Разработка приложений баз данных

Лабораторный практикум

ИГХТУ 2015

Министерство образования и науки Российской Федерации

Ивановский государственный химико-технологический университет

Разработка приложений баз данных

Лабораторный практикум

Иваново 2015

УДК 004.652

Авторы: Э.Г. Галиаскаров, А.Е. Хоченков, А.М. Кострома, А.П. Мицык.

Разработка приложений баз данных: лабораторный практикум / Э.Г. Галиаскаров и др.; Иван. гос. хим.-технол. ун-т. – Иваново, 2015. – 112 с.

Лабораторный практикум ставит себе целью познакомить студентов с практикой разработки приложений баз данных, привить навыки проектирования приложений на основе объектно-ориентированного подхода, реализации программного кода и графического интерфейса пользователя в рамках технологии WPF. В практикуме подробно и последовательно рассмотрены вопросы реализации инструментов работы с таблицами: добавление, редактирование и удаление записей, поиск записей по заданным критериям, выгрузки данных в заданном формате, формирования отчетности в текстовом и графическом виде. Реализация учебного проекта является образцом, которому студенты могут следовать при выполнении курсового проекта по проектированию баз данных.

Практикум предназначен для студентов направления «Информационные системы и технологии», занимающихся по дисциплине «Управление данными» и может быть полезен при курсовом и дипломном проектировании.

Печатается по решению редакционно-издательского совета Ивановского государственного химико-технологического университета

Рецензенты:

доктор технических наук, профессор А.Н. Лабутин (Ивановский государственный химико-технологический университет);

доктор технических наук, профессор В. Е. Мизонов (Ивановский государственный энергетический университет).

© ФГБОУ ВПО «Ивановский государственный химико-технологический университет», 2015

Содержание

Введение 6

Лабораторная работа №1
Знакомство с процессом разработки приложения VRA 10

1.1. Постановка задачи 10

1.2. Создание класса для хранения данных 10

1.3. Декларация операций 13

1.4. Реализация операций над объектом художника 14

1.5 Реализация пользовательского интерфейса 17

Контрольные вопросы и задания 23

Лабораторная работа №2
Настройка подключения и работа с базой данных 24

2.1. Постановка задачи 24

2.2. Объявление действий 24

2.3. Реализация действий 25

2.3.1. Метод Get(int id). Вариант 1 27

2.3.2. Метод Get(int id). Вариант 2 28

2.3.3. Метод GetAll() 29

2.3.4. Метод Add 29

2.3.5. Метод Delete 29

2.3.6. Метод Update 30

2.4. Конвертер объектов DtoConverter 30

2.5. Новый класс процесса – ArtistProcessDb 31

2.6. Настройка подключения к базе данных 32

Контрольные вопросы и задания 38

Лабораторная работа №3
Создаем главное окно приложения ... 39
3.1. Постановка задачи ... 39
3.2. Концепция формы ... 40
3.3. Создание формы ... 42
Контрольные вопросы и задания ... 50

Лабораторная работа №4
Перепроектирование базы данных. Добавление в проект работы с национальностями художников ... 52
4.1. Постановка задачи ... 52
4.2. Подготовка БД ... 52
4.3. Подготовка объекта DTO ... 53
4.4. Реализация “бизнес-требований” ... 53
4.5. Подготовка класса BaseDao ... 54
4.6. Работа с NationDao ... 55
4.7. Реализация “бизнес-требований” 2 ... 58
4.8. Реализация графического интерфейса ... 59
4.9. Изменение в классе Artist ... 59
Контрольные вопросы и задания ... 66

Лабораторная работа №5
Создаем средства работы с художественными произведениями, их покупки и продажи ... 67
5.1. Постановка задачи ... 67
5.2. Работа с Work ... 67
5.3. Работа с Transaction ... 69
5.4. Форма добавления и редактирования произведения ... 71
5.6. Окно «Транзакция» ... 74
Контрольные вопросы и задания ... 81

Лабораторная работа №6
Найдется все. Как организовать поиск и отбор 83
6.1. Постановка задачи 83
6.2. Реализация формы поиска 83
6.3. Поиск запросом к БД 84
6.4. Реализация формы поиска 85
6.5. Обмен данными между формами 87
6.6. Поиск на уровне приложения 88
Контрольные вопросы и задания 89

Лабораторная работа №7
Отчитайся. Выгрузка данных в формате Xlsx и Html 90
7.1. Постановка задачи 90
7.2. Реализация экспорта данных в таблицу Excel 90
7.3. Реализация отчета «Прайс-лист» 94
Контрольные вопросы и задания 98

Лабораторная работа №8
Сложная отчетность. Визуализация данных 99
8.1. Постановка задачи 99
8.2. Создаем окно визуальных отчетов 99
8.3. Создание структуры для работы со сложными отчетами 101
8.4. Реализация окна визуальных отчетов 106
Контрольные вопросы и задания 110

Библиографический список 111

Введение

В лабораторном практикуме будет описан один из вариантов процесса разработки программы. Студентам назначен проект «View Ridge Assistant» [1, 2]. Будем исходить из предположения, что вы знакомы с задачей этого проекта, и уже, по крайней мере частично, разработана база данных; что вы знакомы с синтаксисом языка C#[1] [3] и не боитесь аббревиатуры ООП[2]. В практикуме будут описаны основные моменты создания приложения. Остальное Вам придется делать самостоятельно. Написание программы, решение с помощью программирования каких-либо задач - это интересное времяпрепровождение. Мы уверены, что вы сможете представить на суд преподавателя свой вариант реализации поставленных задач. Прогресс не стоит на месте, и студенты – самая передовая группа.

Как вы могли уже убедиться при тестировании базы данных [1], достаточно уверенно работать с данными можно, используя интерфейс среды SQL Server Management Studio. Специалист, владеющий SQL языком, способен составлять самые разные запросы, добавляя, изменяя или получая нужную информацию. Включив воображение, можно предположить, что человек, далекий от информационных технологий, сумеет воспользоваться средой SQL Server Management Studio, используя ее графический интерфейс, чтобы вводить, менять и получать нужную информацию. Однако ясно, что в обычных случаях для работы с данными необходимо разработать графический интерфейс, ориентированный на простых пользователей.

Графический интерфейс разрабатывается с помощью тех или иных систем программирования, например, Visual Studio, и составляет часть, так называемых, приложений к базам данных. Естественно, помимо графического интерфейса такие приложения содержат логику обработки данных, соответствующую представлениям пользователей, так называемую бизнес-логику, а также средства доступа к данным. Способы разработки таких приложений составляют основу специализированных дисциплин, таких как, «Технология программирования», «Программная инженерия», «Проектирование информационных систем» и т.п. Методы и подходы к разработке могут быть весьма разнообразны.

В нашем случае можно остановиться на дата-центрированном подходе в разработке таких приложений [4]. При этом подходе каждой таблице или представлению базы данных сопоставляется табличная форма графического интерфейса. Для ввода или изменения записей разрабатывается отдельная форма ввода и редактирования. Для работы в табличной форме обычно предусматриваются инструменты навигации по записям, механизмы поиска, отбора и сортировки данных, экспорта данных, например, в формат csv, в

---

[1] C# (произносится си шарп) — объектно-ориентированный язык программирования.

[2] Объектно-ориентированное программирование.

таблицу Excel, в виде web-страницы и т.п. Во многом такой подход реализован в лабораторном практикуме и может быть использован при подготовке курсовой по дисциплине «Управления данными».

Такой подход хорош в небольших и достаточно простых с точки зрения функциональности проектах. В серьезных проектах все значительно сложнее, и разработка проекта начинается с тщательного изучения требований, выявления заинтересованных лиц, выявления групп пользователей, понимания их нужд и изучения возможных сценариев использования таких приложений для решения текущих задач.

Исходя из постановки задачи [1], вы – работник художественной галереи и вашей задачей является оформление новых поступлений и продажи произведений клиентам. Кто-то, в постановке не уточняется, фиксирует данные клиента при его обращении в галерею, а также фиксирует интересы клиентов в творчестве того или иного художника.

Конечно, информации для полного анализа недостаточно, но тут может помочь собственный опыт, сведения из других источников. Скорее всего, в галерее работает не так много людей: сотрудник галереи, владелец или директор галереи, мастера по производству рамок. Возможно, все эти роли будет совмещать один человек.

Теперь вспомним, какие требования были сформулированы на этапе анализа. Мы определили, что приложение «View Ridge Assistant» должно:

- обеспечивать учет покупателей (клиентов) и их художественных интересов;
- отслеживать приобретения, которые делает галерея;
- обеспечивать регистрацию покупок клиентов;
- обеспечивать ведение списка художников и произведений, когда-либо появлявшихся в галерее;
- отображать на веб-странице список произведений, выставленных на продажу;
- предоставлять отчетность о финансовой деятельности галереи.

Для наглядности отобразим наши требования в виде диаграммы вариантов использования [5], отражающих функциональную модель системы, и дадим их краткое описание:

1. ведение списка клиентов подразумевает ввод нового клиента, изменение данных об имеющемся клиенте, удаление клиента, а также поиск и отбор по списку клиентов;

2. ведение списка художников подразумевает ввод нового художника, изменение данных об имеющемся художнике, удаление художника, а также поиск и отбор по списку художников;

3. ведение списка произведений подразумевает ввод нового произведения, изменение данных об имеющемся произведении, поиск и отбор по списку произведений;

4. ведение интересов клиентов подразумевает установление и удаление связей между клиентом и художниками, творчеством которых он интересуется;

5. приобретение произведений предполагает указание даты и стоимости приобретения;

6. продажа произведений предполагает оформление сделки купли-продажи произведения клиентом с указанием даты и стоимости продажи;

7. формирование прейскуранта подразумевает получение списка картин, выставленных на продажу и цену их продажи;

8. получение отчетности о финансовой деятельности галереи предполагает построение графического или табличного отчета по заданным критериям и отображение сумм затрат и доходов галереи за период времени.

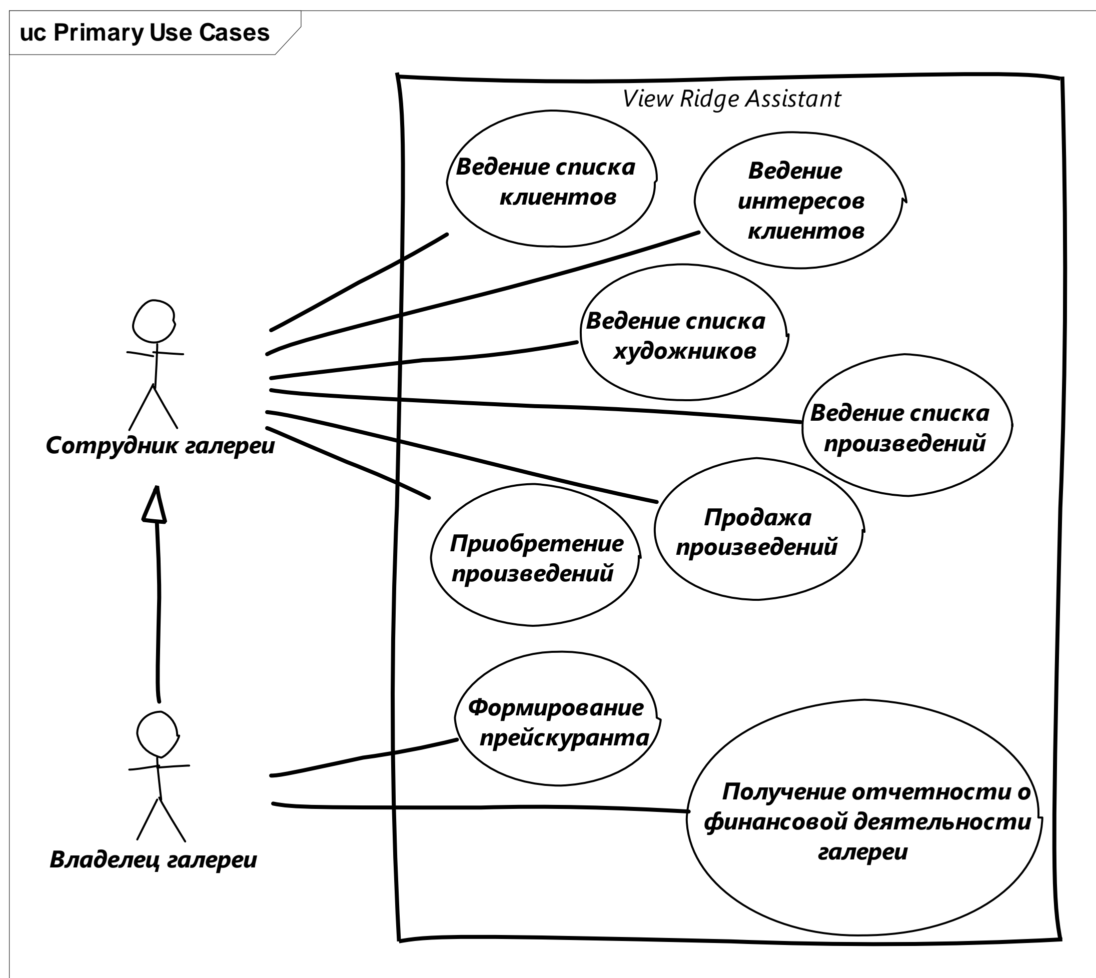

Диаграмма вариантов использования приложения «View Ridge Assistant»

Функциональность первых четырех вариантов использования (1-4) достаточно типичная для приложений к базам данных и предполагает наличие табличной формы, часто полностью аналогичной соответствующей таблице базы данных, и набор определенных действий над записями таблицы, часто оформленных как навигатор: добавление новой записи, изменение имеющейся записи, удаление записи, поиск записи, навигация по записям. Сюда также можно добавить такие типичные действия, как экспорт записей в тот или иной формат: xsl, xml, html, doc, txt и т.п. Поскольку действия над табличными

записями типичны, достаточно разработать обработку необходимых действий над одной таблицей, а остальные выполнить по аналогии.

Реализация вариантов использования приобретения и продажи произведений (5-6) в целом будет похожа на реализацию предыдущих вариантов использования, хотя и более сложная из-за необходимости учета взаимозависимости нескольких таблиц, поэтому этой задаче будет уделено отдельное внимание в рамках лабораторного практикума.

Также особое внимание будет уделено реализации сложных отчетов о финансовой деятельности галереи и визуализации ее результатов.

Помимо функциональных требований, которые в нашем случае представлены вариантами использования, на ход разработки приложений и на их структуру существенное влияние оказывают нефункциональные требования, которые отражают качественные стороны приложения. К таким требованиям следует отнести модифицируемость, производительность, простоту разработки, удобство использования и др. Именно эти требования имеют решающее значение при разработке архитектуры приложения [6].

В ходе выполнения лабораторного проекта будет продемонстрировано преимущество использования шаблонов проектирования. Вместе с тем, вы можете реализовать собственный подход и предложить свою структуру приложения. Также приветствуется использование иных технологий, облегчающих разработку ПО, например, Entity Framework.

Авторы выражают благодарность за помощь, полезные консультации и обсуждения, активное участие в становлении проекта и определяющее влияние на архитектуру приложения Андрею Евгеньевичу Блинчикову, главному инженеру ЗАО Сбербанк-Технологии, и Александру Юрьевичу Крылову, выпускнику кафедры информационных технологий ИГХТУ 2013 года.

Отдельную благодарность авторы выражают Екатерине Сергеевне Ганьшиной, студентке кафедры информационных технологий 2012 года поступления, за потрясающий дизайн обложки пособия.

Лабораторная работа №1
Знакомство с процессом разработки приложения VRA

1.1. Постановка задачи

Любой процесс разработки начинается с постановки задачи. Нам следует чётко понимать, что наша программа должна, а что не должна уметь делать. Написание программы по неясным, не окончательным задачам, или требованиям – дело неблагодарное и негативно сказывается на душевном спокойствии разработчика.

Набор описаний таких задач называется «бизнес-требования». Дословно это можно перевести как «требования бизнеса». Представители бизнеса являлись и остаются основными заказчиками для сферы IT. Таким образом, можно предположить, что термин «бизнес-требования» обозначает требования к ПО, написанные на языке понятном бизнесу, заказчику. Бизнес-требования (БТ) – это задокументированное однозначное описание последовательности операций или процесса. Сам же процесс, уже по аналогии, называется «бизнес-процессом».

Цель текущей работы – продемонстрировать один из вариантов разработки ПО на простом примере. В качестве примера выбор пал на объект Художник.

Список задач сформулируем следующим образом:

1. Объект Художник должен обладать информацией:
   - об имени;
   - о годах рождения и смерти;
   - о национальности.
2. У нас должна быть возможность:
   - получать список Художников;
   - добавлять Художников;
   - редактировать данные о Художнике;
   - удалять Художника.

Вот и все задачи на данный момент. Все очень просто и понятно. Перейдем к самой разработке.

1.2. Создание класса для хранения данных

Этим непонятным описанием называется в нашем случае класс – Художник. Наше будущее приложение должно выполнять какие-либо действия над этим объектом. Поскольку действия, совершаемые над объектом, мы ранее уже назвали «бизнес-процессом», то и сам объект с не меньшим удовольствием назовем «бизнес-объектом».

Данный объект позволяет в себе хранить данные из «бизнес-логики» для передачи их в часть программы, отображающую содержимое этого объекта, и обратно. Ведь мы знаем, что объект – это контейнер для данных.

Тут появляется и иное название этого объекта – объект для передачи

данных. На английском языке это пишется как Data Transfer Object, или сокращенно DTO [7].

С теорией пока закончим и создадим свой первый проект с единственным «бизнес-объектом» – Художником. Назовем его ArtistDto.

Для этого запускаем Visual Studio 2010 (далее студию или VS). Нажимаем Ctrl+Shift+N (или File → New → Project).

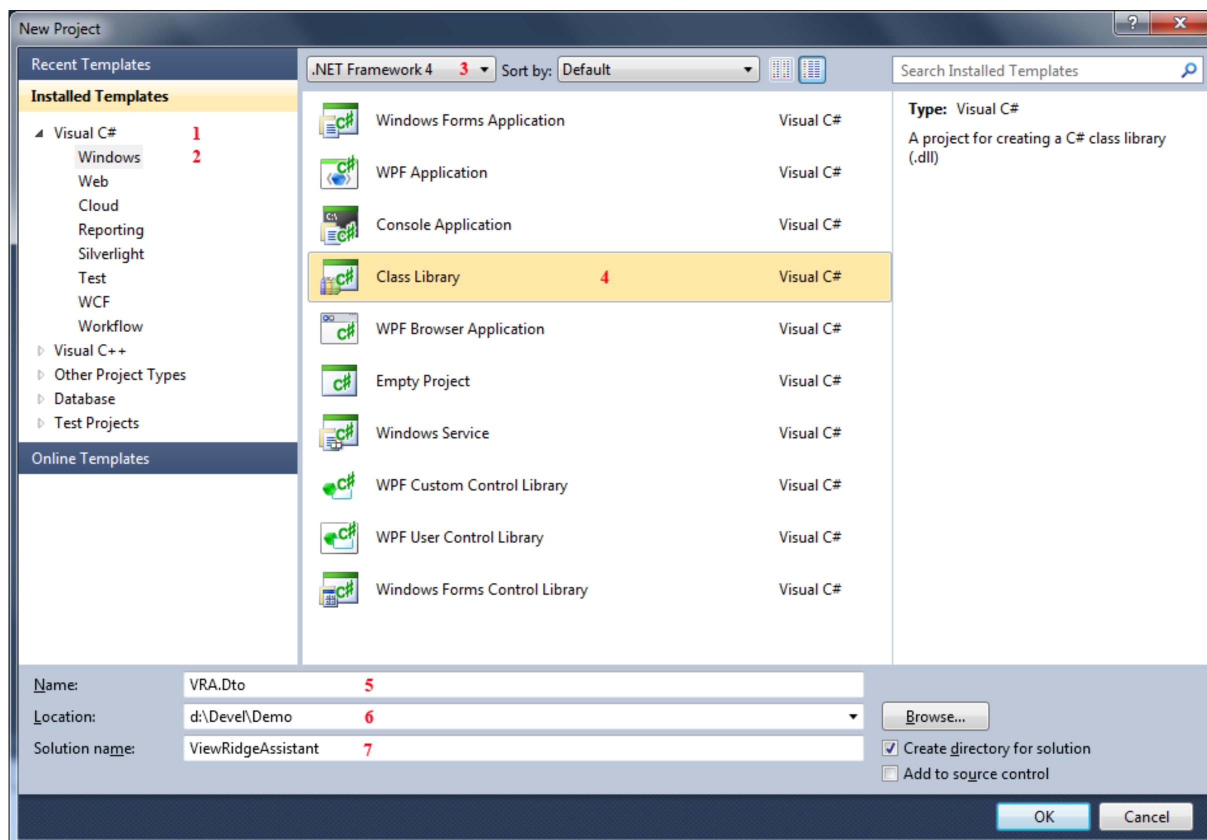

Рис. 1.1. Создание первого проекта

В разделе Installed Templates, выбираем Visual C# (1) \ Windows (2). В верхней части окна оставляем версию.NET Framework 4 (3). Выбираем тип проекта Class Library (4). Указываем имя проекта – VRA.Dto (5): название проекта ViewRidgeAssistant сократили до первых заглавных букв. Часть Dto как раз нам будет подсказывать, что там находятся наши «бизнес-объекты». Указываем место расположения (6) и имя решения (7). Должно получиться как на рис. 1.1. Далее нажимаем кнопку OK.

Среда создаст для нас проект VRA.Dto, в котором будет единственный файл `Class1`. Поскольку нам нужен класс ArtistDto, переименуем этот файл в `ArtistDto`, и название класса в `ArtistDto`.

Код нашего класса будет выглядеть так, как показано на рис. 1.2.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace VRA.Dto
{
	public class ArtistDto
	{
	}
}
```

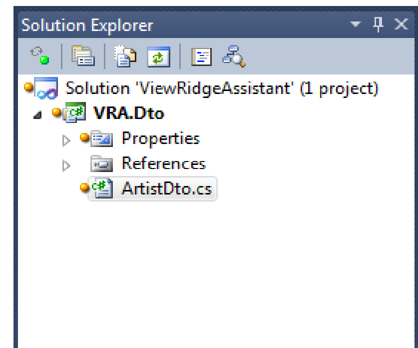

Рис. 1.2. Обозреватель решения

Уже полдела сделано – класс создан. Сделаем и вторую половину – добавим свойства этому классу согласно постановке задачи. Не забывайте писать комментарий к коду, это сделает его более понятным и значительно облегчит документирование проекта.

