для имени вторичного файла данных рекомендуется расширение NDF. СУБД SQL Server позволяет определять до 32000 вторичных файлов.

- **Журналы транзакций**

В этих файлах содержатся данные журнала - информация об операциях, произведенных над базой данных. Протоколирование производимых операций необходимо для обеспечения возможности восстановления базы данных после сбоев или неверных изменений данных. В каждой базе данных должен быть, как минимум, один файл журнала транзакций, а может быть и несколько таких файлов. Для файлов журнала транзакций рекомендуется расширение LDF.

По умолчанию и данные, и журналы транзакций помещаются на один и тот же диск и имеют один и тот же путь. Это сделано для поддержки однодисковых систем, но для производственных сред это решение может быть неоптимальным.

Файлы базы данных могут быть объединены в файловые группы. Файловая группа — это логическое объединение файлов данных, позволяющее администраторам баз данных управлять всеми файлами группы как единым целым. Файловые группы предоставляют возможность выборочно выполнять операции резервного копирования и восстановления или перераспределять нагрузку по вводу и выводу. В системе предусмотрено применение только одной первичной файловой группы (Primary), однако пользователь по своему усмотрению может использовать до 255 вторичных файловых групп. Вторичные файловые группы создаются на момент создания или модификации БД.

Данные внутри файловой группы распределяются по файлам пропорционально свободному месту в файлах. Например, если в файле f1 свободно 100 МБ, а в файле f2 - 300 МБ, то в файл f1 будет записана одна часть данных, а в файл f2 - три части, при этом оба файла будут заполнены примерно в одно и то же время.

Если в базе данных создаются объекты без указания файловой группы, к которой они относятся, они назначаются файловой группе по умолчанию. В любом случае только одна файловая группа создается как файловая группа по умолчанию. Файлы в файловой группе по умолчанию должны быть достаточно большими, чтобы вмещать новые объекты, не назначенные другим файловым группам.

Файловая группа PRIMARY является группой по умолчанию, если только она не была изменена инструкцией ALTER DATABASE.

3. Вход в систему

Работа с базой данных осуществляется с использованием среды SQL Server Management Studio(SSMS).

При запуске программы появляется диалоговое окно Connect to Server (Соединение с сервером)). Естественно для работы с СУБД пользователь должен быть зарегистрирован администратором базы данных в системе.

Внешний вид окна входа в систему может оказаться и другим, в зависимости от того был ли до этого выполнен вход в систему, на каком компьютере осуществляется регистрация пользователя в системе и какая учетная запись используется.

Поле «**Тип сервера**» позволяет выбрать одну из подсистем (сам сервер базы данных или службы Analysis Services, Report Services или Integration Services). Для наших целей необходимо выбрать Компонент Database Engine.

В поле **Имя Сервера** необходимо указать сервер, с которым происходит соединение. Это зависит от того какой экземпляр MS SQL сервера стоит на вашей машине или на другом компьютеры в локальной сети. Если установлена одна из полных версия MS SQL Server, то в этом поле указывается имя компьютера или ключевое слово **(local)**. Такой вариант означает, что необходимо

подключиться к серверу, который применяется по умолчанию. В случае подключения к серверу, запущенному на компьютере в локальной сети, задаем имя этого компьютера. При использовании бесплатной версии MS SQL Server Express доступ осуществляется по имени сервера **(local)\sqlexpress**. Для доступа к компьютеру в сети - имя_компьютера\sqlexpress. Также возможен вариант, когда вы не устанавливали никого экземпляра MS SQL сервера, но у вас есть Visual Studio 2012 или более поздняя версия. В этих версиях Visual Studio есть встроенный экземпляр упрощенной версии MS SQL Server Express LocalDB. При помощи SSMS мы можем подключиться и к этому серверу, но только локально. Для версии SQL Server 2012 или меньше имя сервера **(localdb)\v11.0**, для более поздних **(localdb)\MSSQLLocalDB**.

В поле «**проверка подлинности**» возможен выбор одного из двух вариантов:

- Проверка подлинности **WINDOWS**
- Проверка подлинности **SQL SERVER**

Вариант «Проверка подлинности WINDOWS» всегда доступен пользователю независимо от того, как была выполнена настройка конфигурации сервера. В случае выбора опции «Проверка подлинности WINDOWS» сведения о пользователе проверяются в домене ОС WINDOWS и отображаются на роли, соответствующие учетной записи, а роли указывают какие действия с системой может выполнять пользователь. Наиболее удобной особенностью такого подхода является то, что для того чтобы войти в систему пользователю не придется заполнять о себе все сведения, СУБД использует регистрационную информацию с которой пользователь в данный момент времени подключен к WINDOWS.

При использовании варианта организации защиты на основе параметра «Проверка подлинности SQL SERVER» полностью игнорируется информация, какие права предусмотрены для пользователя в сети, а рассматриваются только права явно заданные в СУБД SQL SERVER. Преимуществом такого варианта организации защиты является то, что администратор СУБД не обязан

брать на себя функции администратора домена, чтобы представить права доступа пользователя к базе данных.

Имя пользователя

В это поле необходимо ввести имя пользователя для соединения. Этот параметр доступен только при соединении с использованием метода проверки подлинности Windows.

Имя входа

В это поле вводится имя входа для подключения. Этот параметр доступен только в том случае, если выбрано соединение с использованием проверки подлинности SQL Server.

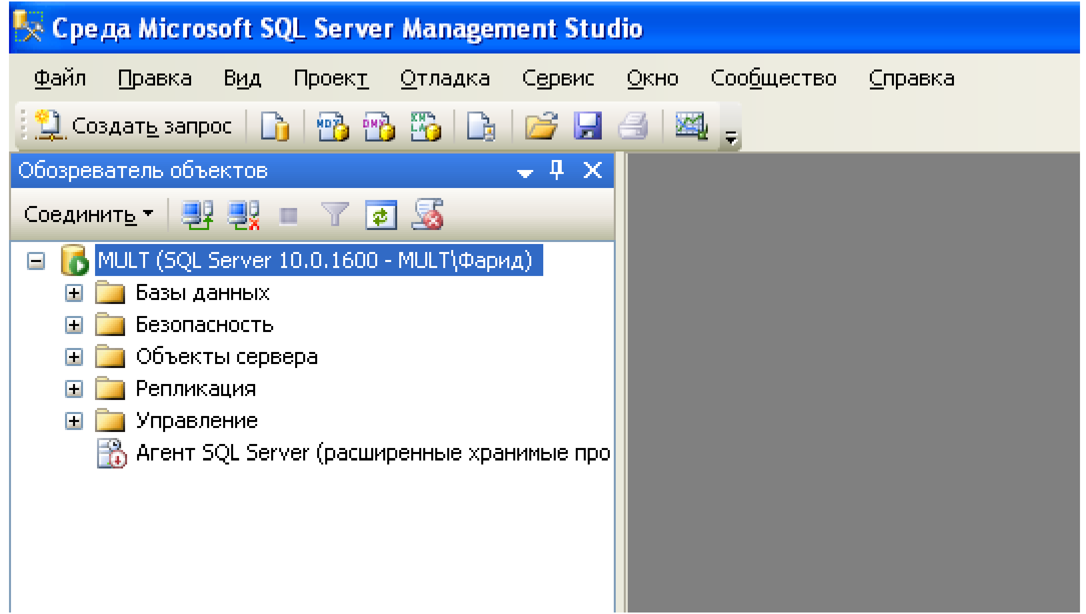
Пароль

Введите пароль для этого имени входа. Этот параметр доступен для редактирования только при подключении с проверкой подлинности SQL Server.

После установки всех параметров необходимо нажать кнопку «Соединить»

4. Создание базы данных. Работа в среде Microsoft SQL Server Management Studio

После вызова среды MicroSoft SQL Server Management Studio мы получим рабочий экран программы:



В списке «Обозреватель объектов» перечислены различные разделы, связанные с сервером, Большинство разделов преимущественно связано с администрированием базы данных. Нас с Вами будет в основном интересовать раздел «Базы данных»

Откроем пункт меню, связанный с базами данных. В окне «Обозреватель объектов» появится список зарегистрированных сервере баз данных. Этот список может сильно варьироваться в зависимости от момента времени, но в любом списке имеется информация о системных базах данных. Существуют четыре системных базы данных

Имя БД	Тип БД	Описание
Master	Системная	Содержит информацию о всех БД, установленных на сервере. Изменяется каждый раз, когда создаются БД, изменяются учетные записи или параметры конфигурации. Содержит все расширенные или

		системные хранимые процедуры. В этой базе хранится почти вся информация, которая описывает конкретную инсталляцию. Желательно всегда иметь актуальную копию системной базы данных Master.
Model	Системная	Представляет шаблоны новых БД. Если требуется, чтобы все новые БД имели определенные свойства или набор разрешений, нужно ввести их в эту БД. Все объекты из Model переносятся во вновь создаваемую базу данных. В базу данных model можно добавлять любые объекты, такие как таблицы, представления, хранимые процедуры, типы данных и т.д., которые войдут в состав всех вновь созданных баз данных.
Tempdb	Системная	Представляет временное рабочее пространство для обработки запросов и выполнения других задач. Данная БД воссоздается заново при каждом запуске SQL Server из БД Model. Имена объектов временной базы данных имеют префикс #.
Msdb	Системная	Используется службой SQL Agent. База данных Msdb содержит задания (task-scheduling), обработку исключений, аварийное управление и информацию об операторах системы, то есть содержит информацию для всех операторов об их адресах электронной почты или номерах пейджера, а также информацию об истории по всем сеансам резервного копирования или восстановления баз данных.

Любая БД включает в себя следующие основные объекты:

- таблицы
- представления
- ограничения
- умолчания
- индексы
- ключи
- хранимые процедуры
- расширенные хранимые процедуры

- триггеры
- пользовательские функции

Для любых объектов БД в MS_SQL предусмотрено соглашение, допускающее использование четырех уровней наименования. Полностью имя имеет такую структуру:

[ServerName.[DatabaseName.[ShemaName.]]]ObjectName[1]

server_name

Указывает имя связанного или удаленного сервера.

database_name

Указывает имя базы данных SQL Server, если объект хранится на локальном экземпляре SQL Server. Когда объект находится на связанном сервере, аргумент database_name указывает каталог OLE DB.

schema_name

Если объект находится в базе данных SQL Server, указывает имя схемы, которая содержит объект. Когда объект находится на связанном сервере, аргумент schema_name указывает имя схемы OLE DB.

object_name

Ссылается на имя объекта.

Параллельно могут использоваться несколько серверов, быть доступными несколько баз данных. Поэтому при указании имени объекта необходимо в каких-то случаях оговаривать полное имя объекта. Если работа идет с одним сервером, то имя сервера опускается. При ссылке на конкретный объект нет необходимости всякий раз указывать сервер, базу данных и схему — компонент SQL Server Database Engine попытается определить этот объект. Однако если объект не удается найти, возвращается ошибка.

Активная БД может быть назначена оператором с использованием инструкции

USE <database name>

[1] Информация, заключенная в квадратные скобки может быть опущена

4.1 Создание базы данных

Создадим базу данных в **MS_SQL SERVER,** состоящую из описанных выше трех таблиц. Для хранения базы данных создадим папку **D:\SFI\bd\KURS_PROJ.** Для создания базы данных воспользуемся средой **Microsoft SSMS.**

Создание базы данных (впрочем, как и многие другие операции) можно выполнить несколькими способами:

- Используя команды языка Transact-SQL
- Используя диалоговые окна
- Используя готовые шаблоны

Синтаксис любого оператора в T_SQL имеет вид:

ОПЕРАТОР <object type> <object name>

Для создания базы данных используется оператор

CREATE DATABASE <имя базы данных>. Ниже приведен не полный синтаксис этого оператора

```
CREATE DATABASE database_name
[ ON
    [ PRIMARY ] [ <filespec> [ ,...n ]
    [ , <filegroup> [ ,...n ] ]
[ LOG ON { <filespec> [ ,...n ] } ]
]
[ COLLATE collation_name ]
[ WITH <external_access_option> ]
]
[;]

<filespec> ::=
{(
    NAME = logical_file_name ,
     FILENAME = { 'os_file_name' | 'filestream_path' }
```

```
         [ , SIZE = size [ KB | MB | GB | TB ] ]
        [ , MAXSIZE = { max_size [ KB | MB | GB | TB ] |
UNLIMITED } ]
         [ , FILEGROWTH = growth_increment [ KB | MB | GB | TB |
% ] ]
) [ ,...n ]
}

<filegroup> ::=
{
FILEGROUP filegroup_name [ CONTAINS FILESTREAM ] [ DEFAULT ]
    <filespec> [ ,...n ]
}
```

Перечислим основные параметры оператора **CREATE DATABASE:**

database_name

Имя создаваемой базы данных. Имена баз данных должны быть уникальны внутри одного экземпляра SQL Server и должны соответствовать правилам для написания идентификаторов

ON

Указывает, что дисковые файлы, используемые для хранения разделов данных в базе данных, определяются явно. Параметр ON необходимо применять, если за ним следует список разделенных запятыми элементов <filespec>, которые определяют файлы данных первичной файловой группы. За списком файлов в первичной файловой группе может следовать необязательный список элементов <filegroup>, разделенных запятыми, которые определяют файловые группы пользователей и принадлежащие им файлы.

Если параметр PRIMARY не указан, то первый файл списка в инструкции CREATE DATABASE становится первичным файлом.

PRIMARY

Указывает, что связанный список <filespec> определяет первичный файл. Первый файл, указанный в элементе <filespec> в первичной файловой группе, становится первичным файлом. В базе

данных может быть только один первичный файл. Если параметр PRIMARY не указан, то первый файл списка в инструкции CREATE DATABASE становится первичным файлом.

LOG ON

Указывает, что дисковые файлы, используемые для хранения журнала базы данных, то есть файлы журналов, определяются явно. За параметром LOG ON следует список элементов <filespec>, разделенных запятыми, которые определяют файлы журналов. Если параметр LOG ON не указан, автоматически создается один файл журнала, размер которого определяется большей из следующих двух величин: 512 КБ или 25 процентов от суммы размеров всех файлов с данными в базе данных.

COLLATE *collation_name*

Задает параметры сортировки по умолчанию для базы данных. Именем параметров сортировки может быть либо имя параметров сортировки Windows, либо имя параметров сортировки SQL. Если этот параметр не указан, базе данных назначаются параметры сортировки по умолчанию экземпляра SQL Server.

Ключевое слово COLLATE имеет отношение к проблемам выбора порядка сортировки, чувствительности к регистру, а также зависимости от выбранной кодовой страницы.

При создании базы данных определяются различные ее параметры, связанные с моделью восстановления базы данных, с репликацией и зеркалированием.

<filespec>

Управляет свойствами файла.

NAME *logical_file_name*

Задает логическое имя файла.

logical_file_name

Логическое имя, используемое в SQL Server при указании ссылки на файл. Аргумент Logical_file_name должен быть уникальным в базе данных и соответствовать правилам для идентификаторов. Имя может быть символом или константой Юникода (в этом случае строка предваряется префиксом N), а также обычным идентификатором или идентификатором с разделителями.

FILENAME { '*os_file_name*' | '*filestream_path*' }

Задает имя файла в операционной системе (физическое имя).'os_file_name' Путь и имя файла, используемые операционной системой при создании файла. Указанный путь должен существовать до выполнения инструкции CREATE DATABASE.

SIZE *size*

Указывает начальный размер файла. Если аргумент size не задан для первичного файла, то компонент Database Engine использует размер первичного файла, указанный в базе данных model. Если указан вторичный файл данных или журнала, но параметр size для этого файла не указан, компонент Database Engine задает размер файла равным 1 МБ. Размер, указанный для первичного файла, должен быть не менее размера первичного файла базы данных model.

MAXSIZE *max_size*

Задает максимальный размер, до которого может расти файл. Можно использовать суффиксы KB, MB, GB и TB. Значение по умолчанию — MB. Укажите целое число, не включая десятичную часть. Если аргумент max_size не указан, размер файла будет увеличиваться до заполнения диска.

FILEGROWTH *growth_increment*

Задает автоматический шаг роста файла. Значение параметра FILEGROWTH для файла не может превосходить значение

параметра MAXSIZE. growth_increment - объем пространства, добавляемого к файлу каждый раз, когда требуется увеличение пространства. Значение может быть указано в килобайтах, мегабайтах, гигабайтах, терабайтах или процентах (%). Если указано число без суффикса МБ, КБ или %, то по умолчанию используется значение MB. Если размер шага роста указан в процентах (%), размер увеличивается на заданную часть в процентах от размера файла. Указанный размер округляется до ближайших 64 КБ. Значение 0 указывает, что автоматическое приращение отключено и добавление пространства запрещено.

FILEGROUP *filegroup_name*

Логическое имя файловой группы. Аргумент filegroup_name должен быть уникальным в базе данных и не может быть именем PRIMARY или PRIMARY_LOG, предоставленным системой.

CONTAINS FILESTREAM

Указывает, что файловая группа хранит большие двоичные объекты (BLOB) FILESTREAM в файловой системе.

Каждый раз при создании базы данных автоматически устанавливаются несколько параметров базы данных. Список этих параметров и их значения по умолчанию берутся из базы данных Model. Значения этих параметров можно изменить с помощью инструкции ALTER DATABASE.

Рассмотрим некоторые примеры:

ПРИМЕР 1

```
USE master;
GO
CREATE DATABASE Archive
ON
PRIMARY
```

```
    (NAME = Arch1,
    FILENAME = 'D:\SalesData\archdat1.mdf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20),
    (NAME = Arch2,
    FILENAME = 'D:\SalesData\archdat2.ndf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20),
    (NAME = Arch3,
    FILENAME = 'D:\SalesData\archdat3.ndf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20)
LOG ON
   (NAME = Archlog1,
    FILENAME = 'D:\SalesData\archlog1.ldf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20),
   (NAME = Archlog2,
    FILENAME = 'D:\SalesData\archlog2.ldf',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 10);
GO
```

В этом примере речь идет о создании базы данных Archive, состоящей из трех файлов (одного первичного и двух вторичных файлов), объединенных в группу PRIMARY, а также двух файлов журнала. Поскольку информация о новой базе данных фиксируется в базе данных Master, перед созданием любой базы данных необходимо активизировать эту базу данных. Команда GO разделяет пакет на части, чтобы синхронизировать последовательность выполнения инструкций в пакете.

Каждый из создаваемых файлов базы данных имеет свою спецификацию, содержащую логическое имя (NAME), адрес физического расположения на диске (FILENAME), размер(SIZE), максимальный размер(MAXSIZE) и размер возможного роста(MAXSIZE).

Пример 2

```
USE master;
GO
CREATE DATABASE Sales
ON PRIMARY
( NAME = SPri1_dat,
    FILENAME = 'D:\SalesData\SPri1dat.mdf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 15% ),
( NAME = SPri2_dat,
    FILENAME = 'D:\SalesData\SPri2dt.ndf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 15% ),
FILEGROUP SalesGroup1
( NAME = SGrp1Fi1_dat,
    FILENAME = 'D:\SalesData\SG1Fi1dt.ndf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 ),
( NAME = SGrp1Fi2_dat,
    FILENAME = 'D:\SalesData\SG1Fi2dt.ndf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 ),
FILEGROUP SalesGroup2
( NAME = SGrp2Fi1_dat,
    FILENAME = 'D:\SalesData\SG2Fi1dt.ndf',
    SIZE = 10,
```

```
    MAXSIZE = 50,
    FILEGROWTH = 5 ),
( NAME = SGrp2Fi2_dat,
    FILENAME = 'D:\SalesData\SG2Fi2dt.ndf',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 )
LOG ON
( NAME = Sales_log,
    FILENAME = 'E:\SalesLog\salelog.ldf',
    SIZE = 5MB,
    MAXSIZE = 25MB,
    FILEGROWTH = 5MB ) ;
```

Второй пример отличается от первого наличием групп вторичных файлов. Первичная группа PRIMARY состоит из двух файлов SPri1_dat и SPri2_dat, определены две группы вторичных файлов SalesGroup1 и SalesGroup2, а также один файл журнала. В этом примере файлы, входящие в одну группу расположены на разных дисках.

Создадим базу данных для нашей задачи[2], используя команду CREATE DATABASE.

С этой целью, войдя в раздел Базы данных обозревателя объектов, нажмем на кнопку «Создать запрос» и в открывшемся окне для написания команд языка T_SQL наберем следующий текст:

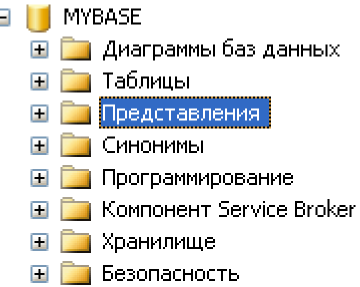
```
USE [master]
GO
CREATE DATABASE [MYBASE] ON  PRIMARY
( NAME = N'MYBASE', FILENAME = N'D:\SFI\bd\KURS_PROJ\MYBASE.mdf'
, SIZE = 3072KB , MAXSIZE = UNLIMITED, FILEGROWTH = 1024KB )
 LOG ON
```

[2] При решении нашей задачи мы предполагаем, что вопросы, связанные с проектированием и нормализацией уже проработаны, и на этом останавливаться не будем.

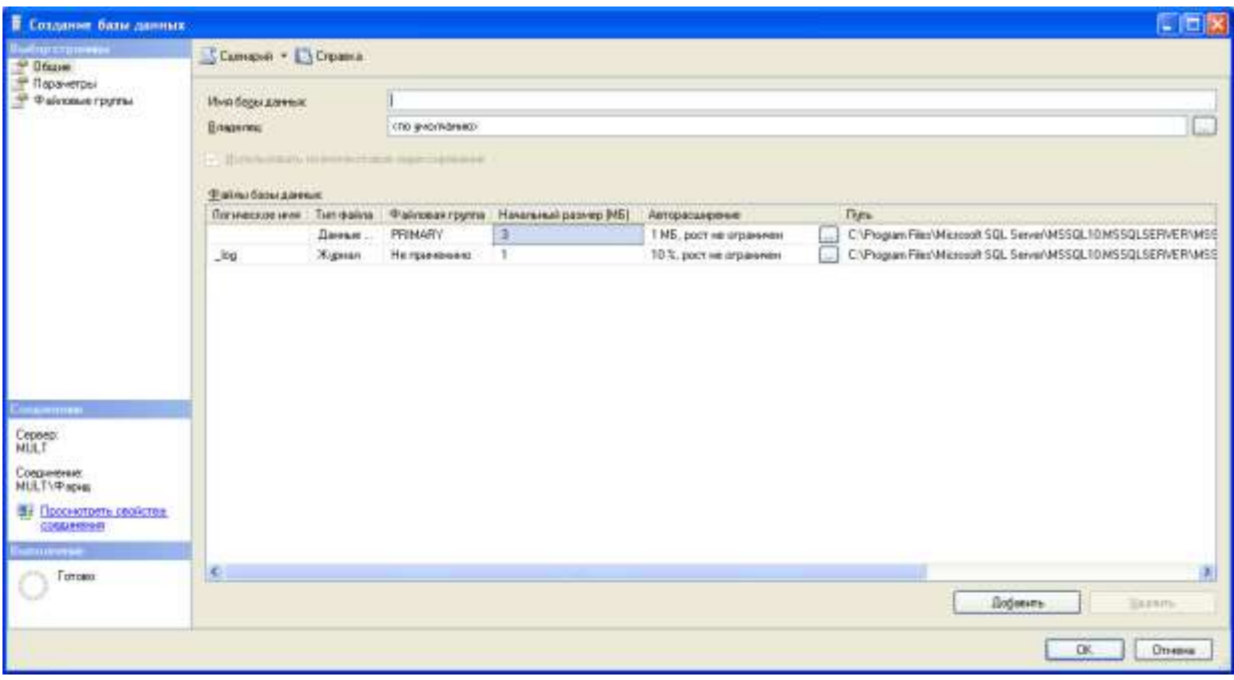
```
( NAME = N'MYBASE_log', FILENAME =
N'D:\SFI\bd\KURS_PROJ\MYBASE_log.ldf' , SIZE = 1024KB , MAXSIZE
= 2048GB , FILEGROWTH = 10%)
GO
```

Префикс N перед символьными строками означает, что кодирование информации производится в кодировке UNICODE. При создании базы данных Вы должны откорректировать пути к папкам согласно месту расположения Вашей базы данных.

После завершения оформления команды создания базы данных необходимо нажать кнопку . В результате таких действий должна быть создана пустая база данных. Не всегда можно увидеть результат выполнения команды непосредственно на экране. Часто бывает необходимо «освежить» соединение с базой данных. Для этой цели предназначена кнопка Refresh(«обновить») . Если после такой операции имя создаваемой базы данных не появится в списке зарегистрированных баз на сервере, отсоединитесь (кнопка) и затем вновь подключитесь (кнопка) к обозревателю объектов. После появления базы данных MYBASE в списке обозревателя объектов, можно войти в соответствующий контейнер и продолжить работу по созданию объектов базы данных.



Альтернативный способ создания базы данных заключается в использовании диалогового сценария, заложенного в системе. Для этого необходимо поставить указатель мыши на объект «Базы данных», нажать правую кнопку мыши и в появившемся контекстном меню выбрать пункт «Создать базу данных». В появившейся экранной форме



ввести информацию о логическом имени базы данных, о ее владельце (информацию о владельце можно почерпнуть из списка доменов ОС), а также добавить описание файлов базы данных, которое включает путь в операционной системе к файлу, размер файла, опции, связанные с расширением дискового пространства, отведенного под файл, информацию по файловой группе, к которой относится файл.

На вкладке «Параметры» можно отредактировать список параметров, которые приписаны к данной базе данных. **Важными** из них являются параметры сортировки (**Collate**), а также модель восстановления, которая определяет режим журнализации данных.

Во вкладке «Файловые группы» доступен инструмент добавления или удаления файловых групп.

Имеется возможность просмотра результата работы с экранным сценарием в виде SQL оператора, используя кнопку «сценарий».

Третий способ определения базы данных заключается в использовании шаблонов.

SQL сервер содержит библиотеку шаблонов. Доступ к библиотеке шаблонов можно получить из главного меню (подпункт вид\обозреватель шаблонов). В правой части экрана появится доступ к библиотеке, которая содержит шаблоны на основные операторы языка T-SQL. Вызвав из библиотеки шаблон соответствующего оператора базы данных, далее можно настроить

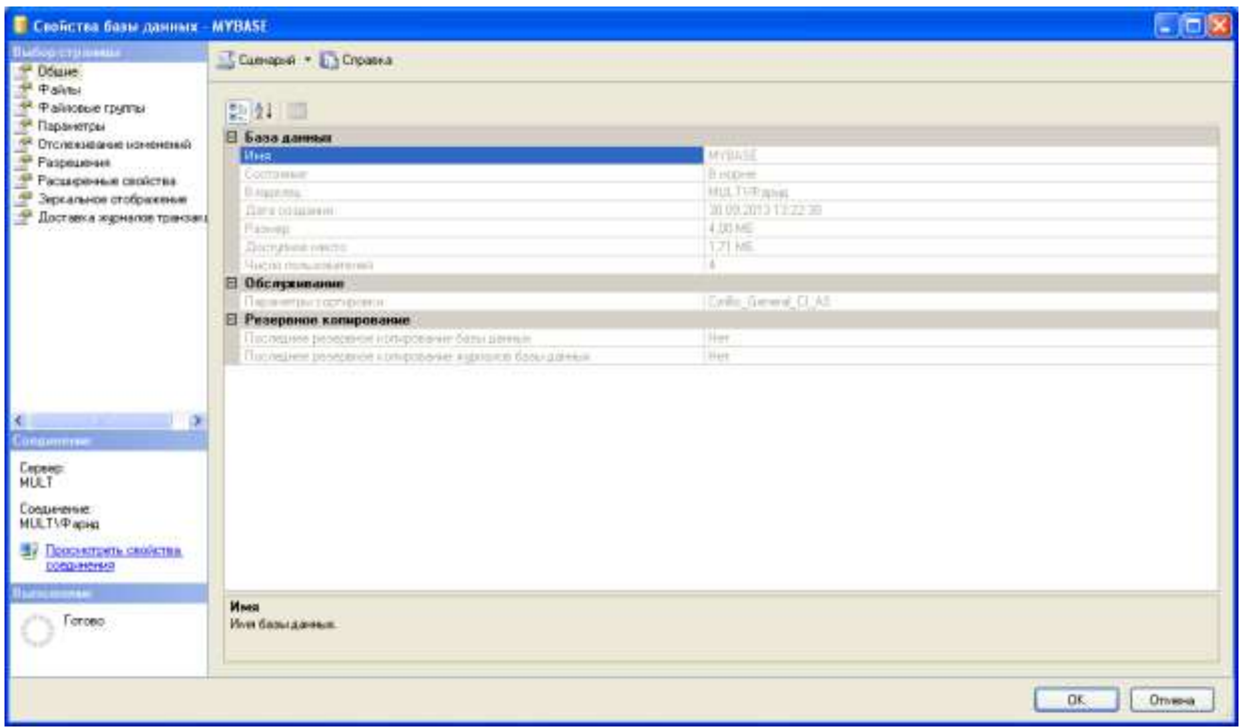
его для конкретной задачи и запустить полученный оператор. В частности, шаблон для оператора CREATE DATABASE выглядит следующим образом:

```
USE master
GO

-- Drop the database if it already exists
IF  EXISTS (
     SELECT name
          FROM sys.databases
          WHERE name = N'<Database_Name, sysname, 
Database_Name>'
)
DROP DATABASE <Database_Name, sysname, Database_Name>
GO

CREATE DATABASE <Database_Name, sysname, Database_Name>
GO
```

Этот сценарий сначала проверяет наличие базы данных с заданным именем в системе (таблица sys.databases из базы Master), если соответствующее имя найдено удаляет базу данных с этим именем из сервера, а дальше создает новую базу данных с указанным именем. Заметим, что в этом коде практически все параметры базы данных определяются по умолчанию. Пользователь может, редактируя шаблон, получить нужный ему сценарий.

После создания объекта базы данных можно использовать различные сценарии ее создания повторно, используя соответствующие пункты контекстного меню базы данных, появляющееся при нажатии правой кнопки мыши при фокусировании указателя мыши на конкретной базе данных.

Сведения о созданной базе данных можно получить, выбрав пункт «свойства» того же контекстного меню.



При попытке скопировать файлы базы данных возникают сложности, связанные с тем, что сервер не позволит это сделать для зарегистрированных баз данных. Поэтому, сначала необходимо отсоединить базу данных от сервера, скопировать файлы базы данных на другой носитель, а затем вновь присоединить базу данных. Делается это с использованием специальной модификации команды CREATE DATABASE. При этом необходимо перевести базу данных в монопольный режим с одним пользователем.

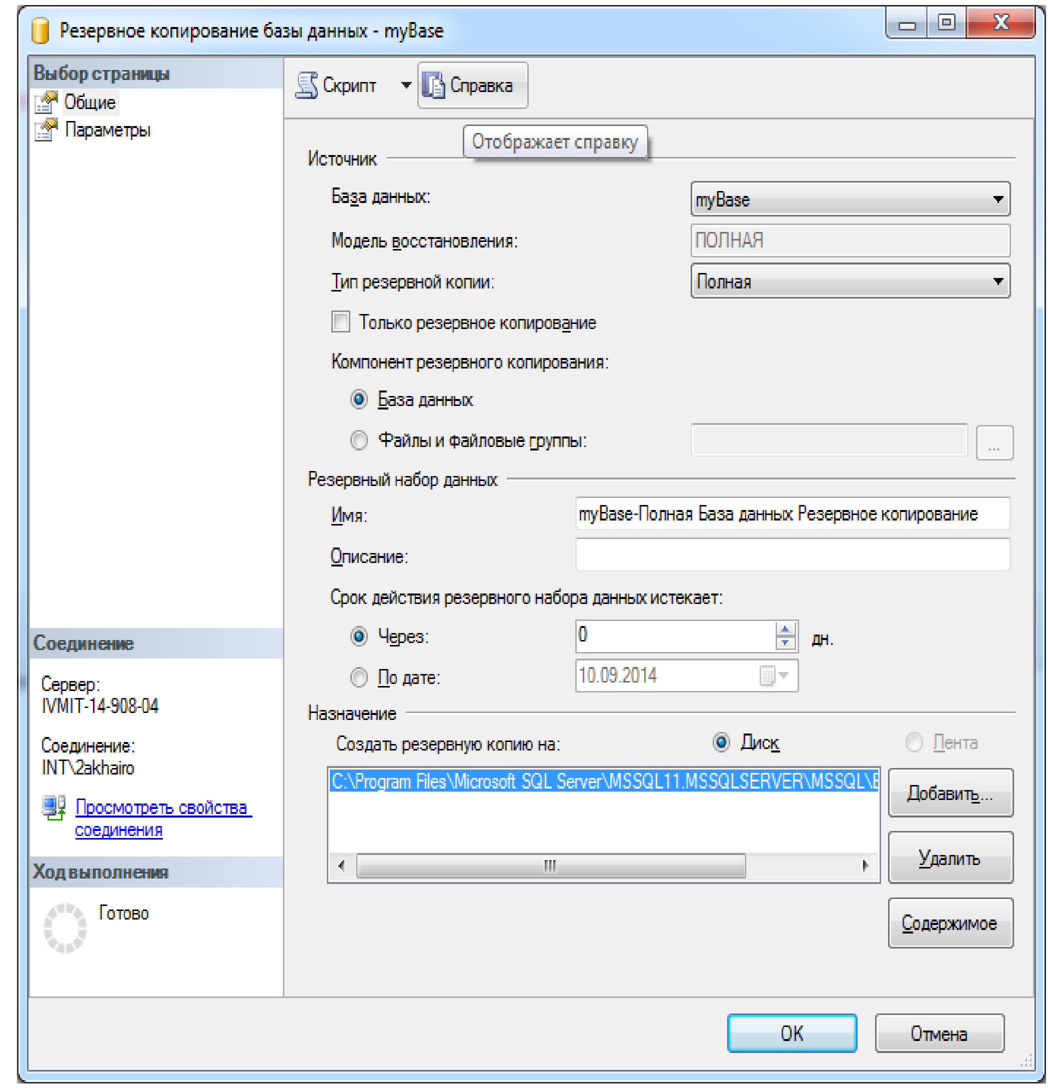
В следующем примере база данных Archive отсоединяется, а затем присоединяется с помощью предложения FOR ATTACH. База данных Archive определена, как база данных с несколькими файлами данных и журналов. Однако, поскольку местоположение файлов не изменилось со времени их создания, в предложении FOR ATTACH должен быть задан только первичный файл. Начиная с SQL Server 2005 любые полнотекстовые файлы, являющиеся частью присоединяемой базы данных, будут присоединены вместе с базой данных.

```
USE master;
GO
sp_detach_db MYBASE;
GO
```

```
CREATE DATABASE MYBASE
        ON (FILENAME = N'D:\SFI\bd\KURS_PROJ\MYBASE.mdf')
        FOR ATTACH ;
GO
```

В данном примере системная процедура sp_detach_db отсоединяет базу данных MYBASE. После этого пользователь может проделать с файлами базы данных необходимые действия. Вариант команды CREATE DATABASE FOR ATTACH вновь подсоединяет базу данных к серверу, при этом системе достаточно информации, содержащейся в основном файле с расширением mdf.

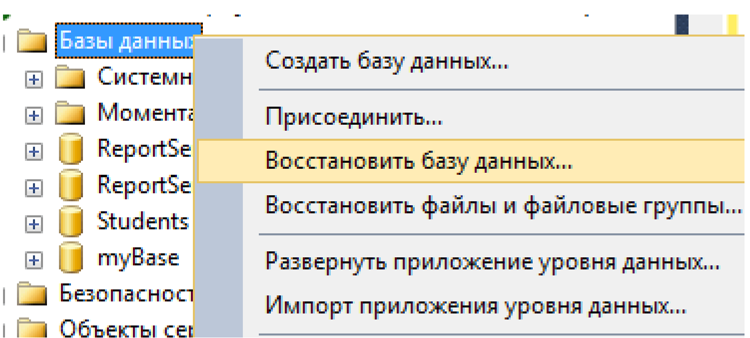
Для переноса БД с одного компьютера на другой также можно воспользоваться механизмом резервного копирования.



. Он сохраняет как и метаданные вашей базы, так и все хранящиеся в ней данные. Заходим в контекстное меню в списке баз данных и

выбираем пункт «Задачи» -> «Создать резервную копию…». Выбираем БД, которую хотим выгрузить и задаем имена файлов и пути куда будет выгружаться резервная копия. По умолчанию, SQL Server сохраняет в файл с именем БД и расширением .bak в специальную папку Backup

Обратная процедура, извлечения базы данных из резервной копии производится при помощи команды контекстного меню «Восстановить базы данных…». В диалоге выбираем файл, из которого будет производиться восстановление. Далее можно будет задать остальные параметры БД.



5. Создание объектов базы данных

5.1 Таблицы

База данных основную информацию хранит в таблицах. Таблица представляет собой объект, состоящий из набора поименованных столбцов (атрибутов), в каждом из которых можно записывать информацию определенного типа. На эту информацию могут быть наложены определенные ограничения. Кроме того, информация, записанная в различных столбцах таблицы может быть связана определенными соотношениями (вычислимые поля), при создании таблицы можно определить значения полей по умолчанию и т.д. Как и в случае базы данных создание таблицы может быть запрограммировано в системе различными способами. Рассмотрим подробнее способы создания таблицы с использованием языка T-SQL и создание объекта таблица, используя диалоговые средства системы.

В SQL сервер используется следующие типы данных:

- **Битовые типы данных** которые содержат последовательности нулей и единиц: Binary(n) и Varbinary(n), где n длина. Длина содержимого полей типа Binary всегда равна n, разница заполняется пробелами. Varbinary размер поля равен n или меньшему;
- **Целочисленные типы данных** - типы данных для хранения целых чисел (в скобках указан диапазон значений типа данных, примерно): Tinyint (0-255), Smallint ($\pm 2^{15}$), Int ($\pm 2^{31}$), Bigint ($\pm 2^{63}$);
- **Типы данных для хранения дробных чисел:** Real семь знаков после запятой, Float(m) может хранить числа из m знаков, максимальное m=38, Decimal(m, n) дробные числа с m знаков до запятой и n после;
- **Специальные типы данных:** Bit - логический тип данных.является заменой логическому типу Boolean в

Visual Basic, Text - тип для хранения больших объемов текста, одно поле может хранить до 2 Гб текста, Image - тип данных для хранения до 2Гб рисунков, RowGUID - уникальный идентификатор строки таблицы, SQL_Variant - аналогичен типу Variant в Visual Basic;

- **Типы данных даты и времени:** Datetime (1 января 1753 года — 31 декабря 9999 года). SmallDatetime (от 1.01.1900 до 06.06.2079);
- **Денежные типы данных для хранения финансовой информации:** Money (от -922 337 203 685 477,5808 до 922 337 203 685 477,5807), Smallmoney (от -214 748,3648 до 214 748,3647);
- **Автоматически обновляемые типы данных** - аналоги счетчиков, но в данной роли они не используются: RowVersion уникальный идентификатор строки. TimeStamp - закодированное дата и время создания строки.

Оператор CREATE TABLE имеет довольно сложную структуру. Ниже приводится синтаксис усеченного варианта оператора:

```
CREATE TABLE [ database_name . [ schema_name ] . | schema_name . ] table_name
    ( { <column_definition> | <computed_column_definition> }
        [ <table_constraint> ] [ ,...n ] )
        [ ON { partition_scheme_name ( partition_column_name )
        [ WITH ( <table_option> [ ,...n ] ) ]
    [ ; ]
```

При определении таблицы указывается имя создаваемого объекта, перечисляется набор столбцов, для каждого из которых указывается тип содержащегося в нем информации, а также системы правил, которые регламентируют какие данные могут храниться в столбце. Попытка нарушить эти правила вызывает негативную реакцию системы, вплоть до отказа выполнения определенных действий с хранимыми данными. Каждая таблица может иметь множество индексов, каждый из которых представляет собой набор ключевых

значений и в основном используется при поиске информации. Существуют два класса индексов:

1) кластиризованный
2) некластиризованный

Кластиризованный индекс существует в единственном числе. Задание кластиризованного индекса означает физическую сортировку данных в порядке, задаваемым в соответствие с этим индексом.

Некластиризованные индексы определяют логический порядок следования записей в соответствие с некоторыми условиями. Одна таблица может содержать несколько некластиризованных индексов.

```
<column_definition> ::=
column_name <data_type>
    [ FILESTREAM ]
         [ COLLATE collation_name ]
     [ NULL | NOT NULL ] [ CONSTRAINT constraint_name ]
      DEFAULT constant_expression ]
      | [ IDENTITY [ ( seed ,increment ) ]
```

Параметр COLLATE определяет способ сортировки для содержимого столбца, DEFAULT – задает значение по умолчанию, IDENTITY [(seed ,increment) – указывает, что поле является счетчиком, seed – начальное значение счетчика, increment – значение разности между соседними значениями. NULL – специальное значение, предусмотренное стандартом SQL, которое означает неопределенное значение.

```
USE MYBASE
GO
CREATE TABLE [dbo].[TOVARY](
     [KOD_TOVAR] [int] IDENTITY(1,1) NOT NULL CONSTRAINT
PK_TOVARY PRIMARY KEY CLUSTERED,
     [TOVAR] [varchar](20) NOT NULL,
     [ED_IZM] [varchar](10) NULL,
```

```
    [ZENA] [int] NULL DEFAULT (0) CONSTRAINT CH_ZENA CHECK
([ZENA]>=(0)),
    [COUNT_TOV] [int] NULL DEFAULT (0),
) ON [PRIMARY]
GO
```

Приведенный выше сценарий создает таблицу TOVARY с набором из 5 полей **KOD_TOVAR**, **TOVAR**, **ED_IZM**, **ZENA**, **COUNT_TOV**, поле **KOD_TOVAR** объявляется ключевым (первичный ключ, таблица физически упорядочена по этому полю). Параметр **NOT NULL** вводится для контроля начального заполнения поля при его обработке (в этом случае, если поле не получило никакого значения, на уровне сервера вырабатывается исключительная ситуация; мы сознательно не поставили этот параметр на полях **ED_IZM** и **COUNT_TOV** для демонстрации обработки соответствующей ошибки на уровне клиентского места). Кроме того, параметр **DEFAULT** задает значение соответствующего поля по умолчанию. Поскольку поле **ZENA** не может принимать отрицательных значений, то для этого поля определен предикат **ZENA >= 0**, исполняющий роль сторожа. Ложное значение предиката генерирует исключительную ситуацию. Поле **COUNT_TOV** также должно принимать неотрицательные значения. При проектировании базы данных проверка этого предиката оставлена клиентской части программы. **PRIMARY KEY** – ограничение, указывающее, что в данной таблице данное поле представляет собой первичный ключ (составной первичный ключ таким образом объявлять нельзя!). При использовании этого ограничения создается первичный индекс. В случае составного ключа ограничение выносится на уровень таблицы. Другим вариантом ограничения может быть свойство **UNIQUE**, указывающее, что в данном поле могут храниться только уникальные значения. При использовании этого ограничения создается уникальный индекс.

Для нашего примера первичным ключом является целочисленное поле **KOD_TOVAR**, которое не несет смысловую нагрузку. Значения такого поля для различных записей должны быть разными по определению. Поскольку это поле не имеет содержательного смысла и используется только для организации связей между таблицами, то

заполнение этого поля (проверку уникальности его значения) можно поручить серверу базы данных. Для этого удобно использовать свойство **IDENTITY.** Каждому полю со свойством **IDENTITY** SQL Server ставит в соответствие свой счетчик и будет следить за нумерацией вновь создаваемых записей.

SQL операторы могут быть загружены (сохранены) из текстового файла с расширением .sql

Перед созданием таблицы «Покупатели» создадим пользовательский тип **KEY_TYPE**, используя оператор:

```
CREATE TYPE [dbo].[KEY_TYPE] FROM [int] NOT NULL
```

В данном примере пользовательский тип создается из базового типа integer с добавлением параметра ***NOT NULL.*** Определение пользовательского типа предпочтительно в том случае, когда в базе данных присутствуют различные столбцы, обладающие одними и теми же характеристиками. Пользовательский тип появится в БД в разделе «Программирование» - «Типы» - «Определяемые пользователем типы данных»

Таблица «Покупатели» определяется следующим сценарием:

```
CREATE TABLE [dbo].[POKUPATELI](
	[KOD_POKUP] KEY_TYPE IDENTITY(1,1),
	[POKUP] [varchar](30) NOT NULL,
	[GOROD] [varchar](20) DEFAULT (''),
	[ADRES] [varchar](25) DEFAULT (''),
	[TEL] [varchar](8) ,
PRIMARY KEY CLUSTERED
(
	[KOD_POKUP] ASC
) ON [PRIMARY]
) ON [PRIMARY]
```

Отметим, что при определении поля **KOD_POKUP** используется ранее определенный пользовательский тип **KEY_TYPE**, который уточняется информацией, что указанное поле представляет счетчик. Для таблицы определен первичный ключ **KOD_POKUP**, который в данном

контексте определяется на уровне таблицы (хотя мог с таким же успехом быть определен на уровне поля). Записи таблицы будут упорядочены по возрастанию (для упорядочивания записей по убыванию значения поля по убыванию необходимо заменить ключевое слово **ASC** на **DESC**)

Таблица «РАСХОД ТОВАРА» определяется оператором:

```
CREATE TABLE [dbo].[RASXOD](
     [KOD_RASH] KEY_TYPE IDENTITY(1,1),
     [DATA_RASH] [datetime] NOT NULL,
     [KOLVO] [int] DEFAULT (0),
     [STOIM] KEY_TYPE,
     [KOD_TOVAR] KEY_TYPE,
     [KOD_POKUP] KEY_TYPE,
 CONSTRAINT [PK_RASXOD] PRIMARY KEY CLUSTERED
(
     [KOD_RASH] ASC
) ON [PRIMARY]
) ON [PRIMARY]
```

Заметим, что для таблицы RASXOD можно было в качестве первичного ключа выбрать набор полей **(KOD_TOVAR,KOD_POKUP)**, дополненный, быть может, полем DATA_RASH (если один и тот же покупатель может многократно купить один и тот же товар). Поле **KOD_RASH** является искусственным и непонятным конечному пользователю. Однако значением этого поля управляет сервер, и пользователь к нему доступ не имеет, что может быть использовано при программировании межтабличных связей. Кроме того, более короткая длина ключа позволяет создавать кроме того более компактный индекс.

Поскольку в нашей задаче существует связь между таблицами (каждая покупка, описанная в таблице ***RASXOD,*** связывает конкретного покупателя, описанного в таблице **POKUPATELI** с конкретным товаром, описание которого дано в таблице **TOVARY**), необходимо зафиксировать эту связь на уровне базы данных. Отсутствие такой информации может привести к ситуации, когда

зафиксирована покупка отсутствующего товара, или товар приобретен не зафиксированным покупателем[3]. Поддержка концепции целостности данных обеспечивается установкой специальных связей. Установку подобных связей можно осуществить с применением декларативных средств обеспечения ссылочной целостности. Добавим связи между таблицами

```
ALTER TABLE [dbo].[RASXOD]
   ADD CONSTRAINT TOV_RASH
      FOREIGN KEY(KOD_TOVAR) REFERENCES TOVARY
        ON DELETE CASCADE ON UPDATE CASCADE;
GO
ALTER TABLE [dbo].[RASXOD]
    ADD CONSTRAINT POK_RASH
        FOREIGN KEY(KOD_POKUP) REFERENCES POKUPATELI
               ON DELETE CASCADE ON UPDATE CASCADE;
```

Команда **ALTER TABLE** позволяет модифицировать структуру таблиц. В нашем примере в таблицу «Расход» добавлена связь по полю **KOD_TOVAR** с родительской таблицей «Товары». Теперь SQL-сервер не допустит появления в таблице «Расход» строк со значениями поля **KOD_TOVAR**, которые не встречаются в таблице «Товары». Причем, благодаря дополнительным указаниям ON..., удаление товара или изменение кода товара в таблице «Товары» повлечет соответствующие изменения в соответствующих записях дочерней таблицы «Расход» (режим каскадного обновления содержимого таблиц)[4]. Если бы в заданной конструкции не был введен режим каскадного обновления таблиц, то сервером была бы использована RESTRICT-стратегия поддержания ссылочной целостности. В этом случае SQL-сервер не допустит удаления строки или изменения кода в родительской таблице (например, кода покупателя таблице «Покупатели», если на этого покупателя /по коду/ ссылается по меньшей мере одна строка таблицы «Расход») при наличии связанных

---

[3] Такая ситуация называется нарушением условия целостности базы данных.

[4] Рекомендуется указывать явным образом мнемоничное имя вводимого условия (CONSTRAINT). В приведенном случае - TOV_RASH. В этом случае при возникновении исключительных ситуаций сервер ссылается на это условие по данному имени. Если его не указать, то сервер сам генерирует имя. Появление пользовательских наименований в сообщениях сервера облегчает анализ исключительной ситуации.

с ней записей в дочерней таблице. Программист может запрограммировать соответствующие действия каскадного обновления и удаления в процедурах-триггерах базы данных.

5.2 Триггеры и хранимые процедуры.

Программирование триггеров представляет достаточно трудный раздел. Триггеры представляют собой специального вида хранимые процедуры, которые автоматически вызываются в ответ на определенные события. Триггеры подразделяются на два типа: триггеры языка определения данных (**DDL-триггеры**) и триггеры языка манипулирования данными (**DML-триггеры**). Триггеры первого типа используются при изменениях, вносимых в структуру данных (команды **CREATE**, **ALTER**, **DROP**) и в целом имеют достаточно узкую область применения. Триггеры DML представляют собой фрагменты кода, которые закрепляются за определенной таблицей или представлением. В отличие от хранимых процедур триггеры вызываются автоматически при наступлении определенного события, связанного с выполнением некоторых операций (обычно – вставка, модификация или удаление данных) над таблицей, за которой закреплен триггер. Триггеры не могут быть вызваны явно. В отличие от хранимых процедур входные и выходные параметры у триггеров отсутствуют.
Приведем упрощенный синтаксис оператора **CREATE TRIGGER** в T-SQL.

```
CREATE TRIGGER <trigger_Name>
ON [schema name.][table or view name]
{{(FOR|AFTER} <[DELETE}[,] [INSERT][,][UPDATE]>}|INSTEAD OF}
AS
Sql_statement;
```

ON служит для указания объекта, к которому применяется триггер.

{(FOR|AFTER} <[DELETE}[,] [INSERT][,][UPDATE]> указывает с каким типом запросов связан запуск триггера. Триггеры **AFTER** выполняются после выполнения инструкций **INSERT, UPDATE** или **DELETE.** Предложение **AFTER** по смыслу эквивалентно предложению **FOR.** Триггеры **AFTER** могут быть определены только в таблицах.

INSTEAD OF – особый тип триггера, который предназначен для выполнения кода, предусмотренного программистом вместо кода, обусловленного запросом, активизирующим триггер.

Правила работы с триггерами:

- триггеры запускаются только после выполнения вызвавшего их оператора (В случае, когда необходимо выполнить набор некоторых операций перед основным оператором, необходимо использовать триггер типа **INSTEAD OF,** в котором сначала прописываются подготовительные действия, а затем вызывается основной оператор. При этом, для того чтобы исключить рекурсивный вызов триггера, триггеры типа **INSTEAD OF** *вызываются* ровно один раз; В любом случае триггер вызывается на выполнение прежде, чем любые изменения действительно зафиксированы в базе данных;
- если при выполнении оператора возникает нарушение какого-либо ограничения или другая ошибка, триггер не срабатывает;
- триггер и вызвавший его оператор образует одну транзакцию. Если нужно из триггера отменить вызвавшую его операцию, следует выполнить откат транзакции **ROLLBACK;**
- триггер срабатывает ровно один раз для каждого оператора, независимо от количества изменяемых им записей.

При выполнении триггера создаются две рабочие таблицы **Inserted** и **Deleted.** В одной из этих таблиц (**Inserted**) хранится копии всех вставляемых строк, а в другой (**Deleted**) – копии всех удаляемых строк. Триггер, связанный с операцией обновления таблицы создает обе таблицы. Эти таблицы используются при программировании триггера, существуют с момента его запуска и уничтожаются по завершению работы триггера.

Мы не будем приводить пример триггера, который используется для принудительной поддержки правил ссылочной целостности. Если имеется возможность, рекомендуется использовать декларативные средства поддержки. Пример программирования триггеров для нашей учебной базы данных рассмотрим при реализации других связей. Анализ таблиц «Товары» и «Расход товара» показывает, что поля этих таблиц связаны по формулам:

- ***Стоимость купленного товара = Количество купленного товара * цена единицы товара,***
- ***Количество товара на складе = Количество товара на складе – Количество купленного товара.***

Эти соотношения также являются требованиями целостности (правильности) базы данных, однако имеют специальный связанный с конкретной предметной областью характер. Контроль первого требования оставим клиентской части программы.