```
namespace VRA.Dto
{
	/// <summary>
	/// Класс - художник
	/// </summary>
	public class ArtistDto
	{
		/// <summary>
		/// id художника
		/// </summary>
		public int Id { get; set; }

		/// <summary>
		/// Имя
		/// </summary>
		public string Name { get; set; }

		/// <summary>
		/// Год рождения
		/// </summary>
		public int BirthYear { get; set; }

		/// <summary>
		/// Год смерти
		/// </summary>
		public int? DeceaseYear { get; set; }

		/// <summary>
		/// Национальность
		/// </summary>
		public string Nationality { get; set; }
	}
}
```

Примечание. Обнуляемые ('nullable') типы-значения, обозначаемые вопросительным знаком, например, int? (эквивалентно Nullable<int>) i = null, представляют собой те же самые типы-значения, способные принимать также значение null. Такие типы позволяют улучшить взаимодействие с базами данных через язык SQL.

Мы полностью выполнили первую из поставленных задач. Приступим к выполнению следующей задачи.

1.3. Декларация операций

Для начала мы опишем свои взгляды на функциональность. А потом их реализуем. Для этого создадим новый проект VRA.BusinessLayer типа – Библиотека классов, по аналогии с VRA.Dto, и добавим в него ссылку на VRA.Dto. Создадим интерфейс `IArtistProcess` в новом проекте. В нем мы и опишем те действия, которые должны будут выполняться над объектами Художник.

Примечание. В данном контексте интерфейс — это шаблон проектирования, являющийся общим методом для структурирования компьютерных программ для того, чтобы их было проще понять. В общем случае, интерфейс — это абстрактный тип, не содержащий данные или код, но определяющий поведение через объявление сигнатуры методов. Интерфейс обеспечивает программисту простой или более программно-специфический способ доступа к другим классам.

Новый интерфейс в новом проекте у Вас должен получиться примерно так, какпоказано на рис 1.3.

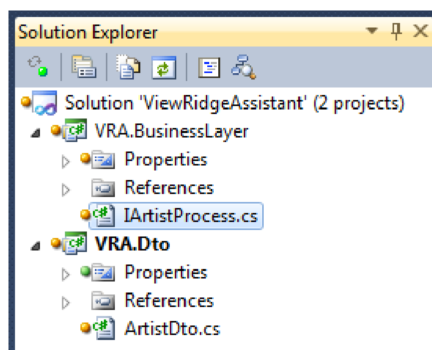

Рис. 1.3. Обозреватель решения

У нас получился следующий код интерфейса:

```
using System.Collections.Generic;
using VRA.Dto;

namespace VRA.BusinessLayer
{
	/// <summary>
	/// Декларация действий по работе с Художником
	/// </summary>
	public interface IArtistProcess
	{
		/// <summary>
		/// Возвращает список Художников
		/// </summary>
		/// <returns>список художников</returns>
		IList<ArtistDto> GetList();

		/// <summary>
		/// Возвращает художника по его id
		/// </summary>
		/// <param name=«id»>id художника</param>
```

```
        /// <returns>Художник</returns>
        ArtistDto Get(int id);

        /// <summary>
        /// Добавляет художника
        /// </summary>
        /// <param name=«artist»></param>
        void Add(ArtistDto artist);

        /// <summary>
        /// Обновляет данные о художнике
        /// </summary>
        /// <param name=«artist»>Художник, изменения которого надо сохранить</param>
        void Update(ArtistDto artist);

        /// <summary>
        /// Удаляет художника
        /// </summary>
        /// <param name=«id»>id художника, которого надо удалить</param>
        void Delete(int id);
    }
}
```

Тут описаны все необходимые операции, которые следует реализовать по заданию. Теперь напишем код, который будет выполнять эти действия.

1.4. Реализация операций над объектом художника

Поскольку мы еще не начали объединения нашего ПО с базой данных, которую вы уже должны иметь, то реализуем работу с художниками как можно проще и без использования базы. Как мы хотим это сделать? Подумайте пару минут, как бы вы решили эту задачу, а потом продолжайте читать.

Предположим, что вы честно подумали, у вас есть свои предположения, тогда данный абзац смело пропустите, так как вас посетили аналогичные, или еще более верные мысли. А мы поступим очень просто. Будем использовать самую обыкновенную коллекцию для хранения списка художников. В дальнейшем, работа с коллекцией будет заменена работой с базой. А для окна приложения, которое будет использовать реализацию написанного выше интерфейса `IArtistProcess`, ничего не изменится.

Если вы читаете данный абзац, то вы решили продолжить с нами реализацию этой задачи. В проект VRA.BusinessLayer добавим класс `ArtistProcess` и унаследуем его от интерфейса `IArtistProcess`. Реализуем все методы интерфейса заглушками (`throw new NotImplementedException();`). Это позволит собирать наше приложение без ошибок. У нас получился такой вариант:

```
using System;
using System.Collections.Generic;
using VRA.Dto;

namespace VRA.BusinessLayer
{
    public class ArtistProcess : IArtistProcess
    {
        public IList<ArtistDto> GetList()
        {
```

```
            throw new NotImplementedException();
        }

        public ArtistDto Get(int id)
        {
            throw new NotImplementedException();
        }

        public void Add(ArtistDto artist)
        {
            throw new NotImplementedException();
        }

        public void Update(ArtistDto artist)
        {
            throw new NotImplementedException();
        }

        public void Delete(int id)
        {
            throw new NotImplementedException();
        }
    }
}
```

Это и будет базовая реализация задачи по операциям с объектами Художника.

Для хранения списка художников будем использовать коллекцию `IDictionary`[3]. Ее использование позволит быстро получать доступ к объекту по его `id`. Добавим соответствующую строку в самое начало созданного класса. Коллекция будет статичной (`static`), что означает единственность экземпляра для всего приложения:

```
private static readonly IDictionary<int, ArtistDto> Artists
    = new Dictionary<int, ArtistDto>();
```

Ключевое слово `readonly` запрещает изменение значения переменной `Artists` после создания объекта `ArtistProcess`.

Реализуем методы. Начнем с `GetList()`. Его задача – вернуть список художников. Художники находятся в коллекции `Artists`, в свойстве `Values`. То есть, заменяя заглушку кодом реализации, получаем:

```
public IList<ArtistDto> GetList()
{
    return new List<ArtistDto>(Artists.Values);
}
```

Очень просто, в одну строчку уложились. А вот для следующего метода `Get(int id)` мы напишем немного больше кода. Нам надо убедиться, что объект находится в коллекции, и вернуть его. Иначе вернуть `null`.

```
public ArtistDto Get(int id)
{
    if (Artists.ContainsKey(id))
```

[3] Универсальная коллекция пар ключ/значение (http://msdn.microsoft.com/ru-ru/library/s4ys34ea.aspx).

```
            return Artists[id];
        else return null;
}
```

Если вы знакомы с инструкцией «?:»[4], то данная реализация сократится до одной строки:

```
public ArtistDto Get(int id)
{
        return Artists.ContainsKey(id) ? Artists[id] : null;
}
```

По аналогии реализуем операции `Update` и `Delete`:

```
public void Update(ArtistDto artist)
{
        if (Artists.ContainsKey(artist.Id))
            Artists[artist.Id] = artist;
}

public void Delete(int id)
{
        if (Artists.ContainsKey(id))
            Artists.Remove(id);
}
```

А вот код для добавления нового художника в данном случае займет немногим более. Попробуйте написать его самостоятельно и понять, почему ниже представлен именно этот вариант.

```
public void Add(ArtistDto artist)
{
        int max = Artists.Keys.Count == 0 ? 1 : Artists.Keys.Max(p => p) + 1;
        artist.Id = max;
        Artists.Add(max, artist);
}
```

Вот и все. Мы практически реализовали вторую задачу.

Нам стоит подумать, как другие компоненты приложения, которые мы еще не начали реализовывать, будут получать данный экземпляр реализации «бизнес-логики». Можно позволить напрямую создавать экземпляры класса `ArtistProcess`. Но так обычно не делают, потому что создание такого объекта может требовать дополнительную инициализацию, которую нет возможности поместить в конструктор. Не рекомендуется, чтобы другие модули знали конкретные классы реализации.

Если позже потребуется заменить реализацию интерфейса `IArtistProcess` с нынешнего варианта `ArtistProcess` на какой-либо иной, например, `ArtistProcessDb`, то замене будет подлежать весь код, который явно знает о классе `ArtistProcess`. Чтобы этого избежать, позволим сторонним модулям знать только об интерфейсе `IArtistProcss`. Также зададим единственное место, в котором все сторонние модули будут получать объект реализации интерфейса `IArtistProcess`. Это место называется фабрикой классов (паттерн[5]) [8].

---

[4] Терна́рная усло́вная опера́ция - операция, возвращающая свой второй или третий операнд в зависимости от значения логического значения, заданного первым операндом.

[5] **Шаблон проектирования** или **паттерн** (англ. *design pattern*) в разработке программного обеспечения — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Добавим в проект VRA.BusinessLayer класс `ProcessFactory`:

```
namespace VRA.BusinessLayer
{
	/// <summary>
	/// Фабрика классов бизнес-логики
	/// </summary>
	public class ProcessFactory
	{
		/// <summary>
		/// Возвращает объект, реализующий <seealso cref=«IArtistProcess»/>
		/// </summary>
		/// <returns></returns>
		public static IArtistProcess GetArtistProcess()
		{
			return new ArtistProcess();
		}
	}
}
```

Таким образом, создание экземпляров класса, реализующих интерфейс `IArtistProcess`, будет в одном месте. Мы всегда с легкостью поменяем класс реализации данного интерфейса на любой иной, и сторонние модули даже не узнают об этом.

А сейчас мы сделаем небольшое окошко в нашем приложении. Вы хотите видеть, как работает написанный нами код, как добавляются и удаляются художники? Мы хотим это увидеть и убедиться, что написанный выше код работает корректно. Вот это и будет наша следующая задача:

Обеспечить возможность использования реализованных нами ранее действий создания, изменения и удаления художников.

1.5 Реализация пользовательского интерфейса

Для выполнения вновь поставленной задачи добавим новый проект WPF Application и назовем его VRA. Добавим в этот проект ссылки на другие два проекта.

Примечание. Описание работы XAML-разметки для компонентов в WPF будет подробно описано в лабораторной работе №3.

Главным окном приложения (рис. 1.4) будет класс `MainWindow`. Поместим на окно компонент `DataGrid` (таблицу) и назовем его (`dgArtists`). Разместим в нижней части окна кнопку для добавления новых художников и назовем ее (`btnAdd`). При нажатии на эту кнопку, должно открываться диалоговое окно, в котором можно вводить данные о новом художнике. Также разместим в таблице соответствующие колонки.

Добавим в проект компонент типа Window (окно) и назовем его `AddArtistWindow`. Это будет то самое окно, в которое мы будем вводить данные о новом художнике. Поместите на окно все компоненты, необходимые для создания нового художника: текстовые поля для имени (`tbName`), годов рождения (`tbBirth`) и смерти (`tbDeath`); ниспадающий список для национальности (`cbNationality);` две кнопки: “Сохранить” (`btnSave`) и “Отмена” (`btnCancel`). Результат представлен на рис. 1.5.

Для начала надо научить форму открываться и закрываться. Откроем окно MainWindow в дизайнере VS2010 и кликнем дважды на кнопку “Добавить” (btnAdd). VS сгенерирует пустой обработчик события:

```
private void btnAdd_Click(object sender, RoutedEventArgs e)
{
}
```

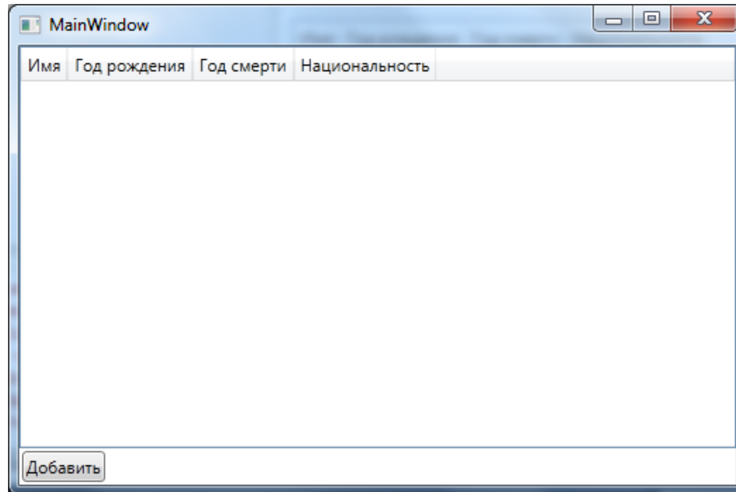

Рис. 1.4. Вид главного окна

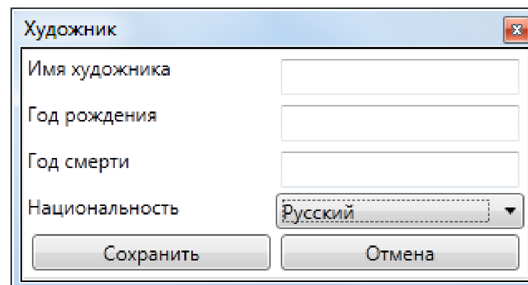

Рис. 1.5. Вид окна добавления художника

Добавим в него код для отображения окна добавления художника.

```
private void btnAdd_Click(object sender, RoutedEventArgs e)
{
	AddArtistWindow window = new AddArtistWindow();
	window.ShowDialog();
}
```

Откроем в дизайнере окно AddArtistWindow и дважды кликнем на кнопке “Отмена” (btnCancel). Впишем в сгенерированный пустой обработчик всего одно слово Close();

```
private void btnCancel_Click(object sender, RoutedEventArgs e)
{
	Close();
}
```

Теперь можно запустить приложение на выполнение. При нажатии на кнопку “Добавить”, откроется окно для добавления художника. При нажатии на кнопку “Отмена”, оно закроется. Попробуйте это выполнить.

Теперь приступим к операции сохранения из окна AddArtistWindow.

Для начала заполним ниспадающий список (cbNationality) списком

допустимых национальностей. Для этой цели мы создадим массив строк – национальностей, и передадим их компоненту (cbNationality).

```
/// <summary>
/// Список допустимых национальностей
/// </summary>
private static readonly string[] Nationalities =
            {«Русский», «Немец», «Испанец», «Итальянец»};
```

Для передачи этого списка компоненту (cbNationality) добавим две строки в конструктор AddArtistWindow() после вызова метода InitializeComponent();

```
public AddArtistWindow()
{
      InitializeComponent();
      //Передаем допустимые значения
      cbNationality.ItemsSource = Nationalities;

      //Задаем начальное значение
      cbNationality.SelectedIndex = 0;
}
```

Теперь список национальностей всегда будет иметь значения.

Создадим обработчик для кнопки “Сохранить” и перейдем в него. Visual Studio сгенерирует код:

```
private void btnSave_Click(object sender, RoutedEventArgs e)
{
}
```

Нам предстоит запрограммировать несколько проверок на введенные пользователем значения. Например, имя не должно быть пустым, или год смерти, если и указан, не должен быть меньше года рождения.

Проверка на не введённое имя художника:

```
if (string.IsNullOrEmpty(tbName.Text))
{
      MessageBox.Show(«Имя художника не должно быть пустым», «Проверка»);
      return;
}
```

По аналогии сделаем проверку на то, что годы должны быть целыми числами, год смерти не должен быть меньше года рождения. Выполните эти проверки самостоятельно, а потом вернитесь к документации.

У нас получился такой код:

```
int birth;
int? death = null;

if (string.IsNullOrEmpty(tbName.Text))
{
      MessageBox.Show(«Имя художника не должно быть пустым», «Проверка»);
      return;
}

if (!int.TryParse(tbBirth.Text, out birth))
{
      MessageBox.Show(«Год рождения должен быть целым числом», «Проверка»);
      return;
}
```

```
if (!string.IsNullOrEmpty(tbDeath.Text))
{
	int intDeath;

	if (!int.TryParse(tbDeath.Text, out intDeath))
	{
		MessageBox.Show(«Год смерти должен быть целым числом», «Проверка»);
		return;
	}

	if (intDeath < birth)
	{
		MessageBox.Show(«Год смерти должен быть больше года рождения»,
«Проверка»);
		return;
	}
	death = intDeath;
}
```

Если какая-либо проверка не сработает, программа сообщит пользователю о найденной ошибке.

Нам осталось сохранить верные данные о художнике и закрыть окно добавления. Именно тут мы впервые используем реализованную ранее задачу о добавлении художника. Допишем в этот же обработчик события следующий код:

```
//Создаем объект для передачи данных
ArtistDto artist = new ArtistDto();

//Заполняем объект данными
artist.Name = tbName.Text;
artist.BirthYear = birth;
artist.DeceaseYear = death;
artist.Nationality = cbNationality.SelectedItem.ToString();

//Именно тут запрашиваем реализованную ранее задачу по работе с художниками
IArtistProcess artistProcess = ProcessFactory.GetArtistProcess();

//Сохраняем художника
artistProcess.Add(artist);

//и закрываем форму
Close();
```

При успешном сохранении данных о художнике окно добавления закроется.

Мы научились добавлять художника. Но нам этого мало. Мы хотим еще и видеть, что именно мы добавили. Для этого вернемся в класс MainWindow, откроем обработчик btnAdd_Click. Допишем в его конец несколько строк кода:

```
//Получаем список художников
IList<ArtistDto> items = ProcessFactory.GetArtistProcess().GetList();

//Передаем этот список таблице на отображение
dgArtists.ItemsSource = items;
```

При желании можно сократить 2 строки кода в одну:

```
//Получаем список художников и передаем его на отображение таблице
```

```
dgArtists.ItemsSource = ProcessFactory.GetArtistProcess().GetList();
```

Запустите приложение, добавьте в него несколько художников. Наше главное окно теперь выглядит как на рис. 1.6:

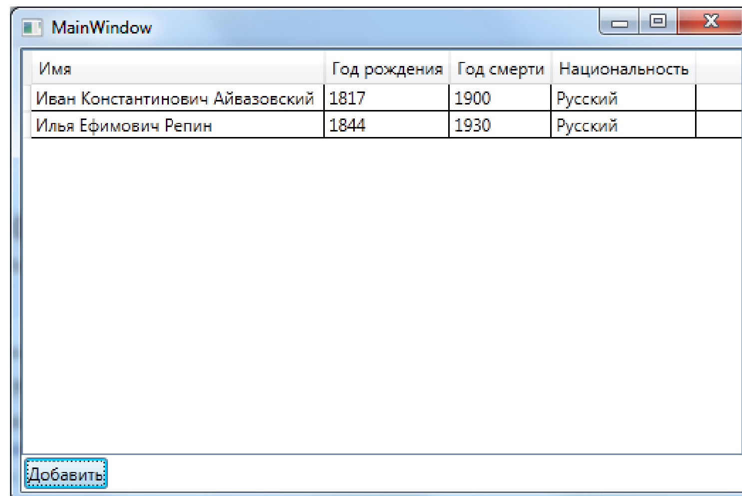

Рис. 1.6. Вид главного окна

На данный момент у нас реализована возможность добавления художника. После добавления загружается список всех художников в таблицу.

Добавим кнопку для перезагрузки данных на форму. Назовем ее (`btnRefresh`). Создадим обработчик, нажатия на кнопку `btnRefresh_Click` и поместим в него написанную выше строчку кода.

Добавим кнопку для удаления художника. Назовем ее (`btnDelete`). Создадим обработчик, нажатия на кнопку `btnDelete_Click` и поместим в него следующий код.

```
private void btnDelete_Click(object sender, RoutedEventArgs e)
{
	//Получаем выделенную строку с объектом художника
	ArtistDto item = dgArtists.SelectedItem as ArtistDto;

	//если там не художник или пользователь ничего не выбрал сообщаем об этом
	if (item == null)
	{
		MessageBox.Show(«Выберите запись для удаления», «Удаление художника»);
		return;
	}

	//Просим подтвердить удаление
	MessageBoxResult result = MessageBox.Show(«Удалить художника « + item.Name +
«?»,»Удаление художника», MessageBoxButton.YesNo, MessageBoxImage.Warning);

	//Если пользователь не подтвердил, выходим
	if (result != MessageBoxResult.Yes)
		return;

	//Если все проверки пройдены и подтверждение получено, удаляем художника
	ProcessFactory.GetArtistProcess().Delete(item.Id);

	//И перезагружаем список художников
	btnRefresh_Click(sender, e);
}
```

Теперь нам осталось сделать редактирование художника. Мы поступим хитро и будем использовать уже готовую форму AddArtistWindow для этих целей. Добавим кнопку для редактирования художника, назовем ее (btnEdit). В обработчике несколько первых строк кода будут такими же, как и в предыдущем случае за исключением текста сообщения.

```
//Получаем выделенную строку с объектом художника
ArtistDto item = dgArtists.SelectedItem as ArtistDto;

//если там не художник или пользователь ничего не выбрал сообщаем об этом
if (item == null)
{
	MessageBox.Show(«Выберите запись для редактирования», «Редактирование»);
	return;
}
```

Теперь надо передать художника для редактирования в объект AddArtistWindow. Для этого в классе AddArtistWindow добавим поле для хранения идентификатора художника.