Решение второй задачи возложим на SQL-сервер. Для этого добавим в БД триггер, отрабатывающий после ввода новых данных в таблицу **RASXOD**. Рассмотрим пример такого триггера для операции вставки данных.

```
CREATE TRIGGER AI_RASXOD ON dbo.RASXOD
     FOR INSERT
AS
BEGIN
    DECLARE @Count int;
    SET @Count = @@ROWCOUNT;
    IF @Count = 0
        RETURN;
    SET NOCOUNT ON;
    UPDATE TOVARY
         SET COUNT_TOV = COUNT_TOV-(SELECT SUM(Kolvo)
             FROM INSERTED i WHERE i.Kod_Tovar=tovary.Kod_Tovar)
      WHERE Kod_Tovar IN (SELECT Kod_Tovar FROM INSERTED i1);
END
```

В хранимых процедурах (соответственно, и в триггерах) можно определять локальные переменные, именам которых предшествует символ @. Объявление переменных имеет вид

DECLARE имя_переменной тип_переменной [(длина)]

Оператор присваивания значения переменной имеет вид

SET переменная = значение

В приведенном коде глобальная переменная **@@ROWCOUNT** возвращает число строк таблицы (в данном контексте – таблицы Rasxod), затронутых при выполнении последней инструкции вставки, которая инициализировала вызов триггера. Поскольку любая очередная команда обработки данных сбрасывает значение этой переменной, то сначала в переменной @Count запоминается это значение, и при условии, что ни одна строка не была затронута операцией вставки, триггер не срабатывает. Оператор **UPDATE** обновляет поле **COUNT_TOV** в таблице **TOVARY**, при этом информация для обновления берется из таблицы **Inserted**. В таблице **Inserted** вычисляется сумма количества проданного товара, далее происходит обновление поля **COUNT_TOV** для соответствующей записи в таблице **TOVARY.** С точки зрения оптимальности времени выполнения запроса, такая реализация является не лучшей, поскольку оператор **UPDATE** содержит подзапросы, которые выполняются для каждой строки таблицы **TOVARY.** Даже если предположить, что SQL-сервер оптимизирует запрос и создаст специальную таблицу-список для кодов товаров, включенных в таблицу **Inserted**, все равно получаем квадратичный по времени алгоритм, связанный с тем, что для каждого товара необходимо вычислить сумму изменений, зафиксированных в таблице **Inserted.** Улучшить производительность триггера можно за счет запоминания результатов группировки во временной таблице **#tmp_tbl**[5], что отражено в следующем варианте.

```
CREATE TRIGGER AI_RASXOD ON dbo.RASXOD
     FOR INSERT
```

```
AS
BEGIN
    DECLARE @Count int;
    SET @Count = @@ROWCOUNT;
    IF @Count = 0
        RETURN;
    SELECT SUM(i.Kolvo) AS kolvo, i.Kod_Tovar INTO #tmp_tbl
       FROM INSERTED i
       GROUP BY Kod_Tovar
    UPDATE TOVARY
         SET COUNT_TOV = COUNT_TOV-kolvo
          FROM TOVARY t INNER JOIN #tmp_tbl tmp ON
                    t.Kod_Tovar=tmp.Kod_Tovar;
END
```

Для того, чтобы проверить работу триггера нужно выполнить оператор обновления таблицы RASXOD.

```
INSERT INTO RASXOD (DATA_RASH,KOLVO,STOIM,KOD_TOVAR,KOD_POKUP)
    VALUES ('2008-03-06',20,500,1,5)
```
[6]

Мы рассмотрели случай обновления поля **COUNT_TOV** в таблице **TOVARY** при вставке новых записей в таблицу **RASXOD.** Однако содержимое этого поля должно меняться также при удалении записей из таблицы **RASXOD** (при условии возврата купленного ранее товара[7]), а также при модификации записей таблицы **RASXOD** (случай, когда продажа товара была оформлена с ошибкой: неправильно был внесено количество купленного товара или при регистрации покупки перепутан товар). Приняв за основу сценарий для триггера вставки, можно легко написать программный код триггера, связанного с удалением записей, используя таблицу **Deleted.**

---

[5] Временная таблица определяется в системной базе tempdb и уничтожается при завершении сеанса работы с базой данных. Имя временного объекта должно быть предварено префиксом #.

[6] Заметим, что поле KOD_RASH в операторе вставки не присутствует, поскольку оно автоматически обновляется сервером

[7] Удаление записей в таблице Rasxod предполагается чисто с учебной точки зрения. Более естественно предполагать, что таблица Rasxod ведет полный протокол, связанный с регистрацией покупок и возврат товара оформляется как покупка с отрицательным значением количества товара. Впрочем, сценарии подобных действий прописываются при проектировании базы данных.

```
CREATE TRIGGER BD_RASXOD ON dbo.RASXOD[8]
AFTER DELETE
AS
   BEGIN
    DECLARE @Count int;

    SET @Count = @@ROWCOUNT;
    IF @Count = 0
        RETURN;
    UPDATE TOVARY
        SET COUNT_TOV = COUNT_TOV+(SELECT SUM(Kolvo)
         FROM DELETED d WHERE d.Kod_Tovar=tovary.Kod_Tovar)
          WHERE Kod_Tovar IN (SELECT Kod_Tovar FROM DELETED d1);
   END;
```

Триггер, отрабатывающий после обновления данных в таблице **RASXOD** (а точнее при изменении количества проданного товара или изменении его номенклатуры) производит перерасчет количества товаров, для которых произведены изменения. В отличие от предыдущих случаев, этот триггер связан сразу с двумя таблицами **Inserted** и **Deleted**.

```
CREATE TRIGGER [dbo].[AU_RASXOD] ON [dbo].[RASXOD]
   AFTER UPDATE
AS
BEGIN
     DECLARE @Count int;
     SET @Count = @@ROWCOUNT;
     IF @Count = 0
         RETURN;
     SET NOCOUNT ON;

   IF UPDATE(KOLVO) OR UPDATE(KOD_TOVAR)
```

[8] Самостоятельно составьте сценарий для триггера удаления, который использует временную таблицу.

```
    BEGIN
      SELECT * FROM DELETED
      SET @Count = @@ROWCOUNT;
      IF @COUNT>0
        UPDATE TOVARY
           SET COUNT_TOV = COUNT_TOV+(SELECT SUM(Kolvo)
             FROM DELETED d
             WHERE d.Kod_Tovar=tovary.Kod_Tovar
             GROUP BY Kod_Tovar)
       WHERE Kod_Tovar IN (SELECT Kod_Tovar FROM DELETED d1);
      SELECT * FROM INSERTED
      SET @Count = @@ROWCOUNT;
      IF @COUNT>0
         UPDATE TOVARY
             SET COUNT_TOV = COUNT_TOV-(SELECT SUM(Kolvo)
                 FROM INSERTED i
                 WHERE i.Kod_Tovar=tovary.Kod_Tovar
                 GROUP BY Kod_Tovar)
      WHERE Kod_Tovar IN (SELECT Kod_Tovar FROM INSERTED i1);
    END
END;
```

Проанализируем более детально представленный код. В этом коде встречаются уже знакомые нам конструкции, связанные с обновлением таблицы **TOVARY** с использованием информации, содержащейся в таблицах **Deleted** и **Inserted.** Вообще говоря, обновление таблицы **TOVARY** прежде всего связано с обновлением поля **KOLVO** в таблице **RASXOD.** Это обновление должно зафиксироваться в строках каждой из таблиц **Deleted** (старое значение) и **Inserted** (новое значение) с соответствующим пересчетом значения поля **COUNT_TOV** таблицы **TOVARY.** Но триггер запускается при обновлении любого поля таблицы **RASXOD.** В библиотеке функций MS SQL Server имеется функция UPDATE(имя поля), которая принимает значение ИСТИНА на полях, значения которых изменяются основным оператором, и коде триггера происходит проверка менялись ли данные этого поля перед запуском

основной (а следовательно затратной по времени) частью кода триггера. Тем не менее, проведем более детальный анализ этой ситуации:

1. Триггер должен отрабатывать при любых изменениях содержимого таблицы **RASXOD,** которое может быть вызвано косвенно при изменениях, вносимых в родительские таблицы, связанные с таблицей **RASXOD** по внешнему ключу в каскадном не только при изменении значения поля **KOLVO**, а и в случае, когда меняется наименование купленного товара. Это может произойти при ошибке регистрации покупки. В этом случае поле **KOLVO** не меняет своего значения, а пересчет общего количества товара все равно происходит, только в таблицах **Deleted** и **Inserted** коды товаров различаются.
2. Изменение кода товара в таблице **RASXOD** может быть вызвано каскадным обновлением записей (а, следовательно, и запуском триггера) при изменении соответствующего поля в таблице **TOVARY.** В нашем случае такой вариант заблокирован свойством **IDENTITY**, которое не позволяет менять значения ключевого поля в таблице **TOVARY.** Более того, даже если бы связь строилась по полю **TOVAR** и не была защищена от изменений свойством **IDENTITY**, при изменении значения этого поля, триггер бы отрабатывал вхолостую, поскольку значения поля **KOLVO** в соответствующих записях таблицы **RASXOD** не изменялись.
3. При программировании можно было вычислить сначала разность между старым и новым значением поля **KOLVO**, построив запрос на таблицах **Deleted** и **Inserted** (связывая их по ключевым параметрам), а затем использовать эту разность при изменении таблицы **TOVARY.** Этот вариант возможен только в случае, когда содержимое ключевых полей таблицы **RASXOD** не меняется. Напомним, при создании таблицы, мы обсуждали две возможности определения первичного ключа в виде некоторого числового поля **KOD_RASH** или составного ключа **(KOD_TOVAR,KOD_POKUP)**. Во втором случае изменение наименования купленного товара приводит изменению значения ключевого параметра **KOD_TOVAR**(см. предыдущий пункт) и

разрыву связей между соответствующими записями таблиц **Deleted** и **Inserted**. Если при этом эти изменения происходят для целой группы товаров, для восстановления соответствующих связей необходимо предпринимать специальные шаги. Выбор в качестве ключа поля **KOD_RASH** (напомним, пользователь не имеет доступ к этому полю) позволяет обойти подобные проблемы.

Технология Клиент-Сервер предполагает обработку различных запросов на стороне сервера. Очень часто предполагаемые действия можно оформить в виде хранимой процедуры, которая в оттранслированном виде хранится на сервере. Предположим нам необходимо получить информацию о количественных изменениях некоторого товара за определенный промежуток времени. Этот запрос может быть сформирован следующим образом:

```
SELECT  t.tovar as 'Товар', SUM(r.Kolvo) as 'Изменение
количества'
FROM Tovary t INNER JOIN Rasxod r ONt.Kod_Tovar = r.Kod_Tovar
WHERE r.DATA_RASH>='2009-01-01 00:00:00' AND
      r.DATA_RASH<='2011-01-01 00:00:00' AND t.Tovar=N'Сахар'
GROUP BY t.Tovar;
```

Если такой запрос осуществляется достаточно часто для различных товаров и различных временных отрезков, имеются определенные доводы оформления этого запроса в виде хранимой процедуры.

```
CREATE PROCEDURE [dbo].[Get_Kol_Tov]
     (@Tovar Nvarchar(20), @DataL DateTime,
      @DataH DateTime, @SumKol Int OUTPUT)
AS
BEGIN
     SET NOCOUNT ON;
     SELECT @SumKol = SUM(r.Kolvo)
     FROM Tovary t INNER JOIN Rasxod r ON
```

```
            t.Kod_Tovar = r.Kod_Tovar
      WHERE r.DATA_RASH>=@DataL AND r.DATA_RASH<=@DataH
            AND t.Tovar=@Tovar
      GROUP BY t.Tovar;
END
```

В данном случае наша процедура имеет три входных параметра
@Tovar – наименование товара
@DataL - нижняя граница интересующего нас временного отрезка
@DataH - верхняя граница временного отрезка
И один выходной - **@SumKol** – здесь формируется результат нашего запроса.
Тело процедуры содержит тот же самый запрос, в котором конкретные значения заменены на переменные-параметры.
Вызов хранимой процедуре можно выполнить, используя сценарий

5.3 Работа с курсором

Команды манипулирования данными - **SELECT**, **UPDATE**, **DELETE** работают сразу с группами строк. Эти группы, вплоть до отдельных строк, можно выбрать с помощью опции **WHERE**. Критерии, задаваемые подобного рода условиями, распространяются на множество всех строк набора данных, что может оказаться неудобным для решения некоторых задач. С другой стороны, в процедурных языках чаще всего встречается построчная обработка данных. На самом деле после поступления данных в клиентское приложение, обработка этих данных происходит последовательно строка за строкой. Такую построчную обработку данных можно также организовать с помощью средств языка SQL. Для этого в языке SQL существует такое понятие, как *курсор*. *Курсор* (**cur**rent **s**et of **r**ecord) – это временный набор строк, которые можно перебирать последовательно, с первой до последней. Целесообразность применения курсоров обычно требует обоснования, поскольку они чаще всего снижают производительность системы.

Для работы с курсорами существуют следующие команды.

Объявление курсора:

Любой курсор создается на основе некоторого оператора

SELECT DECLARE имя_курсора CURSOR FOR SELECT текст_запроса

Открытие курсора:

OPEN имя_курсора

Только после открытия курсора он становится активным, и из него можно читать строки.

Чтение следующей строки из курсора:

FETCH имя_курсора INTO список_переменных

Переменные в списке должны иметь тот же количество и тип, что и столбцы курсора. Глобальная переменная @@FETCH_STATUS принимает ненулевое значение, если строк в курсоре больше нет. Если же набор строк еще не исчерпан, то @@FETCH_STATUS равна нулю, и оператор FETCH перепишет значения полей из текущей строки в переменные.

Закрытие курсора:

CLOSE имя_курсора

Для удаления курсора из памяти используется команда

DEALLOCATE имя_курсора

Для иллюстрации использования курсора оформим процедуру, вычисляющую наиболее популярный товар (приносящий максимальную прибыль определенный промежуток времени).

```
CREATE PROCEDURE [dbo].[Get_Kol_Tov_Kur]
     (@DataL DateTime, @DataH DateTime,
       @Tovar Nvarchar(20) OUTPUT)
AS
BEGIN
     SET NOCOUNT ON;
     DECLARE @MaxStoim int, @Stoim int, @Kod_t int,
             @Old_Kodt int, @Sum_Stoim int;
     DECLARE @MaxKod_t int
     SET @Stoim = 0;
     SET @Sum_Stoim=0;
     DECLARE my_cur CURSOR FOR
```

```
    SELECT r.KOD_TOVAR, t.ZENA*r.KOLVO AS STOIM
    FROM TOVARY t INNER JOIN RASXOD r ON
         t.KOD_TOVAR=r.KOD_TOVAR
    WHERE r.DATA_RASH>='01.01.1900' AND
          r.DATA_RASH<=GETDATE()
    ORDER BY KOD_TOVAR
    OPEN my_cur
    FETCH my_cur INTO @Kod_t, @Stoim
    SET @Old_Kodt=@Kod_t
    SET @MaxStoim=0
    SET @MaxKod_t=0
    WHILE @@FETCH_STATUS=0
    BEGIN
         WHILE @@FETCH_STATUS=0 AND @Kod_t=@Old_Kodt
         BEGIN
        SET @Sum_Stoim=@Sum_Stoim+@Stoim
        FETCH my_cur INTO @Kod_t, @Stoim
         END
         IF @MaxStoim<@Sum_Stoim
         BEGIN
         SET @MaxStoim=@Sum_Stoim
         SET @MaxKod_t=@Old_Kodt
         END
         SET @Old_Kodt=@Kod_t
         SET @Sum_Stoim=0
    END
     IF @MaxStoim<@Sum_Stoim
     BEGIN
         SET @MaxStoim=@Sum_Stoim
         SET @MaxKod_t=@Old_Kodt
     END
    SELECT @Tovar=TOVAR from TOVARY
          where Kod_tovar=@MaxKod_t
    CLOSE my_cur
    DEALLOCATE my_cur
END
```

Рассмотрим эту процедуру более подробно.

Параметром этой процедуры являются границы временного отрезка, для которого выполняется запрос, выходной параметр задает имя товара.

При объявлении курсора выбирается упорядоченный по коду товара набор данных, относящийся к продажам, попадающим заданный временной интервал, в котором каждому коду товара приписана стоимость покупки, вычисляемая как произведение цены на количество проданного товара.

Открываем курсор и читаем из него в цикле последовательно по одной строке. Каждая строка содержит информацию о стоимости отдельной продажи. Поскольку информация в курсоре отсортирована по коду товара, за один проход подсчитываем максимальную сумму стоимости товара, проданного в заданный промежуток времени, вычисляя одновременно в переменной **@MaxKod_t** код товара, на котором этот максимум достигается. Далее простым запросом к таблице **TOVARY** вычисляем наименование интересующего нас товара.

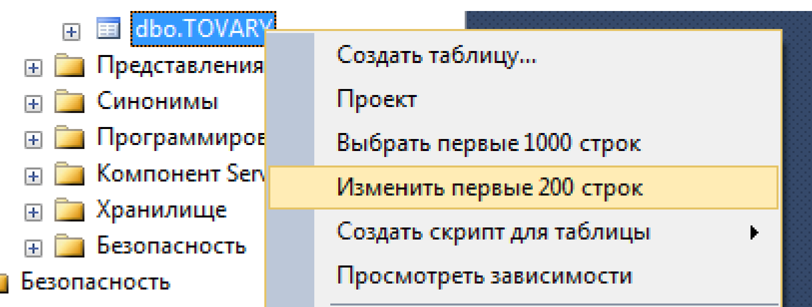
Этот пример приведен в иллюстративных целях, эту задача легко решается SQL запросом без обращения к курсору.

5.4 Заполнение таблиц.

Заполнение данных в таблице может быть выполнено в двух режимах:

1. Заполнение записей таблицы в интерактивном режиме с использованием средств SQL SERVER Management Studio.

Для этого необходимо вызвать на экран интерактивное окно свойств соответствующей таблицы (например, таблицы **TOVARY**) и выбрать пункт «Изменить первые 200 строк»

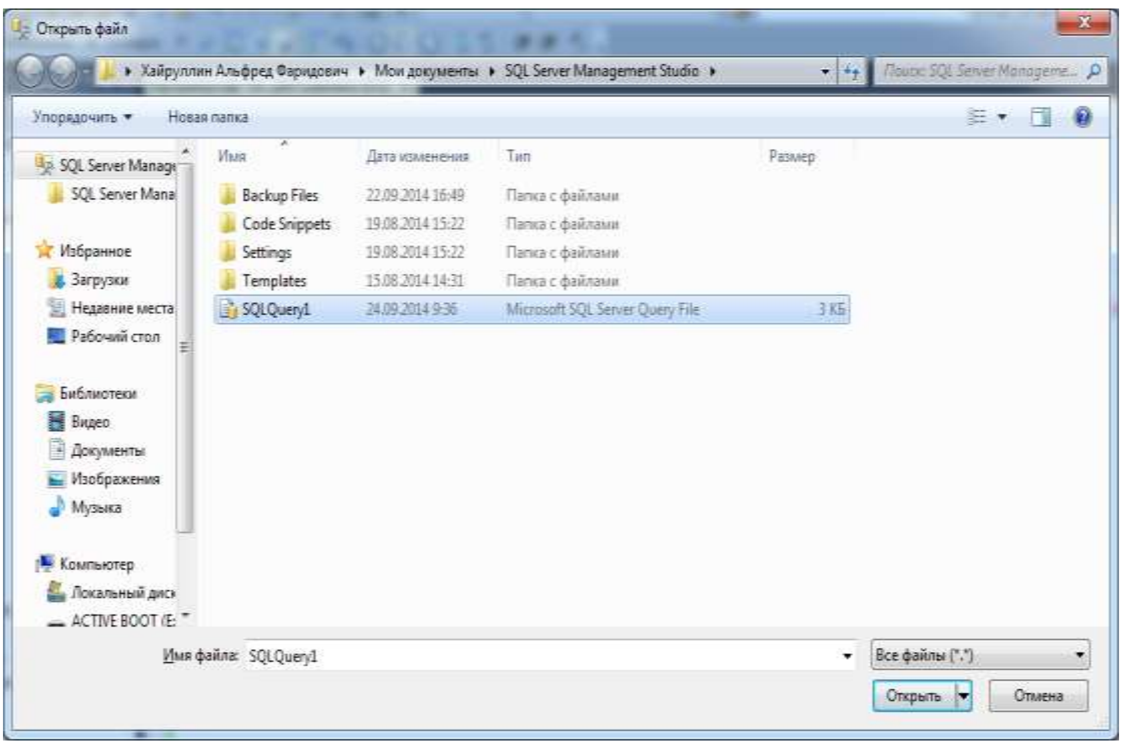


Далее можно начать набор данных. При вводе данных отрабатывают все триггера, определенные в базе данных. Поле **KOD_TOVAR**, объявленное как счетчик со свойством **IDENTITY** не доступно для изменения. Новые значения счетчика генерируются сервером без участия пользователя.

ALFREDNET.myBase - dbo.TOVARY

	KOD_TOVAR	TOVAR	ED_IZM	ZENA	COUNT_TOV
	1	Сахар	кг	30	100
	2	Макароны	упак	25	50
▸*	NULL	NULL	NULL	NULL	NULL

2. Другим вариантом заполнения таблицы является использование оператора **INSERT** языка T-SQL. Для этого (используя любой редактор текстов, например Блокнот) подготовим набор SQL операторов (SQL-Script) и выполним ее в окне редактора запросов. Поскольку в этом случае также отрабатывают декларативные средства поддержки целостности и установленные триггеры, заполнение данных необходимо производить в определенной последовательности: сначала заполняются родительские таблицы, во вторую очередь – дочерние. Поместим сценарий ввода в файл **SQLQuery1.sql** и вызвав этот код в новое окно запросов запустим его на выполнение.



Ниже приведен протокол работы этой программы:

```
USE MYBASE;
DELETE FROM RASXOD;
DELETE FROM POKUPATELI;
DELETE FROM TOVARY;

--SET IDENTITY_INSERT POKUPATELI ON;
INSERT INTO POKUPATELI (POKUP, GOROD, ADRES, TEL)
 VALUES ('Алиса','Казань','ул.Латышских стрелков','92-45-67'),
        ('Буратино','Рим','Италия','23-45-35'),
        ('Тротилла','Пруд','Италия',NULL),
        ('Шекспир','Лондон','Англия',NULL);
--SET IDENTITY_INSERT POKUPATELI OFF;

SELECT * FROM POKUPATELI;
```

```
KOD_POKUP   POKUP                          GOROD                ADRES                     TEL
----------- ------------------------------ -------------------- ------------------------- -----
1           Алиса                          Казань               ул.Латышских стрелков     92-45-67
2           Буратино                       Рим                  Италия                    23-45-35
3           Тротилла                       Пруд                 Италия                    NULL
4           Шекспир                        Лондон               Англия                    NULL

(строк обработано: 4)
```

Как видно из текста запроса сначала удаляются данные из дочерних таблиц, а потом из родительских. Далее идет группа команд, вставляющая данные в таблицу **POKUPATELI.** Включение при вводе данных параметра **IDENTITY_INSERT** в положение **ON** позволяет отключать счетчик **IDENTITY** и вводить собственные значения в поле **KOD_POKUP.** Поскольку в данном контексте соответствующий оператор закомментирован, то поле **KOD_POKUP** формируется из последовательных значений соответствующего счетчика.

При заполнении таблицы **TOVARY** отключим счетчик. В этом случае соответствующие значения кодов поля **KOD_TOVAR** задаются пользователем.

```
SET IDENTITY_INSERT TOVARY ON;
INSERT INTO TOVARY  (KOD_TOVAR,TOVAR,ED_IZM,ZENA,COUNT_TOV)
 VALUES (1,'Сахар','кг',12,1442),
        (2,'Макароны','кг',5,86),
        (3,'Огурцы весовые','кг',6,250),
        (4,'Огурцы баночные','банки',10,475),
        (5,'Крупа манная','кг',8,1010);

INSERT INTO TOVARY  (KOD_TOVAR,TOVAR,ED_IZM,ZENA,COUNT_TOV)
 VALUES (2,'Масло подсолнечное','л',10,1977);
```

В этом случае подготовленный сценарий не будет полностью отработан. При вводе записи сервер обнаружит нарушение ограничения PRIMARY KEY. Будет выдано следующее сообщение об ошибке:

Сообщение 2627, уровень 14, состояние 1, строка 12

Нарушено "PK_TOVARY" ограничения PRIMARY KEY. Невозможно вставить повторяющийся ключ в объект "dbo.TOVARY".
Выполнение данной инструкции было прервано.

Поскольку транзакции в данном случае устанавливаются на каждый оператор **INSERT**, то первые пять записей, которые не вызывают конфликтов, будут введены. Повторение ввода этих записей приведет к описанной выше конфликтной ситуации

Попробуем вставить очередное значение посредством оператора

```
INSERT INTO TOVARY  (KOD_TOVAR,TOVAR,ED_IZM,ZENA,COUNT_TOV)
 VALUES (6,'Масло подсолнечное','л',-10,1977);
```

Сообщение 547, уровень 16, состояние 0, строка 2
Конфликт инструкции INSERT с ограничением CHECK "CH_ZENA". Конфликт произошел в базе данных "MYBASE", таблица "dbo.TOVARY", column 'ZENA'.
Выполнение данной инструкции было прервано.

В данном случае срабатывает ограничение **CHECK** на поле **ZENA** (должно быть положительным). Исправим значение -10 на 10 и повторим ввод.

```
INSERT INTO TOVARY  (KOD_TOVAR,TOVAR,ED_IZM,ZENA,COUNT_TOV)
 VALUES (6,'Масло подсолнечное','л',10,1977);
SET IDENTITY_INSERT TOVARY OFF;
```

После завершения ввода можно посмотреть содержимое таблицы **TOVARY**, используя оператор

```
SELECT * FROM TOVARY;
```

KOD_TOVAR	TOVAR	ED_IZM	ZENA	COUNT_TOV
1	Сахар	кг	12	1442
2	Макароны	кг	5	86
3	Огурцы весовые	кг	6	250
4	Огурцы баночные	банки	10	475
5	Крупа манная	кг	8	1010
6	Масло подсолнечное	л	10	1977

(строк обработано: 6)

Заполним таблицу **RASXOD:**

```
SET IDENTITY_INSERT RASXOD ON;
INSERT INTO RASXOD
(KOD_RASH,DATA_RASH,KOLVO,STOIM,KOD_TOVAR,KOD_POKUP)
 VALUES (2,'12.04.98',10,120,1,4),
        (5,'12.07.00',12,120,4,4),
        (6,'13.09.00',5,25,2,2),
        (11,'12.04.03',10,50,2,3),
        (7,'15.09.00',1,10,6,4),
        (8,'16.07.00',7,70,6,1);
```

Отметим, что при вводе информации в таблицу RASXOD отрабатывают триггеры (в частности триггер AFTER INSERT, поэтому сообщение о числе обработанных строк включает значительно больше строк.

```
INSERT INTO RASXOD
(KOD_RASH,DATA_RASH,KOLVO,STOIM,KOD_TOVAR,KOD_POKUP)
 VALUES (9,'23.05.00',11,110,400,1);
```

Сообщение 547, уровень 16, состояние 0, строка 14
Конфликт инструкции INSERT с ограничением FOREIGN KEY "TOV_RASH". Конфликт произошел в базе данных "myBase", таблица "dbo.TOVARY", column 'KOD_TOVAR'.

```
INSERT INTO RASXOD
(KOD_RASH,DATA_RASH,KOLVO,STOIM,KOD_TOVAR,KOD_POKUP)
 VALUES (9,'23.05.00',11,110,4,100);
```

Сообщение 547, уровень 16, состояние 0, строка 15
Конфликт инструкции INSERT с ограничением FOREIGN KEY "POK_RASH". Конфликт произошел в базе данных "MYBASE", таблица "dbo.POKUPATELI", column 'KOD_POKUP'.
Выполнение данной инструкции было прервано.

```
INSERT INTO RASXOD
(KOD_RASH,DATA_RASH,KOLVO,STOIM,KOD_TOVAR,KOD_POKUP)
 VALUES (9,'23.05.00',11,110,4,1);
SET IDENTITY_INSERT RASXOD OFF;
```

```
SELECT * FROM RASXOD;
```

KOD_RASH	DATA_RASH	KOLVO	STOIM	KOD_TOVAR	KOD_POKUP
2	1998-04-12 00:00:00.000	10	120	1	4
5	2000-07-12 00:00:00.000	12	120	4	4
6	2000-09-13 00:00:00.000	5	25	2	2
7	2000-09-15 00:00:00.000	1	10	6	4
8	2000-07-16 00:00:00.000	7	70	6	1
9	2000-05-23 00:00:00.000	11	110	4	1
11	2003-04-12 00:00:00.000	10	50	2	3

(строк обработано: 7)

```
SELECT * FROM TOVARY;
```

KOD_TOVAR	TOVAR	ED_IZM	ZENA	COUNT_TOV
1	Сахар	кг	12	1432
2	Макароны	кг	5	71

```
3        Огурцы весовые       кг      6       250
4        Огурцы баночные      банки   10      452
5        Крупа манная         кг      8       1010
6        Масло подсолнечное   л       10      1969
```

Отметим, что в процессе работы этой программы SQL-сервер проверял описанные ограничения целостности БД и прерывал выполнение операторов, которые нарушали эти ограничения:

- Первое сообщение об ошибке ***(Сообщение 2627, уровень 14...)*** связано с попыткой вставить в таблицу **TOVARY** строку с ключом **(KOD_TOVAR=2)**, который уже был использован во второй строке.
- Второе сообщение связано с попыткой вставить в таблицу TOVARY строку со значением поля **ZENA=-10**, для которого объявлен предикат **CHECK(ZENA>=0)**.
- Следующие два сообщения связаны с попытками нарушить требование межтабличной целостности – вставить в таблицу **RASXOD** строки с кодом товара **(KOD_TOVAR=400)** и кодом покупателя **(KOD_POKUP=100)**, которые не встречаются в таблицах **TOVARY** и **POKUPATELI**, соответственно.

Отметим, что после загрузки данных в таблицу **RASXOD** содержимое таблицы **TOVARY** тоже изменилось (значение поля **COUNT_TOV** в некоторых записях) – это отрабатывал триггер **AI_RASXOD.**

Остальные триггеры не отрабатывали, поскольку не происходили события их запускающие.

Не выполнялись также и хранимые процедуры. В отличие от триггеров, хранимые процедуры вызываются явно оператором процедуры, а таковых в нашей программе не было.

5.5 Создание представлений.

Теперь определим представление **View** для пользователей, которых интересуют покупки только не Казанских покупателей. Для этого опять подготовим SQL-программу и выполним ее. Протокол работы этой программы:

```
USE MYBASE;
DROP VIEW RasxodDoc;
DROP VIEW RasxodNoKz;

CREATE VIEW RasxodDoc(KOD_RASH,DATA_RASH,KOLVO,STOIM,
        KOLVO_ZENA, POKUP,GOROD,ADRES,TEL,
     TOVAR,ED_IZM,ZENA,COUNT_TOV)
 AS
SELECT   RASXOD.KOD_RASH, RASXOD.DATA_RASH, RASXOD.KOLVO,
RASXOD.STOIM, RASXOD.KOLVO*TOVARY.ZENA,       POKUPATELI.POKUP,
POKUPATELI.GOROD, POKUPATELI.ADRES, POKUPATELI.TEL,
TOVARY.TOVAR, TOVARY.ED_IZM, TOVARY.ZENA, TOVARY.COUNT_TOV
      FROM RASXOD, POKUPATELI, TOVARY
      WHERE ((RASXOD.KOD_POKUP=POKUPATELI.KOD_POKUP) AND
             (RASXOD.KOD_TOVAR=TOVARY.KOD_TOVAR) AND
             (POKUPATELI.GOROD<>'Казань'));


SELECT * FROM RasxodDoc ORDER BY KOD_RASH;
INSERT INTO RasxodDoc (KOD_RASH,DATA_RASH,KOLVO,STOIM)
 VALUES (20,'23.05.02',110,1100);
```

Сообщение 544, уровень 16, состояние 1, строка 1
Невозможно вставить явное значение для столбца идентификаторов в таблице "RASXOD", когда параметр IDENTITY_INSERT имеет значение OFF.

Исправим ошибку, установив флаг **IDENTITY_INSERT** таблицы **RASXOD** в положение **ON.**

```
SET IDENTITY_INSERT RASXOD ON;
INSERT INTO Rasxod
(KOD_RASH,DATA_RASH,KOLVO,STOIM,KOD_TOVAR,KOD_POKUP)
 VALUES (20,'23.05.02',110,1100,4,5);
SELECT * FROM RasxodDoc ORDER BY KOD_RASH;
SET IDENTITY_INSERT RASXOD OFF;
```

Сообщение 515, уровень 16, состояние 2, строка 2 Не удалось вставить значение NULL в столбец "KOD_POKUP", таблицы "MYBASE.dbo.RASXOD"; в столбце запрещены значения NULL. Ошибка в INSERT.
Выполнение данной инструкции было прервано.

*SELECT * FROM Rasxod ORDER BY KOD_RASH;*

KOD_RASH	DATA_RASH	KOLVO	STOIM	KOD_TOVAR	KOD_POKUP
2	1998-04-12 00:00:00.000	10	120	1	4
5	2000-07-12 00:00:00.000	12	120	4	4
6	2000-09-13 00:00:00.000	5	25	2	2
7	2000-09-15 00:00:00.000	1	10	6	4
8	2000-07-16 00:00:00.000	7	70	6	1
9	2000-05-23 00:00:00.000	11	110	4	1
11	2003-04-12 00:00:00.000	10	50	2	3

(строк обработано: 7)

Создадим представление на базе одной таблицы

```
CREATE VIEW RasxodNoKz
 AS SELECT * FROM RASXOD WHERE
     ('Казань'<>(SELECT GOROD FROM POKUPATELI
                    WHERE
               (RASXOD.KOD_POKUP=POKUPATELI.KOD_POKUP)));
```

*SELECT * FROM RasxodNoKz ORDER BY KOD_RASH;*

KOD_RASH	DATA_RASH	KOLVO	STOIM	KOD_TOVAR	KOD_POKUP
2	1998-04-12 00:00:00.000	10	120	1	4
5	2000-07-12 00:00:00.000	12	120	4	4
6	2000-09-13 00:00:00.000	5	25	2	2
7	2000-09-15 00:00:00.000	1	10	6	4
11	2003-04-12 00:00:00.000	10	50	2	3

(строк обработано: 5)

Вставим новую запись в представление

```
SET IDENTITY_INSERT RASXOD ON;
INSERT INTO RasxodNoKz
(KOD_RASH,DATA_RASH,KOLVO,STOIM,KOD_TOVAR,KOD_POKUP)
 VALUES (21,'23.05.02',120,1200,4,2);
```

```
SET IDENTITY_INSERT RASXOD OFF;
```

*SELECT * FROM RasxodNoKz ORDER BY KOD_RASH;*

```
KOD_RASH    DATA_RASH               KOLVO       STOIM       KOD_TOVAR   KOD_POKUP
----------- ----------------------- ----------- ----------- ----------- -----------
2           1998-04-12 00:00:00.000 10          120         1           4
5           2000-07-12 00:00:00.000 12          120         4           4
6           2000-09-13 00:00:00.000 5           25          2           2
7           2000-09-15 00:00:00.000 1           10          6           4
11          2003-04-12 00:00:00.000 10          50          2           3
21          2002-05-23 00:00:00.000 120         1200        4           2

(строк обработано: 6)
```

*SELECT * FROM Rasxod ORDER BY KOD_RASH;*

```
KOD_RASH    DATA_RASH               KOLVO       STOIM       KOD_TOVAR   KOD_POKUP
----------- ----------------------- ----------- ----------- ----------- -----------
2           1998-04-12 00:00:00.000 10          120         1           4
5           2000-07-12 00:00:00.000 12          120         4           4
6           2000-09-13 00:00:00.000 5           25          2           2
7           2000-09-15 00:00:00.000 1           10          6           4
8           2000-07-16 00:00:00.000 7           70          6           1
9           2000-05-23 00:00:00.000 11          110         4           1
11          2003-04-12 00:00:00.000 10          50          2           3
21          2002-05-23 00:00:00.000 120         1200        4           2

(строк обработано: 8)
```

- Представление **RasxodDoc** для каждого документа о покупке содержит полную пользовательскую информацию – включает реквизиты покупателя и товара вместо их кодов. Отметим, что реквизит «стоимость покупки» вычислим и определен в этом представлении **KOLVO_ZENA.** Использование хранимого значения этого реквизита в поле **STOIM** таблицы **RASXOD** требует обоснования...

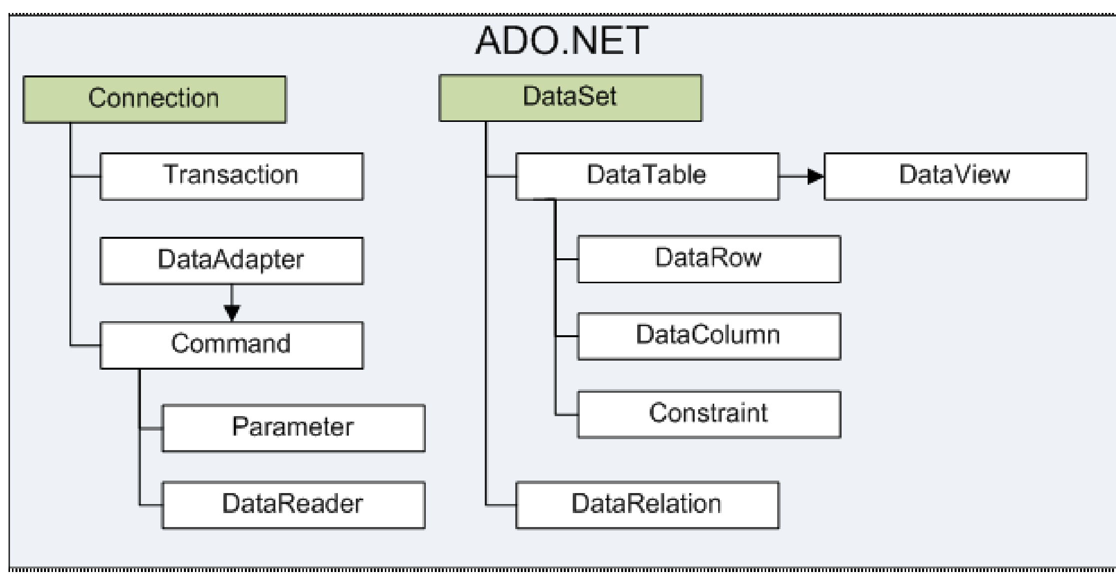
  К сожалению, таким образом определенное представление не является модифицируемым, сообщение об ошибке к первому **INSERT** это иллюстрирует. Второй оператор **INSERT** иллюстрирует специфику представлений – изменение внесено в таблицу **Rasxod,** но оно отобразилось и в **RasxodDoc.**
- Представление **RasxodNoKz** имеет такие же поля, как и таблица **Rasxod,** но оно включает строки только о покупках не Казанских

покупателей. **RasxodNoKz** – модифицируемое представление, это иллюстрирует третий **INSERT,** который добавляет строку в **RasxodNoKz.** Эта строка появляется и в таблице **Rasxod,** что опять же иллюстрирует специфику представлений.

6. Принципы работы с источниками данных в среде MS Visual Studio.

ADO.NET — это набор библиотек, поставляемый с Microsoft .NET Framework и предназначенный для взаимодействия с различными хранилищами данных из клиентских приложений. Библиотеки ADO.NET включают классы для подсоединения к источнику данных, выполнения запросов и обработки их результатов. Кроме того, ADO.NET можно использовать в качестве надежного, иерархически организованного, отсоединенного кэша данных для автономной работы с данными. Главный отсоединенный объект, DataSet, позволяет сортировать, осуществлять поиск, фильтровать, сохранять отложенные изменения и перемещаться по иерархичным данным. Кроме того, объект DataSet включает ряд функций, сокращающих разрыв между традиционным доступом к данным и программированием с использованием XML. Теперь разработчики получили возможность работать с XML-данными через обычные интерфейсы доступа к данным и наоборот. Если вкратце, при создании приложений для работы с данными нужно использовать ADO.NET

В Microsoft Visual Studio .NET есть ряд функций, предназначенных для создания приложений для доступа к данным. Одни из них экономят время на разработку, генерируя большие объемы скучного кода. Другие повышают производительность ваших приложений, добавляя метаданные и логику обновления в код вместо того, чтобы выбирать эту информацию в период выполнения. Большинство функций доступа к данным, предоставляемых Visual Studio .NET, выполняют сразу обе этих задачи.

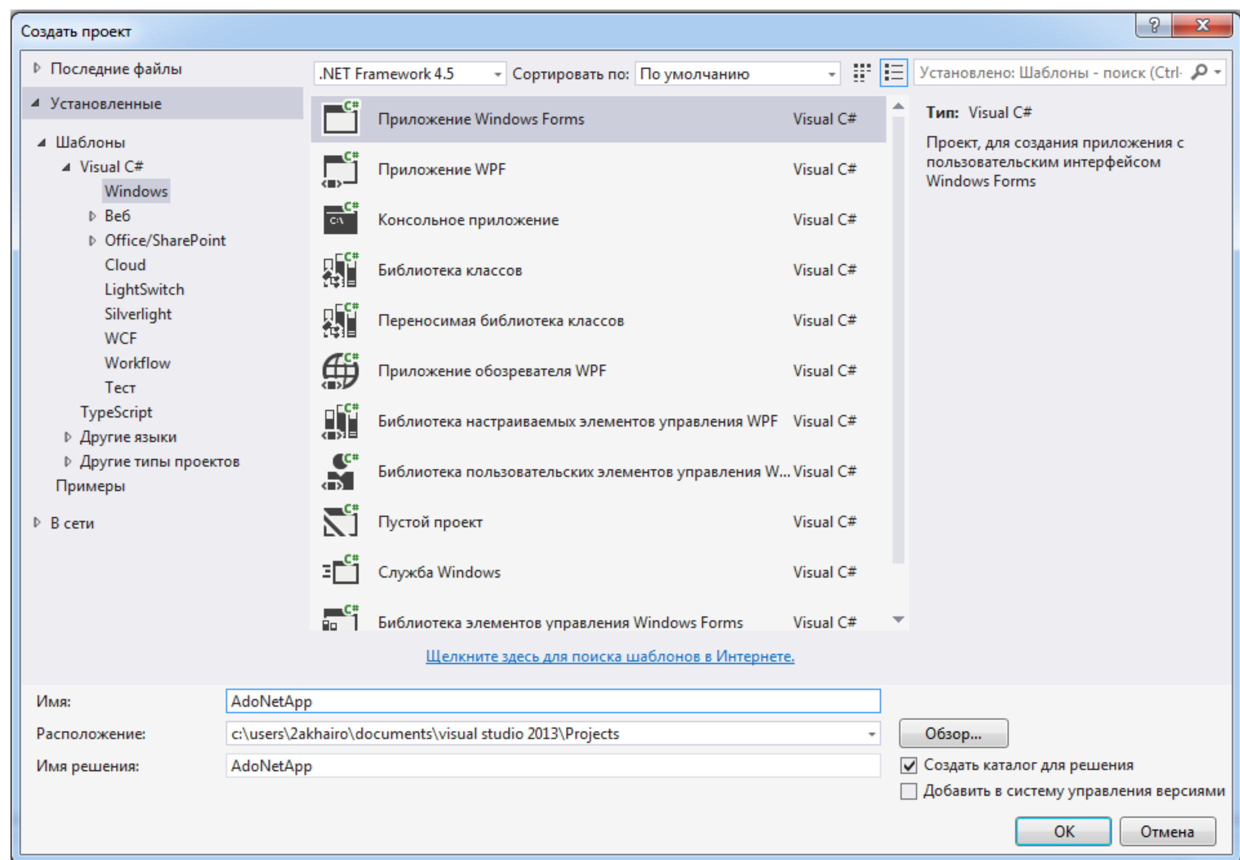


На рисунке показаны классы, составляющие объектную модель ADO.NET. Объекты в левой части называются подсоединенными (connected). Для управления соединением, транзакциями, выборки данных и передачи изменений они взаимодействуют непосредственно с БД. Объекты в правой части называются отсоединенными (disconnected), они позволяют работать с данными автономно.

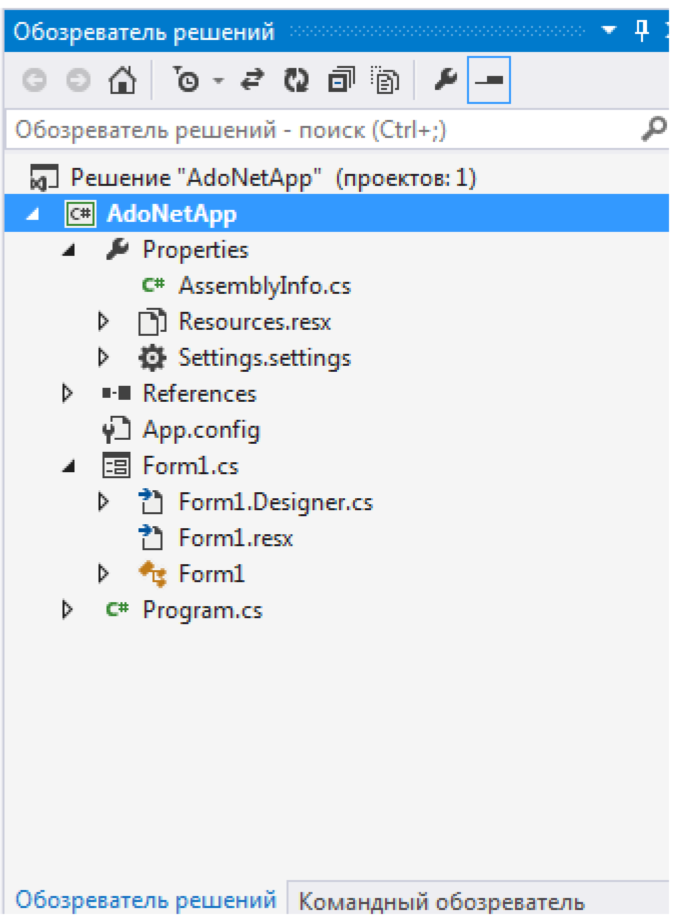
7. Разработка клиентского места

В данной методической разработке будет рассмотрена реализация клиентского приложения на C# в среде разработки Microsoft Visual Studio 2013. Для разработки также можно использовать язык программирования C++ или Visual Basic. Отличия в коде связаны только с особенностью используемых языков программирования, а объектная библиотека останется одной и той же. То есть наименования пространств имен, классов и методов будут совпадать на обоих языках. При выполнении курсового проекта студент может выбрать любую реализацию.

Начинаем разработку приложения с создания проекта в Visual Studio, выбираем шаблон проекта «Приложение Windows Forms Visual C#», задаём имя и путь расположения проекта.



Будет создан проект, содержащий одну пустую форму, которая и будет открыта на экране. По умолчанию форме присвоено имя **Form1**, кроме этого в проект содержит вспомогательные файлы. Во-первых, проект помещён в решение (Solution) с тем же именем. В дальнейшем вы можете в это решение добавлять другие проекты необходимые для решения поставленных задач. Причем никаких ограничений на типы проектов нет, одно решение может содержать проекты на различных языках, созданных из различных шаблонов. В разделе свойств проекта ***Properties*** хранятся различные настройки проекта – информация о сборке, авторах, уникальный идентификатор GUID, дополнительные ресурсы, используемые в проекте – иконки, картинки и т.д. **References** – ссылки на пространства имён (namespace) включаемые в проект.



Файл **Program.cs** содержит следующий код

```
static class Program
{
    [STAThread]
```

```
        static void Main()
        {
          Application.EnableVisualStyles();
          Application.SetCompatibleTextRenderingDefault(false);
          Application.Run(new Form1());
        }
    }
```

Метод *static void Main()* является точкой входа в приложение, и работа приложения начинается с создания формы *new Form1().* Здесь можно вызвать создание любой формы из вашего проекта, то есть регулировать какая форма будет основной для приложения.

7.1 Основная экранная форма

Имеются несколько основных компонент (классов), которые используются для доступа к БД. Эти классы делятся на две группы:

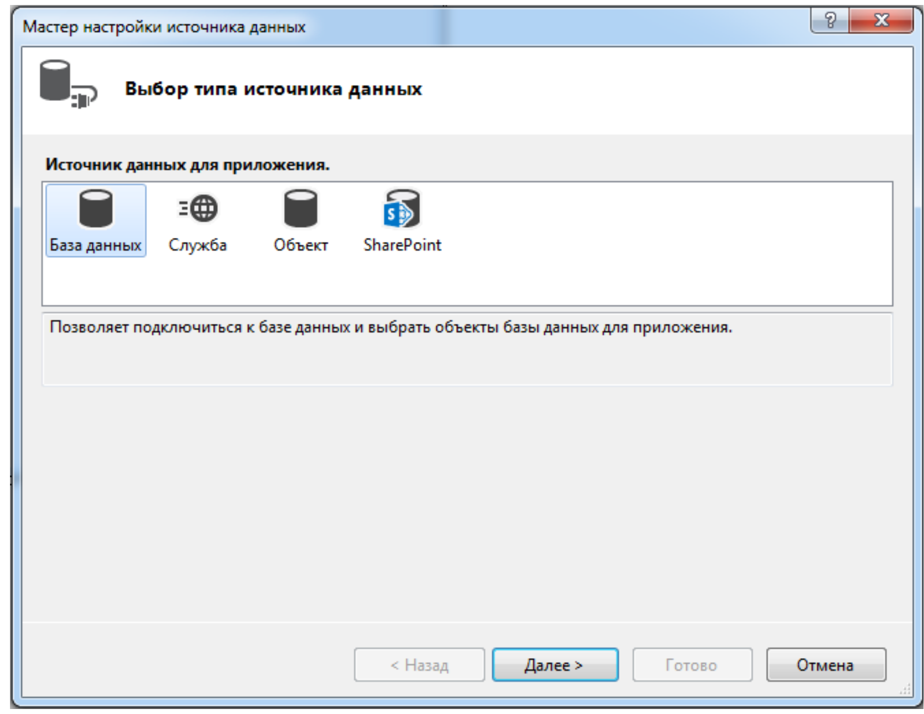
- невизуальные: DataSet, DataTable, DataView, DataColumn, DataRow, TableAdapter, BindingSource
- визуальные: DataGridView, TextBox, ComboBox, Label

Первая группа включает невизуальные классы, которые используются для управления базами данных, таблицами и связей между базой данных и формой. Эта группа сосредотачивается вокруг компонент типа **DataSet**, **TableAdapter** и **BindingSource**. В Панели Элементов они расположены на странице «Данные».

Вторая важная группа классов - визуальные, которые показывают данные пользователю, и позволяют ему просматривать и модифицировать их. Эта группа классов включает компоненты типа **DataGridView**, **ComboBox**, **TextBox**, **Label**. В Панели Элементов они расположены на странице «Стандартные Элементы Управления».

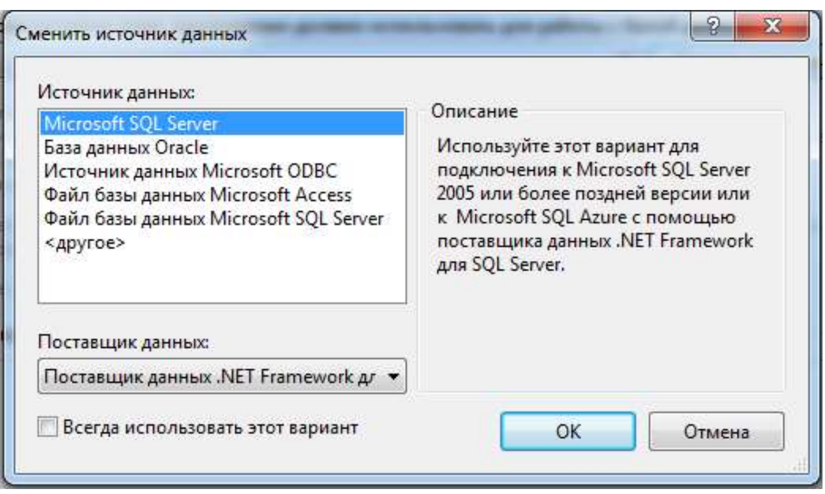
Рассмотрим последовательность действий построения клиентской части программы.

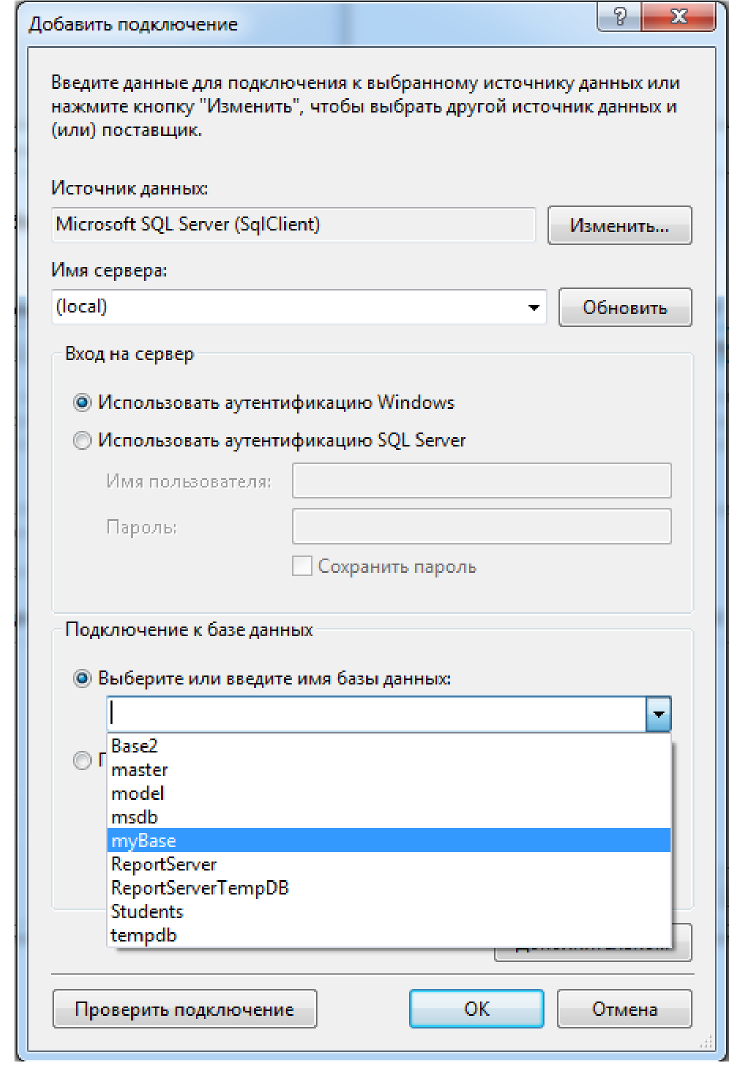
Первым делом нам необходимо подключиться к базе данных. Для этого открываем панель «Источники данных» проекта, если её нет среди видимых панелей, то выбираем пункт меню «Вид»-«Другие окна»-«Источники данных». У нашего проекта пока нет источников данных, поэтому панель пуста и Visual Studio предлагает добавить новый источник данных.



Запускается «Мастер настройки источника данных» - определяем, что источником данных будет база данных. Моделью данных будет «Набор данных» – **DataSet** – снимок части нашей базы данных, необходимой в клиентском приложении.