```
/// <summary>
/// Поле хранит идентификатор художника
/// </summary>
private int _id;
```

Если это значение равно нулю, то мы будем добавлять художника. Если больше нуля, то редактируем его. Добавим метод для загрузки объекта художника на редактирование

```
/// <summary>
/// Метод загружает объект художника для редактирования
/// </summary>
/// <param name=«artist»>редактируемый объект художника</param>
public void Load(ArtistDto artist)
{
	//если объект не существует, или его национальность не в списке допустимых, 
выходим
	if (artist == null || !Nationalities.Contains(artist.Nationality))
		return;

	//сохраняем id художника
	_id = artist.Id;

	//заполняем визуальные компоненты для отображения данных
	tbName.Text = artist.Name;
	tbBirth.Text = artist.BirthYear.ToString();
	if (artist.DeceaseYear.HasValue)
		tbDeath.Text = artist.DeceaseYear.Value.ToString();
	cbNationality.SelectedItem = artist.Nationality;
}
```

Теперь изменим метод сохранения художника btnSave_Click класса AddArtistWindow. Изменим код сохранения объекта, чтобы можно было выполнять операции и обновления и сохранения. Для этого заменим строки:

```
//Сохраняем художника
artistProcess.Add(artist);
```

На:

```
//если это новый объект - сохраняем его
if (_id == 0)
{
	//Сохраняем художника
	artistProcess.Add(artist);
}
else //иначе обновляем
{
	//копируем обратно идентификатор объекта
	artist.Id = _id;

	//обновляем
	artistProcess.Update(artist);
}
```

Код для редактирования объекта написан. Осталось его вызвать. Для этого вернемся в метод `btnEdit_Click` класса `MainWindow`. Допишем в него следующие строки:

```
//Создаем окно
AddArtistWindow window = new AddArtistWindow();

//Передаем объект на редактирование
window.Load(item);

//Отображаем окно с данными
window.ShowDialog();

//Перезагружаем список объектов
btnRefresh_Click(sender, e);
```

На этом мы закончили осуществление задачи обеспечения возможности использования реализованных нами ранее действий создания, изменения и удаления художников. Приложение можно запустить и выполнить основные операции с художником.

Контрольные вопросы и задания

1. Что такое приложение баз данных?
2. Какую задачу играют бизнес-требования, что это такое?
3. Что такое Data Transfer Object, какую задачу он решает?
4. В чем преимущество деления приложения на отдельные модули? Каким образом это реализуется в данном решении?
5. Что такое обнуляемые типы-значения? В каких случаях целесообразно их применять?
6. Какую задачу играет так называемый бизнес-уровень или бизнес-слой?
7. Что такое интерфейс?
8. Чем удобно использование коллекции `IDictionary`?
9. Чем удобно использование паттерного проектирования?
10. Познакомьтесь с инструкцией «?:» (Терна́рная усло́вная опера́ция). Объясните, для чего ее используют.

Лабораторная работа №2
Настройка подключения и работа с базой данных

2.1. Постановка задачи

База данных является универсальным хранилищем данных с возможностью поиска. Именно в этом контексте мы и будем её рассматривать. Наша задача – реализовать редактирование списка художников. Мы будем использовать СУБД MS SQL Server 2008.

Добавим проект Vra.DataAccess типа Class Library. Первым делом создадим класс `Artist`, который будет отражать данные в базе, согласно таблице Artist. Поместим класс в папку Entities текущего проекта. Добавим поля в наш класс согласно полям в базе данных.

```
namespace Vra.DataAccess.Entities
{
	public class Artist
	{
		public int ArtistId;
		public string Name;
		public int BirthYear;
		public int? DeceaseYear;
		public string Nationality;
	}
}
```

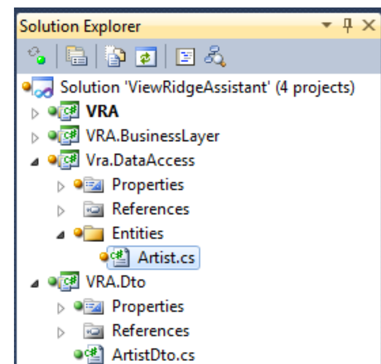

Рис. 2.1. Структура класса Artist

2.2. Объявление действий

Добавим в проекте Vra.DataAccess ссылку System.Configuration и интерфейс `IArtistDao` с описанием необходимых нам действий и строку: (`using Vra.DataAccess.Entities;`).

```
/// <summary>
/// Описание действий с объектом художника в базе
/// </summary>
public interface IArtistDao
{
	/// <summary>
	/// Получить художника по id
	/// </summary>
	/// <param name=«id»>id художника</param>
	/// <returns>художник</returns>
	Artist Get(int id);

	/// <summary>
	/// Получить список всех художников в базе
	/// </summary>
	/// <returns>список всех художников в базе</returns>
	IList<Artist> GetAll();

	/// <summary>
	/// Добавить художника в базу
	/// </summary>
	/// <param name=«artist»>новый художник</param>
```

```
        void Add(Artist artist);

        /// <summary>
        /// Обновить художника
        /// </summary>
        /// <param name=«artist»>обновленный художник</param>
        void Update(Artist artist);

        /// <summary>
        /// Удалить художника
        /// </summary>
        /// <param name=«id»>id удаляемого художника</param>
        void Delete(int id);
}
```

Как мы видим, всё те же элементарные операции над объектами.

2.3. Реализация действий

Добавим класс `ArtistDao`, реализующий созданный ранее интерфейс `IArtistDao` и строки:

```
using Vra.DataAccess.Entities;
using System.Configuration;
using System.Data.SqlClient;
```

Реализуем интерфейс заглушками по умолчанию. Dao – Data Access Object – объект для доступа к данным. Dao-объекты обеспечивают возможность работы с данными. В нашем случае класс `ArtistDao` позволит выполнять операции с объектами художников в базе.

```
public class ArtistDao : IArtistDao
{
        public Artist Get(int id)
        {
                throw new NotImplementedException();
        }

        public IList<Artist> GetAll()
        {
                throw new NotImplementedException();
        }

        public void Add(Artist artist)
        {
                throw new NotImplementedException();
        }

        public void Update(Artist artist)
        {
                throw new NotImplementedException();
        }

        public void Delete(int id)
        {
                throw new NotImplementedException();
        }
}
```

По аналогии с проектом VRA.BusinessLayer, создадим фабрику классов `DaoFactory`. Её будет использовать слой “бизнес-логики” вместо работы с

коллекцией.

```
public class DaoFactory
{
	public static IArtistDao GetArtistDao()
	{
		return new ArtistDao();
	}
}
```

Теперь следует сообщить программе о базе данных, с которой будем работать. Для этого добавим в проект VRA файл app.config со следующим содержимым.

```
<?xml version=«1.0» encoding=«utf-8» ?>
<configuration>
	<connectionStrings>
		<add name=«vradb»
			connectionString=«Data Source=PC\SQL2008R2;
				Initial Catalog=vra;
				User ID=vrauser;Password=123456»
			 providerName=«System.Data.SqlClient»/>
	</connectionStrings>
</configuration>
```

Значение атрибута `connectionString` – это и есть строка подключения к базе данных.

Для Вас адрес сервера (Data Source), база данных (Initial Catalog), имя пользователя (User ID) и пароль (Password) могут быть иными. Однако значение атрибута `name` следует оставить неизменным. Оно нам пригодится.

Вернемся к исходному коду класса `ArtistDao`. Есть два объекта, которые нам понадобятся в каждом из методов этого класса. Первый объект – это строка подключения, чтобы знать, к какой базе подключаться. Второй – сам объект подключения к базе данных. Добавим статичные методы, возвращающие эти объекты, в конец класса `ArtistDao`.

```
/// <summary>
/// Возвращает строку подключения к базе
/// </summary>
/// <returns></returns>
private static string GetConnectionString()
{
	return ConfigurationManager.ConnectionStrings[«vradb»].ConnectionString;
}

/// <summary>
/// Возвращает объект подключения к базе
/// </summary>
/// <returns></returns>
private static SqlConnection GetConnection()
{
	return new SqlConnection(GetConnectionString());
}
```

В первом методе мы используем значение атрибута `name` из файла app.config. Если Вы пожелаете изменить это значение, его следует изменить в двух местах.

Перейдём к реализации первого метода `Artist Get(int id)`. В этом

методе надо подключиться к базе данных, запросить запись в таблице ARTIST по указанному id, если таковая имеется, создать объект класса Artist и заполнить поля объекта, соответствующими значениями из базы данных. Попробуйте написать код этого метода самостоятельно, прежде чем перейти к предложенным вариантам его реализации.

2.3.1. Метод Get(int id). Вариант 1

Будем считать, что вы уже написали свой собственный вариант реализации, а вот какой код получился у нас:

```
public Artist Get(int id)
{
      //Получаем объект подключения к базе
      using (var conn = GetConnection())
      {
            //Открываем соединение
            conn.Open();
            //Создаем sql команду
            using (var cmd = conn.CreateCommand())
            {
                  //Задаём текст команды
                  cmd.CommandText = «SELECT ArtistID, Name, BirthYear, DeceaseYear,
                                     Nationality FROM ARTIST WHERE ArtistID = @id»;
                  //Добавляем значение параметра
                  cmd.Parameters.AddWithValue(«@id», id);
                  //Открываем SqlDataReader для чтения полученных в результате
                   //выполнения запроса данных
                  using (var dataReader = cmd.ExecuteReader())
                  {
                        //Если есть запись, то работаем с ней
                        if (dataReader.Read())
                        {
                            //Создаём пустой объект
                            Artist artist = new Artist();
                            //Заполняем поля объекта в соответствии с названиями
                            //полей результирующего набора данных
                             artist.ArtistId =
                                  dataReader.GetInt32(
                                       dataReader.GetOrdinal(«ArtistID»));
                             artist.BirthYear =
                                     Convert.ToInt32(dataReader[«BirthYear»]);
                            //Помните, что у нас поле DeceaseYear может иметь
                            //значение NULL?
                            //Следующие 3 строки корректно обрабатывают этот случай
                            object decease = dataReader[«DeceaseYear»];
                            if (decease != DBNull.Value)
                                      artist.DeceaseYear = Convert.ToInt32(decease);
                              artist.Name =
                                     dataReader.GetString(
                                         dataReader.GetOrdinal(«Name»));
                              artist.Nationality =
                                     dataReader.GetString(
                                         dataReader.GetOrdinal(«Nationality»));
                            return artist;
                        }
                        return null;
                  }
            }
```

```
        }
}
```

В общем, ничего сложного, хотя довольно много кода. Но это единственное место, где его будет так много. Да и то его можно упростить. У нас также есть метод `GetAll()`, который должен будет возвращать коллекцию объектов. И в нём также надо будет создавать объекты `Artist` из данных, полученных из базы. Давайте вынесем операцию создания объекта в новый метод `LoadArtist`, чтобы этот код можно было использовать более одного раза.

```
private static Artist LoadArtist(SqlDataReader reader)
{
        //Создаём пустой объект
        Artist artist = new Artist();
        //Заполняем поля объекта в соответствии с названиями полей результирующего
         // набора данных
        artist.ArtistId = reader.GetInt32(reader.GetOrdinal(«ArtistID»));
        artist.BirthYear = Convert.ToInt32(reader[«BirthYear»]);
        //Помните, что у нас поле DeceaseYear может иметь значение NULL?
        //Следующие 3 строки корректно обрабатывают этот случай
        object decease = reader[«DeceaseYear»];
        if (decease != DBNull.Value)
                artist.DeceaseYear = Convert.ToInt32(decease);
        artist.Name = reader.GetString(reader.GetOrdinal(«Name»));
        artist.Nationality = reader.GetString(reader.GetOrdinal(«Nationality»));
        return artist;
}
```

2.3.2. Метод Get(int id). Вариант 2

Метод `Get(int id)` немного изменится для нового случая:

```
public Artist Get(int id)
{
        //Получаем объект подключения к базе
        using (var conn = GetConnection())
        {
                //Открываем соединение
                conn.Open();
                //Создаем sql команду
                using (var cmd = conn.CreateCommand())
                {
                        //Задаём текст команды
                        cmd.CommandText = «SELECT ArtistID, Name, BirthYear, DeceaseYear,
                                           Nationality FROM ARTIST WHERE ArtistID = @id»;
                        //Добавляем значение параметра
                        cmd.Parameters.AddWithValue(«@id», id);
                        //Открываем SqlDataReader для чтения полученных в результате
                         // выполнения запроса данных
                        using (var dataReader = cmd.ExecuteReader())
                        {
                                return !dataReader.Read() ? null : LoadArtist(dataReader);
                        }
                }
        }
}
```

Код стал короче и выглядит уже не так пугающе. Не так ли?

Предлагаем Вам реализовать оставшиеся методы самостоятельно. Пока вы это делаете, мы напишем свой вариант.

2.3.3. Метод GetAll()

```
public IList<Artist> GetAll()
{
	IList<Artist> artists = new List<Artist>();
	using (var conn = GetConnection())
	{
		conn.Open();
		using (var cmd = conn.CreateCommand())
		{
			cmd.CommandText = «SELECT ArtistID, Name, BirthYear, DeceaseYear,
						Nationality FROM ARTIST»;
			using (var dataReader = cmd.ExecuteReader())
			{
				while (dataReader.Read())
				{
					artists.Add(LoadArtist(dataReader));
				}
			}
		}
	}
	return artists;
}
```

2.3.4. Метод Add

```
public void Add(Artist artist)
{
	using (var conn = GetConnection())
	{
		conn.Open();
		using (var cmd = conn.CreateCommand())
		{
			cmd.CommandText =
			  «INSERT INTO ARTIST (Name,BirthYear,DeceaseYear,Nationality)
			  VALUES (@Name,@BirthYear,@DeceaseYear,@Nationality)»;
			cmd.Parameters.AddWithValue(«@Name», artist.Name);
			cmd.Parameters.AddWithValue(«@BirthYear», artist.BirthYear);
			cmd.Parameters.AddWithValue(«@Nationality», artist.Nationality);
			object decease = artist.DeceaseYear.HasValue ?
				          (object)artist.DeceaseYear.Value : DBNull.Value;
			cmd.Parameters.AddWithValue(«@DeceaseYear», decease);
			cmd.ExecuteNonQuery();
		}
	}
}
```

2.3.5. Метод Delete

```
public void Delete(int id)
{
	using (var conn = GetConnection())
	{
		conn.Open();
		using (var cmd = conn.CreateCommand())
		{
			cmd.CommandText = «DELETE FROM ARTIST WHERE ArtistID = @ID»;
			cmd.Parameters.AddWithValue(«@ID», id);
			cmd.ExecuteNonQuery();
		}
	}
```

```
}
```

2.3.6. Метод Update

```
public void Update(Artist artist)
{
	using (var conn = GetConnection())
	{
		conn.Open();
		using (var cmd = conn.CreateCommand())
		{
			cmd.CommandText = «UPDATE ARTIST
			                    SET Name = @Name,
			                        BirthYear = @BirthYear,
			                        DeceaseYear = @DeceaseYear,
			                        Nationality = @Nationality
			                    WHERE ArtistID = @ID»;
			cmd.Parameters.AddWithValue(«@Name», artist.Name);
			cmd.Parameters.AddWithValue(«@BirthYear», artist.BirthYear);
			cmd.Parameters.AddWithValue(«@ID», artist.ArtistId);
			cmd.Parameters.AddWithValue(«@Nationality», artist.Nationality);
			object decease = artist.DeceaseYear.HasValue ?
			                  (object)artist.DeceaseYear.Value : DBNull.Value;
			cmd.Parameters.AddWithValue(«@DeceaseYear», decease);
			cmd.ExecuteNonQuery();
		}
	}
}
```

Реализация класса ArtistDto закончена. Теперь нам следует использовать только что написанный класс в проекте VRA.BusinessLayer.

Поскольку класс ArtistProcess уже существует, мы его оставим и создадим новый в том же проекте – ArtistProcessDb. Унаследуем его от интерфейса IArtistProcess.

Теперь у нас всплыла довольно интересная задача. Нам надо как-то преобразовывать объекты типа Artist в объекты типа ArtistDto, и обратно. Чтобы не делать это каждый раз, где только потребуется, добавим специальный класс DtoConverter в проекте VRA.BusinessLayer и поместим его в папку Converters этого проекта.

Примечание. Для корректной работы приложения необходимо добавить в проект VRA.BusinessLayer ссылку на проект Vra.DataAccess.

Код класса DtoConverter довольно длинный, но куда проще, чем был ранее.

2.4. Конвертер объектов DtoConverter

```
using System.Collections.Generic;
using VRA.Dto;
using Vra.DataAccess.Entities;

namespace VRA.BusinessLayer.Converters
{
	public class DtoConverter
	{
		public static ArtistDto Convert(Artist artist)
		{
```

```
			if (artist == null)
				return null;
			ArtistDto artistDto = new ArtistDto();
			artistDto.Id = artist.ArtistId;
			artistDto.BirthYear = artist.BirthYear;
			artistDto.DeceaseYear = artist.DeceaseYear;
			artistDto.Name = artist.Name;
			artistDto.Nationality = artist.Nationality;
			return artistDto;
		}

		public static Artist Convert(ArtistDto artistDto)
		{
			if (artistDto == null)
				return null;
			Artist artist = new Artist();
			artist.ArtistId = artistDto.Id;
			artist.BirthYear = artistDto.BirthYear;
			artist.DeceaseYear = artistDto.DeceaseYear;
			artist.Name = artistDto.Name;
			artist.Nationality = artistDto.Nationality;
			return artist;
		}

		public static IList<ArtistDto> Convert(IList<Artist> artists)
		{
			if (artists == null)
				return null;
			IList<ArtistDto> artistDtos = new List<ArtistDto>();
			foreach (var artist in artists)
			{
				artistDtos.Add(Convert(artist));
			}
			return artistDtos;
		}
	}
}
```

Теперь вы увидите, каким простым стал код нового класса `ArtistProcessDb`.

2.5. Новый класс процесса – ArtistProcessDb

```
public class ArtistProcessDb : IArtistProcess
{
	private readonly IArtistDao _artistDao;

	public ArtistProcessDb()
	{
		//Получаем объект для работы с художниками в базе
		_artistDao = DaoFactory.GetArtistDao();
	}

	public IList<ArtistDto> GetList()
	{
		return DtoConverter.Convert(_artistDao.GetAll());
	}

	public ArtistDto Get(int id)
	{
		return DtoConverter.Convert(_artistDao.Get(id));
```

```
	}

	public void Add(ArtistDto artist)
	{
		_artistDao.Add(DtoConverter.Convert(artist));
	}

	public void Update(ArtistDto artist)
	{
		_artistDao.Update(DtoConverter.Convert(artist));
	}

	public void Delete(int id)
	{
		_artistDao.Delete(id);
	}
}
```

Несравненно меньше и проще, чем созданный ранее `ArtistProcess`.

Остался последний штрих в нашей работе – начать использовать новый класс `ArtistProcessDb`. Для этого мы сделаем правку всего в одном месте – в фабрике классов `ProcessFactory`. Заменим `ArtistProcess` на `ArtistProcessDb`. Должен получиться следующий код:

```
/// <summary>
/// Возвращает объект, реализующий <seealso cref=«IArtistProcess»/>
/// </summary>
/// <returns></returns>
public static IArtistProcess GetArtistProcess()
{
	return new ArtistProcessDb();
}
```

Если бы мы в самом начале не стали использовать эту фабрику классов, нам бы сейчас потребовалось изменить название класса, по крайней мере, в четырёх местах.

Теперь запускайте приложение. При условии, что база существует, Вы сможете проводить любые операции с художниками. Все данные будут сохранены в базе.

2.6. Настройка подключения к базе данных

Вы не задумывались, что будет, если развернуть полученное приложение на другом компьютере? Который имеет, скорее всего, имя отличное от вашего, на котором вы разрабатываете это приложение. Имя базы данных, пользователь и пароль – все может отличаться. И что в этом случае делать? Верно, нужно будет изменить строку подключения в файле app.config. Конечно, это можно сделать вручную, открыв файл в текстовом редакторе. Но, согласитесь, гораздо удобнее иметь некоторую настройку, позволяющую изменять параметры подключения непосредственно из приложения.

Решение такой задачи может быть разным. Вашему вниманию предлагается следующий вариант.

Архитектура решения подобна тому, что было сделано для Художника. Для этого нам понадобится класс уровня доступа к данным, работающего с

настройками, и класс формы xaml, заключающий в себе логику представления.

Начнем с формы, которая может иметь вид, показанный на рис.2.2.

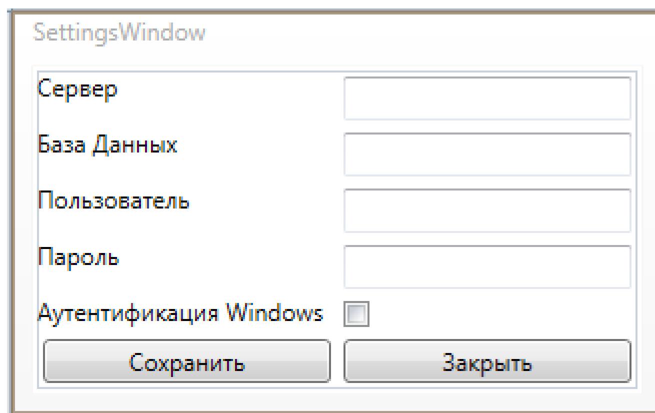

Рис.2.2. Вид формы подключения к базе данных

Опытные программисты при рисовании xaml-форм редко пользуются визуальным конструктором, а просто пишут код-разметку. Код окна будет выглядеть так:

```
<Window x:Class=«VRA.SettingsWindow»
        xmlns=«http://schemas.microsoft.com/winfx/2006/xaml/presentation»
        xmlns:x=«http://schemas.microsoft.com/winfx/2006/xaml»
        Title=«SettingsWindow» ResizeMode=«NoResize» SizeToContent=«WidthAndHeight»
WindowStartupLocation=«CenterOwner» WindowStyle=«ToolWindow» Closed=«Window_Closed»
Closing=«Window_Closing»>

    <Grid Margin=«2» HorizontalAlignment=«Stretch» VerticalAlignment=«Stretch»>

        <Grid.RowDefinitions>
            <RowDefinition Height=«Auto»/>
            <RowDefinition Height=«Auto»/>
            <RowDefinition Height=«Auto»/>
            <RowDefinition Height=«Auto»/>
            <RowDefinition Height=«Auto»/>
            <RowDefinition Height=«*»/>
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width=«150»/>
            <ColumnDefinition Width=«150»/>
        </Grid.ColumnDefinitions>

        <TextBlock Text=«Сервер» Grid.Row=«0» Grid.Column=«0»/>
        <TextBlock Text=«База Данных» Grid.Row=«1» Grid.Column=«0»/>
        <TextBlock Text=«Пользователь» Grid.Row=«2» Grid.Column=«0»/>
        <TextBlock Text=«Пароль» Grid.Row=«3» Grid.Column=«0»/>
        <TextBlock Text=«Аутентификация Windows» Grid.Row=«4» Grid.Column=«0»/>

        <TextBox Name=«tbServer» Grid.Row=«0» Grid.Column=«1» Margin=«3»/>
        <TextBox Name=«tbDataBase» Grid.Row=«1» Grid.Column=«1» Margin=«3»/>
        <TextBox Name=«tbUser» Grid.Row=«2» Grid.Column=«1» Margin=«3»/>
        <TextBox Name=«tbPassword» Grid.Row=«3» Grid.Column=«1» Margin=«3»/>

        <Button Name=«btnSave» Grid.Row=«5» Grid.Column=«0» Content=«Сохранить»
Margin=«3» Click=«btnSave_Click» />
        <Button Name=«btnCancel» Grid.Row=«5» Grid.Column=«1» Content=«Закрыть»
```