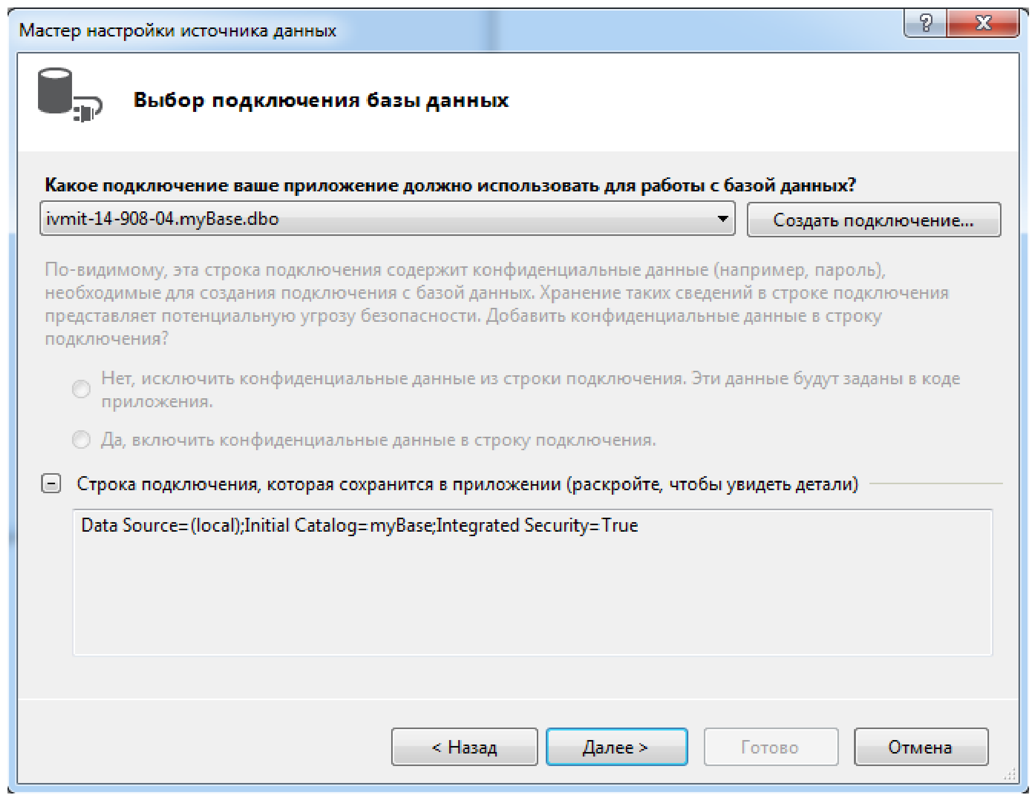
Теперь надо создать строку подключения – «Создать подключение». Определяем источник данных, в нашем случае это Microsoft SQL Server при помощи поставщика данных .Net Framework. Если у вас база данных хранится в другом источнике, то выбираем соответствующий.





Далее необходимо задать имя сервера и выбрать имя вашей базы данных. Если вы подключаетесь к полной версии SQL Server, то, по умолчанию, он именуется по имени компьютера. Также можно использовать идентификатор **(local)**. Если у вас экспресс-версия сервера, тогда **(local)\sqlexpress**. Для LocalDB-версий сервера

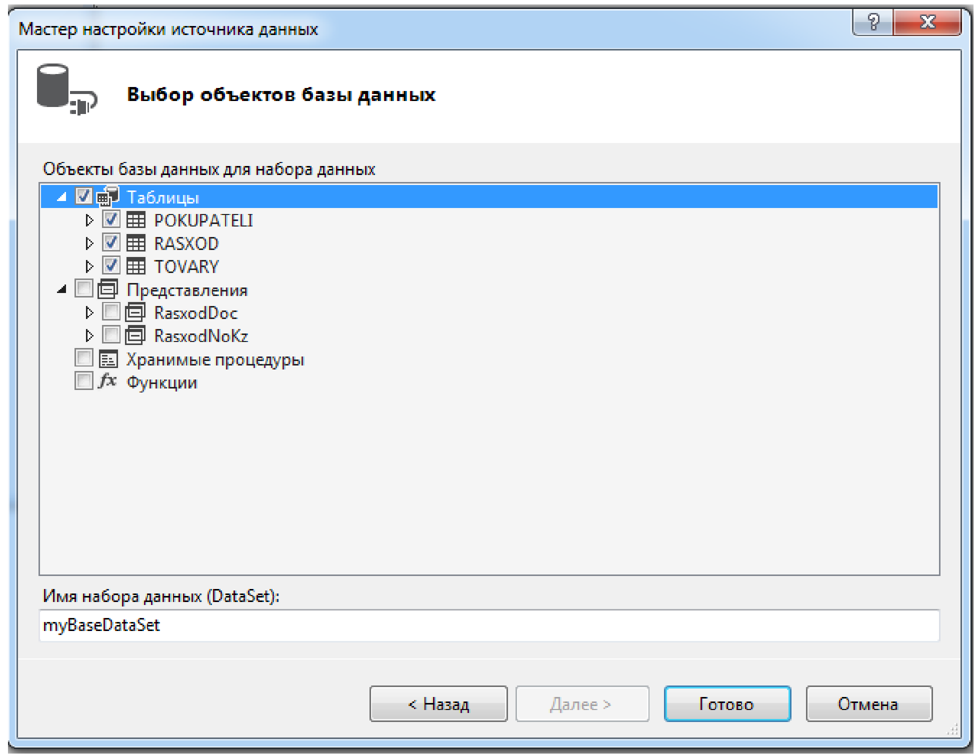
строка подключений либо **(localdb)\v11.0**, либо (**localdb)\mssqllocaldb.** Имена серверов можно посмотреть в настройках SQL Server и в службах Windows. Если вы выбрали правильный и запущенный сервер баз данных, то в списке появятся все базы, хранящиеся на этом сервер, к которым у вас есть доступ, включая системные. Выбираем нужную базу данных и получаем строку подключения к базе.



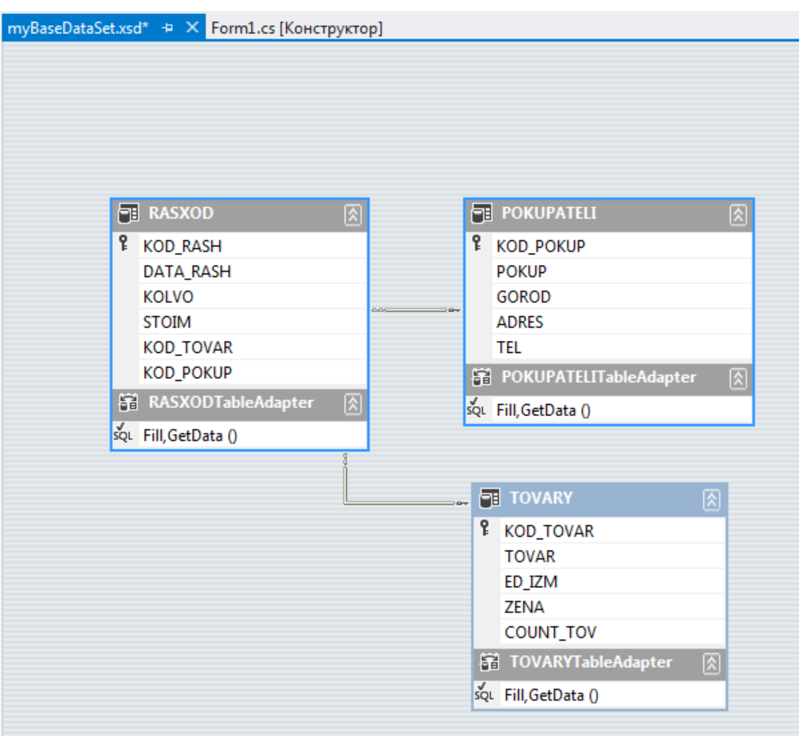
После задания имени, строка подключения сохранится в настройках проекта и в файле ***App.congif.***

```
<connectionStrings>
    <add
name= "AdoNetApp.Properties.Settings.myBaseConnectionString"
connectionString="Data Source=(local);Initial Catalog=myBase;
Integrated Security=True"
     providerName="System.Data.SqlClient" />
</connectionStrings>
```

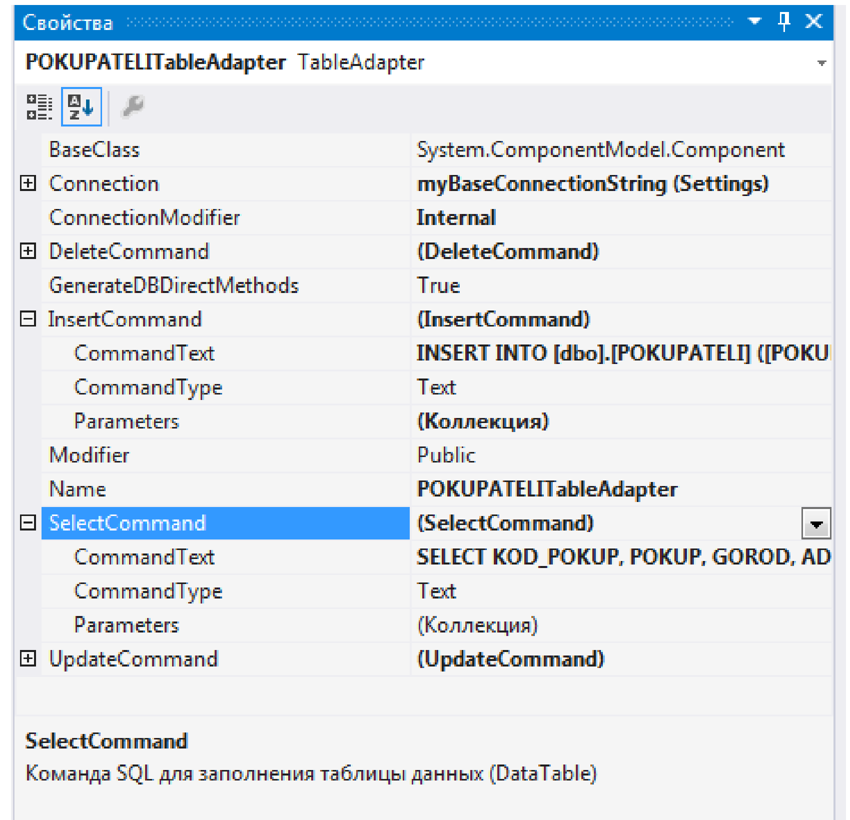
Теперь надо выбрать объекты базы данных, которые нам понадобятся в проекте. На форме отображаются все таблицы, представления, хранимые процедуры и функции базы. Отмечаем нужные объекты и задаём имя набора данных проекта.



На этом работа мастера настройки заканчивается, в панели «Источники данных» появится новый **DataSet** с выбранными объектами базы данных.



В проекте создается несколько файлов с именем набора данных и различными расширениями. Они хранят информацию о новом **DataSet**, создаётся класс **myBaseDataSet** расширение класса **DataSet** из пространства **System.Data.** Он хранит коллекцию таблиц, каждая из них является объектом расширения класса **DataTable.** Для каждой таблицы кроме этого создается соответствующий класс **TableAdapter.** Адаптер таблицы подключается к базе данных, выполняет запросы или хранимые процедуры и либо возвращает новую заполненную таблицу данных, либо заполняет существующую **DataTable** возвращаемыми данными. Адаптеры таблиц также используются для отправки обновленных данных из приложения обратно в базу данных.

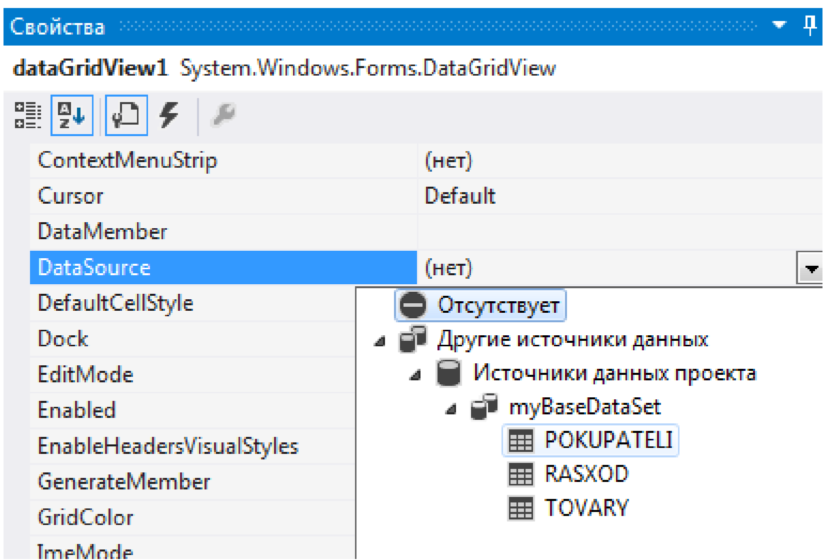


Visual Studio сама генерирует для адаптера таблицы все необходимые SQL-команды – Select, Insert, Update и Delete. Вы можете отредактировать любые из них или заменить их на вызов хранимых процедур. Кроме этого определяются методы: **Fill()**, заполнение **DataTable** из базы данных на SQL Server; **GetData()** – возвращает заполненную таблицу; **Update()** – отправляет изменения

произошедшие с таблицей **DataTable** в базу данных на сервере. Вы можете добавить свои методы с дополнительными параметрами.

Также создаётся класс **DataAdapterManager** – он содержит ссылки на адаптеры всех таблиц набора данных и имеет метод **UpdateAll** – отправка всей измененной информации из **DataSet** в базу данных на SQL Server. Кроме таблиц в **DataSet** хранятся связи между ними **DataRelation,** загруженные из базы данных или добавленные непосредственно в конструкторе Visual Studio.

Теперь у нас в проекте появился набор данных и можно отобразить его объекты на форме. Можно напрямую из панели «Источники данных» перетянуть любую таблицу на форму drag-n-drop – Visual Studio автоматически добавит все необходимые компоненты на форму и сгенерирует нужный код. Если мы хотим сами контролировать, откуда что появилось, добавим на форму элемент из «Панели элементов» **DataGridView.** Изменим свойство ***name*** объекта на **pOKUPATELIdataGridView,** по умолчанию оно **dataGridView1.** Выберем источник данных для таблицы, он задаётся свойством **DataSource.** Выбираем таблицу **POKUPATELI** из **DataSet** проекта.

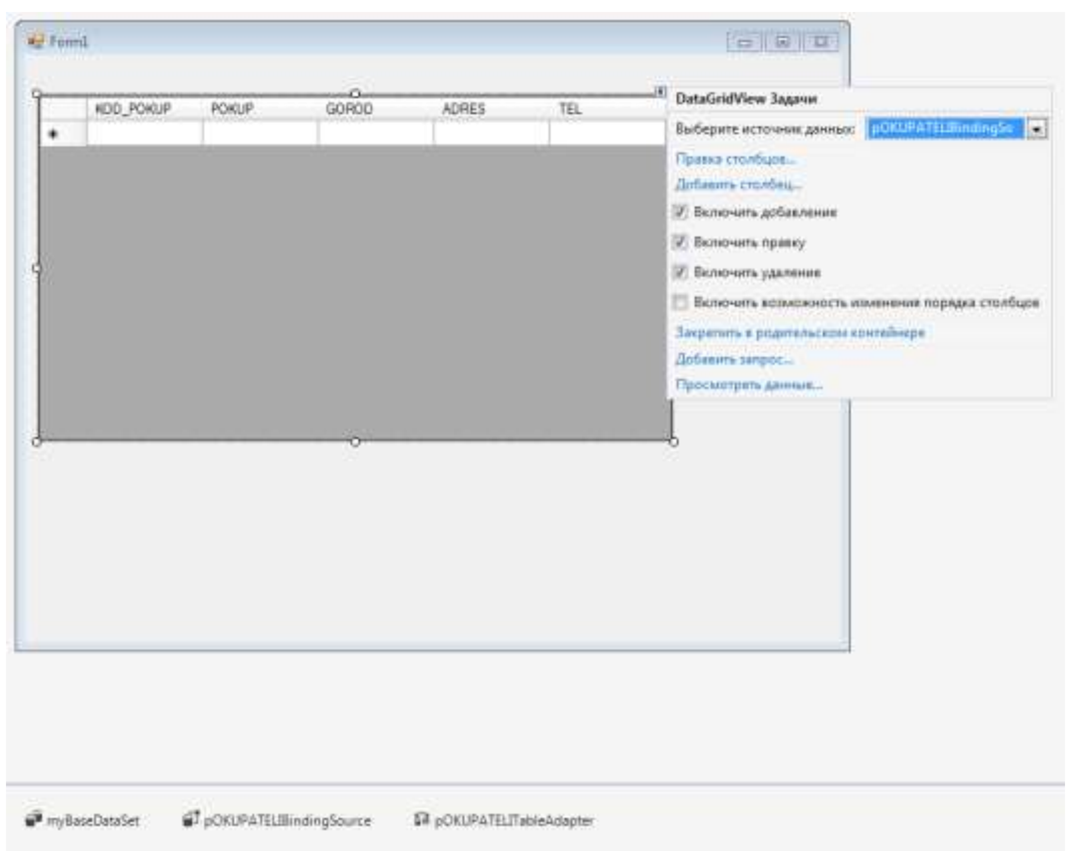


На форму добавляется ссылка на набор данных проекта **myBaseDataSet** и адаптер соответствующей таблицы **pOKUPATELITableAdapter.** Кроме этого создался объект **pOKUPATELIBindingSource** класса **BindingSource** из **System.Windows.Forms,** которые связывает элементы на форме с

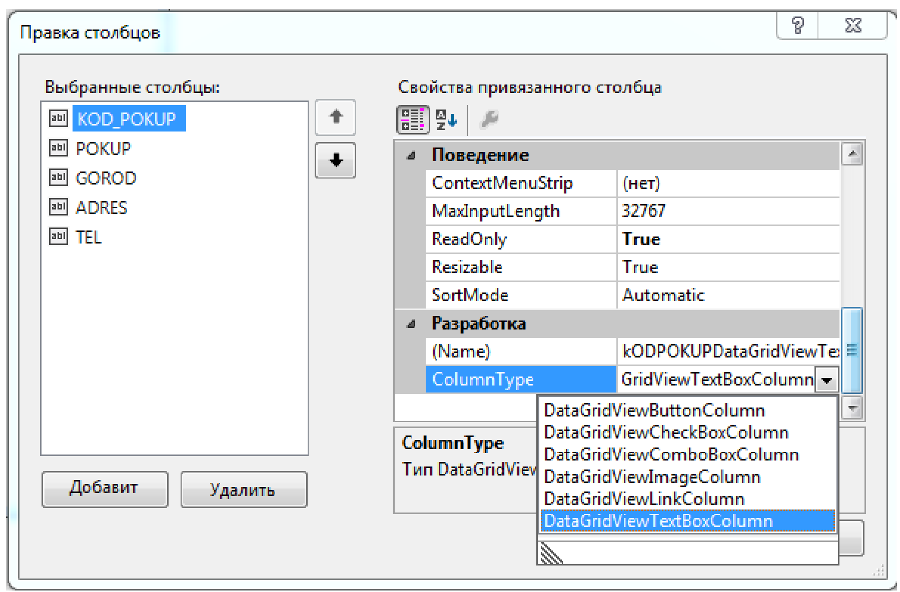
источниками данных. В нашем случае связь определяется свойствами данного объекта **DataSource** – **myBaseDataSet** и **DataMember** – ***POKUPATELI***. Для фильтрации и сортировки строк таблицы необходимо соответствующим образом заполнить свойства **Filter** *и* **Sort**. Далее объект **pOKUPATELIDataGridView** можно настраивать под нужды пользователя – настраивать список столбцов, регулировать возможности изменять данные в таблице.

В классе формы Visual Studio создаёт обработчик события загрузки формы, в ней таблица **DataSet** заполняется при помощи адаптера таблицы из базы данных.

```
private void Form1_Load(object sender, EventArgs e)
{
// TODO: данная строка кода позволяет загрузить данные в таблицу
// "myBaseDataSet.POKUPATELI". При необходимости она может быть
// перемещена или удалена.
this.pOKUPATELITableAdapter.Fill(this.myBaseDataSet.POKUPATELI);
}
```

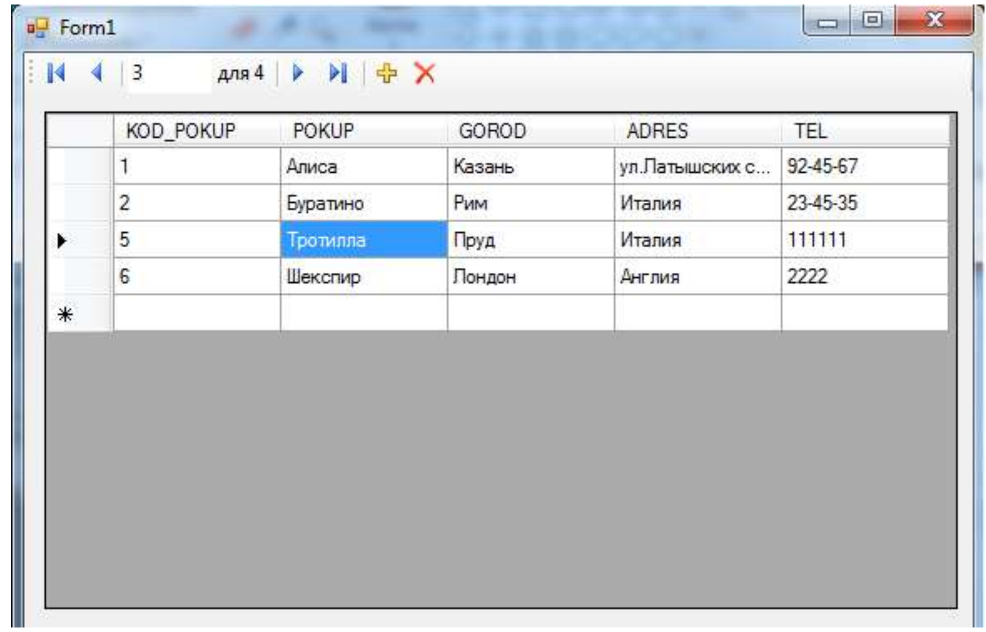


Свойство **Columns** объекта **pOKUPATELIdataGridView** содержит коллекцию столбцов. Каждый из столбцов может быть объектом одного из специализированного типа: **DataGridViewTextBoxColumn** – текстовое поле, **DataGridViewButtonColumn** – поле кнопка, **DataGridViewCheckBoxColumn** – поле переключатель и т.д. В свойствах колонок можно настроить форматы отображения информации заголовки столбцов. В свойстве **Name** в разделе «Разработка» задается имя поля, которое будет в дальнейшем использовано при программировании различных действий с соответствующим столбцом.



Теперь можно запустить приложение и посмотреть, как будет выглядеть форма в режиме выполнения. Таблица заполнится данными из базы данных, в зависимости от выбранных настроек их можно или нельзя редактировать, добавлять или удалять строки. Но все эти изменения остаются только на форме и не сохраняются на сервере. Для их сохранения необходимо вызвать метод Update адаптера таблицы.

Для упрощения навигации по таблице добавим компонент **BindingNavigator** на форму приложения. Оставим имя по умолчанию **BindingNavigator1**. Связь навигатора и таблицы производится по свойству **BindingSource**, выбираем **pOKUPATELIBindingSource**, устанавливается связь с таблицей **POKUPATELI**, а значит и с **pOKUPATELIdataGridView**.



Кнопки объекта **bindingNavigator1** имеют слева направо следующие значения:

Первая запись

Предыдущая

Номер текущей записи

Всего записей

Следующая

Последняя запись

Вставка записи

Удаление записи

Обновление данных (Кнопка появляется при перетаскивании таблицы из панели источников данных)

Аналогичным образом можно получить доступ к записям таблиц «Товары» и «Расход товара». Добавляем два новых **DataGridView** на форму (соответственно под именами **tOVARYDataGridView** ***и*** **rASXODDataGridView**) задаём для них свойство **DataSource.** Создадутся соответствующие невизуальные компоненты **BindingSource** и **TableAdapter.** Навигаторы для новых таблиц пока не добавляем. В обработчике события загрузки формы добавятся вызовы методов заполнения этих таблиц набора данных проекта.

```
this.tOVARYTableAdapter.Fill(this.myBaseDataSet.TOVARY);
this.pOKUPATELITableAdapter.Fill(this.myBaseDataSet.POKUPATELI);
```

7.1.1 Динамическое изменение свойств элементов управления.

В качестве примера динамического перепрограммирования свойств объектов приведем пример обслуживания одним навигатором двух таблиц «Товары» и «Покупатели». Как уже упоминалось выше, в навигаторе свойство **BindingSource** необходимо настроить на имя компонента **BindingSource**, связывающий навигатор с определенной таблицей. Изменяя динамически эту связь можно одним навигатором обслуживать различные таблицы.

Определим обработчики для события **Click** таблиц **tOVARYdataGridView** и **rASXODdataGridView.** Добавим на форму элемент **Label**, он находится в разделе «Все формы Windows Forms» в панели элементов. Зададим новое имя **lblTable**, изменим шрифт и свойство **Text** - «Покупатели».

```
private void pOKUPATELIdataGridView_Click(object sender, EventArgs e)
{
    bindingNavigator1.BindingSource = pOKUPATELIBindingSource;
    lblTable.Text = "Покупатели";
}

private void tOVARYdataGridView_Click(object sender, EventArgs e)
{
    bindingNavigator1.BindingSource = tOVARYBindingSource;
    lblTable.Text = "Товары";
}

private void rASXODdataGridView_Click(object sender, EventArgs e)
    {
       rASXODdataGridView.DataSource = rASXODBindingSource;
       lblTable.Text = "Покупки";
    }
```

Запустив программу можно убедиться, что любое переключение между объектами **DataGridView** для таблиц «Товары» и «Покупатели» изменяет состояние метки **lblTable**, которое показывает, какой таблицей в настоящий момент управляет навигатор. Для таблицы «Расход товара» построим собственный навигатор. В частности, если на форме имеется объект **rASXODdataGridView**, то изменяя свойство **CurrentCell** можно перейти к любой ячейке таблицы. Переход осуществляется заданием новых индексов строки и столбца **rASXODdataGridView[nCol, nRow]**. Добавим на форму четыре кнопки: назовём их **btnPrev**, **btnNext**, **btnFirst**, **btnLast**. Заполним поле text для этих кнопок соответственно значениями «предыдущая», «следующая», «первая» и «последняя».