```
Margin=«3» Click=«btnCancel_Click» />

        <CheckBox Name=«cbAuth» Grid.Row=«4» Grid.Column=«1» Margin=«3»
Checked=«cbAuth_Checked» Unchecked=«cbAuth_Uncheked»/>

    </Grid>
</Window>
```

Представленный код требует небольшого пояснения.

Во-первых, стиль окна `WindowStyle=«ToolWindow»`. Впрочем, тут и так все ясно. Если нет, воспользуйтесь Интернетом, чтобы получить дополнительные сведения о стилях окон xaml.

Во-вторых, строчка `Closing=«Window_Closing»` – это метод, который срабатывает при событии закрытия формы. Для чего он нужен? Он реализует логику поведения в момент закрытия окна. Ни секундой раньше закрытия и ни секундой после закрытия, а именно, во время закрытия. Это требуется для сохранения настроек в случае какого-то непредвиденного сценария.

В-третьих, это события `Checked=«cbAuth_Checked»` и `Unchecked=«cbAuth_Uncheked»`. Как вы уже догадались – это методы, которые обрабатывают событие постановки птички в чекбокс.

Перейдем на уровень доступа к данным Vra.DataAccess. И там создадим интерфейс для считывания и загрузки настроек `ISettingsDao`:

```
public interface ISettingsDao
{
    string GetSettings();
    bool SetSettings(string server, string db, string user, string password);
}
```

Реализация класса `SettingsDao` будет такой:

```
using System.Configuration;
using System.Data.SqlClient;

namespace Vra.DataAccess
{
    public class SettingsDao : ISettingsDao
    {
        public string GetSettings()
        {
            try
            {
              return ConfigurationManager.ConnectionStrings[«vradb»].ConnectionString;
            }
            catch
            {
                return null;
            }

        }

        public bool SetSettings(string server, string db, string user, string password)
        {
            SqlConnectionStringBuilder conStr = new SqlConnectionStringBuilder();
            conStr.DataSource = server;
            conStr.InitialCatalog = db;
```

```
            if (user == ««)
            {
                conStr.IntegratedSecurity = true;
            }
            else
            {
                conStr.UserID = user;
                conStr.Password = password;
            }
            try
            {
                SqlConnection con = new SqlConnection(conStr.ConnectionString);
                con.Open();
            }
            catch
            {
                return false;
            }

            Configuration config =
ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None);
            config.ConnectionStrings.ConnectionStrings[«vradb»].ConnectionString =
conStr.ConnectionString;
            config.Save();
            ConfigurationManager.RefreshSection(«connectionStrings»);
            config = ConfigurationManager.OpenExeConfiguration(«VRA.exe»);
            config.ConnectionStrings.ConnectionStrings[«vradb»].ConnectionString =
conStr.ConnectionString;
            config.Save();
            ConfigurationManager.RefreshSection(«connectionStrings»);

            return true;
        }
    }
}
```

Примечание. («VRA.exe») это имя нашего приложения, если вы соблюдаете наши именования, то оно будет таким.

Добавим в класс DaoFactory реализацию SettingsDao:

```
public static SettingsDao GetSettingsDao()
{
    return new SettingsDao();
}
```

Теперь следует внести изменения на уровне “бизнес-логики”. Для этого переходим в проект VRA.BusinessLayer и добавляем:

Интерфейс ISettingsProcess:

```
namespace VRA.BusinessLayer
{
    public interface ISettingsProcess
    {
        string GetSettings();
        bool SetSettings(string server, string db, string user, string password);
    }
}
```

Класс SettingsProcess:

```
using Vra.DataAccess;
```

```
namespace VRA.BusinessLayer
{
    public class SettingsProcess : ISettingsProcess
    {
        private readonly ISettingsDao _settingsdao;

        public SettingsProcess()
        {
            _settingsdao = new SettingsDao();
        }

        public string GetSettings()
        {
            return _settingsdao.GetSettings();
        }

        public bool SetSettings(string server, string db, string user, string password)
        {
            return _settingsdao.SetSettings(server, db, user, password);
        }
    }
}
```

В классе ProcessFactory надо добавить новый метод:

```
public static ISettingsProcess GetSettingsProcess()
{
    return new SettingsProcess();
}
```

Теперь мы переходим на уровень VRA.

В логике управления формой SettingsWindow пишем следующее:

```
private bool recstatus;
private bool applySettings = false;

private void LoadSettings()
{
    if (ProcessFactory.GetSettingsProcess().GetSettings() != null)
    {
        SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
        builder.ConnectionString = ProcessFactory.GetSettingsProcess().GetSettings();

        this.tbServer.Text = builder.DataSource;
        this.tbDataBase.Text = builder.InitialCatalog;

        if (builder.IntegratedSecurity & !recstatus)
        {
            this.cbAuth.IsChecked = true;
        }
        else
        {
            this.tbPassword.Text = builder.Password;
            this.tbUser.Text = builder.UserID;
        }
    }
    else
    {
        this.tbServer.Text = «127.0.0.1»;
        this.tbDataBase.Text = «vra_db»;
        this.cbAuth.IsChecked = true;
    }
```

```
}

public SettingsWindow()
{
    InitializeComponent();
    LoadSettings();
}

private void btnCancel_Click(object sender, RoutedEventArgs e)
{
    this.Close();
}

private void cbAuth_Checked(object sender, RoutedEventArgs e)
{
    this.tbUser.Text = ««;
    this.tbPassword.Text = ««;
    this.tbUser.IsEnabled = false;
    this.tbPassword.IsEnabled = false;
}

private void cbAuth_Uncheked(object sender, RoutedEventArgs e)
{
    this.tbUser.IsEnabled = true;
    this.tbPassword.IsEnabled = true;
    this.recstatus = true;
    LoadSettings();
}

private void btnSave_Click(object sender, RoutedEventArgs e)
{
    if (ProcessFactory.GetSettingsProcess().SetSettings(this.tbServer.Text,
this.tbDataBase.Text, this.tbUser.Text, this.tbPassword.Text))
    {
        this.applySettings = true;
        this.Close();
    }
    else
    {
        MessageBox.Show(«Не удалось записать настройки! Что-то пошло не так»);
        return;
    }
}

private void Window_Closed(object sender, EventArgs e)
{
}

private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
    if (!this.applySettings)
    {
        MessageBoxResult result = MessageBox.Show(«Сохранить изменения?», «Изменение
настроек подключения», MessageBoxButton.YesNo, MessageBoxImage.Warning);

        if (result != MessageBoxResult.Yes) return;

        if (!ProcessFactory.GetSettingsProcess().SetSettings(this.tbServer.Text,
this.tbDataBase.Text, this.tbUser.Text, this.tbPassword.Text))
        {
            return;
```

```
        }
    }
}
```

При этом мы имеем поле `private bool recstatus;` на форме, которое подсказывает нам, аутентифицировался ли пользователь личным аккаунтом при предыдущих настройках.

Поле `private bool applySettings = false;` подсказывает форме, были ли уже приняты настройки при текущем их изменении до закрытия формы.

На форме MainWindow добавляем кнопку (`btnDatabase`) и присваиваем ей метод:

```
private void btnDatabase_Click(object sender, RoutedEventArgs e)
{
    SettingsWindow window = new SettingsWindow();
    window.ShowDialog();
}
```

На этом настройка подключения к базе данных завершена.

Теперь запускайте приложение, подключитесь к базе данных, попробуйте добавить новые записи и убедитесь, что приложение работает, как и ожидалось.

Контрольные вопросы и задания

1. С какой целью добавляем проект VRA.DataAccess?
2. Какова роль классов Entities?
3. Что такое DAO, какую задачу он играет в проекте?
4. Объясните структуру строки подключения.
5. Что такое фабрика классов? Какую задачу она играет?
6. Какие методы реализует класс `ArtistDao`, объясните алгоритм их работы?
7. С какой целью реализован метод `LoadArtist`?
8. Зачем и каким образом происходит преобразование объектов типа `Artist` в объекты типа `ArtistDto`?
9. Что такое стиль окна, и какие они бывают?
10. Что такое Binding?
11. Попробуйте построить модель классов полученного нами решения. Что у вас получилось? Используйте Enterprise Architect для получения модели по разработанному коду.

Лабораторная работа №3
Создаем главное окно приложения

3.1. Постановка задачи

В этой работе мы рассмотрим принцип создания главного окна приложения. Нам наиболее удачным показалось использование XAML[6] разметки [9]. Главное ее назначение – это конструирование графических пользовательских интерфейсов WPF[7].

Разработчики давно поняли, что создавать сложные графически насыщенные приложения намного проще, если отделить графическую часть от лежащего в основе кода. Таким образом, дизайнеры могут заниматься графикой, а разработчики – кодом.

Графический пользовательский интерфейс GUI[8] позволяет пользователю осуществлять взаимодействие с приложением. GUI может различаться по способу организации [10].

Однодокументный интерфейс (англ. Single document interface или SDI) — способ организации графического интерфейса приложений в отдельных окнах. Не существует «фонового» или «родительского» окна, содержащего меню или панели инструментов, по отношению к активному — каждое окно несёт в себе эти элементы. Такие приложения, позволяют редактировать более одного документа одновременно.

Многодокументный интерфейс (англ. Multiple document interface или MDI) — способ организации графического интерфейса пользователя, предполагающий использование оконного интерфейса, в котором большинство окон (исключая, как правило, только модальные[9] окна) расположены внутри одного общего окна. Этим он и отличается от SDI, в котором окна располагаются независимо друг от друга.

Вопрос о том, какой тип интерфейса предпочтителен — MDI или SDI — часто становится предметом обсуждений в сообществе разработчиков и пользователей программного обеспечения. SDI, в частности, более удобен

---

[6] XAML (Extensible Application Markup Language – расширяемый язык разметки приложений) представляет собой язык разметки, используемый для создания экземпляров разметки .Net.

[7] WPF (Windows Presentation Foundation) – система для построения клиентских приложений Windows с визуально привлекательными возможностями взаимодействия с пользователем, графическая (презентационная) подсистема в составе .NET Framework (начиная с версии 3.0), использующая язык XAML.

[8] Графи́ческий интерфе́йс по́льзователя, графический пользовательский интерфейс, ГИП (англ. Graphical user interface, GUI) — разновидность пользовательского интерфейса, в котором элементы интерфейса (меню, кнопки, значки, списки и т. п.), представленные пользователю на дисплее, исполнены в виде графических изображений.

[9] Модальным называется окно, которое блокирует работу пользователя с родительским приложением до тех пор, пока пользователь это окно не закроет. Модальными преимущественно реализованы диалоговые окна. Также модальные окна часто используются для привлечения внимания пользователя к важному событию или критической ситуации.

тогда, когда пользователь одновременно работает с несколькими приложениями разных типов. Разработчики широко используют оба типа интерфейса, а зачастую и интерфейс смешанного типа.

Мы же будем придерживаться использования однодокументного интерфейса, однако помимо различного рода модальных окон, нам придется добавлять окна для реализации различных операций, что указывает на преимущество интерфейса смешанного типа.

3.2. Концепция формы

Разработка дизайна формы – дело вкуса, но у всех форм есть нечто общее [11].

Изначально нам сложно представить, как будет выглядеть главное окно нашего приложения, да и в процессе разработки оно может измениться до неузнаваемости. Однако разработка дизайнов на XAML и WPF требует некоторой определенности с внешним видом еще до начала работ. Мы же пишем код в приложении по заранее продуманным алгоритмам – и тут примерно, то же самое. Обычно с формами много помогают заказчики, указывая как им удобно, а как нет. В конечном итоге именно удобство пользования и определяет, хорошо спроектирована форма или нет.

Итак, главное окно приложения будет выглядеть, как показано на рис. 3.1:

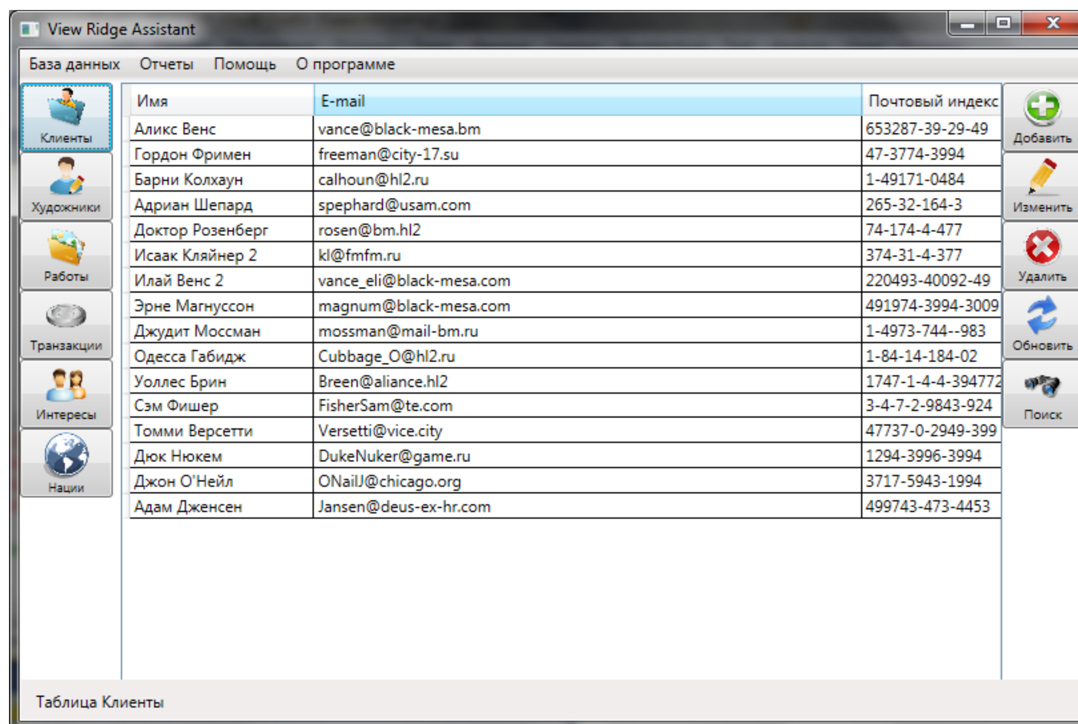

Рис. 3.1. Вид главного окна

У главной формы должны быть элементы, помогающие осуществлять навигацию по приложению, отображать и управлять данными.

Рассмотрим более подробно общую структуру главного окна приложения. Верх формы занимает командное меню.

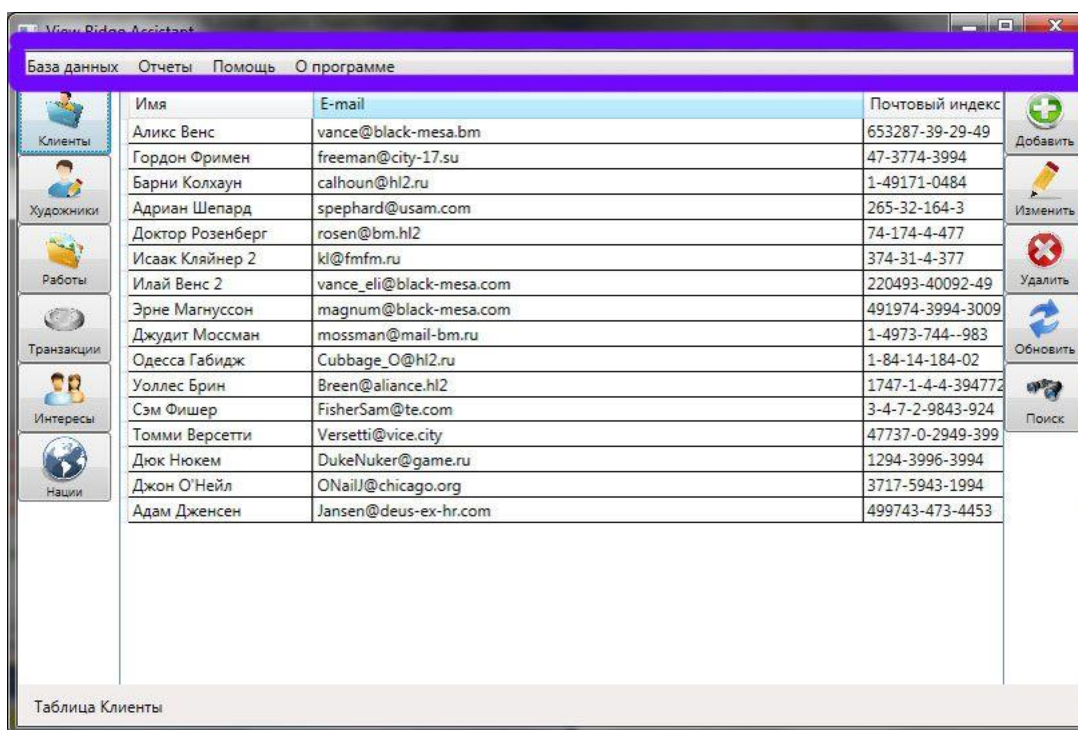

Рис. 3.2. Элемент “Меню” формы

Внизу формы располагается панель, на которой будем отображать для пользователя подсказку: с какой таблицей он сейчас работает.

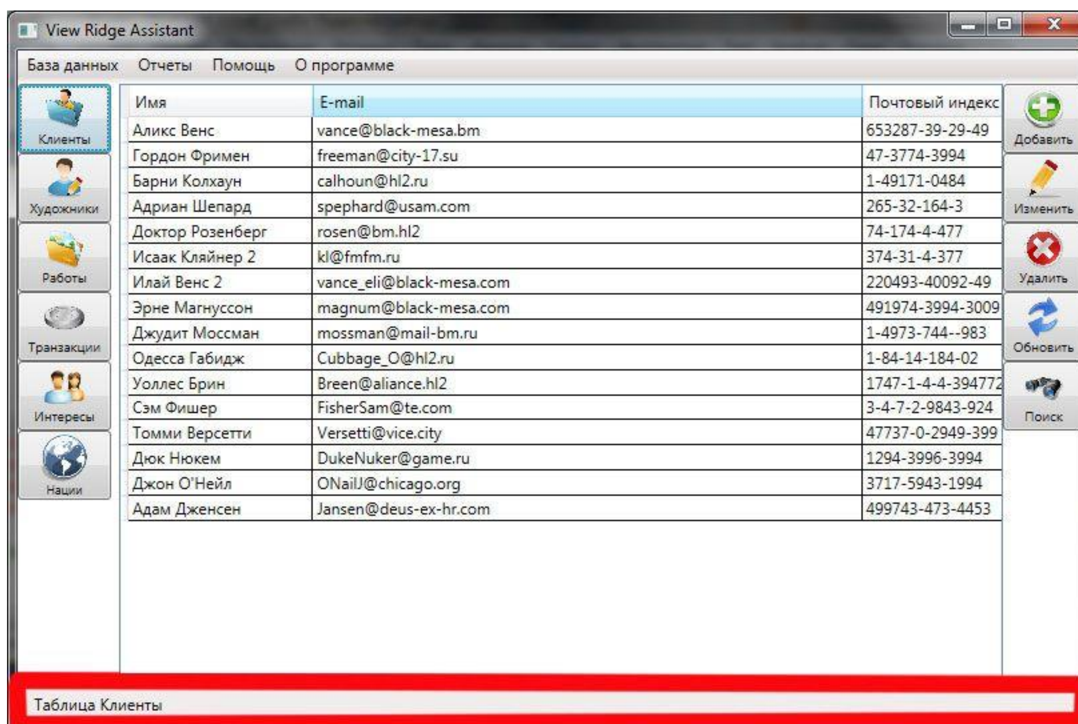

Рис. 3.3. Элемент “Панель-подсказка” формы

Слева и справа на форме расположены кнопки. Слева – кнопки навигации по объектам базы данных: таблицам, справа – кнопки действий над записями этих таблиц.

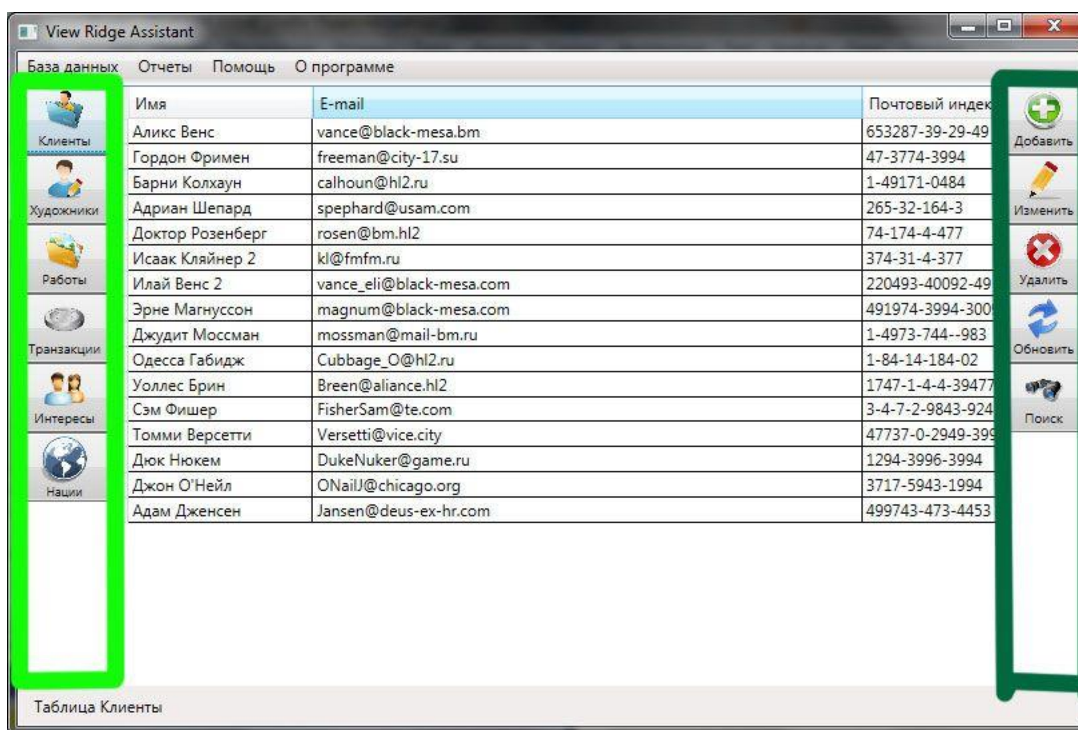

Рис. 3.4. Элементы “Кнопки” формы

По центру соответственно расположим объект типа DataGrid, в нем будет указана вся интересующая пользователя информация в виде списка, отображающего данные об объекте.

Поскольку вся суть WPF – это использование контейнеров, то наша первоочередная задача – настройка Grid (сетка) таким образом, чтобы у окна получалось пять упомянутых ранее зон.

3.3. Создание формы

Существует два способа разбить Grid на ячейки при помощи:

– визуального редактора;
– кода XAML.

Оба эти способа дадут одинаковый результат, некоторые предпочитают ручное написание кода, однако, на первых порах это может оказаться достаточно сложной задачей. Поэтому воспользуемся первым способом.

Приступим к формированию Grid. Для этого переключитесь на панель разработки MainWindow и кликните мышью по центру окошка, так чтобы гарантированно попасть в невидимый пока Grid, добейтесь результата, наблюдаемого на рис. 3.5. Наведите курсор на подсвеченную бледно-голубым цветом область. Если вы все сделали правильно, то увидите, как стрелка курсора превратилась в черный крест.

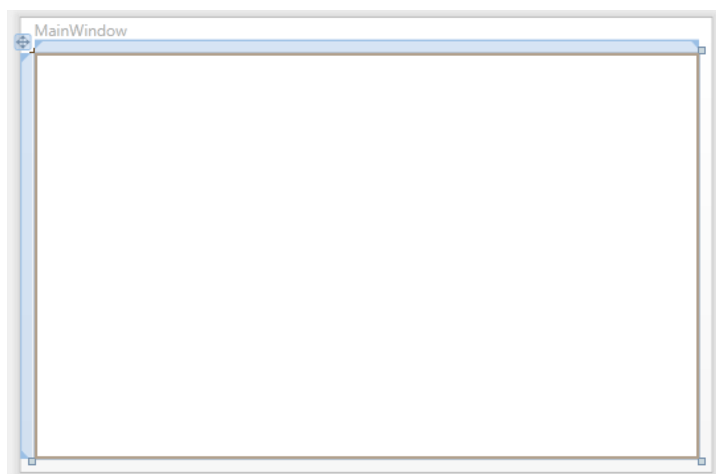

Рис. 3.5. Окно с активированным элементом Grid

Представьте, что теперь мы будем резать прямоугольный торт. Отмерьте от левого края некоторое пространство для первой части нашей сетки и кликните левой кнопкой мыши (другими словами, нарисуйте первый крайний левый столбец). Таких столбцов нам потребуется три. Результат должен стать похожим на рис. 3.6.

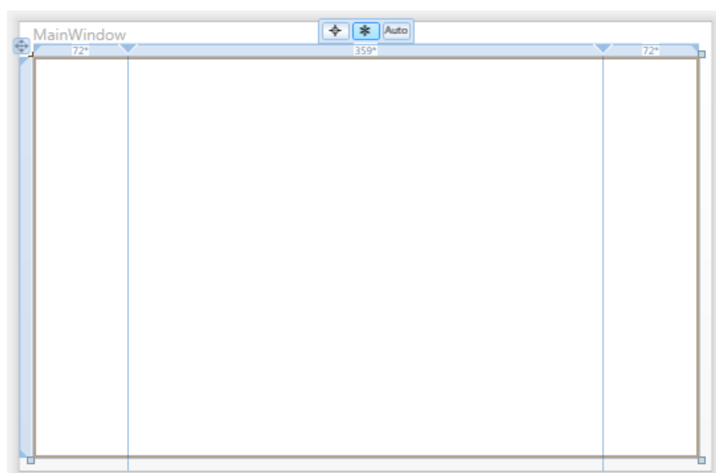

Рис. 3.6. Окно с разметкой на столбцы

В процессе рисования среда показывает нам маленькую вкладку, расположенную в середине окошка приложения (её хорошо видно на рис. 3.6) с изображением трех кнопок: «мишень», «звездочка» и «auto» (абсолютные, автоматические и пропорциональные размеры).

Абсолютные размеры. Выбирается точный размер с использованием независимых от устройства единиц измерения. Это наименее удобная стратегия, поскольку она недостаточно гибка, чтобы справиться с изменением размеров содержимого, изменением размеров контейнера или локализацией.

Автоматические размеры. Каждая строка и колонка получает в точности то пространство, которое нужно, и не более. Это один из наиболее удобных режимов изменения размеров.

Пропорциональные размеры. Пространство разделяется между группой

строк и колонок.

Для хранения на форме кнопок можно поступать несколькими путями: хранить каждую кнопку в отдельной ячейке грида или же разместить их в какой-нибудь контейнер, потому как философия WPF: одна ячейка – один виджет[10]. Мы будем пользоваться в дальнейшем StackPanel. Но до этого пока еще далеко.

То, что должно получиться на первом этапе рисования формы, представлено на рис. 3.7.

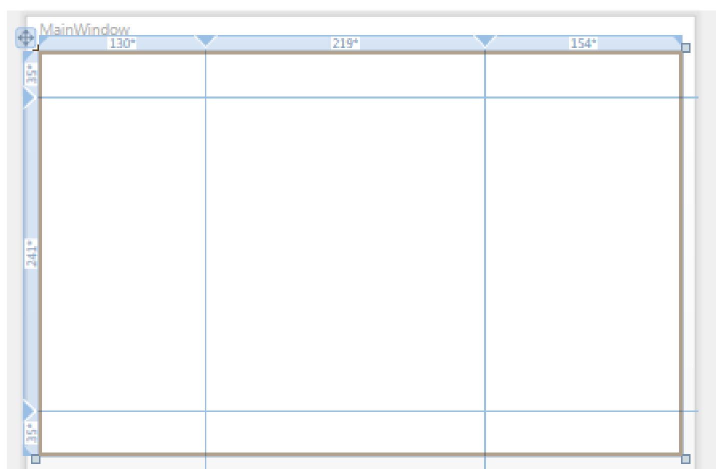

Рис. 3.7. Размещение виджета Grid на форме

0-я строка для меню;

1-я строка для кнопочек и таблицы;

2-я строка для панели статуса.

Мы предлагаем начать рисование с вытаскивания в центральную область виджета DataGrid (координаты этой области 1;1). Его легко можно найти в панели элементов.

Если мы теперь заглянем в редактор кода XAML на форме, то там что-то незримо поменялось, предлагаем еще больше это поменять и вот в какую сторону:

```
<DataGrid x:Name=«dgArtists» Grid.Row=«1» Grid.Column=«1» IsReadOnly=«True»
AutoGenerateColumns=«False»  Visibility=«Hidden»>

    <DataGrid.Columns>
        <DataGridTextColumn Header=«Имя» Binding=«{Binding Path=Name}» />
        <DataGridTextColumn Header=«Год рождения» Binding=«{Binding Path=BirthYear}» />
        <DataGridTextColumn Header=«Год смерти» Binding=«{Binding Path=DeceaseYear}» />
        <DataGridTextColumn Header=«Национальность» Binding=«{Binding
Path=Nationality}» />

    </DataGrid.Columns>
</DataGrid>
```

[10] **Ви́джет** (англ. *widget*) или элемент интерфейса, или элемент управления – примитив графического интерфейса пользователя, имеющий стандартный внешний вид и выполняющий стандартные действия.

Если изменить свойство `Visibility=«Visible»`, то можно добиться вида формы, как на рис. 3.8. но мы будем изменять это свойство в коде программы при переходе между таблицами, поэтому сейчас оставим `Hidden`. При всех этих манипуляциях в конструкторе меняем руками размеры строчек, столбцов, да и самого окна – чтобы содержимое у нас нормально умещалось.

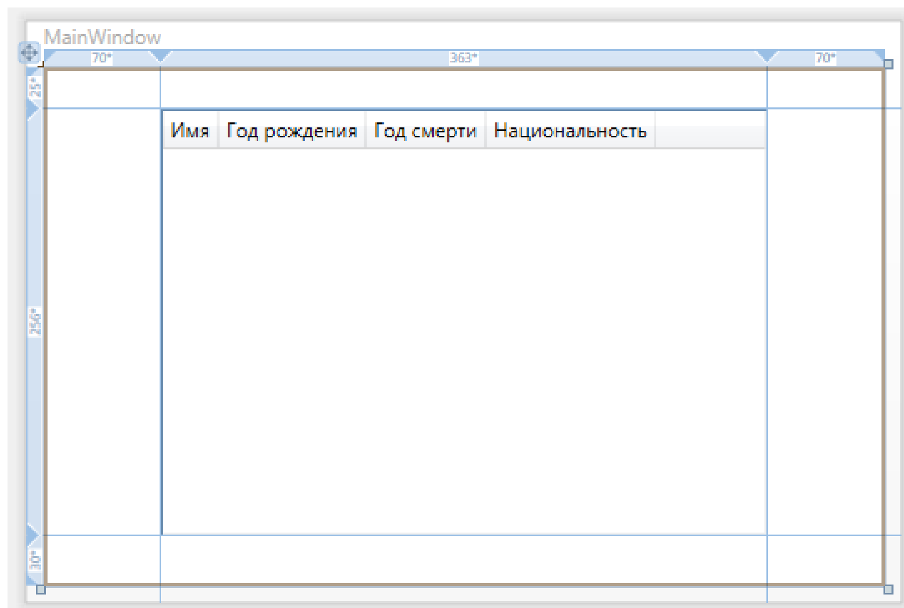

Рис. 3.8. Размещение виджета DataGrid на форме

Попробуем описать полученный XAML код:

```
<DataGrid x:Name=«dgArtists» Grid.Row=«1» Grid.Column=«1» IsReadOnly=«True»
AutoGenerateColumns=«False»  Visibility=«Hidden»>
```

Это начало определения объекта, кроваво-красным цветом обозначен собственно класс объекта. В нашем случае это виджет `DataGrid`.

Цветом пионерского галстука перечисляются названия атрибутов-свойств Имя объекта: `x:Name`.

Grid – это механизм размещения компонентов (виджетов). Однозначно определяют местонахождение компонентов на форме свойства: `Grid.Row` и `Grid.Column`.

Ну и так далее. Начинающим важно узнать, что найти эти свойства можно на панели свойств Visual Studio. Некоторые свойства, правда, там найти не так-то и просто из-за применяемой иерархии свойств.

Синим цветом, как вы уже, наверное, поняли, обозначаются инициализированные значения этих самых свойств.

Это мы сейчас разобрали открывающий тег нашего объекта.

Закрывается объект вот так: `</DataGrid>` это значит, что его описание окончено. Все, что находится между этими двумя тегами, – это описание содержимого объекта.

```
<DataGrid.Columns>

    <DataGridTextColumn Header=«Имя» Binding=«{Binding Path=Name}» />
    <DataGridTextColumn Header=«Год рождения» Binding=«{Binding Path=BirthYear}» />
```

```
    <DataGridTextColumn Header=«Год смерти» Binding=«{Binding Path=DeceaseYear}» />
    <DataGridTextColumn Header=«Национальность» Binding=«{Binding Path=Nationality}» />

</DataGrid.Columns>
```

Поскольку мы описываем таблицу, где строки – это сущности или объекты, имеющие определенные свойства, то вполне логично, что сами эти свойства – это столбцы таблицы. Собственно в вышенаписанном участке кода XAML мы и делим таблицу на колонки, присваивая им заголовки `Header` и источники данных: `Binding=«{Binding Path=Name}»`, которые означают, что конкретная колонка с заголовком «ИМЯ» должна содержать в себе то, что в объекте, связанном с таблицей, хранится в свойстве под названием `Name`.

Перейдем к созданию меню. Как ни странно нужный нам виджет называется `Menu`, найдем его на панели элементов и кинем на форму. После небольшого редактирования кода у нас получилось следующее:

```
<Menu Height=«25» HorizontalAlignment=«Stretch» Name=«mainMenu» VerticalAlignment=«Top»
Grid.ColumnSpan=«3»>
    <MenuItem Name=«DataBase» Header=«База данных»>
        <MenuItem Name=«dataBaseS» Header=«Настроить соединение с имеющейся БД»
Click=«btnDatabase_Click»/>
    </MenuItem>
    <MenuItem Name=«Reports» Header=«Отчеты»>
        <MenuItem Name=«ExelExporterButton» Header=«Экспорт таблицы Excel» />
        </MenuItem>
    <MenuItem Name = «About» Header=«О программе» ></MenuItem>
</Menu>
```

Вы пока можете не создавать такое количество `MenuItem`, а ограничиться ровно тем, что вам будет нужно в данный момент.

Добавить события можно на соответствующей вкладке в окне свойств. Событий может быть очень много, но хотя бы одно событие элемент управления должен поддерживать (если события нет – элемент управления не работает). Довольно часто – это событие нажатия левой кнопкой мыши. Отражение в коде это находит таким образом: `Click=«btnDatabase_Click»`.

Второй интересный момент этого участка кода – это атрибут `Grid.ColumnSpan=«3»` в открывающем теге. Поскольку наше меню занимает не одну колонку, это необходимо указывать явно. Иначе меню будет, как вы уже догадались, вписанным только в нулевой столбик. Аналогичное свойство есть для строк, но нам оно не требуется.

Теперь давайте займемся «низом». Элемент управления, который мы поместим в эту зону, называется `StatusBar`. По умолчанию он, как и меню пуст, но на него можно с легкостью добавлять другие виджеты. Нам потребуется виджет `Label` для отображения текста-подсказки. На языке XAML это выглядит таким образом:

```
<StatusBar Name=«statusBar» Grid.Row=«3» Grid.ColumnSpan=«3»
HorizontalAlignment=«Stretch» Margin=«1,0»  VerticalAlignment=«Bottom» Height=«30»>
    <Label Name =«statusLabel» Content=«Работа с таблицей:» Visibility=«Visible»>
    </Label>
</StatusBar>
```

А на экране отображается примерно так, как показано на рис. 3.9.

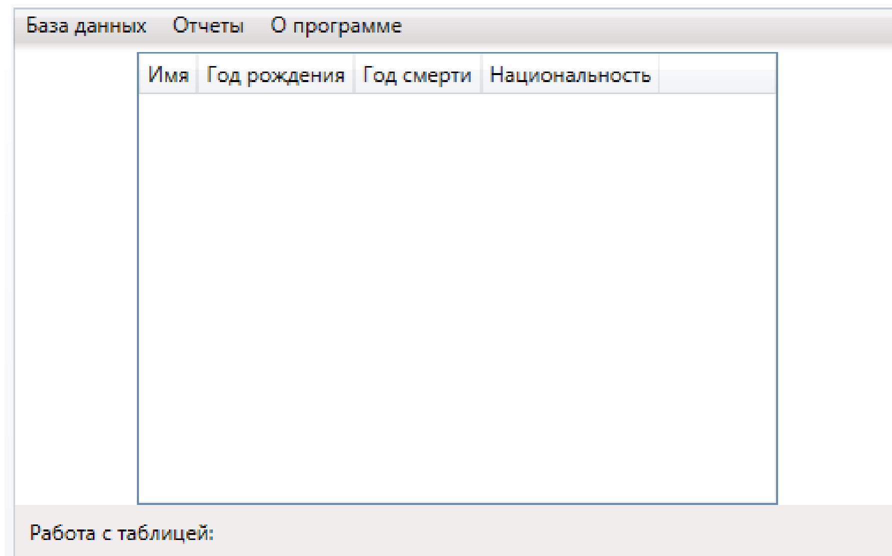

Рис. 3.9. Размещение виджетов DataGrid, Menu и StatusBar на форме

Осталось немного, а именно, добавить кнопки.

Кнопки у нас на форме лежат стопками. Ничего не напоминает? Правильно – стек. Для формирования вертикального стека используется специальный инструмент компоновки – контейнер `StackPanel`.

Кнопки можно создавать прямо в коде, не пользуясь drag-n-drop средствами среды. Код XAML вообще достаточно удобное средство, если к нему привыкнуть.

```
<StackPanel Grid.Row=«1» Grid.Column=«0» Orientation=«Vertical»>
    <Button  HorizontalAlignment=«Left»  Name=«btnArtists»  VerticalAlignment=«Top»
Height=«52» Width=«70» Click=«btnArtists_Click»>
        <Button.Content>
            <StackPanel>
                <Image Source=«/VRA;component/Images/Artist.ico» Width=«30»></Image>
                <TextBlock HorizontalAlignment=«Center» Margin=«1»
FontSize=«10»>Художники</TextBlock>
            </StackPanel>
        </Button.Content>
        <Button.ToolTip>
            <TextBlock> Информация о художниках</TextBlock>
        </Button.ToolTip>
    </Button>
</StackPanel>
```

Что тут написано? Во-первых, объект класса `StackPanel` размещен в первую строку нулевой колонки. Элементы на панели будут располагаться вертикально.

Вторая и третья строка создают кнопку. Кнопке при необходимости присваивается имя, а также размеры и событие нажатия кнопки. Размеры подбираются, исходя из чувства прекрасного разработчика.

Что происходит дальше: поскольку мы хотим иметь кнопку с текстом и рисунком, то об этом надо как-то рассказать среде. Через элемент `<Button.Content>` мы можем управлять внешним видом. Содержимое описывается несколькими элементами, это значит, что нам потребуется

контейнер. Этим контейнером будет еще один компонент StackPanel.

Добавим рисунок. Он может быть любым по желанию, мы добавим в формате .ico и соответственно небольшого для иконки размера.

Добавляем код:

```
<Image></Image>
```

Переходим в окно свойств и находим свойство Source. Открываем его и указываем путь к нашему рисунку. Не забудем выставить размер рисунка, иначе он заполнит пространство, которое ему отдавать излишне и расточительно.

Добавляем подпись:

```
<TextBlock HorizontalAlignment=«Center» Margin=«1» FontSize=«10»>Художники</TextBlock>
```

Тут все должно быть понятно. Настраиваются позиция внутри контейнера, отступ от краев, размер шрифта и собственно сам текст.

Закрываем теги.

А вот что представляет собой следующий код, попробуйте разобраться самостоятельно:

```
<Button.ToolTip>
    <TextBlock> Информация о художниках</TextBlock>
</Button.ToolTip>
```

Давайте теперь подробней разберемся с событием нажатия кнопки Click=«btnArtists_Click». Напомним, что это событие управляет переходом между таблицами и связанных с этим визуальных эффектов. Реализация имеет вид:

```
private string status; //Устанавливает, с какой таблицей в текущий момент работает пользователь

private void btnArtists_Click(object sender, RoutedEventArgs e)
        {
            switch (status)
            {
                case «Customer»: this.dgCustomers.Visibility = Visibility.Hidden;
                    break;
                case «Work»: this.dgWork.Visibility = Visibility.Hidden;
                    break;
                case «Trans»: this.dgTrans.Visibility = Visibility.Hidden;
                    break;
                case «Interests»: this.dgInterests.Visibility = Visibility.Hidden;
                    break;
                case «Nations»: this.dgNations.Visibility = Visibility.Hidden;
                    break;
            }

            this.dgArtists.Visibility = Visibility.Visible;//отображаем DateGrid художников
            status = «Artist»;//устанавливаем таблицу, с которой работаем в данный момент
            this.btnRefresh_Click(sender, e);//загружаем данные в DateGrid
            this.statusLabel.Content = «Работа с таблицей: Художники»;
            //устанавливаем видимость кнопок управления записями таблицы
            this.btnAdd.Visibility = Visibility.Visible;
            this.btnPurchase.Visibility = Visibility.Collapsed;
            this.btnSale.Visibility = Visibility.Collapsed;
            this.btnEdit.Visibility = Visibility.Visible;
            this.btnDelete.Visibility = Visibility.Visible;
```

```
        this.btnRefresh.Visibility = Visibility.Visible;
        this.btnSearch.Visibility = Visibility.Visible;
    }
```

Попробуем объяснить, что же мы написали. Блок `switch` будет скрывать активную в данный момент таблицу. Поскольку мы еще не реализовали классы для работы с остальными таблицами, кроме таблицы Художников, то блок в данной ситуации не работает.

Далее мы активируем и обновляем содержимое таблицы Artist и изменяем текущий статус работы. И, наконец, устанавливаем, нужно или нет отображать кнопки действий над записями таблицы, используя свойство видимости `Visible` или `Collapsed` соответственно.

Перейдем к реализации кнопок действия. Обработка события нажатия кнопок поиска (`btnSearch`), покупки (`btnPurchase`) и продажи (`btnSale`) картины будут описаны позже в следующих работах. Здесь рассмотрим реализацию кнопок добавления новой записи, редактирования и удаления записи и обновления данных текущей таблицы.

Метод кнопки `btnAdd_Click` будет иметь такую реализацию:

```
private void btnAdd_Click(object sender, RoutedEventArgs e)
{
    switch (status)
    {
        case «Artist»: this.btnAddA_Click();
            break;
        case «Work»: this.btnAddW_Click();
            break;
        case «Customer»: this.btnAddC_Click();
            break;
        case «Nations»: this.btnAddN_Click();
            break;
        case «Interests»: this.btnAddI_Click();
            break;
        default: MessageBox.Show(«Необходимо выбрать таблицу, в которую добавляется
элемент!»); return;
    }
}
```

Стандартный блок `switch` в зависимости от текущего значения `status` выполнит действие “Add”, соответствующее активной таблице. Обратите внимание, что в данном блоке отсутствует событие добавления записи в таблицу Transaction. Это сделано не случайно, т.к. по условию задачи картина может покупаться или продаваться. Для этого мы и ввели кнопки `btnPurchase` и `btnSale`.

Программный код обработки событий нажатия на кнопки остальных действий будет мало чем отличаться друг от друга. Просто “Add” нужно будет заменить на “Edit”, “Delete” или “Refresh” и отредактировать выводимое по умолчанию сообщение, соответствующее обрабатываемой ситуации.

Теперь рассмотрим реализацию методов добавления, изменения, и удаления записи, а также обновления всех записей для таблицы Artist. Начнем с метода `this.btnAddA_Click();`. Ранее мы уже реализовали этот метод, когда только начали знакомство с приложением. Поэтому просто найдите и

переименуйте его:

```
private void btnAddA_Click()
{
    AddArtistWindow window = new AddArtistWindow();
    window.ShowDialog();

    //Получаем список художников и передаем его на отображение таблице
    dgArtists.ItemsSource = ProcessFactory.GetArtistProcess().GetList();
}
```

Аналогично найдите и переименуйте методы редактирования, удаления и обновления записей: `btnEditA_Click();` `btnDeleteA_Click();` `btnRefreshA_Click();`.

Единственный момент, который нужно учесть в данном случае, в коде методов редактирования и удаления записи найти строку

```
    //И перезагружаем список художников
    btnRefresh_Click(sender, e);
```

и заменить на

```
    //И перезагружаем список художников
    btnRefreshA_Click(sender, e);
```

После добавления всех кнопок вы можете запустить приложение и проверить его работу. У вас должно получиться выполнить следующие действия:

1) подключиться к базе данных;
2) вызвать кнопку Художники и получить записи по художникам;
3) добавить художника;
4) отредактировать художника;
5) удалить запись;
6) обновить таблицу.

После того, как вы добавите соответствующий код для работы с таблицами Клиент, Работа, Транзакция и Интересы, реализация методов типа `this.btnAddW_Click();` останется лишь формальностью, вам надо будет скопировать аналогичный метод для “Artist” и изменить в нем несколько строк соответственно, например:

```
private void btnAddW_Click()
{
    AddWorkWindow window = new AddWorkWindow();
    window.ShowDialog();
    dgWork.ItemsSource = ProcessFactory.GetWorkProcess().GetList();
}
```

Контрольные вопросы и задания

1. Что такое XAML разметка, в чем её преимущества?
2. Какие способы организации графических интерфейсов бывают? В чем их особенности?
3. Что такое модальное окно?
4. Что такое WPF?
5. Каким образом можно создавать экранные формы в технологии WPF?

6. Что такое виджет?
7. Назовите основные элементы управления, используемые при разработке главного окна приложения.
8. Используя образец XAML кода, объясните применение его элементов и их свойств.
9. Объясните, каким образом можно сделать кнопки с рисунком и текстом?
10. Объясните назначение оператора `switch`.
11. Каким образом можно управлять видимостью на экране элементов управления?

Лабораторная работа №4
Перепроектирование базы данных. Добавление в проект работы с национальностями художников

4.1. Постановка задачи

Часто в ходе эксплуатации реальной системы возникают моменты, которые требуют внесения изменений в уже существующее решение. Это продиктовано тем, что мир меняется. Меняются задачи, условия и, соответственно, требования, предъявляемые к системе. Изменений не стоит бояться, к ним нужно быть готовым и успешно их осуществлять. Например, оказалось, что использование фиксированного списка возможных национальностей художников не очень удобно. Решено было выделить национальности в отдельную таблицу.

4.2. Подготовка БД

Первое, что необходимо сделать – внести изменения в базу данных. Делается это при помощи SQL-скриптов [2]. Естественно, мы уже подготовили свой сценарий изменения базы данных, попробуйте и вы сделать это. Давайте подумаем, что необходимо предпринять.

Сначала необходимо создать новую таблицу, которая будет хранить записи с национальностями. Затем заполнить ее записями на основании тех национальностей, которые уже есть в таблице художников. Изменить таблицу художников, добавив столбец, ссылающийся на таблицу национальностей. Заполнить этот столбец значениями первичного ключа таблицы национальностей, соответствующих национальности художника. Установить связь между таблицей национальностей и художников и задать органичение ссылочной целостности. И, наконец, удалить ограничение на список национальностей и сам столбец национальности из таблицы художников.

Вот, что у нас получилось в результате.

На случай, если ранее уже были какие-то неудачные попытки создания подобной таблицы, используется скрипт:

```
IF OBJECT_ID('Nation', N'U') IS NOT NULL
	DROP TABLE Nation
GO
```

Далее необходимо произвести манипуляции с имеющимися ограничениями и таблицами, накладывающими их. А также создать новую таблицу с национальностями, по возможности заполнив ее информацией без потерь.