Определим обработчики нажатия на кнопку – **btnNext_Click**.

```
private void btn_Click(object sender, EventArgs e)
{
// Считываем номер строки и столбца активной ячейки.
int nRow = rASXODdataGridView.CurrentCell.RowIndex;
int nCol = rASXODdataGridView.CurrentCell.ColumnIndex;
// Если строка – не последняя, увеличиваем номер строки на 1,
// в противном случае соответству
if (nRow < rASXODdataGridView.RowCount - 1)
 rASXODdataGridView.CurrentCell = rASXODdataGridView[nCol,
++nRow];
}
```

Для ограничения движения указателя записи размерами таблицы напишем обработчик события **CurrentCellChanged**[9].

```
 private void rASXODdataGridView_CurrentCellChanged(object
sender, EventArgs e)
        {
            if (rASXODdataGridView.CurrentCell != null)
            { //если существует выбранная ячейка
```

[9] иначе будем получать исключения при попытке пользователя выйти за пределы таблицы.

```
                int nRow =
rASXODdataGridView.CurrentCell.RowIndex;
                //первая строка
                if (nRow == 0)
                {
                    btnPrev.Enabled = false;
                    btnFirst.Enabled = false;
                }
                else
                {
                    btnPrev.Enabled = true;
                    btnFirst.Enabled = true;
                }
                //последняя строка
                if (nRow == rASXODdataGridView.RowCount - 1)
                {
                    btnNext.Enabled = false;
                    btnLast.Enabled = false;
                }
                else
                {
                    btnNext.Enabled = true;
                    btnLast.Enabled = true;
                }
            }
        }
```

Заметим, что этот метод отрабатывает при изменении адреса активной ячейки. Но он отрабатывает и при управлении таблицей через навигатора: нажатие любой из клавиш навигации изменяет положение текущей ячейки с вызовом метода **rASXODdataGridView_CurrentCellChanged**.

Добавление новой и удаление текущей строки таблицы осуществляется стандартными средствами Visual Studio. После внесения изменений в таблицу на форме надо сохранить их в базе данных на сервере. Для этого воспользуемся методом **Update**

адаптера таблицы, вызовем его при нажатии на кнопку формы **btnSave**:

```
private void btnSave_Click(object sender, EventArgs e)
        {
            //сохранение данных
this.rASXODTableAdapter.Update(this.myBaseDataSet.RASXOD);
            //обновление данных из источника
this.rASXODTableAdapter.Fill(this.myBaseDataSet.RASXOD);
            //обновление состояния навигатора
this.rASXODdataGridView_CurrentCellChanged(rASXODdataGridView,
e);
    }
```

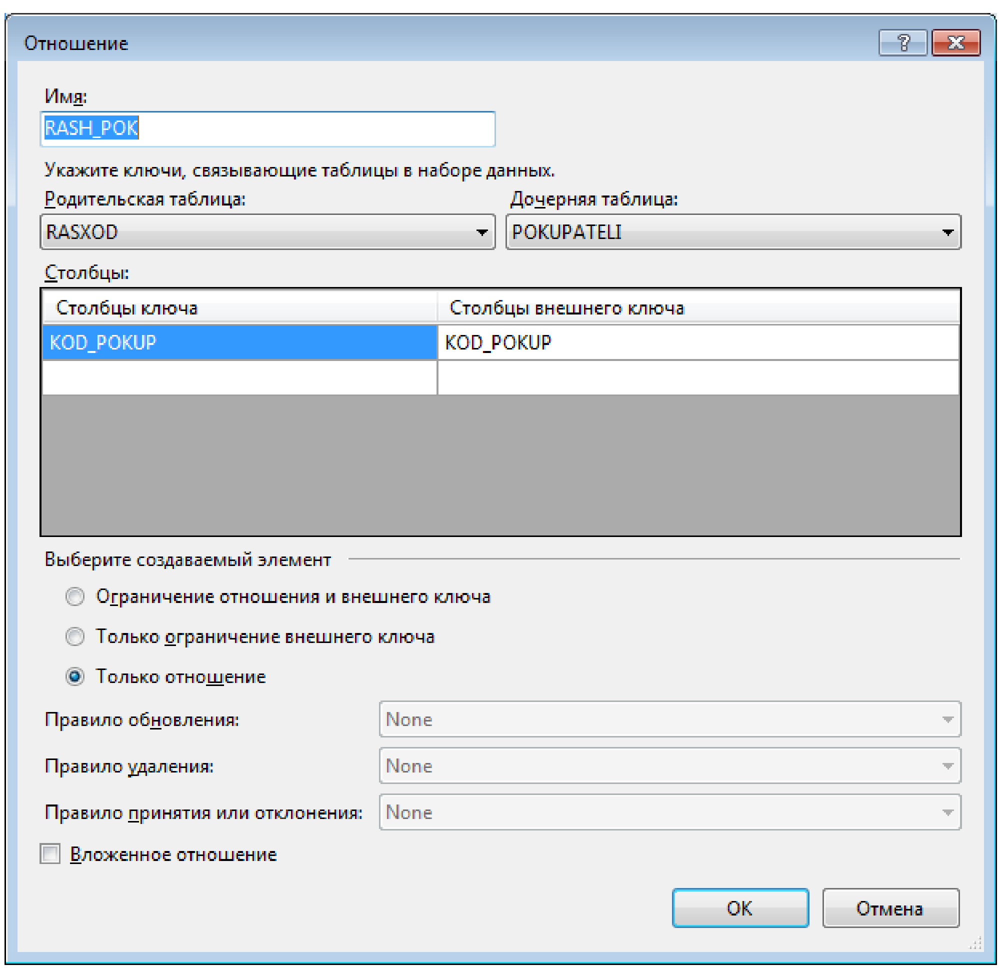
Если мы не хотим сохранять внесенные изменения, и нам надо получить состояние таблицы до внесения изменений, то можно заново заполнить её при помощи метода **Fill** адаптера таблицы. Напишем соответствующий код в обработчике нажатия кнопки **btnRefresh**.

```
        private void btnRefresh_Click(object sender, EventArgs e)
        {
            //обновление данных из источника
        this.rASXODTableAdapter.Fill(myBaseDataSet.RASXOD);
            //обновление состояния навигатора
this.rASXODdataGridView_CurrentCellChanged(rASXODdataGridView,
e);
        }
```

Задача: *Построить код для своего навигатора для одной из таблиц курсового проекта, смоделировав работу навигатора из визуальной библиотеки компонент.*

7.1.2 Отображение связи между таблицами «Master – Detal».

Мы имеем визуализацию всех трех таблиц нашего проекта. Одна из таблиц («Покупки»), вообще говоря, связана с таблицами «Товары» и «Покупатели», но это никак не отражается на форме. Создадим операционные связи между таблицей «Покупки» и таблицами «Товары» и «Покупатели», так чтобы синхронизировать указатели записей в таблицах. Конечно, у в базе данных мы, определяя внешние ключи связывали эти таблицы, эти связи отражаются в созданном наборе данных **myBaseDataSet** под именами **TOV_RASH** и **POK_RASH.** Можно использовать эти связи так, что при фокусировке одной из таблиц «Товары» или «Покупатели», таблица «Покупки» показывала только записи, связанные с выделенной записью одной из вышеупомянутых таблиц. Но более предпочтительным решением является реализация обратного отношения между таблицами так, что при фокусировании записи на таблице «Покупки», мы видели подробную информацию по покупателю и товару, которые участвуют в процессе покупок.



С этой целью откроем контейнер базы данных **MyBaseDataSet.xsd** и добавим соответствующие связи между таблицами. Выше показана связь между таблицами **RASXOD** и **POKUPATELI** по ключевому полю **KOD_POKUP.**

Аналогичным образом добавим связь между таблицами **RASXOD** и **TOVARY** по ключевому полю **KOD_TOVAR.**

Связь между таблицами устанавливается изменением свойств источника данных у таблиц. Добавим в процедуру **rASXODdataGridView_Click** операторы

this.tOVARYdataGridView.DataSource = rASHTOVBindingSource;

this.pOKUPATELIdataGridView.DataSource = rASHPOKBindingSource;

в процедуру **tOVARYdataGridView_Click** оператор

this.tOVARYdataGridView.DataSource = tOVARYBindingSource;

а в процедуру **pOKUPATELIdataGridView_Click** оператор

this.pOKUPATELIdataGridView.DataSource = pOKUPATELIBindingSource;

Объявленная нами связь «Detal-ведомая -> Master-ведущая» для таблиц RASXOD -> TOVARY ограничила область видимости в гриде таблицы «Покупки». Пока мы работаем в таблице «Покупки» — это возможно даже и очень удобно. Но когда мы переключаемся на таблицу «Товары» хотелось бы видеть весь список товаров. Для устранения этих проблем добавим в обработчик нажатия мышью на таблицу **tOVARYdataGridView** оператор:

this.tOVARYdataGridView.DataSource = tOVARYBindingSource;

Соответствующие изменения производим с методом **pOKUPATELIdataGridView_Click**, добавляя оператор

this.pOKUPATELIdataGridView.DataSource = pOKUPATELIBindingSource;

Теперь наши процедуры фокусировки таблиц выглядят следующим образом:

```
private void rASXODdataGridView_Click(object sender, EventArgs e)
```

```
        {
            rASXODdataGridView.DataSource = rASXODBindingSource;
            this.tOVARYdataGridView.DataSource =
rASHTOVBindingSource;
            this.pOKUPATELIdataGridView.DataSource =
rASHPOKBindingSource;
            lblTable.Text = "Покупки";
        }

      private void tOVARYdataGridView_Click(object sender,
EventArgs e)
        {
            bindingNavigator1.BindingSource =
tOVARYBindingSource;
            lblTable.Text = "Товары";
            this.tOVARYdataGridView.DataSource =
tOVARYBindingSource;
        }

      private void pOKUPATELIdataGridView_Click(object sender,
EventArgs e)
        {
            bindingNavigator1.BindingSource =
pOKUPATELIBindingSource;
            lblTable.Text = "Покупатели";
            this.pOKUPATELIdataGridView.DataSource =
pOKUPATELIBindingSource;
        }
```

Теперь мы будем видеть полный список товаров и покупателей в своем «окне просмотра», а находясь в окне просмотра «Покупки» будем видеть информацию только о товаре и покупателе, которые связаны с текущей покупкой.

Выше уже отмечалось, что таблицы «Товары» и «Покупки» связаны по формулам:

- ***Стоимость купленного товара = Количество купленного товара * цена единицы товара,***
- ***Количество товара на складе = Количество товара на складе – Количество купленного товара.***

При этом заметим, *«Количество купленного товара» и «цена единицы товара» («Количество товара на складе») лежат в разных таблицах.* Решением второй задачи мы уже занимались.

- Теперь для события **CellValueChanged** объекта **rASXODdataGridView** напишем обработчик

```
private void rASXODdataGridView_CellValueChanged(object sender,
DataGridViewCellEventArgs e)
        {
            if (rASXODdataGridView.Columns[e.ColumnIndex].Name
== "KOLVO")
            {
             int kod, zena, kolvo;
           //получение кода товара в таблице RASXOD
            kod = Convert.ToInt32(rASXODdataGridView.Rows[
e.RowIndex].Cells["KOD_TOVAR"].Value);
           //Поиск цены товара в таблице Товары по коду товара
            zena = this.myBaseDataSet.TOVARY.FindByKOD_TOVAR(
kod).ZENA;
            kolvo = Convert.ToInt32(rASXODdataGridView.Rows[
e.RowIndex].Cells["KOLVO"].Value);
           //Вычисление нового значения стоимости
          rASXODdataGridView.Rows[e.RowIndex].Cells[
"STOIM"].Value = zena * kolvo;
            }
         }
```

При этом естественно значение свойства **ReadOnly** столбца **STOIM** объекта **rASXODdataGridView** положить равным **True**, чтобы запретить независимые корректировки этого реквизита. С этой целью необходимо дважды щелкнуть мышью на объекте

rASXODdataGridView, вызвать редактор столбцов и манипулируя левой кнопкой мыши получить доступ ко всем столбцам, в появившемся списке выбрать столбец, характеризующий стоимость и в свойстве **ReadOnly** выбрать значение **True.**

- Остается еще один случай, в котором не обеспечено корректное обновление значение поля «стоимость» - при изменении цены товара в таблице «Товары». Для этого добавим в БД триггер:

```
CREATE TRIGGER AU_TOVARY FOR TOVARY
AFTER UPDATE
     AS BEGIN
          IF (UPDATE(ZENA))
          BEGIN
               UPDATE RASXOD SET STOIM=inserted.ZENA*KOLVO
               FROM inserted
               WHERE RASXOD.KOD_TOVAR = inserted.KOD_TOVAR;
          END
     END
```

Установим контроль условия KOLVO>=0 – обработчик события **CellValidating** поля **Kolvo** объекта **dataGridView3:**

```
private void rASXODdataGridView_CellValidating(object sender,
DataGridViewCellValidatingEventArgs e)
        {
            try
            {
                if (rASXODdataGridView.Columns[
e.ColumnIndex].Name == "KOLVO")
                    if (Convert.ToInt32(e.FormattedValue) < 0)
                        throw new Exception("Количество не может
быть отрицательным");
            }
            catch (Exception a)
            {
                e.Cancel = true;
```

```
                MessageBox.Show(a.Message);
                rASXODdataGridView.Rows[e.RowIndex].ErrorText =
a.Message;
            }
        }
```

Таким способом порожденная в **CellValidating** исключительная ситуация не просто выдает сообщение, но и блокирует подтверждение введенного значения поля.

Прежде чем двигаться дальше, полезно посмотреть поведение программы в действии:

- Как отрабатывает каскадное обновление таблицы «Расход товара» при удалении строки в таблице «Товары». Аналогично, для таких же операций с таблицей «Покупатели.
- Как отрабатывает контроль ограничений целостности в таблице «Покупки» при изменении кодов товара или покупателя на отсутствующие в родительских таблицах.
- Как отрабатывает контроль условия **ZENA>=0** и условия **KOLVO>=0.**

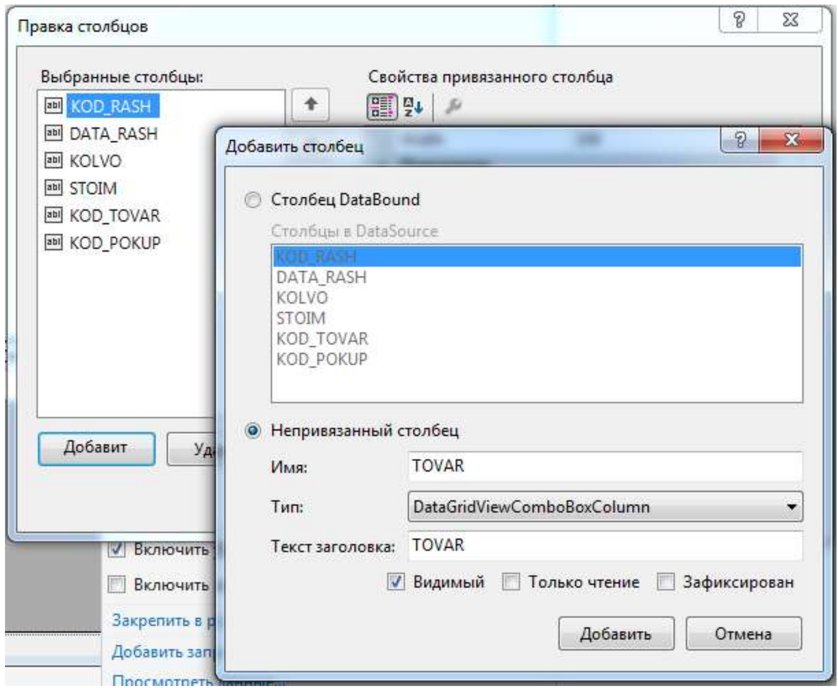
  Обратите внимание: реакция на нарушение первого условия появляется только после попытки сохранить запись с ошибочными данными, а для второго условия реакция почти мгновенная – программа не позволяет покинуть поле.

  Это связано с тем, что второе условие контролирует клиентская программа, причем именно в момент выхода из поля. А первое условие контролирует SQL-сервер, причем, только тогда, когда он получает от клиента ошибочную запись.
- Как отрабатывает перерасчет «стоимости» при изменении вместе и по отдельности «количества» и «кода товара» в таблице «Расход товара», а также изменении цены товара в таблице «Товары».
- Как отрабатывает перерасчет «количества на складе» при таких же изменениях в таблице «Расход товара».
- Как отрабатывает перерасчет «количества на складе» при удалении и добавлении строк в таблице «Расход товара». И как при добавлении отрабатывает генератор «кода покупки».

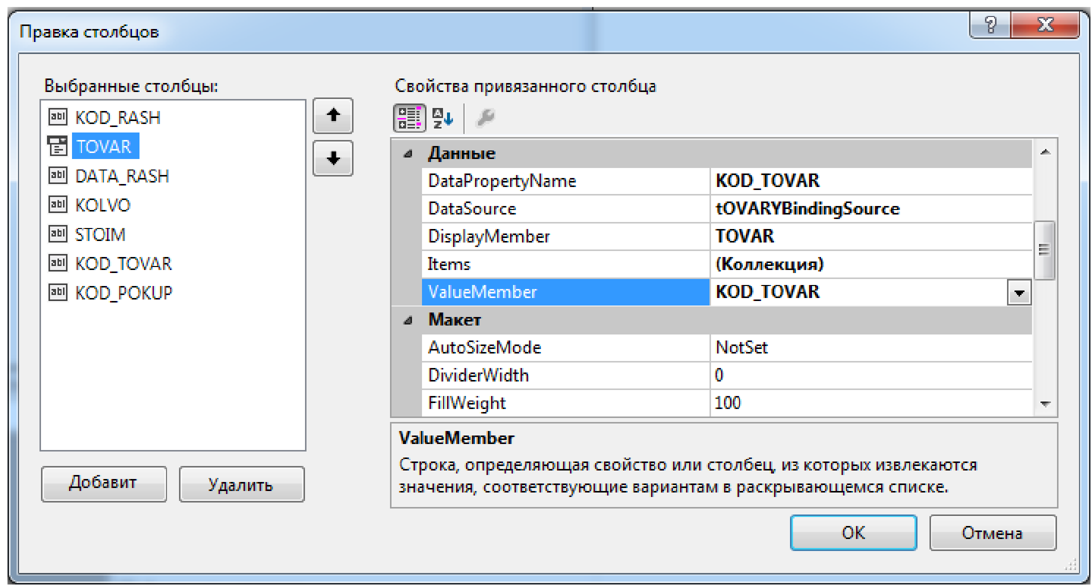
7.1.3 Использование Lookup – полей.

Можно заметить, что значение полей, описывающих коды записей и необходимых для обеспечения уникальности записей, не носят содержательный характер, и могут быть исключены из состава столбов визуальной компоненты **dataGridView**. Также желательно наименования столбцов таблиц выводить на русском языке. Для исключения столбца из состава столбцов визуальной компоненты необходимо выбрать мышью объект **datadGridView**, в инспекторе объектов найти свойство **Columns**, открыть окно редактирования столбцов, через контекстное меню создать список, включающий все поля таблицы (далее можно удалить ненужные столбцы из списка). Каждый столбец, являясь объектом, имеет ряд свойств. Для изменения наименования столбца необходимо исправить для каждого поля свойство **HeaderText**.

Записи в Таблице «Расход товара» не содержат наименований товаров и покупателей, а содержат только коды этих реквизитов. Поскольку код товара не несет содержательной информации, а конечный пользователь привык работать с наименованием товара, желательно при показе таблицы «Расход товара» выводить на экран не коды товара и покупателя, а их наименования. Этого можно достичь, используя поля выбора данных (LookUp - поля). Создадим LookUp-поле для наименования товара. Выберем компоненту **rASXODdataGridView** отображающую таблицу «Расход товара» и список полей таблицы.



Выбрав кнопку «Добавить», перейдём в окно добавления нового столбца. Добавляем «Непривязанный столбец», задаём имя и выбираем тип поля **DataGridViewComboBoxColumn**, так как новое поле будет позволять выбирать значения из списка.



На данном экране описана LookUp-связь по ключевому полю **KOD_TOVAR** с таблицей «Товары». В результате всех этих действий в таблице «Расход товара» образуется новое текстовое поле **TOVAR**, которое связано по полю **KOD_TOVAR** с соответствующим полем таблицы «Товары». Совершенно аналогично поступим с полем POKUP.

В столбцах компоненты **rASXODdataGridView** появляются элементы управления с выпадающим списком (с наименованием товара или покупателя) через которые можно осуществить выбор значения соответствующего реквизита.

KOD_RASH	TOVAR	DATA_RASH	KOLVO	STOIM	KOD_TOVAR	KO
2	Сахар	12.04.1998	10	120	1	6
5		12.07.2000	12	111	4	5
6		13.09.2000	5	25	2	2
8		16.07.2000	7	70	6	1
9		23.05.2000	11	110	4	1
11	Макароны	12.04.2003	10	50	2	6

Сахар
Макароны
Огурцы весовые
Огурцы баночные
Крупа манная
Масло подсолнечн

Можно изменить свойство **FlatStyle** поля на **Flat** для изменения цвета заливки элемента колонки.

Также для улучшения интерфейса можно убрать из гридов поля с числовыми кодами, а также задать именования колонок на русском языке.

7.1.4 Использование клиентским приложением хранимых процедур и функций.

У нас в базе данных есть хранимая процедура **Get_Kol_Tov** с тремя входными параметрами и одним и выходным. Передав имя товара и две даты процедуре, мы получим общее количество продаж данного товара за промежуток времени. Реализуем эту возможность на форме нашего клиентского приложения. Для этого добавим два элемента **DateTimePicker** – назовем их **dateFrom**, **dateTo**. Имя товара будем брать из текущей строки грида отображающего таблицу «Товары». Для вызова функции используем кнопку **btnSProc**, результат покажем в метке на форме **lblKolvo**.

Наша хранимая процедура не возвращает табличные данные, поэтому её нельзя использовать в контейнере **DataSet**. Придется работать напрямую с базой данных. Для начала получим из текущей строки значение поля **Tovar**. Далее создаем **SqlConnection** - соединение с БД, строку соединения можно написать вручную, либо извлечь из свойств проекта.

Следующий шаг – создание команды, при помощи которой мы вызовем нашу хранимую процедуру. Свойства этого объект нужные нам:

- **Connection** – подключение к БД
- **CommandType** – тип команды: SQL-команда, хранимая процедура или таблица.
- **CommandText** – собственно текст команды, имя хранимой процедуры или таблицы.
- **Parameters** – коллекция параметров команды.

Для добавления входных параметров достаточно имени и значения параметра. Для выходного нам придется задать тип **SqlDbType.Int** и направленность параметра **ParameterDirection.Output.**

Когда все необходимые свойства команды заданы, можем её выполнить. Предварительно надо открыть соединение с БД. Команда может запускаться тремя способами, в зависимости от типа команды и возвращаемых значений. Если результатом выполнения команды будет набор записей, таблица. Тогда запускаем метод **ExecuteReader** – он возвращает **SqlDataReader**, из которого можно получить строки результата запроса. Если результат выполнения команды – одно число, скалярное значение, то нужно вызвать метод **ExecuteScalar**. Если у команды нет возвращаемых значений, или как в нашем случае все данные находятся в параметрах. Тогда для исполнения команды выполняем метод **ExecuteNonQuery**.

Теперь мы можем получить значение выходного параметра и закрыть соединение с базой данных.

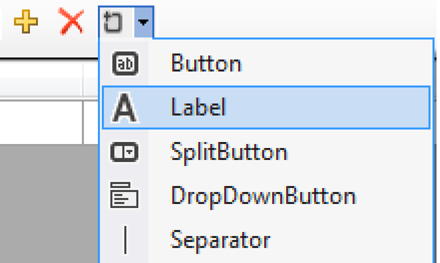
```
private void btnSProc_Click(object sender, EventArgs e)
{
  try {
     String tovar =
dataGridView2.CurrentRow.Cells["tOVARDataGridViewTextBoxColumn"]
.Value.ToString();
     SqlConnection con = new SqlConnection();
     con.ConnectionString =
Properties.Settings.Default.myBaseConnectionString;
```

```
    SqlCommand cmd = new SqlCommand();
    cmd.Connection = con;
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.CommandText = "Get_Kol_Tov";

    SqlParameter param = new SqlParameter("@SumKol",
SqlDbType.Int);
    param.Direction = ParameterDirection.Output;
    cmd.Parameters.Add(param);
    cmd.Parameters.Add(new SqlParameter("@Tovar", tovar));
    cmd.Parameters.Add(new SqlParameter("@DataL",
dateFrom.Value));
    cmd.Parameters.Add(new SqlParameter("@DataH",
dateTo.Value));

    con.Open();
    cmd.ExecuteNonQuery();
    string kolvo = cmd.Parameters["@SumKol"].Value.ToString();
    con.Close();

    lblKolvo.Text = tovar + " израсходован в количестве " +
kolvo;
    }
  catch(Exception ex)
    {
          MessageBox.Show(ex.Message);
    }
}
```

7.2 Создание дополнительных форм клиентского приложения. Построение альтернативной формы

На основной форме мы имеем три визуальные компоненты **dataGridView**, которые одновременно показывают состояние трех таблиц. Рассмотрим несколько другую реализацию доступа к данным таблиц через единственную визуальную компоненту. С этой целью определим новую форму, выбрав пункт «Проект» -> *«Добавить форму Windows»* основного меню. Зададим имя формы, и на экране появится чистый бланк, где можно начать построение.

Поместим на новую форму: компонент **dataGridView**, компонент **BindingNavigator**, 5 компонент **Button**. Настроим **dataGridView** на источники данных, поочередно выберем все три таблицы из контейнера **DataSet** проекта. Visual Studio создаст невизуальные компоненты **BindingSource** и **TableAdapter** для каждой таблицы. Оставим грид и навигатор настроенными на таблицу «Покупатели». В навигатор добавим **Label**, будет создан компонент **toolStripLabel**.



Назовём его **lblTableName**, увеличим шрифт и зададим свойство **Text** – «Покупатели». Три кнопки назовём именами таблиц – «Товары», «Покупатели» и «Расход товара». Остальные кнопки будут отправлять изменения на сервер в базу данных и закрывать форму.

Добавим в обработчик события **Form2_Load** строку

```
dataGridView1.AutoGenerateColumns = true;
```

Она задаёт свойство автоматической генерации столбцов, при изменении источника данных. Далее на каждую кнопку, напишем

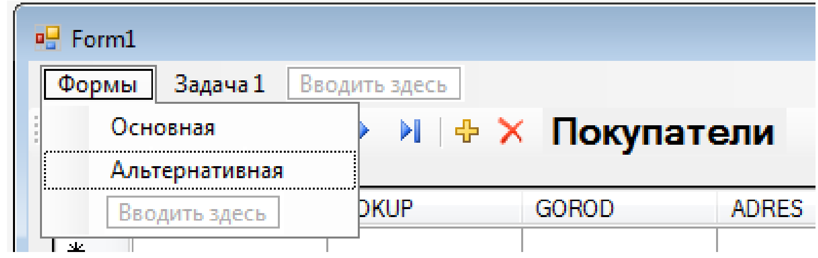
обработчик меняющий источник данных на соответствующую таблицу. Если выбирается текущая таблица, то ничего не происходит.

```
private void btnTovary_Click(object sender, EventArgs e)
{
     if (lblTableName.Text != "Товары")
     {
        dataGridView1.Columns.Clear();
        dataGridView1.DataSource = tOVARYBindingSource;
        bindingNavigator1.BindingSource = tOVARYBindingSource;
        lblTableName.Text = "Товары";
      }
}
```

Аналогично для остальных таблиц. Для кнопки «Сохранить» нам понадобится ещё один невизаульный компонент **tableAdapterManager**, его можно найти в «Панели элементов» в разделе компоненты проекта **AdoNetApp**. Метод **UpdateAll** сохраняет все изменения, произошедшие в **DataSet**.

```
private void btnUpdate_Click(object sender, EventArgs e)
{
     tableAdapterManager.UpdateAll(myBaseDataSet);
     MessageBox.Show("Изменения сохранены");
}
```

Нам осталось попасть на альтернативную форму из основной формы приложения. Для этого возвратимся на первую форму и добавим компонент ***MenuStrip***, из раздела «Меню и панели инструментов». Далее заполняем пункты меню.