```
CREATE TABLE Nation
(
	NatID	int IDENTITY ( 1,1 ) NOT NULL CONSTRAINT pk_Nation_NatID PRIMARY KEY
CLUSTERED,
	Value	varchar(25)  NOT NULL CONSTRAINT uk_Nation_Value UNIQUE
)
go
```

```
insert into Nation
(Value)
	select distinct Nationality from Artist

ALTER TABLE Artist ADD NatID int

update Artist set
Artist.Nationality = (Select NatID from Nation where Artist.Nationality=Nation.Value)

alter table Artist
	ADD CONSTRAINT R FOREIGN KEY (NatID) REFERENCES Nation(NatID)
		ON DELETE CASCADE
		ON UPDATE CASCADE
go

Alter table Artist
Drop constraint NationalityList

ALTER table Artist
Drop Column Nationality
```

Примечание. `NationalityList` это имя нашего ограничения на национальности. При создании своей БД вы могли указать его под другим именем, поэтому будьте внимательней.

В конце концов, можно заполнить таблицу национальностей известным способом, добавив новые национальности:

```
insert into Nation(Value)
values ('Русский'), ('Немец'), ('Француз')
```

4.3. Подготовка объекта DTO

Теперь, когда база данных подготовлена и изменена, займемся доводкой приложения. Каких-либо особых действий это не потребует, задача схожа с созданием модулей работы с Художниками. Возможно, начинать с объекта DTO несколько не естественно, ведь это даже, по сути, не объект предметной области, а вспомогательный класс, используемый внутри приложения, но мы предлагаем начать с него.

Уже знакомым способом создадим пустой класс в решении VRA.Dto:

```
namespace VRA.Dto
{
    public class NationDto
    {
        public int Id { get; set; }
        public string Nationality { get; set; }
    }
}
```

4.4. Реализация “бизнес-требований”

Перейдем на уровень “бизнес-решения” и в соответствии с предлагаемым шаблоном реализации создадим интерфейс `INationProcess`.

Для национальностей мы будем предполагать операции извлечения из списка, добавление новой национальности в БД и удаление национальности из списка доступных.

```
using System.Collections.Generic;
using VRA.Dto;

namespace VRA.BusinessLayer
{
    public interface INationProcess
    {
        IList<NationDto> GetList();
        void Add (NationDto n);
        void Delete(int id);
    }
}
```

Создадим класс NationProcess, на месте реализаций пока оставим заглушки и перейдем на уровень DAO, там нам нужно подготовить объекты, извлекающие данные из базы данных. На этом этапе было принято решение о создании класса BaseDao, чтобы вынести в него общий для всех DAO классов код.

4.5. Подготовка класса BaseDao

Поскольку на уровне Dao мы всегда подключаемся к серверу, вне зависимости от того, каким именно объектом уровня БД мы манипулируем, то эту операцию и вынесем в BaseDao.

```
using System.Configuration;
using System.Data.SqlClient;

namespace Vra.DataAccess
{
    public class BaseDao
    {
        /// <summary>
        /// Возвращает объект подключения к базе
        /// </summary>
        /// <returns></returns>
        protected static SqlConnection GetConnection()
        {
            return new SqlConnection(GetConnectionString());
        }
        /// <summary>
        /// Возвращает строку подключения к базе
        /// </summary>
        /// <returns></returns>
        private static string GetConnectionString()
        {
            return ConfigurationManager.ConnectionStrings[«vradb»].ConnectionString;
        }
    }
}
```

Примечание. Обратите внимание на модификаторы доступа (в UML их называют кванторы видимости), о них часто спрашивают на собеседованиях и для программиста C# важно разбираться в отличиях protected от private. Protected – метод будет доступен классу-наследнику, private – нет.

Сам по себе класс BaseDao можно сделать абстрактным, ведь конкретной реализации у него не будет – только наследники. Пока этого мы делать не

станем, возможно, вам будет интересно в этом разобраться самостоятельно?

4.6. Работа с NationDao

В соответствии с принятым шаблоном реализации в папку Entites добавим новый класс сущность: `Nation`.

```
namespace Vra.DataAccess.Entities
{
    public class Nation
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
}
```

И перейдем к декларации требований через интерфейс:

```
using System.Collections.Generic;
using Vra.DataAccess.Entities;

namespace Vra.DataAccess
{
    public interface INationDao
    {
        /// <summary>
        /// Загрузить все национальности
        /// </summary>
        /// <returns></returns>
        IList<Nation> Load();

        /// <summary>
        /// Получить национальность по id
        /// </summary>
        /// <param name=«id»></param>
        /// <returns></returns>
        Nation Get(int id);

        void Add(Nation Nation);
        void Delete(int id);
    }
}
```

В этот раз, подход к извлечению конкретной национальности с конкретным ключом, мы будем производить на уровне приложения, а не на уровне БД. Обратите особое внимание на метод `Nation Get(int id)`.

```
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
using Vra.DataAccess.Entities;

namespace Vra.DataAccess
{
    public class NationDao : BaseDao, INationDao
    {
        /// <summary>
        /// Делаем своего рода кэш с национальностями
        /// </summary>
        private static IDictionary<int, Nation> Nations;
```

```
private IList<Nation> LoadInternal()
{
    IList<Nation> nations = new List<Nation>();
    using (var conn = GetConnection())
    {
        conn.Open();
        using (var cmd = conn.CreateCommand())
        {
            cmd.CommandText = «SELECT NatID, Value FROM Nation»;
            cmd.CommandType = CommandType.Text;
            using (var reader = cmd.ExecuteReader())
            {
                while (reader.Read())
                {
                    nations.Add(LoadNation(reader));
                }
            }
        }
    }
    return nations;
}

public IList<Nation> Load()
{
        Nations = new Dictionary<int, Nation>();
        var nations = LoadInternal();
        foreach (var nation in nations)
        {
            Nations.Add(nation.Id, nation);
        }
    return Nations.Values.ToList();
}

public Nation Get(int id)
{
    if (Nations == null)
        Load();
    return Nations.ContainsKey(id) ? Nations[id] : null;
}

public void Reset()
{
    if (Nations == null)
        return;
    Nations.Clear();
}

private static Nation LoadNation(SqlDataReader reader)
{
    Nation nation = new Nation();
    nation.Id = reader.GetInt32(reader.GetOrdinal(«NatID»));
    nation.Name = reader.GetString(reader.GetOrdinal(«Value»));
    return nation;
}

public void Add(Nation Nation)
{
    using (var conn = GetConnection())
    {
        conn.Open();
        using (var cmd = conn.CreateCommand())
```

```
                {
                    cmd.CommandText =
                         «INSERT INTO Nation (Value) VALUES (@Nation)»;
                    cmd.Parameters.AddWithValue(«@Nation», Nation.Name);
                    cmd.ExecuteNonQuery();
                }
            }
        }
        public void Delete(int id)
        {
            using (var conn = GetConnection())
            {
                conn.Open();
                using (var cmd = conn.CreateCommand())
                {
                    cmd.CommandText = «DELETE FROM Nation WHERE NatID = @ID»;
                    cmd.Parameters.AddWithValue(«@ID», id);
                    cmd.ExecuteNonQuery();
                }
            }
        }
    }
}
```

Самостоятельно необходимо разобраться со структурой данных `Dictionary<int, Nation>`, реализующей интерфейс `IDictionary<int, Nation>` (`IDictionary<TKey, TValue>`).

Давайте, чтобы было понятно, попытаемся отследить последовательность действий при получении списка всех национальностей из БД.

Вызывая метод `Load()`, мы получаем:

```
Nations = new Dictionary<int, Nation>();
var nations = LoadInternal();
```

Вызов метода `LoadInternal()` влечет за собой выполнение запроса к базе данных при помощи подключения, открываемого методом `GetConnection()`, унаследованным от базового класса (см. код определения методов класса `NationDao`).

Здесь же мы впервые сталкиваемся с типом данных `var`. Возвращаемся к методу `Load()`, который уже получил список национальностей:

```
foreach (var nation in nations)
{
    Nations.Add(nation.Id, nation);
}
return Nations.Values.ToList();
```

В этом участке кода происходит заполнение словаря `Nations`, обладающего свойством `static`, найдите в Интернет информацию об этом ключевом слове и разберитесь с ограничениями, которые оно накладывает, это тоже один из вопросов, которые вам могут задать на собеседовании.

Теперь, когда приложение получило список национальностей, с ним сможет работать метод `Get()`. Подумайте и скажите, почему такая реализация не вполне удобна?

Теперь, когда объект Dao подготовлен, создадим для него метод в фабрике классов `DaoFactory`:

```
public static INationDao GetNationDao()
{
    return new NationDao();
}
```

Возвращаемся на “бизнес-уровень”.

4.7. Реализация “бизнес-требований” 2

Как и в случае других классов, теперь мы можем создать конвертер в соответствующей папке, в класс `DtoConverter` дописываем:

```
public static NationDto Convert(Nation nation)
{
    if (nation == null)
        return null;
    NationDto nationDto = new NationDto();
    nationDto.Id = nation.Id;
    nationDto.Nationality = nation.Name;
    return nationDto;
}

public static Nation Convert(NationDto nationDto)
{
    if (nationDto == null)
        return null;
    Nation nation = new Nation();
    nation.Id = nationDto.Id;
    nation.Name = nationDto.Nationality;
    return nation;
}

internal static IList<NationDto> Convert(IList<Nation> nationList)
{
    var nations = new List<NationDto>();
    foreach (var nation in nationList)
    {
        nations.Add(Convert(nation));
    }
    return nations;
}
```

Примечание. internal - это разновидность модификатора доступа в C#, означающая доступность только внутри содержащей сборки.

Таким образом, файл `NationProcess` примет вид:

```
using System.Collections.Generic;
using VRA.BusinessLayer.Converters;
using VRA.Dto;
using Vra.DataAccess;

namespace VRA.BusinessLayer
{
    public class NationProcess : INationProcess
    {
        private static INationDao NatDao = new NationDao();

        public IList<NationDto> GetList()
        {
            return DtoConverter.Convert(DaoFactory.GetNationDao().Load());
        }
```

```
        public void Add(NationDto n)
        {
           NatDao.Add(DtoConverter.Convert(n));
        }
        public void Delete(int id)
        {
            NatDao.Delete(id);
        }
    }
}
```

Не забудем включить новый класс в фабрику классов этого слоя:

```
public static INationProcess GetNationProcess()
{
    return new NationProcess();
}
```

4.8. Реализация графического интерфейса

Реализация GUI для национальностей не содержит никаких новых нюансов, выполните её в соответствии с рекомендациями по реализации, описанными в предыдущих лабораторных работах. Также не забудьте подключить работу с национальностями в главном окне приложения и убедиться, что все работает правильно.

4.9. Изменение в классе Artist

Изменения в работе с национальностями влекут изменения в работе с художниками, которые проявляются на уровнях DTO и DAO.

В папке Entites изменим класс сущности `Artist`:

```
namespace Vra.DataAccess.Entities
{
    public class Artist
    {
        public int ArtistId;
        public string Name;
        public int? BirthYear;
        public int? DeceaseYear;
        public int NationId;
    }
}
```

А также класс `ArtistDto`:

```
namespace VRA.Dto
{
    public class ArtistDto
    {
        /// <summary>
        /// Уникальный идентификатор объекта
        /// </summary>
        public int Id { get; set; }

        /// <summary>
        /// Имя
        /// </summary>
        public string Name { get; set; }
```

```
        /// <summary>
        /// Год рождения
        /// </summary>
        public int? BirthYear { get; set; }

        /// <summary>
        /// Год смерти
        /// </summary>
        public int? DeceaseYear { get; set; }

        /// <summary>
        /// Национальность
        /// </summary>
        public NationDto Nation { get; set; }
    }
}
```

Теперь объект DTO будет хранить внутри себя не просто строку, а целый объект. Это повлечет за собой изменения в работе формы. Точнее в реализации биндинга.

```
<DataGrid x:Name=«dgArtists» Grid.Row=«1» Grid.Column=«1» IsReadOnly=«True»
AutoGenerateColumns=«False» Visibility=«Visible»>
    <DataGrid.Columns>
        <DataGridTextColumn Header=«Имя» Binding=«{Binding Path=Name}» />
        <DataGridTextColumn Header=«Год рождения» Binding=«{Binding Path=BirthYear}» />
        <DataGridTextColumn Header=«Год смерти» Binding=«{Binding Path=DeceaseYear}» />
        <DataGridTextColumn Header=«Национальность» Binding=«{Binding
Path=Nation.Nationality}» />
    </DataGrid.Columns>
</DataGrid>
```

Так будет выглядеть грид с национальностями на главной форме. Изменения на форме Художника связаны с нюансами реализации ее программистом, поэтому конкретный рецепт дать сложно, однако принцип связывания будет точно таким же.

Вернемся с форм, но не на уровень DTO, а немного глубже – на уровень DAO, добавим ссылку System.Windows.Forms.

Листинг класса `ArtistDao` изменится следующим образом:

```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using Vra.DataAccess.Entities;
using System.Windows.Forms;

namespace Vra.DataAccess
{
    public class ArtistDao : BaseDao, IArtistDao
    {
        private static Artist LoadArtist(SqlDataReader reader)
        {
            //Создаём пустой объект
            Artist artist = new Artist();
            //Заполняем поля объекта в соответствии с названиями полей результирующего
            // набора данных
            artist.ArtistId = reader.GetInt32(reader.GetOrdinal(«ArtistID»));
            object birth = reader[«BirthYear»];
            if (birth != DBNull.Value)
```

```
            artist.BirthYear = Convert.ToInt32(birth);
        //Помните, что у нас поле DeceaseYear может иметь значение NULL?
        //Следующие 3 строки корректно обрабатывают этот случай
        object decease = reader[«DeceaseYear»];
        if (decease != DBNull.Value)
            artist.DeceaseYear = Convert.ToInt32(decease);
        artist.Name = reader.GetString(reader.GetOrdinal(«Name»));
        int pos = reader.GetOrdinal(«NatID»);
        artist.NationId = reader[pos] == DBNull.Value ? -1 : reader.GetInt32(pos);
        return artist;
    }

    public Artist Get(int id)
    {
      //Получаем объект подключения к базе
      using (var conn = GetConnection())
      {
         //Открываем соединение
         conn.Open();
         //Создаем sql команду
         using (var cmd = conn.CreateCommand())
         {
            //Задаём текст команды
             cmd.CommandText = «SELECT ArtistID, Name, BirthYear, DeceaseYear,
NatID FROM ARTIST WHERE ArtistID = @id»;
            //Добавляем значение параметра
            cmd.Parameters.AddWithValue(«@id», id);
            //Открываем SqlDataReader для чтения полученных в результате
            // выполнения запроса данных
            using (var dataReader = cmd.ExecuteReader())
            {
                return !dataReader.Read() ? null : LoadArtist(dataReader);
            }
         }
      }
    }

    public IList<Artist> GetAll()
    {
        IList<Artist> artists = new List<Artist>();
        using (var conn = GetConnection())
        {
            conn.Open();
            using (var cmd = conn.CreateCommand())
            {
                cmd.CommandText = «SELECT ArtistID, Name, BirthYear, DeceaseYear,
NatID FROM ARTIST»;
                using (var dataReader = cmd.ExecuteReader())
                {
                    while (dataReader.Read())
                    {
                        artists.Add(LoadArtist(dataReader));
                    }
                }
            }
            return artists;
        }
    }

    public void Add(Artist artist)
    {
```

```
            using (var conn = GetConnection())
            {
                conn.Open();
                using (var cmd = conn.CreateCommand())
                {
                    cmd.CommandText = «INSERT INTO ARTIST
(Name,BirthYear,DeceaseYear,NatID) VALUES
(@Name,@BirthYear,@DeceaseYear,@Nationality)»;
                    cmd.Parameters.AddWithValue(«@Name», artist.Name);
                    cmd.Parameters.AddWithValue(«@BirthYear», artist.BirthYear);
                    cmd.Parameters.AddWithValue(«@Nationality», artist.NationId);
                    object decease = artist.DeceaseYear.HasValue ?
                          (object)artist.DeceaseYear.Value : DBNull.Value;
                    cmd.Parameters.AddWithValue(«@DeceaseYear», decease);
                    cmd.ExecuteNonQuery();
                }
            }
        }

        public void Update(Artist artist)
        {
            using (var conn = GetConnection())
            {
                conn.Open();
                using (var cmd = conn.CreateCommand())
                {
                    cmd.CommandText = «UPDATE ARTIST SET Name = @Name, BirthYear =
@BirthYear, DeceaseYear = @DeceaseYear, NatID = @Nationality WHERE ArtistID = @ID»;
                    cmd.Parameters.AddWithValue(«@Name», artist.Name);
                    cmd.Parameters.AddWithValue(«@BirthYear», artist.BirthYear);
                    cmd.Parameters.AddWithValue(«@ID», artist.ArtistId);
                    cmd.Parameters.AddWithValue(«@Nationality», artist.NationId);
                    object decease = artist.DeceaseYear.HasValue ?
                        (object)artist.DeceaseYear.Value : DBNull.Value;
                    cmd.Parameters.AddWithValue(«@DeceaseYear», decease);
                    cmd.ExecuteNonQuery();
                }
            }
        }

        public void Delete(int id)
        {
            using (var conn = GetConnection())
            {
                conn.Open();
                using (var cmd = conn.CreateCommand())
                {
                    cmd.CommandText = «DELETE FROM ARTIST WHERE ArtistID = @ID»;
                    cmd.Parameters.AddWithValue(«@ID», id);
                    try
                    {
                        cmd.ExecuteNonQuery();
                    }
                    catch (Exception ex)
                    {
                        MessageBox.Show(«Исключение базы данных: « + ex.ToString(),
«WARNING!»);
                    }
                }
            }
        }
```

```
    }
}
```

Везде, где до этого были названия национальностей, сущность, извлекаемая из БД, имеет целочисленный идентификатор национальности. Как же этот идентификатор внутри сущности, становится объектом внутри DTO класса? Мы не зря писали метод, который возвращает национальность по идентификатору внутри “бизнес-логики” национальности. Изменяя в конвертерах всего одну строчку, мы добиваемся необходимого нам эффекта, и в качестве свойства DTO-объекта храним теперь другой DTO.