Двойной щелчок на пункте меню, создаст обработчик нажатия. Добавим код создания альтернативной формы и отобразим её на экране. Также можно использовать метод **ShowDialog**, тогда форма будет отображаться модально.

```
 private void альтернативнаяToolStripMenuItem_Click(object sender, EventArgs e)
{
	Form2 f2 = new Form2();
	f2.Show();
}
```

8. Запросы

8.1 Задача 1

Рассмотрим задачу: ***Используя таблицы «Товары» и «Расход товаров» построить выборку товаров, купленных за период с 2012 по 2013 годы, включающую:***

- ***наименование товара***
- ***дату покупки***
- ***объем покупки***
- ***цена за единицу товара***
- ***стоимость покупки***

в количестве большем некоторого заданного числа.

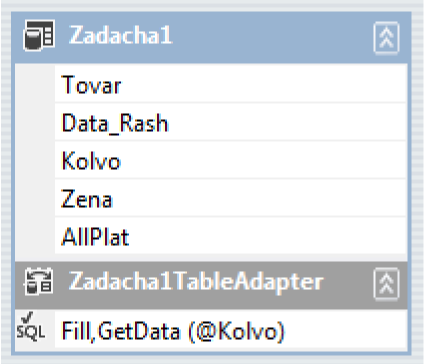
Рассмотрим несколько вариантов решения задачи:

- В **первом варианте** решения поставленной задачи воспользуемся *Set*-ориентированными средствами *SQL* для работы с таблицами и объектом типа **TableAdapter** для связи с *SQL*-сервером.

  Откроем в панели «Обозреватель решений» файл, в котором хранится набор данных проекта. Откроется окно конструктора редактирования **DataSet**. Добавим новый **TableAdapter,** в качестве команды SQL для заполнения таблицы используем:

```
SELECT Tovary.TOVAR,Rasxod.DATA_RASH,Rasxod.KOLVO,
      Tovary.ZENA, Rasxod.Kolvo*Tovary.Zena AS AllPlat
  FROM Tovary,Rasxod
  WHERE Tovary.Kod_Tovar=Rasxod.Kod_Tovar AND
        Rasxod.Data_Rash>='01.01.2010' AND
        Rasxod.Data_Rash<='31.12.2012' AND
        Rasxod.Kolvo >= @Kolvo
```

Зададим имя **DataTable** – **Zadacha1,** тогда и адаптер таблицы станет соответственно – **Zadacha1TableAdapter.**



Для заполнения таблицы, вызовем метод **Fill.** Но теперь у него появился новый параметр – это значение параметра нашего запроса **@Kolvo**, при первоначальном заполнении поставим его равным нулю.

```
private void Zadacha1_1_Load(object sender, EventArgs e)
{
 dataGridView1.AutoGenerateColumns = true;
 this.zadacha1TableAdapter.Fill(myBaseDataSet.Zadacha1, 0);
 this.Text = "Запрос на основании объекта TableAdapter";
}
```

Для того чтобы пользователь мог задавать значения параметра, добавим на форму поле **txtKolvo** и кнопку **btnFill.**

```
private void btnFill_Click(object sender, EventArgs e)
 {
 int kolvo = Convert.ToInt32(txtKolvo.Text);
 this.zadacha1TableAdapter.Fill(myBaseDataSet.Zadacha1,  kolvo);
 }
```

- Во **втором варианте** решения поставленной задачи воспользуемся *Record*-ориентированными средствами *Visual Studio* для работы с таблицами, наиболее близкими к традиционным процедурным средствам обработки файлов.

Ниже в методе **QueryZapr1** мы будем использовать **DataTable,** который был создан в **DataSet** при выполнении первого варианта. В этой таблице будем накапливать результат запроса. Все объекты этого примера размещены в новой форме **Zadacha1_2.** Визуальные компоненты можно перенести из формы первого варианта, или выбрать из панели элементов. Из невизуальных нам понадобятся компоненты проекта – **myBaseDataSet, rasxodTableAdapter, tovaryTableAdapter** и **zadacha1BindingSource.** Датагрид и навигатор настроим на **zadacha1BindingSource.**

```
private void Zadacha1_2_Load(object sender, EventArgs e)
{
   this.tOVARYTableAdapter.Fill(this.myBaseDataSet.TOVARY);
   this.rASXODTableAdapter.Fill(this.myBaseDataSet.RASXOD);
   //Заполним данными таблицы в датасете
   QueryZapr1();
}

public void QueryZapr1()
{
   myBaseDataSet.Zadacha1.Clear();
   //очистим таблицу от предыдущих значений
   foreach(myBaseDataSet.RASXODRow rRow in
   myBaseDataSet.RASXOD.Rows)
   {     //пройдём по всем строкам таблицы «Расход товара»
      if((rRow.DATA_RASH>=Convert.ToDateTime("01.01.2010"))
   && (rRow.DATA_RASH<=Convert.ToDateTime("31.12.2012")))
      {  //проверка выполнения ограничения на дату
         if (rRow.KOLVO >= Convert.ToInt32(txtKolvo.Text))
        {   //проверка выполнения условия на количество
        foreach (myBaseDataSet.TOVARYRow tRow in
        myBaseDataSet.TOVARY.Rows)
         {  //второй цикл по строкам таблицы «Товары»
         if (rRow.KOD_TOVAR == tRow.KOD_TOVAR)
           {  //нашли нужный товар
```

```
                myBaseDataSet.Zadacha1Row zRow =
            myBaseDataSet.Zadacha1.NewZadacha1Row();
                  //создали новую строку таблицы «Zadacha1»
                zRow.TOVAR = tRow.TOVAR;
                zRow.DATA_RASH = rRow.DATA_RASH;
                zRow.KOLVO = rRow.KOLVO;
                zRow.ZENA = tRow.ZENA;
                zRow.AllPlat = rRow.KOLVO * tRow.ZENA;
                myBaseDataSet.Zadacha1.AddZadacha1Row(zRow);
                  //добавили строку в результирующую таблицу
                }
            }
          }
      }
    }
    dataGridView1.Refresh();
    this.Text = "Запрос через двойной цикл";
    //редактирование заголовка формы
}
```

- В **третьем варианте** решения поставленной задачи, воспользуемся первичным ключами для поиска в таблице **TOVARY** товара с интересующим нас кодом товара. Поиск строки в таблице по её ключу можно выполнить с помощью метода **Find** объекта **DataRowCollection.** У этого метода две перегрузки для одинарного и для составного первичного ключа.

public DataRow Find(Object key) или

public DataRow Find(Object[] keys)

Для любого первичного ключа таблицы, SQL Server строит поисковый индекс. Поиск по индексированному полю быстрее, чем полный цикл обхода. Получим время поиска порядка O(log n), где n – количество записей. Теперь мы можем написать метод заполнения результирующей таблицы в одном цикле.

```
private void QueryZapr2()
{
   myBaseDataSet.Zadacha1.Clear();
   foreach (myBaseDataSet.RASXODRow rRow in
myBaseDataSet.RASXOD.Rows)
   {
   if ((rRow.DATA_RASH >= Convert.ToDateTime("01.01.2010"))
&& (rRow.DATA_RASH <= Convert.ToDateTime("31.12.2012")))
      {
      if (rRow.KOLVO >= Convert.ToInt32(txtKolvo.Text))
        {
            myBaseDataSet.TOVARYRow tRow =
            myBaseDataSet.TOVARY.Rows.Find(rRow.KOD_TOVAR) as
            myBaseDataSet.TOVARYRow;
//Ищем по первичному ключу, получив DataRow приводим к типу
// myBaseDataSet.TOVARYRow при помощи оператора as
            myBaseDataSet.Zadacha1Row zRow =
            myBaseDataSet.Zadacha1.NewZadacha1Row();
            zRow.TOVAR = tRow.TOVAR;
            zRow.DATA_RASH = rRow.DATA_RASH;
            zRow.KOLVO = rRow.KOLVO;
            zRow.ZENA = tRow.ZENA;
            zRow.AllPlat = rRow.KOLVO * tRow.ZENA;
            myBaseDataSet.Zadacha1.AddZadacha1Row(zRow);
         }
      }
   }
   dataGridView1.Refresh();
```

```
    this.Text = "Запрос через поиск по ключу";
}
```

Мы считаем, что база данных удовлетворяет условию ссылочной целостности. То есть для всех внешних ключей существуют записи в родительской таблице. А значит метод **Find** не вернёт **NULL**. Для полной уверенности, можно было поставить эту проверку, но оставим её на совести СУБД.

Если первичный ключ составной, то в качестве параметра надо передавать **Object**. Допустим, в таблице **TOVARY** ключ состоит из полей **KOD_TOVAR** и **NUM_SKLAD**. Тогда поиск нужной строки выглядит так:

```
Object[] keys = {rRow.KOD_TOVAR, rRow.NUM_SKLAD};
myBaseDataSet.TOVARYRow tRow =
myBaseDataSet.TOVARY.Rows.Find(keys) as myBaseDataSet.TOVARYRow;
```

- В **четвертом варианте** решения поставленной задачи воспользуемся операционной связью ***DataRelation*** для таблиц ***TOVARY*** -> ***RASXOD*** в наборе данных.

Такой возможностью мы уже пользовались для того, чтобы таблица «Расход товаров» показывала операции только с товаром, на который позиционирована таблица «Товары» Теперь нас интересует обратная зависимость, по записи расхода товара определить наименование товара.

Использование реляции позволит устранить необходимость использования **TOVARY.Rows.Find()**. Существование связи между таблицами в наборе данных позволяет переходить от строк дочерней таблице к строкам родительской, и наоборот. Для перехода от строки дочерней таблицы используется метод **GetParentRow**, по условию ссылочной целостности у нас есть только одна родительская запись.

```
public DataRow GetParentRow(string relationName)
```

Обратная операция, возвращает уже коллекцию дочерних строк для одной строки родительской таблицы. Коллекция может оказаться и пустой.

```
public DataRow[] GetChildRows(string relationName)
```

Элементы на форме те же, как и в всех предыдущих вариантах. Теперь напишем метод заполняющий результирующую таблицу.

```
private void QueryZapr3()
{
  myBaseDataSet.Zadacha1.Clear();
  foreach (myBaseDataSet.RASXODRow rRow in
myBaseDataSet.RASXOD.Rows)
    {
    if ((rRow.DATA_RASH >= Convert.ToDateTime("01.01.2010"))
&& (rRow.DATA_RASH <= Convert.ToDateTime("31.12.2012")))
      {
      if (rRow.KOLVO >= Convert.ToInt32(txtKolvo.Text))
       {
        myBaseDataSet.TOVARYRow tRow =
        rRow.GetParentRow("TOV_RASH") as
        myBaseDataSet.TOVARYRow;
// Получили строку из родительской таблицы, связанной с таблицей
// RASXOD связью с именем TOV_RASH и привели к типу

       myBaseDataSet.Zadacha1Row zRow =
myBaseDataSet.Zadacha1.NewZadacha1Row();
       zRow.TOVAR = tRow.TOVAR;
       zRow.DATA_RASH = rRow.DATA_RASH;
       zRow.KOLVO = rRow.KOLVO;
       zRow.ZENA = tRow.ZENA;
       zRow.AllPlat = rRow.KOLVO * tRow.ZENA;
       myBaseDataSet.Zadacha1.AddZadacha1Row(zRow);
       }
    }
  }
```

```
    dataGridView1.Refresh();
    this.Text = "Запрос через DataRelation";
}
```

- В **пятом варианте** решения поставленной задачи будем использовать присоединенный режим работы с базой данных. То есть сами открываем соединение с БД, выполняем команду SQL и закрываем соединение. Основной объект получения данных **SqlDataReader.** Основные отличия от **SqlDataAdapter** и **SqlTableAdapter:**
  - Данные только для чтения, то есть используется только одна команда Select
  - Читать можно только один раз с первой записи до последней, для повторного чтения надо заново запустить **SqlDataReader.**
  - На текст команды Select нет ограничений, можно использовать любые конструкции языка SQL.

  SqlDataReader создается при помощи метода **ExecuteReader** объекта **SqlCommand.** Далее по нему можно пробежаться в цикле, получая по очереди каждую запись результата запроса. В нашем случае мы отправим все записи в таблицу. Таблица **DataTable** автоматически формируется и заполняется при помощи метода **Load**, в параметр которого передается **SqlDataReader.**

  Для унификации результата на экране, добавим на форму те же самые визуальные элементы, как и в предыдущих вариантах. Невизуальные компоненты нам не понадобятся, так мы не используем **DataSet.** Определим метод **FillGridByReader**, будем вызывать его при загрузке формы и при нажатии кнопки после редактирования текстового поля со значение ограничения по количеству. Так как источник данных для **DataGridView** определяется во время выполнения, то среда исполнения не знает какими должны быть столбцы грида. За возможность автоматический генерации колонок **DataGridView** отвечает булево

свойство **AutoGenerateColumns**. По умолчанию Visual Studio устанавливает его в **false**.

```
 private void Zadacha1_5_Load(object sender, EventArgs e)
 {
   dataGridView1.AutoGenerateColumns = true;
   FillGridByReader();
 }

 private void FillGridByReader()
 {
    SqlConnection con = new SqlConnection(
Properties.Settings.Default.myBaseConnectionString);
// создаем объект связь с бд, строку соединения берём из
// свойств проекта, можно задать самим строкой
    con.Open();
// подключаемся к бд
   String str = "SELECT Tovar,Data_Rash,Kolvo,Zena,Kolvo*Zena
AS AllPlat FROM Tovary,Rasxod " +
   WHERE Tovary.Kod_Tovar=Rasxod.Kod_Tovar and Data_Rash
between '01.01.2010' AND '31.12.2012' AND KOLVO >= " +
txtKolvo.Text;
// задаем текст запроса, добавляем текст из txtKolvo

   SqlCommand cmd = new SqlCommand(str, con);
   SqlDataReader rdr = cmd.ExecuteReader();
// создали команду и выполнили метод ExecuteReader

        DataTable dt = new DataTable();
        dt.Load(rdr);
        con.Close();
        // при помощи ридера заполнили таблицу и закрыли
        // соединение с бд

        BindingSource bs = new BindingSource();
        bs.DataSource = dt;
```

```
    // программно создали объект BindingSource и связали
    // его с таблицей, далее грид и навигатор укажем на
    // него для связи с таблицей

   dataGridView1.DataSource = bs;
   bindingNavigator1.BindingSource = bs;
   dataGridView1.Refresh();
  }
```

8.2 Задача 2.

Вторая задача связана с построением запроса, использующего группировку данных:

Найти список товаров общий объем продаж, которых после определенной даты превышал некоторую заданную сумму.

Сразу заметим, что в данной задаче не конкретизированы ни общая сумма продаж, ни дата, после которой необходимо проводить расчеты. Эти данные задаются в виде параметров. С другой стороны, при реализации необходимо уметь подсчитывать суммарный объем стоимости проданных товаров. Ниже приведен пример реализации запроса с использованием SQL оператора, использующего группировку данных.

- Как и ранее для решения поставленной задачи воспользуемся *Set*-ориентированными средствами *SQL* для работы с таблицами и объектом типа **SqlTableAdapter** для связи с *SQL*-сервером.

  Все объекты этого примера размещены на форме **Zadacha2_1**, на которой результаты запроса визуализируются в **dataGridView1.** Перед началом работы добавим в источник данных нашего проекта **myBaseDataSet** новый **TableAdapter.** Команда SELECT новой таблицы **Zadacha2** будет содержать следующий текст:

```
SELECT MAX(A.TOVAR) AS TOVAR, SUM (B.STOIM) AS ALLST
   FROM TOVARY A,RASXOD B
     WHERE A.KOD_TOVAR=B.KOD_TOVAR AND
           B.DATA_RASH>=@FromData
     GROUP BY A.KOD_TOVAR
     HAVING SUM(B.STOIM)>@SumAll
```

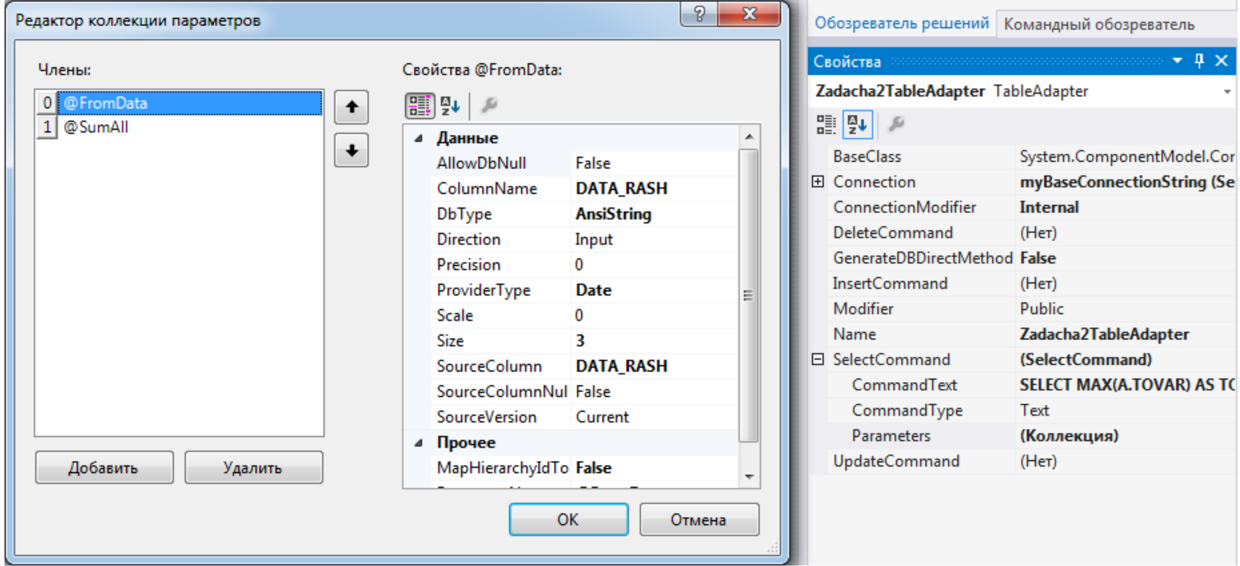
  В данном операторе производится группировка данных по полю **KOD_TOVAR**[10] и подсчитываются суммы продаж для каждого вида товара. Результаты подсчетов перед выводом фильтруются

[10] Группировка по полю Kod_Tovar представляется естественной, поскольку это поле является ключевым. С другой стороны, поскольку мы бы хотели в отчете получить наименование товаров, необходимо обеспечить, чтобы каждая формируемая группа не содержала различных имен. И, хотя в нашем примере каждому значению кода товара соответствует единственное наименование, и условие единственности значения имени выполняется, тем не менее, при написании оператора SELECT необходимо явно указать отсутствие неоднозначности. По этой причине при формировании наименования товара использована функция MAX.

условием **SUM(B.STOIM)>@SumAl.** Заметим также, что оператор SELECT имеет два параметра: **@FromData** и **@SumAll.** Эти параметры задаются пользователем через объекты типа **DateTimePicker** и **TextBox**[11]. Инициализация параметров производится заданием свойств объектов в режиме разработки или в обработчике события загрузки формы **Zadacha2_1_Load.** По нажатию кнопки «Заполнить» на форме – проверяем значение текстового поля **txtSum** и вызываем метод **Fill,** созданного адаптера таблицы.

```
private void btnFill_Click(object sender, EventArgs e)
{
    if( Convert.ToInt32(txtSum.Text) >= 0 )

   this.zadacha2TableAdapter.Fill(this.myBaseDataSet.Zadacha2,
   dtpDateFrom.Value.ToString(), Convert.ToInt32(txtSum.Text));
}
```

Типы параметров можно посмотреть, щелкнув на свойстве **Parameters** команды **SelectCommand** объекта **Zadacha2TableAdapter** в наборе данных проекта.



[11] Вообще-то необходимо проверять правильность форматов вводимых данных. В данной программе это сделано для переменной SumAll.

- Record-ориентированный подход к решению этой задачи подобен методам, описанным для задачи 1. Для однократного прохода по строкам таблицы **RASXOD** потребуем, чтобы таблица была упорядочена по коду товара. Для этого в адаптере таблицы добавим в текст команды **SelectCommand** сортировку.

  SELECT KOD_RASH, DATA_RASH, KOLVO, STOIM, KOD_TOVAR, KOD_POKUP FROM RASXOD ORDER BY KOD_TOVAR

  Тогда можем написать код заполняющий результирующую задачу:

```
public void QueryZ2()
{
    myBaseDataSet.Zadacha2.Clear();
    int kod = -1, sum=0;
    //код товара и сумма продаж этого товара
    myBaseDataSet.Zadacha2Row zRow = null;
    //строка результирующей таблицы
    foreach(myBaseDataSet.RASXODRow rRow in myBaseDataSet.RASXOD.Rows)
    {
        if(rRow.DATA_RASH > dtpDateFrom.Value)
        {
        if(zRow == null)//если первая запись
          {
          kod = rRow.KOD_TOVAR;
          sum = rRow.STOIM;
          zRow = myBaseDataSet.Zadacha2.NewZadacha2Row();
          }
        else if (kod == rRow.KOD_TOVAR) //если тот же товар
          {
           sum += rRow.STOIM;
          }
        else //если новый товар
          {
          if (sum >= Convert.ToInt32(txtSum.Text))
             { //если выполняется условие записываем строку
             zRow.ALLST = sum;
```

```
                zRow.TOVAR=
          myBaseDataSet.TOVARY.FindByKOD_TOVAR(kod).TOVAR;
                myBaseDataSet.Zadacha2.Rows.Add(zRow);
                }
             kod = rRow.KOD_TOVAR;//подготовливаем новую строку
             sum = rRow.STOIM;
             zRow = myBaseDataSet.Zadacha2.NewZadacha2Row();
             }
           }
        }
      if(zRow != null)//проверяем последний товар
         if(sum >= Convert.ToInt32(txtSum.Text))
           {
           zRow.ALLST = sum;
           zRow.TOVAR=
   myBaseDataSet.TOVARY.FindByKOD_TOVAR(kod).TOVAR;
           myBaseDataSet.Zadacha2.Rows.Add(zRow);
           }
      dataGridView1.Refresh();
         }
```

8.3 Задача 3.

Рассмотрим третий более сложный пример запроса, связанного с нашей базой данных:

Найти имена покупателей из Казани, которые до даты Y покупали каждый товар, имеющий цену выше X.

Переформулируем постановку задачи, явно оговорив неявное и явно выделив кванторы: ***найти наименования таких покупателей, которые проживают в Казани и для каждого товара, если он имеет цену выше X, то существуют сведения об отпуске этого товара этому покупателю с датой покупки меньшей Y.***

Устранив импликацию **(A⊃B)≡(¬A∨B)**, получим: ***найти наименования таких покупателей, которые проживают в Казани и для каждого товара, либо его цена не выше X, либо существуют сведения об отпуске этого товара этому покупателю с датой покупки меньшей Y.***

Запрос реляционного исчисления кортежей.

НАЙТИ{(p.POKUP)/p∈POKUPATELI} (p.GOROD='Казань') **&** ∀t∈TOVARY

((t.ZENA <=X) ∨ ∃r∈RASXOD ((r.DATA_RASH<Y) **&**

(r.KOD_TOVAR =t.KOD_TOVAR)**&**(r.KOD_POKUP =p.KOD_POKUP)))

Реализация этого запроса на SQL потребует устранения ∀-квантора с помощью эквивалентностей **∀x A(x) ≡ ¬∃x ¬A(x)** и **¬(A∨B) ≡ ¬A&¬B.** В итоге получим формулировку, не удовлетворяющую требованиям языка реляционного исчисления кортежей, но семантически эквивалентную и подходящую для реализации на SQL:

НАЙТИ{(p.POKUP)/p∈POKUPATELI} (p.GOROD='Казань') **&** ¬∃t∈TOVARY

((t.ZENA >X) **&** ¬∃r∈RASXOD ((r.DATA_RASH<Y) **&**

(r.KOD_TOVAR =t.KOD_TOVAR)**&**(r.KOD_POKUP =p.KOD_POKUP)))

Или в словесной формулировке: ***найти наименования таких покупателей, которые проживают в Казани и не верно, что есть такой товар, цена которого выше X, и нет сведений об отпуске этого товара этому покупателю с датой покупки меньшей Y.***

```
SELECT POKUP FROM POKUPATELI
  WHERE (POKUPATELI.GOROD = 'Казань')
     AND NOT EXISTS (SELECT * FROM TOVARY
       WHERE (TOVARY.ZENA > @Zena) AND NOT EXISTS
        (SELECT * FROM RASXOD  WHERE
            (RASXOD.KOD_POKUP = POKUPATELI.KOD_POKUP) AND
                (RASXOD.KOD_TOVAR = TOVARY.KOD_TOVAR) AND
                    (RASXOD.DATA_RASH  < @Data_Rash) ))
```
[12]

В приведенной реализации параметры **@Zena** и **@DataRash** задаются пользователем, используя объекты классов **TextBox** и **DateTimePicker**.

Код, активизирующий запрос, имеет вид:

```
private void btnFill_Click(object sender, EventArgs e)
{
this.zadacha3TableAdapter.Fill(this.myBaseDataSet.Zadacha3,
dtpDateFrom.Value, Convert.ToInt32(txtZena.Text) );
}
```

Рассмотрим схему (лобовой) реализации этого запроса традиционными средствами уровня языка C#.

Структуру запроса (в первой формулировке) можно представить диаграммой:

[12] Заметим, что если в таблице товар отсутствуют товары, цена которых выше X, запрос выдаст список всех Казанских покупателей, что не очень согласуется с исходным текстом запроса. Связано это с интерпретацией логической импликации A->B. При ложности A импликация всегда истинна. Для учета всех граничных условий необходимы дополнительные усилия.

Учитывая, что семантически

$$\exists x \in File(B(x)) = \bigvee_{x \in File} B(x); \quad \forall x \in File(B(x)) = \mathop{\&}_{x \in File} B(x)$$

- реализация оператора «*b:=∃x∈File(B(x))*»:

```
b=false; file.open();
while(!b && !file.eof())
{
   file.read(x);
   b=B(x);
}
```

- реализация оператора «*b:=∀x∈File(B(x))*»:

```
b=true; file.open();
while(b && !file.eof())
{
   file.read(x);
   b=B(x);
}
```

В итоге получаем программу:

```
void QueryZadacha3()
{
  bool aT, eR;
  this.Text = " Запрос, содержащий подзапросы (Record)";
  myBaseDataSet.Zadacha3.Clear();
  foreach(myBaseDataSet.POKUPATELIRow pRow in
     myBaseDataSet.POKUPATELI.Rows)
  {
    if(pRow["Gorod"].ToString() == "Казань")
    {
      aT = true; //любой товар
      foreach(myBaseDataSet.TOVARYRow tRow in
          myBaseDataSet.TOVARY.Rows)
      {
         aT = (Convert.ToInt32(tRow["Zena"]) <
               Convert.ToInt32(txtZena.Text));
         if(!aT)
         {
            eR = false;
            foreach(myBaseDataSet.RASXODRow rRow in
               myBaseDataSet.RASXOD.Rows)
            {
               eR = (Convert.ToDateTime(rRow["Data_Rash"]) <
dtpDateFrom.Value) && (rRow["Kod_Tovar"].ToString() ==
tRow["Kod_Tovar"].ToString()) &&
(rRow["Kod_Pokup"].ToString() == pRow["Kod_Pokup"].ToString());
               if (eR)
                  break;
             }
             aT = eR;
          }
          if (!aT)
              break;
       }
       if(aT)
       {
```

```
            myBaseDataSet.Zadacha3Row zRow =
               myBaseDataSet.Zadacha3.NewZadacha3Row();
            zRow["Pokup"] = pRow["Pokup"];
            myBaseDataSet.Zadacha3.Rows.Add(zRow);
          }
        }
      }
dataGridView1.Refresh();
}
```

СХЕМА ЗАДАЧИ-3 в заданиях на практике.

Имеется база данных типа: **A <->> C <<-> B**.

В заданиях используются выборки двух типов:

(1) Найти{x∈A}: ***B***1(x) **&** ∀y∈B

(***B***2(y) ⊃ ∃z∈C (***B***3(z) **&** (x,y,z-связаны по ключам)))

(2) Найти{x∈A}: ***B***1(x) & ∃y∈B

(***B***2(y) **&** ∀z∈C ((x,y,z-связаны по ключам) ⊃ ***B***3(z)))

Поскольку в *SQL* имеется только **&**, ∨ (**and**, **or**) и квантор ∃ *(EXISTS)*, устраним ⊃, F⊃G = (**not** F **or** G) , и квантор ∀, ∀u∈T F(u) = (**not** **EXISTS** u ∈ T **not** F(u)).

(1) Найти{x∈A}: ***B***1(x) **and** **not** **EXISTS** y∈B (***B***2(y) **and**

not **EXISTS** z∈C (***B***3(z) **and** (x,y,z-связаны по ключам)))

(2) Найти{x∈A}: ***B***1(x) **and** **EXISTS** y∈B (***B***2(y) **and**

not **EXISTS** z∈C ((x,y,z-связаны по ключам **and** **not**

B3(z))))

Тогда соответствующие *SELECT*-операторы будут иметь вид:

(1) SELECT * FROM A WHERE ***B***1 AND

NOT EXISTS (SELECT * FROM B WHERE ***B***2 AND

NOT EXISTS (SELECT * FROM C WHERE ***B***3 AND

```
                    (строки из A,B,C связаны по ключам)))
(2) SELECT * FROM A WHERE B1 AND
          EXISTS (SELECT * FROM B WHERE B2 AND
               NOT EXISTS (SELECT * FROM C WHERE NOT B3
                AND (строки из A,B,C связаны по ключам)))
```

9. Отчёты

На данный момент мы научились работать формами и основным элементом управления был **DataGridView**. Если мы хотим *вывести* информации на *печать* или перенести в табличный редактор, то нам понадобится новый инструмент - отчеты. Отчеты так же, как и формы состоят из объектов и сами являются объектами, но между отчетами и формами есть отличия:

1. Отчёты содержат только объекты только для отображения информации (Например, подписи, рисунки, текстовые поля, *геометрические фигуры* и линии), но не элементы управления приложением.
2. В отчёт выводит все записи из источника данных (таблицы, запроса или представления).
3. Отчёт требует наличия в системе принтера, так как настройки внешнего вида отчёта берутся из настроек драйвера принтера;
4. В отличие от форм отчёты состоят из пяти разделов:
   - **Заголовок** - верхняя часть первого листа отчёта. В заголовке располагают название отчёта и некоторую служебную информацию. Например, полное наименование и адрес организации, или имя автора отчёта.
   - **Подвал** - нижняя часть последнего листа отчёта. В примечание помещают итоговую информацию по отчёту и место для печати и подписи ответственных исполнителей.
   - **Верхний колонтитул** - верхняя часть каждого листа отчёта. В данный раздел помещают номера листов отчёта и дополнительную служебную информацию. Например, заголовки полей таблиц.
   - **Нижний колонтитул** - нижняя часть каждого листа отчёта. В данном разделе располагается та же информация что и верхнем колонтитуле, но не дублирует информацию из верхнего колонтитула. Например, под итоги по странице.
   - **Строки данных** - средняя часть каждого листа отчёта. Основная часть информации.

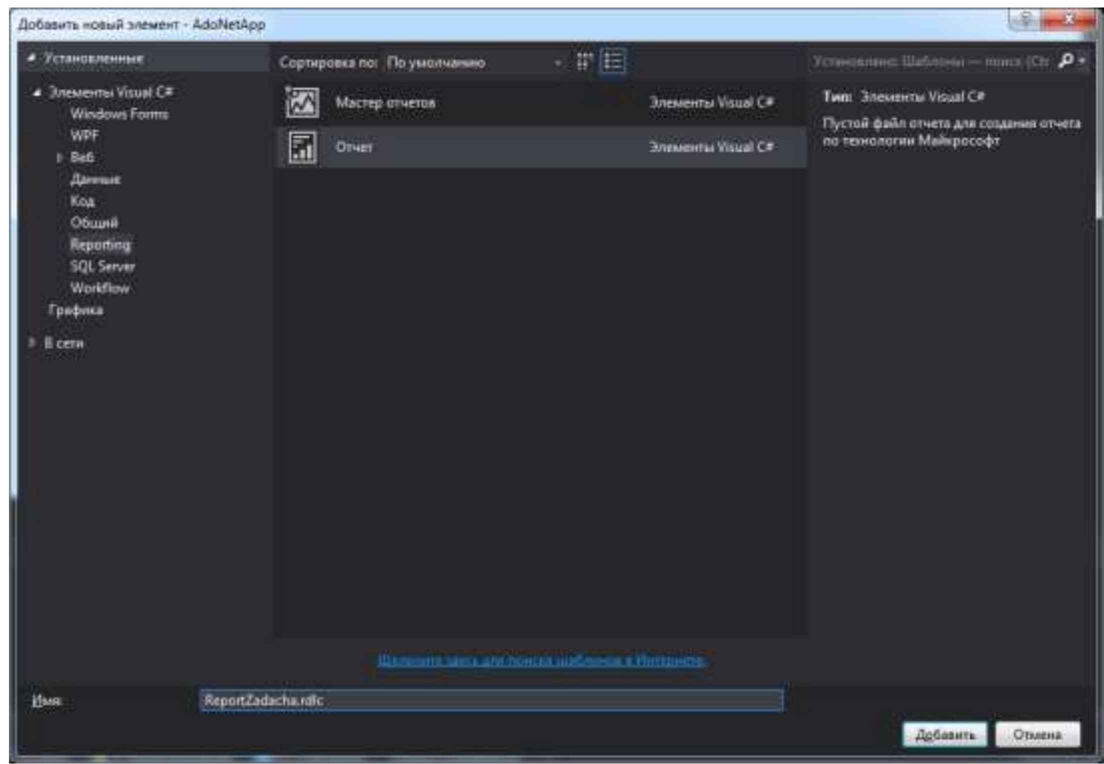
Существует несколько видов дизайна отчетов:

- **Ленточный дизайн** – выводит информацию по каждой записи отдельно. То есть для каждого поля каждой записи отображается название поля и его значение;
- **Списочный дизайн** - выводит информацию в виде списка. То есть информация о каждой записи отображается в отдельном разделе. Элементы идут последовательно по вертикали.
- **Табличный дизайн** - выводит информацию в виде таблицы. То есть в заголовок отчёта помещают названия полей, а в области данных под названием полей отображаются их значения.

Объекты для работы с отчётами

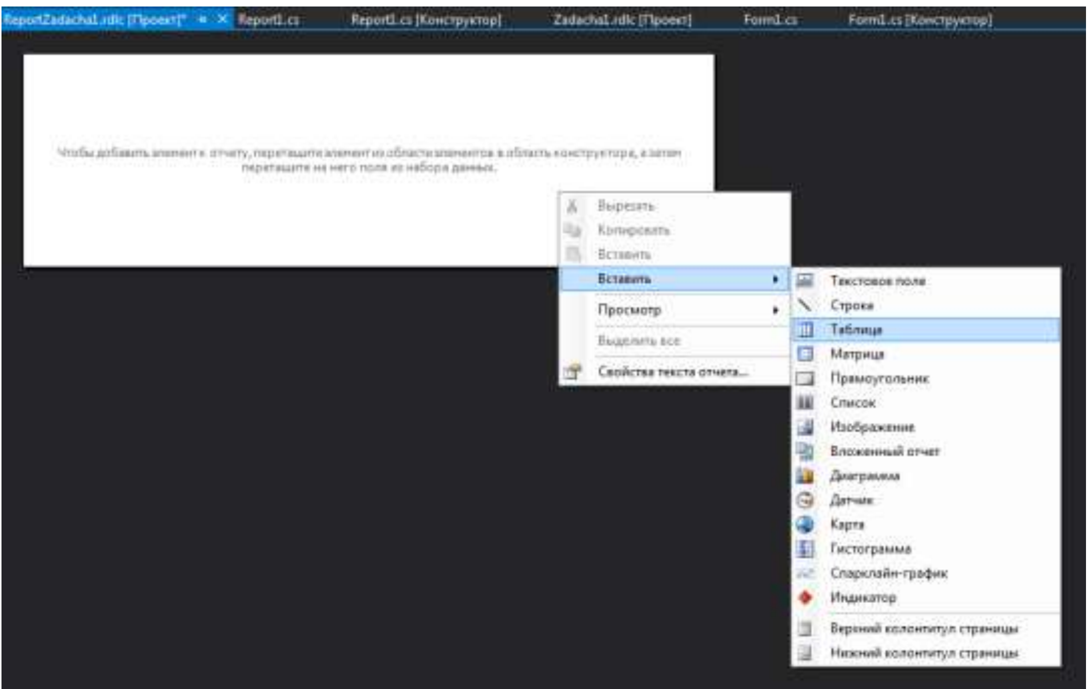
Для работы с отчётами в Visual Studio можно найти много инструментов различных производителей. Мы же остановимся стандартном элементе, включенном в поставку Visual Studio – **ReportViewer.**

Добавим в проект новый элемент **«Отчёт»** - он находится в разделе **Reporting.**

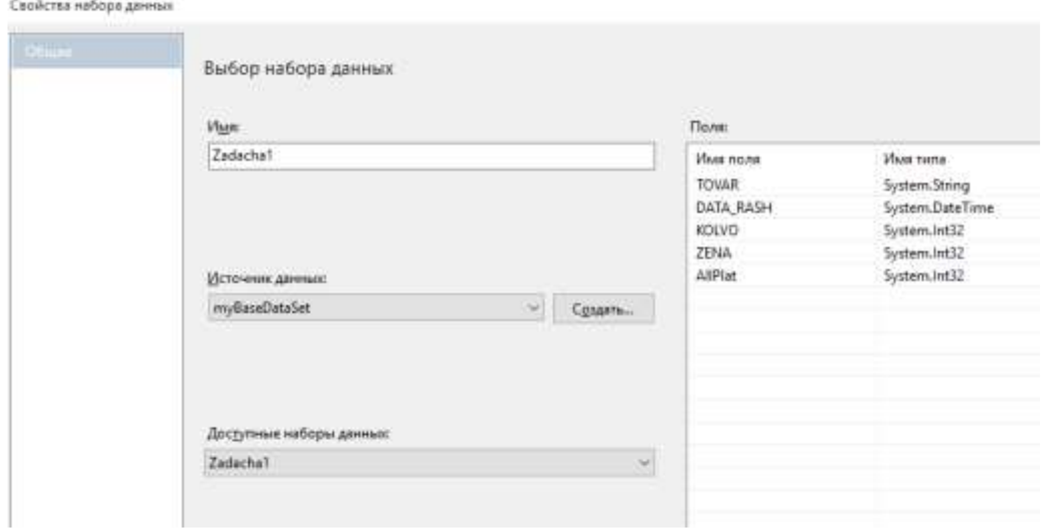


Если у вас нет такого раздела, то необходимо добавить **Microsoft.ReportViewer.WinForms**, а также все необходимые библиотеки.

В проекте создаётся пустой отчёт **ReportZadacha1.rdlc.** В отчёт поместим таблицу при помощи контекстного меню:

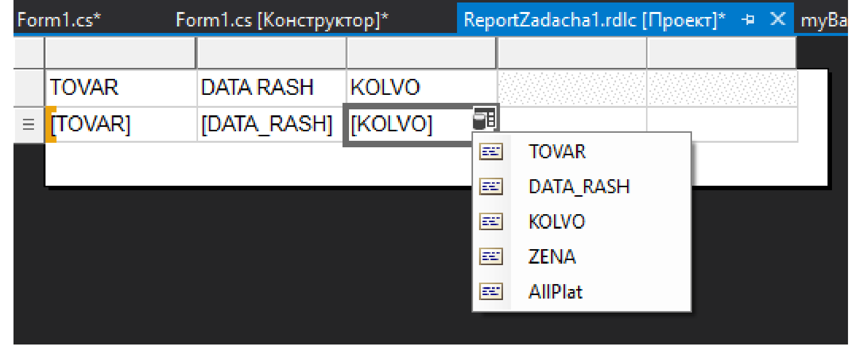


Далее выбираем источник данных и отображаемую таблицу или запрос – это будет набором данных, которые можно будет отобразить в отчёте.

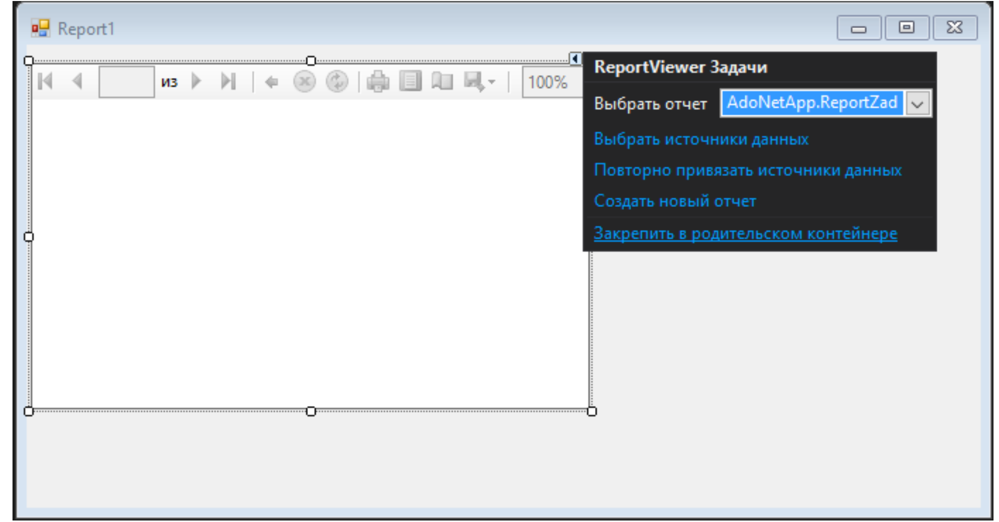


Добавляем в табликс колонки для всех полей запроса, и выбираем отображаемые данные. В верхний колонтитул автоматически

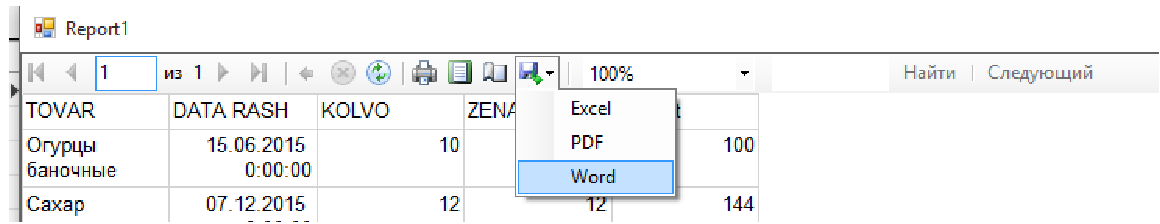
записываются метки с именами полей. В основной части – выражения вычисляющие значения этих полей.



Работа с конструктором отчета завершена. Теперь добавим его на форму нашего приложения.



Для этого создадим новую форму и поместим на неё элемент **ReportViewer** из раздела **Отчёты** панели инструментов. Выбираем отчёт из тех что есть в проекте, далее «Закрепляем в родительском контейнере». Получим форму с отчётом, в котором по умолчанию есть инструменты для работы с табличными документами: навигатор по страницам отчета, обновление данных из источника, настройка печати и печать, режим просмотра, экспорт данных в Excel, PDF, Word и поиск строки в данных ячеек.



Работа с объектами в отчёте полностью аналогична работе с объектами на форме. Мы можем выбирать либо поля из источника данных отчёта, либо можем создать объекты в отчёте вручную, а затем подключить их к полям через панель свойств. В отчётах все объекты делятся на объекты *контейнеры, объекты* для отображения данных и объекты оформления.

- ***Объекты контейнеры*** - это объекты, содержащие объекты для отображения данных и определяющие дизайн отчёта.
- ***Объекты для отображения данных*** - это объекты, отображающие значения полей источника данных, или дополнительную служебную информацию, или вычислимые выражения.
- ***Объекты оформления*** - объекты, применяемые только для оформления отчёта.

Рассмотрим объекты для отображения данных, к ним относятся:

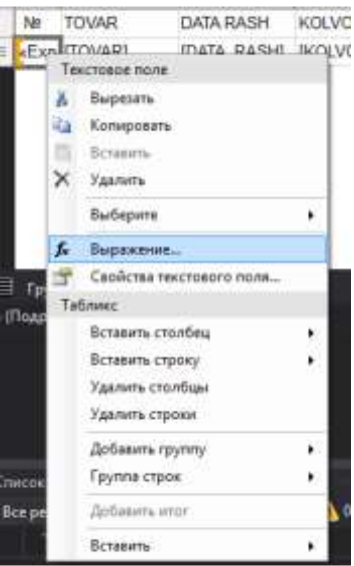
- **TextBox** - текстовое поле, предназначено для отображения значений полей и любой текстовой информации. Если объект **TextBox** используется для отображения информации из источника данных, и он находится вне объекта контейнера, то в нём будет отображено значение выбранного поля только первой записи из источника данных;
- **Image** - объект отображающий содержимое полей с графической информацией либо отображающий рисунки (графические файлы);
- **Chart** - объект, отображающий график или гистограмму, построенную по информации из источника данных.

Теперь рассмотрим объекты контейнеры. В отчёт можно поместить следующие объекты контейнеры:

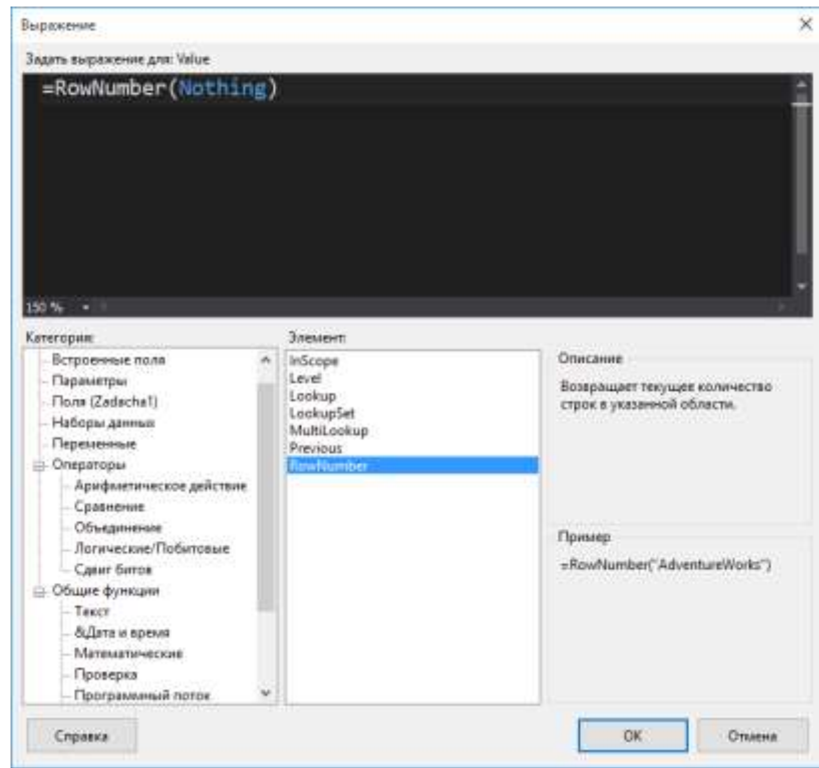
- **Table** - таблица выводит информацию в виде таблицы с ограниченным количеством столбцов и неограниченным количеством строк. То есть в количество строк в таблице зависит от объёма выводимых данных;
- **Matrix** - таблица выводит информацию в виде таблицы с неограниченным количеством столбцов и строк. То есть в количество строк и столбцов в таблице зависит от объёма выводимых данных;
- **List** - объект, выводящий информацию в виде списков;

- **Subreport** – объект, содержащий внутри себя дополнительный отчёт, созданный ранее.

Отдельно рассмотрим вариант заполнения ячеек таблиц вычислимым выражением. То есть когда нам недостаточно текстовых констант и значений полей. В контекстном меню ячейки выбираем пункт **«Выражения»**, откроется конструктор построителя выражений.



Здесь собираем нужное нам выражение, или пишем вручную. В данном случае мы выбрали раздел **Общие функции – Разное** – функция **RowNumber**.

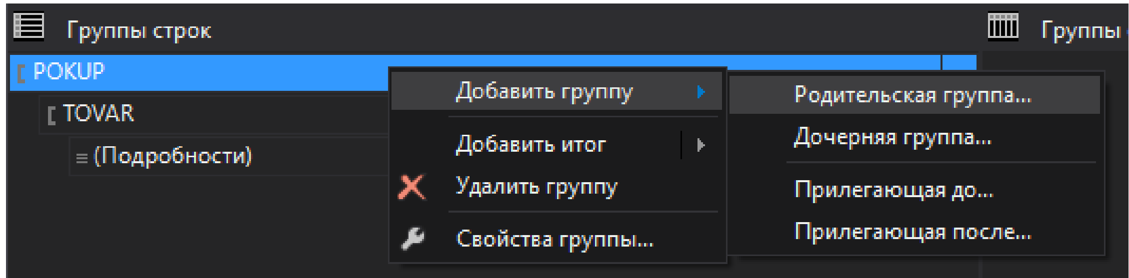


Эта функция возвращает номер текущей строки, если в параметрах передано ключевое слово **Nothing**. Или общее количество строк в источнике данных, с именем переданном в параметре. Теперь в нашем отчете в первой колонке будет отображаться номер строки.

Отчеты для задачи 2 и 3 формируются аналогично.

Рассмотрим отчет по представлению, здесь нам необходимо добавить группировки, причем несколько уровней – по товарам и по покупателям. Создаем новый отчет, выбираем источник данных – **TableAdapter** представления. Если его ещё нет в вашем **DataSet**, добавьте, как обычную таблицу, из базы данных.

Теперь настроим группировку:



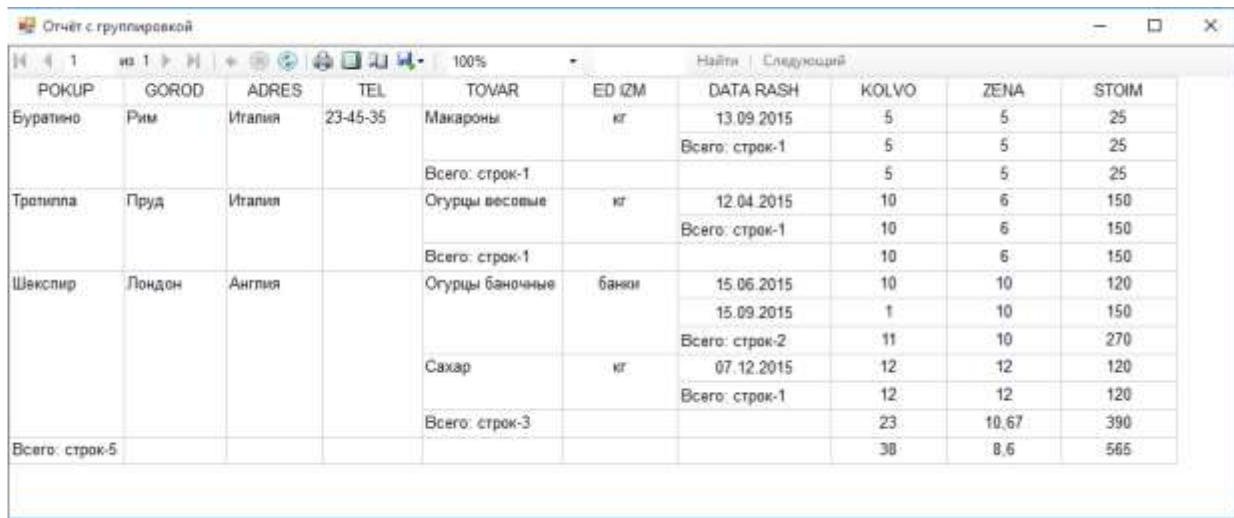
Сначала добавим родительскую группу по одному полю, затем дочернюю по второму. **«Подробности»** - это строки без группировок. Для каждой группы и строк мы также добавим итог «после». По умолчанию все числовые поля в итогах суммируются. Для поля «цена» эта агрегирующая функция нас не устраивает, заменим её на среднее – **Avg**. Также настроили формат выводимый даты: *FormatDateTime(Fields!DATA_RASH.Value, DateFormat.ShortDate)*.

POKUP	GOROD	ADRES	TEL	TOVAR	ED IZM	DATA RASH	KOLVO	ZENA	STOIM
[POKUP]	[GOROD]	[ADRES]	[TEL]	[TOVAR]	[ED_IZM]	«Expr»	[KOLVO]	[ZENA]	[STOIM]
						«Expr»	[Sum(KOLVO)]	«Expr»	[Sum(STOIM)]
				«Expr»			[Sum(KOLVO)]	«Expr»	[Sum(STOIM)]
«Expr»							[Sum(KOLVO)]	«Expr»	[Sum(STOIM)]

Нас также может интересовать количество строк в каждой группе. Выражение для вычисления количества:

"Всего: строк-" & CountRows(), где & - знак конкатенации строк.

Остается лишь запустить приложение и посмотреть, как выглядит отчет во время исполнения.



POKUP	GOROD	ADRES	TEL	TOVAR	ED IZM	DATA RASH	KOLVO	ZENA	STOIM
Буратино	Рим	Италия	23-45-35	Макароны	кг	13.09.2015	5	5	25
						Всего: строк-1	5	5	25
				Всего: строк-1			5	5	25
Тротилла	Пруд	Италия		Огурцы весовые	кг	12.04.2015	10	6	150
						Всего: строк-1	10	6	150
				Всего: строк-1			10	6	150
Шекспир	Лондон	Англия		Огурцы баночные	банки	15.06.2015	10	10	120
						15.09.2015	1	10	150
						Всего: строк-2	11	10	270
				Сахар	кг	07.12.2015	12	12	120
						Всего: строк-1	12	12	120
				Всего: строк-3			23	10,67	390
Всего: строк-5							38	8,6	565