```
public static ArtistDto Convert(Artist artist)
{
    if (artist == null)
        return null;
    ArtistDto artistDto = new ArtistDto();
    artistDto.Id = artist.ArtistId;
    artistDto.BirthYear = artist.BirthYear;
    artistDto.DeceaseYear = artist.DeceaseYear;
    artistDto.Name = artist.Name;
    artistDto.Nation = Convert(DaoFactory.GetNationDao().Get(artist.NationId));
    return artistDto;
}

public static Artist Convert(ArtistDto artistDto)
{
    if (artistDto == null)
        return null;
    Artist artist = new Artist();
    artist.ArtistId = artistDto.Id;
    artist.BirthYear = artistDto.BirthYear;
    artist.DeceaseYear = artistDto.DeceaseYear;
    artist.Name = artistDto.Name;
    artist.NationId = artistDto.Nation.Id;
    return artist;
}
```

Конвертер списка художников остается без изменений.

Теперь перейдем на уровень VRA в класс `AddArtistWindow`. Так как теперь наши национальности хранятся в БД, то и работа формы должна измениться. Ниже приведены изменившиеся методы:

```
/// <summary>
/// Список допустимых национальностей
/// </summary>
private readonly IList<NationDto> Nationalities =
ProcessFactory.GetNationProcess().GetList();

public AddArtistWindow()
{
    InitializeComponent();
    //Передаем допустимые значения
    cbNationality.ItemsSource = (from N in Nationalities orderby N.Nationality select
N);
}

private void btnSave_Click(object sender, RoutedEventArgs e)
{
```

```
    int? birth;
    int? death = null;

    if (string.IsNullOrEmpty(tbName.Text))
    {
        MessageBox.Show(«Имя художника не должно быть пустым», «Проверка»);
        return;
    }

    if (tbBirth.Text != ««)
    {
        try
        {
            birth = int.Parse(this.tbBirth.Text);
        }

        catch
        {
            MessageBox.Show(«Год рождения должен быть целым числом», «Проверка»);
            return;
        }

        if (birth < 1000 || birth > DateTime.Today.Year)
        {
            MessageBox.Show(«Галерея занимается продажей только произведений художников
прошлого тысячелетия», «Проверка»);
            return;
        }
    }
    else
    {
        birth = null;
    }

    if (!string.IsNullOrEmpty(tbDeath.Text))
    {
        int intDeath;

        if (!int.TryParse(tbDeath.Text, out intDeath))
        {
            MessageBox.Show(«Год смерти должен быть целым числом», «Проверка»);
            return;
        }

        if (intDeath < 1000 || intDeath > DateTime.Today.Year)
        {
            MessageBox.Show(«Год смерти введен неверно», «Проверка»);
            return;
        }

        if (intDeath < birth)
        {
            MessageBox.Show(«Год смерти должен быть больше года рождения», «Проверка»);
            return;
        }
        death = intDeath;
    }

    //Создаем объект для передачи данных
    ArtistDto artist = new ArtistDto();
```

```
    //Заполняем объект данными
    artist.Name = tbName.Text;
    artist.BirthYear = birth;
    artist.DeceaseYear = death;
    artist.Nation = cbNationality.SelectedItem as NationDto;

    //Именно тут запрашиваем реализованную ранее задачу по работе с художниками
    IArtistProcess artistProcess = ProcessFactory.GetArtistProcess();

    //если это новый объект - сохраняем его
    if (_id == 0)
    {
        //Сохраняем художника
        artistProcess.Add(artist);
    }
    else //иначе обновляем
    {
        //копируем обратно идентификатор объекта
        artist.Id = _id;
        //обновляем
        artistProcess.Update(artist);
    }

    //и закрываем форму
    Close();
}

/// <summary>
/// Метод загружает объект художника для редактирования
/// </summary>
/// <param name=«artist»>редактируемый объект художника</param>
public void Load(ArtistDto artist)
{
    //если объект не существует выходим
    if (artist == null)
        return;

    //сохраняем id художника
    _id = artist.Id;

    //заполняем визуальные компоненты для отображения данных
    tbName.Text = artist.Name;
    tbBirth.Text = artist.BirthYear.ToString();
    if (artist.DeceaseYear.HasValue)
        tbDeath.Text = artist.DeceaseYear.Value.ToString();

    if (artist.Nation != null)
    {
        foreach (NationDto Nat in Nationalities)
        {
            if (artist.Nation.Id == Nat.Id)
            {
                this.cbNationality.SelectedItem = Nat;
                break;
            }
        }
    }
}
```

Для корректного отображения национальностей на форме присвоим комбобоксу `cbNationality` два свойства:

```
ItemsSource=«{Binding}»
DisplayMemberPath=«Nationality»
```

Теперь можно смело запускать приложения и убедиться, что все работает правильно.

Контрольные вопросы и задания

1. С помощью какого SQL оператора можно удалить таблицу?
2. Расскажите об SQL операторах определения и изменения структуры таблицы? Дайте примеры.
3. Что такое ограничение? Каким образом они задаются? Приведите примеры ограничений.
4. С какой целью был создан класс `BaseDao`?
5. Что такое кванторы видимости? Какие они бывают? В чем их различия и особенности?
6. Что такое абстрактный класс? Какую задачу он выполняет?
7. Что такое структура данных `Dictionary<int, Nation>`?
8. Зачем используется свойство `static`, какую задачу оно играет?
9. Что такое internal? Объясните его назначение и порядок использования.
10. Реализуйте графический интерфейс для национальностей.
11. По аналогии с реализацией для сущности Художник выполните самостоятельно реализацию для сущности Клиент, Интересы.

Лабораторная работа №5
Создаем средства работы с художественными произведениями, их покупки и продажи

5.1. Постановка задачи

В данной работе мы рассмотрим создание средств, для работы с картинами. Этими средствами будут являться два объекта Work (Картины) и Transaction (Транзакции). Они нам необходимы для реализации возможности добавления и продажи картин.

До этого этапа все действия с таблицами художников и клиентов должны быть выполнены. А вы, надеемся, начинаете понимать, как устроено приложение.

5.2. Работа с Work

Ранее мы подробно рассмотрели реализацию сущностей «Художник» и «Национальность». Вам самостоятельно, следуя образцу, предлагалось реализовать работу с сущностью «Клиент». Если вы делали это внимательно, то не могли не заметить общего подхода, общей схемы при реализации каждой рассмотренной выше сущности. Теперь настал черёд разобраться с реализацией сущности «Работа».

Общая схема реализации сущности «Работа» представлена на рис. 5.1 в виде диаграммы классов UML. Если следовать представленной диаграмме, то вам необходимо создать классы `Work`, `WorkDto`, `WorkDao`, `WorkProcessDb` и интерфейсы `IWorkDao` и `IWorkProcess`.

Реализацию сущности «Работа» можно начать с создания соответствующего класса `Work` в папке Entities. Далее перейти на уровень Vra.DataAccess и создать интерфейс для доступа к данным `IWorkDao` и его реализацию в виде класса `WorkDao`.

В классе `DaoFactory` не забудьте добавить метод, отвечающий за создание объектов класса `WorkDao`:

```
public static IWorkDao GetWorkDao()
{
    return new WorkDao();
}
```

Реализация уровня доступа к данным завершена. Теперь перейдем к созданию объекта DTO `WorkDto`.

Вы уже заметили, чем отличаются классы DTO от классов Entity? Во-первых, класс DTO описывает свойства (`property`), а не поля (`field`). Во-вторых, при объявлении свойств мы легко можем ссылаться на объект в целом. Таким образом, каждой картине привязан не просто идентификатор художника, как это отражено в сущности `Work`, а все сведения о художнике: с его именем, фамилией и прочими атрибутами, что существенно облегчает получение нужных данных в ходе реализации приложения.

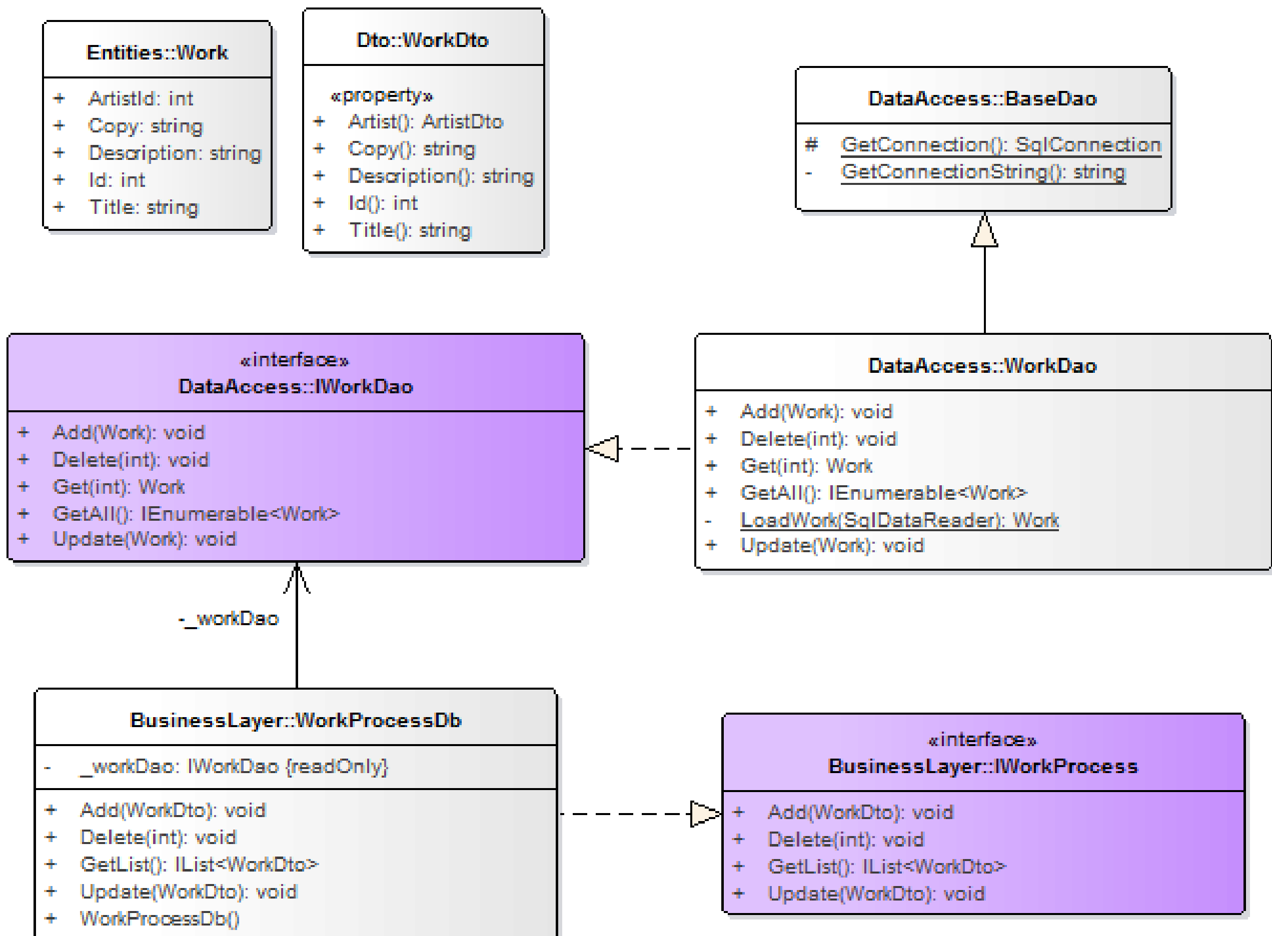

Рис. 5.1. Схема реализации сущности «Работа»

Переходим на уровень «бизнес-логики» и реализуем логику работы с объектами класса Work. Для этого создаем интерфейс IWorkProcess и реализуем его в классе WorkProcessDb, согласно предложенной схеме. Реализацию методов выполните по аналогии с классами Artist и Nation.

Далее заглянете в папку Converters и добавьте методы-конвертеры для Work и WorkDto:

```
public static WorkDto Convert(Work work){}
public static IList<WorkDto> Convert(IList<Work> works){}
public static Work Convert(WorkDto workDto){}
```

Реализовать их предлагаем самостоятельно.

В классе ProcessFactory не забудьте добавить метод, отвечающий за создание объектов класса WorkProcessDb:

```
public static IWorkProcess GetWorkProcess()
{
    return new WorkProcessDb();
}
```

На уровне представления в главном окне приложения (MainWindows.xaml) будет одна маленькая тонкость:

```
<DataGrid x:Name=«dgWork» Grid.Row=«1» Grid.Column=«1» IsReadOnly=«True»
AutoGenerateColumns=«False» Visibility=«Hidden»>
      <DataGrid.Columns>
```

```
            <DataGridTextColumn Header=«Название» Binding=«{Binding Path=Title}» />
            <DataGridTextColumn Header=«Автор» Binding=«{Binding Path=Artist.Name}» />
            <DataGridTextColumn Header=«Копия» Binding=«{Binding Path=Copy}» />
            <DataGridTextColumn Header=«Описание» Binding=«{Binding Path=Description}» />
        </DataGrid.Columns>
</DataGrid>
```

Обратите внимание на строчку:

```
<DataGridTextColumn Header=«Автор» Binding=«{Binding Path=Artist.Name}» />
```

Таким образом, в сетке данных в главном окне приложения мы выведем не идентификатор художника, а его имя.

Казалось бы, на этом реализация работы с классом `Work` закончена. Однако самая пора вспомнить требования, которые формулировались еще в процессе проектирования базы данных. Одним из таких требований было то, что при поступлении (покупке) картины в галерею, необходимо зафиксировать дату поступления и цену приобретения картины. Для отражения этого события нами был реализован триггер, который при добавлении записи о картине в таблицу Work добавлял соответствующую запись в таблицу Transaction. Таким образом, для завершения реализации работы с классом `Work` необходимо реализовать работу с классом `Transaction`.

5.3. Работа с Transaction

Общая схема реализации сущности «Транзакция» представлена на рис. 5.2 в виде диаграммы классов UML. Если следовать представленной диаграмме, Вам необходимо создать классы `Transaction`, `TransactionDto`, `TransactionDao`, `TransactionProcessDb` и интерфейсы `ITransactionDao` и `ITransactionProcess`.

Далее все абсолютно аналогично.

В классе `DaoFactory` не забудьте добавить метод, отвечающий за создание объектов класса `TransactionDao`, а в классе `ProcessFactory` – за создание объектов класса `TransactionProcessDb`.

Также не забудьте добавить методы-конвертеры для `Transaction` и `TransactionDto`:

```
public static TransactionDto Convert(Transaction trans){}
public static Transaction Convert(TransactionDto transDto){}
public static IList<TransactionDto> Convert(IList<Transaction> transes){}
```

Вот, в общем-то, и все.

На уровне представления в главном окне приложения (MainWindows.xaml) добавьте код для отображения записей транзакций:

```
<DataGrid x:Name=«dgTrans» Grid.Row=«1» Grid.Column=«1» IsReadOnly=«True»
AutoGenerateColumns=«False» Visibility=«Hidden»>
    <DataGrid.Columns>
        <DataGridTextColumn Header=«Номер транзакции» Binding=«{Binding
Path=TransactionID}» />
        <DataGridTextColumn Header=«Автор» Binding=«{Binding Path=Work.Artist.Name}» />
        <DataGridTextColumn Header=«Работа» Binding=«{Binding Path=Work.Title}» />
        <DataGridTextColumn Header=«Дата приобретения» Binding=«{Binding
Path=DateAcquired, StringFormat=\{0:dd.MM.yyyy\}}»  />
```

```
        <DataGridTextColumn Header=«Цена приобретения» Binding=«{Binding
Path=AcquisitionPrice}» />
        <DataGridTextColumn Header=«Клиент» Binding=«{Binding Path=Customer.Name}» />
        <DataGridTextColumn Header=«Дата продажи» Binding=«{Binding Path=PurchaseDate,
StringFormat=\{0:dd.MM.yyyy\}}» />
```

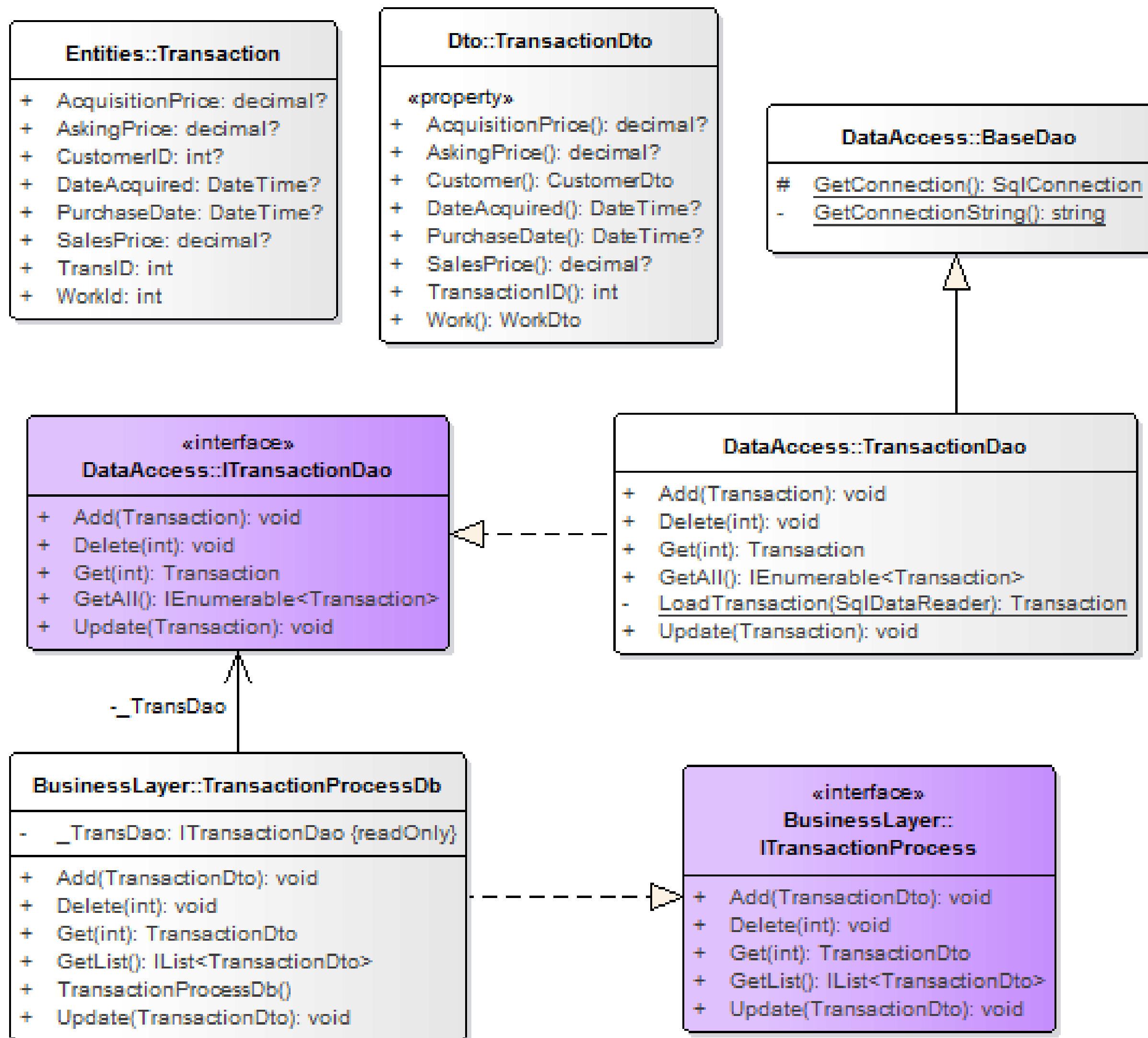

Рис. 5.2. Схема реализации сущности «Транзакция»

```
        <DataGridTextColumn Header=«Цена продажи» Binding=«{Binding Path=SalesPrice}»
/>
        <DataGridTextColumn Header=«Запрашиваемая цена» Binding=«{Binding
Path=AskingPrice}» />
    </DataGrid.Columns>
</DataGrid>
```

Обратите внимание на строки:

```
<DataGridTextColumn Header=«Дата приобретения» Binding=«{Binding Path=DateAcquired,
StringFormat=\{0:dd.MM.yyyy\}}» />
<DataGridTextColumn Header=«Дата продажи» Binding=«{Binding Path=PurchaseDate,
StringFormat=\{0:dd.MM.yyyy\}}» />
```

Параметр `StringFormat=\{0:dd.MM.yyyy\}` позволит корректно отображать дату в сетке данных.

Остается реализовать формы для работы с отдельными произведениями

и транзакциями, и связать их с соответствующими элементами управления на главном окне приложения.

5.4. Форма добавления и редактирования произведения

Для начала добавим в проект VRA новый компонент типа Window (окно) и назовем его AddWorkWindow. Это будет окно, в которое мы будем вводить данные о новом художественном произведении (Work).

Создание этого окна отдаем вам на самостоятельную работу. Оно должно получиться, как на рис.5.3. Как вы можете заметить, на форме есть два поля «Дата приобретения» и «Цена приобретения», которые не имеют прямого отношения к добавляемой работе. Однако это не совсем верно. Почему? Дело в том, как это упоминалось ранее, что при добавлении нового произведения в базу данных необходимо добавить и запись об операции приобретения этого произведения, а при добавлении транзакции эти два поля являются обязательными для заполнения. По этой причине мы и вынесли их на эту форму.

Поле «Дата приобретения» будет иметь следующую реализацию в xaml коде:

```
<DatePicker x:Name=«dpAcuired» Grid.Row=«4» Grid.Column=«1» Margin=«3» />
```

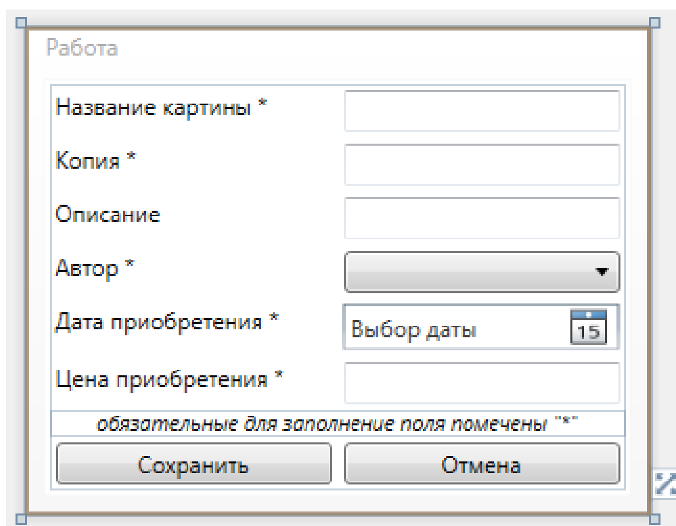

Рис. 5.3. Вид окна Работа

Опишем реализацию кода формы.

Для начала создадим три поля `Artists`, `FreeForSale` и `_id`, в которых будем хранить список всех имеющихся художников и картин, а также идентификатор редактируемой работы:

```
private  readonly IList<ArtistDto> Artists =
ProcessFactory.GetArtistProcess().GetList();

private IList<WorkDto> FreeForSale = ProcessFactory.GetWorkProcess().GetList();

private int _id;
```

В метод инициализации формы добавим строку, которая будет заполнять

комбобокс именами имеющихся художников:

```
public AddWorkWindow()
{
    InitializeComponent();

    this.cbArtist.ItemsSource = (from a in Artists orderby a.Name select
a).ToList<ArtistDto>();
}
```

Теперь следует метод загрузки формы для изменения выбранной картины. Он будет заполнять форму данными выбранного и переданного в него объекта:

```
public void Load(WorkDto work)
{
    if (work == null)
        return;

    this._id = work.Id;

    //Заполняем визуальные компоненты для отображения данных.
    tbTitle.Text = work.Title;
    tbCopy.Text = work.Copy ?? ««;
    tbDescription.Text = work.Description ?? ««;

    foreach (ArtistDto artist in Artists)
    {
        if (artist.Id == work.Artist.Id)
        {
            this.cbArtist.SelectedItem = artist;
            return;
        }
    }
}
```

Осталось лишь добавить события для кнопок «Сохранить» и «Отмена». Со второй кнопкой все просто, закрываем форму и все:

```
private void btnCancel_Click(object sender, RoutedEventArgs e)
{
    this.Close();
}
```

А вот с кнопкой «Сохранить» придется подумать. Для начала мы должны проверить заполнены ли обязательные поля:

```
private void btnSave_Click(object sender, RoutedEventArgs e)

{
      if (string.IsNullOrEmpty(tbTitle.Text))
      {
          MessageBox.Show(«Название работы не может быть пустым!»);
          return;
      }

      if (string.IsNullOrEmpty(tbCopy.Text))
      {
          MessageBox.Show(«Укажите копию работы!»);
          return;
      }

      if (string.IsNullOrEmpty(cbArtist.Text))
```

```
{
    MessageBox.Show(«Укажите автора работы!»);
    return;
}

if (string.IsNullOrEmpty(dpAcuired.Text))
{
    MessageBox.Show(«Укажите дату приобретения работы!»);
    return;
}

if (string.IsNullOrEmpty(tbAcquisitionPrice.Text))
{
    MessageBox.Show(«Укажите цену приобретения работы!»);
    return;
}
```

Затем создать новый объект WorkDto и присвоить атрибутам объекта значения соответствующих полей формы:

```
WorkDto work = new WorkDto
{
    Title = tbTitle.Text,
    Copy = tbCopy.Text,
    Description = tbDescription.Text,
    Artist = (ArtistDto) this.cbArtist.SelectedItem
};
```

И аналогично создать новую транзакцию:

```
TransactionDto transaction = new TransactionDto
{
    AcquisitionPrice = Convert.ToDecimal(tbAcquisitionPrice.Text),
    DateAcquired = Convert.ToDateTime(this.dpAcuired.Text)
};
```

После этого добавляем их в БД:

```
IWorkProcess workProcess = ProcessFactory.GetWorkProcess();
ITransactionProcess transProcess = ProcessFactory.GetTransactionProcess();

if (_id == 0)
{
                workProcess.Add(work);
                FreeForSale = ProcessFactory.GetWorkProcess().GetList();
                transaction.Work = FreeForSale.Last();
                transProcess.Add(transaction);
}
else
{
                work.Id = _id;
                workProcess.Update(work);
}
```

И закрываем форму:

```
    this.Close();
}
```

На этом реализация окна «Работы» завершена. Вам осталось только связать это окно с главным окном приложения, а мы приступаем к реализации окна «Транзакция».

5.6. Окно «Транзакция»

Аналогично добавим в проект VRA еще один компонент типа Window и назовем его AddTransactionWindow, в нем мы будем вводить данные о Транзакциях.

Должно получиться окно, как на рис. 5.4.

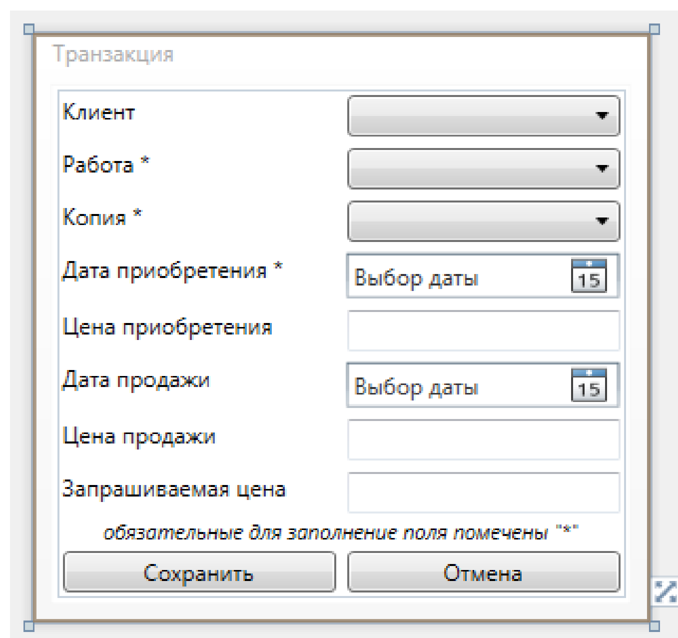

Рис. 5.4. Вид окна «Транзакция»

Сразу оговоримся, во время реализации нам будет необходимо получать список работ, доступных для продажи. Для этого нам придется опуститься на уровень Dao и в WorkDao добавить метод GetInGallery(), выполняющий эту функцию:

```
public IEnumerable<Work> GetInGallery()
{
    IList<Work> works = new List<Work>();

    using (var conn = GetConnection())
    {
        conn.Open();

        using (var cmd = conn.CreateCommand())
        {
            cmd.CommandText = «SELECT WorkID, ArtistID, Title, Copy, Description « +
«FROM WORK « + «WHERE WorkID in ( « + «SELECT WorkID « + «FROM TRANS « + «WHERE
CustomerID is NULL « + «)»;

            using (var dataReader = cmd.ExecuteReader())
            {
                while (dataReader.Read())
                {
                    works.Add(LoadWork(dataReader));
                }
            }
        }
    }
    return works;
```

```
}
```

В IWorkDao добавим строчку:

```
IEnumerable<Work> GetInGallery();
```

На уровне «бизнес-логики» - привычные два изменения. В классе WorkProcessDb появится метод:

```
public IList<WorkDto> GetListInGallery()
{
    return DtoConverter.Convert(_workDao.GetInGallery());
}
```

И в интерфейсе IWorkProcess:

```
IList<WorkDto> GetListInGallery();
```

Все теперь давайте реализуем это окно.

Аналогично предыдущему окну начнем с полей `Work`, `FreeForSale`, `FreeForPurchase`, `id` и `status`. Они будут хранить списки всех работ, работ доступных для продажи, работ доступных для покупки, идентификатор редактируемой операции и статус той таблицы, с которой мы работаем:

```
private readonly IList<WorkDto> Works = ProcessFactory.GetWorkProcess().GetList();

private readonly IList<WorkDto> FreeForSale =
ProcessFactory.GetWorkProcess().GetListInGallery();

private IList<WorkDto> FreeForPurchase = new List<WorkDto>();

private int id;

public string status;
```

В метод инициализации формы добавим строки, которые будут заполнять комбобоксы и задавать формат даты:

```
public AddTransactionWindow()
{
    //Эти строчки переводят формат даты в удобоваримый вид:
    CultureInfo ci =
CultureInfo.CreateSpecificCulture(CultureInfo.CurrentCulture.Name);
    ci.DateTimeFormat.ShortDatePattern = «dd-MM-yyyy»;
    Thread.CurrentThread.CurrentCulture = ci;

    InitializeComponent();

    IList<CustomerDto> customers = ProcessFactory.GetCustomerProcess().GetList();
    Works = ProcessFactory.GetWorkProcess().GetList();
    this.cbCustomer.ItemsSource = (from c in customers orderby c.Name select c);
    this.cbWork.ItemsSource = (from w in Works orderby w.Title select w);
    this.dpAcuired.IsTodayHighlighted = true;
}
```

Метод загрузки формы для изменения выбранной транзакции, будет заполнять форму данными выбранного и переданного в него объекта:

```
public void Load(TransactionDto Trans)
{
    this.Title = «Транзакция: Редактирование»;
    this.id = Trans.TransactionID;
```

```
    if (Trans.Customer != null)
    {
        this.cbCustomer.Text = Trans.Customer.Name;
    }

    if (Trans.Work.Copy != null)
    {
        this.cbCopy.Text = Trans.Work.Copy;
        this.cbWork.Text = Trans.Work.Title;
    }

    this.tbAcquisitionPrice.Text = Trans.AcquisitionPrice.ToString();
    this.tbAskingPrice.Text = Trans.AskingPrice.ToString();
    this.tbSalesPrice.Text = Trans.SalesPrice.ToString();
    this.dpAcuired.Text = Trans.DateAcquired.ToString();
    this.dpPurchase.Text = Trans.PurchaseDate.ToString();

    // Загружаем на форму данные работы участвующей в транзакции.
    this.loadWork(Trans.Work.Title);
}
```

Выше мы указали метод `loadWork()`, он находит в списке работ ту, название которой совпадает с переданным аргументом, и заполняет поле Копии данными найденной работы. Опишем его:

```
private void loadWork(string Title)
{
    this.cbCopy.Items.Clear();
    foreach (WorkDto work in Works)
    {
        if (work.Title == Title)
        {
            this.cbCopy.Items.Add(work);
        }
    }
    this.cbCopy.SelectedIndex = 0;
}
```

При работе с формой мы сможем выбирать работу из списка доступных. Будет удобно, если форма сама заполнит поля данными, связанными с этой работой. Для этого создадим два события `cbWork_LostFocus()` и `cbWork_SelectionChanged()`. Они возникают при взаимодействии с комбобоксом. В xaml коде формы запишите:

```
LostFocus=«cbWork_LostFocus» SelectionChanged=«cbWork_SelectionChanged»
```

Первое событие возникает при наведении стрелки на один из элементов списка. Оно исполняется всегда и заполняет данные по работе. Второе событие возникает при выборе элемента списка и выполняется только при продаже, так как в нем мы заполняем данные по транзакции, связанной с этой работой. Приведем эти методы:

```
private void cbWork_LostFocus(object sender, RoutedEventArgs e)
{
    this.loadWork(this.cbWork.Text);
}

private void cbWork_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (this.status == «sale»)
```

```
    {
        WorkDto work = cbWork.SelectedItem as WorkDto;
        if (work != null) this.id = FindTransaction(work.Id);
        loadTransaction(this.id);
    }
}
```

Как видно, в первом методе мы применяем уже знакомый нам `loadWork()`, а во втором появляются два новых метода `FindTransaction()` и `loadTransaction()`. Второй метод аналогичен `loadWork()`, он применяется в тех же целях, а `FindTransaction()` возвращает номер транзакции соответствующий номеру работы, которая была выбрана на форме. Это необходимо для установки идентификатора редактируемой транзакции. Приведем их:

```
private int FindTransaction(int workId)
{
    IList<TransactionDto> transes = ProcessFactory.GetTransactionProcess().GetList();
    foreach (TransactionDto t in transes)
    {
        if (t.Work.Id == workId & t.Customer == null)
        {
            return t.TransactionID;
        }
    }
    return -1;
}

private void loadTransaction(int transId)
{
    TransactionDto trans = ProcessFactory.GetTransactionProcess().Get(transId);
    this.dpAcuired.Text = trans.DateAcquired.ToString();
    this.dpAcuired.IsEnabled = false;
    this.tbAcquisitionPrice.Text = trans.AcquisitionPrice.ToString();
    this.tbAcquisitionPrice.IsEnabled = false;
    this.tbAskingPrice.Text = trans.AskingPrice.ToString();
    this.tbAskingPrice.IsEnabled = false;
}
```

Переходим к кнопкам «Сохранить» и «Отмена». Вторая реализуется ровно как на предыдущей форме, поэтому сразу переходим к первой. Последовательность действий в ней, как и на предыдущей форме, но проверок полей больше:

```
private void btnSave_Click(object sender, RoutedEventArgs e)
{
```

Проверяем, заполнены ли необходимые поля и параллельно заполняем объект `TransactionDto`:

```
    if (status == «sale»)
    {
        if (this.cbCustomer.SelectedIndex < 0)
        {
            MessageBox.Show(«Укажите клиента, которому продается картина!»); return;
        }
    }

    TransactionDto transaction = new TransactionDto();
    WorkDto SelectedWork = selectWork();

    if (SelectedWork == null)
```

```
{
    MessageBox.Show(«Картина должна быть выбрана!»); return;
}

if (status == «sale»)
{
    if (!workAtGalery(SelectedWork))
    {
        MessageBox.Show(«Запрашиваемая работа уже продана!»); return;
    }
}

if (status == «purchase»)
{
    if (workAtGalery(SelectedWork))
    {
        MessageBox.Show(«Запрашиваемая работа уже находится в галерее!»); return;
    }
}

transaction.Work = SelectedWork;

if (!string.IsNullOrEmpty(tbAcquisitionPrice.Text))
{
    try
    {
        transaction.AcquisitionPrice = Convert.ToDecimal(tbAcquisitionPrice.Text);
    }
    catch (Exception)
    {
        MessageBox.Show(«Введите корректную цену приобретения»); return;
    }
}

if (!string.IsNullOrEmpty(tbAskingPrice.Text))
{
    try
    {
       transaction.AskingPrice = Convert.ToDecimal(tbAskingPrice.Text);
    }
    catch (Exception)
    {
        MessageBox.Show(«Введите корректную запрашиваемую цену»); return;
    }
}

if (!string.IsNullOrEmpty(this.dpAcuired.Text))
{
    transaction.DateAcquired = Convert.ToDateTime(this.dpAcuired.Text);
}
else
{
    MessageBox.Show(«Дата приобретения должна быть указана!»); return;
}

if (!string.IsNullOrEmpty(this.dpPurchase.Text))
{
    if (Convert.ToDateTime(dpPurchase.Text) > Convert.ToDateTime(dpAcuired.Text))
        transaction.PurchaseDate = Convert.ToDateTime(this.dpPurchase.Text);
    else
    {
```

```
            MessageBox.Show(«Нельзя продать работу раньше, чем её купила галерея!
Проверьте правильность ввода данных.»);
            return;
        }
    }

    if (!string.IsNullOrEmpty(cbCustomer.Text))
    {
        CustomerDto SelectedCustomer = (CustomerDto)this.cbCustomer.SelectedItem;
        transaction.Customer = SelectedCustomer;
    }

    if (!string.IsNullOrEmpty(tbSalesPrice.Text))
    {
        try
        {
            if (Convert.ToDecimal(tbSalesPrice.Text) >= 30000 &&
Convert.ToDecimal(tbSalesPrice.Text) <= 1500000)
                transaction.SalesPrice = Convert.ToDecimal(tbSalesPrice.Text);
            else
            {
                MessageBox.Show(«Продажа может проходить только в пределах от 30 тыс.
у.е. до 1,5 млн. у.е.»);
                return;
            }
        }
        catch (Exception)
        {
    MessageBox.Show(«Неверный формат данных при операции с ценой продажи»); return;
        }
    }
```

Добавляем заполненный `TransactionDto` объект в БД:

```
    ITransactionProcess transProcess = ProcessFactory.GetTransactionProcess();

    if (id == 0)
        transProcess.Add(transaction);
    else
    {
        transaction.TransactionID = id;
        transProcess.Update(transaction);
    }
```

И закрываем форму:

```
    this.Close();
}
```

Вы могли заметить, что мы использовали два новых метода. Один из них – это `WorkAtGalery()`, он проверяет, содержится ли выбранная художественная работа в списке работ, доступных для продажи:

```
private bool workAtGalery(WorkDto work)
{
    return FreeForSale.Contains(work);
}
```

Другой – это `selectWork()`, он возвращает работу, в которой Title и Copy соответствуют выбранным на форме:

```
private WorkDto selectWork()
{
```

```
    WorkDto work = null;
    foreach (WorkDto w in Works)
    {
        if (w.Copy == this.cbCopy.Text && w.Title == this.cbWork.Text)
        { work = w; break; }
    }
    return work;
}
```

По условию задачи вместо простого действия добавления транзакции будем использовать два действия: приобретение и продажа работы. Это в свою очередь повлечет небольшие дополнения, так как при реализации этих действий нам необходимо сделать недоступными для изменения поля, которые не соответствуют выполняемому действию. Для этого добавим метод загрузки формы в зависимости от `status`:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    if (status == «purchase»)
    {
        GetWorksWithCustomers();
        this.cbWork.ItemsSource = this.FreeForPurchase;
        this.cbCustomer.IsEnabled = false;
        this.tbSalesPrice.IsEnabled = false;
        this.dpPurchase.IsEnabled = false;
    }

    if (status == «sale»)
    {
        this.cbWork.ItemsSource = FreeForSale;
    }
}
```

Наверно, вы заметили, что в методе загрузки формы используется еще один новый метод `GetWorksWithCustomers()`. Его задача – получение списка проданных ранее работ для повторного их приобретения:

```
private void GetWorksWithCustomers()
{
    IEnumerable<WorkDto> forPurchase = Works.Except(FreeForSale);
    FreeForPurchase = forPurchase.ToList();
}
```

Обратите внимание, что для получения такого списка мы применили теоретико-множественную операцию разности двух множеств: из списка всех работ, представленных коллекцией `Works`, вычитаем список работ на продажу `FreeForSale`. Эта операция реализуется методом `IEnumerable.Except`. Для его корректной работы необходимо добавить в класс `WorkDto` строчки, которые укажут компилятору, что работы нужно сравнивать по Id:

```
public override bool Equals(object obj)
        {
            if (obj == null || GetType() != obj.GetType())
                return false;
            WorkDto workItem = obj as WorkDto;
            return workItem != null && workItem.Id.Equals(this.Id);
        }

        public override int GetHashCode()
```

```
{
    return this.Id.GetHashCode();
}
```

Метод `Window_Loaded()` мы должны присвоить самому окну в xaml коде:

```
Loaded=«Window_Loaded»
```

Теперь перейдем на главную форму и проясним пару моментов. Так как у нас не будет кнопки «Добавить» для транзакции, то мы создадим аналогично ей две другие «Купить» и «Продать», как уже говорилось ранее. Для обработки события нажатия на каждую из этих кнопок, добавим следующие методы, соответственно:

```
private void btnPurchase_Click(object sender, RoutedEventArgs e)
{
    switch (status)
    {
        case «Trans»:
            AddTransactionWindow window = new AddTransactionWindow {status =
«purchase»};
            window.ShowDialog();
            btnRefreshT_Click();
            break;

        default: MessageBox.Show(«Необходимо выбрать таблицу, Транзакции!»); return;
    }
}

private void btnSale_Click(object sender, RoutedEventArgs e)
{
    switch (status)
    {
        case «Trans»:
            AddTransactionWindow window = new AddTransactionWindow {status = «sale»};
            window.ShowDialog();
            btnRefreshT_Click();
            break;

        default: MessageBox.Show(«Необходимо выбрать таблицу, Транзакции!»); return;
    }
}
```

Как вы можете заметить, при создании объекта `window` мы сразу же определяем статус формы: `status == «purchase»` или `status == «sale»`. При этом и выполняется метод `Window_Loaded()`. Остальные кнопки действий будут реализовываться как обычно.

Все, теперь свяжите это окно с основным. После чего можете попробовать запустить приложение и проверить его работу.

Контрольные вопросы и задания

1. Свяжите окно Работы и Транзакции с главным окном.
2. В чем отличие классов DTO от классов Entity?
3. Что такое свойство? В чем его отличие от атрибута?
4. Какие свойства бывают?
5. Что такое триггер?

6. Зачем используются методы-конверторы, какую задачу они решают и какое преимущество дают?
7. Что такое binding?
8. Какие виджеты (элементы управления) используются при реализации формы добавления и редактирования произведения?
9. В каком случае использование выпадающих списков (combobox) не целесообразно? Что следует использовать в этом случае?
10. Опишите алгоритм получения списка работ, доступных для продажи.
11. Каким образом можно узнать, что картина продана?

Лабораторная работа №6
Найдется все. Как организовать поиск и отбор

6.1. Постановка задачи

В данной работе нам предстоит реализовать возможность поиска объекта каждой сущности по определенным для нее параметрам. В первую очередь для реализации поиска необходимо определиться, какие это будут параметры. Собственно параметры поиска определяются исходя из требований, предъявляемых заказчиком.

Сама реализация – дело техники, причем, как всегда, при использовании гибких инструментов у программистов есть различные варианты. Например, поиск можно производить, выполняя запрос к базе данных, или производить поиск по коллекции объектов при помощи LINQ прямо на уровне приложения. Эти два способа мы и попробуем вам объяснить. Как всегда, призываем вас решать данную задачу самостоятельно, используя более удобный на ваш взгляд способ.

6.2. Реализация формы поиска

Форм поиска у приложения может быть несколько, ведь у каждой сущности, которую предполагается искать в БД, параметры поиска различны. Мы для этой цели используем виджет `TabControl`.

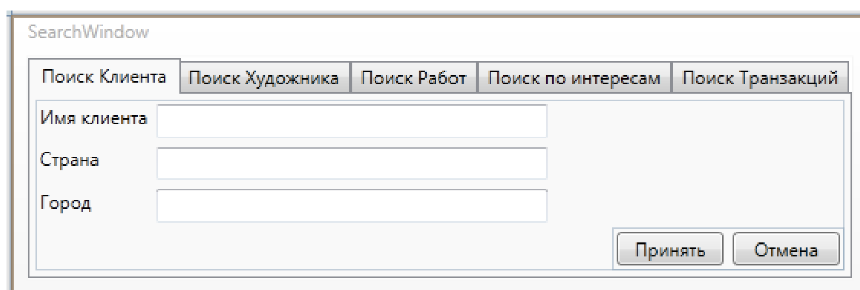

Рис.6.1. Размещение виджета TabControl на форме

При этом видимость каждого «ушка» (правильно именуемого `TabItem`) настраивается в зависимости от состояния главного окна приложения, из которого вызвана форма поиска. Так, например, для транзакции окно будет выглядеть, как на рис.6.2:

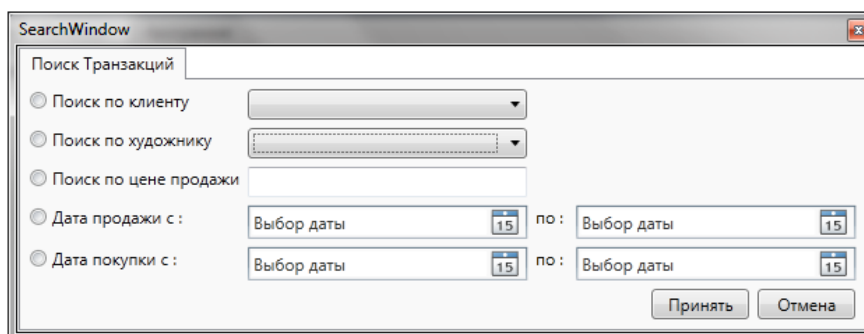

Рис.6.2. Вид формы поиска по транзакции

Можно реализовать поиск по каждой таблице в отдельном окне. Существуют даже специальные развитые компоненты для организации поиска, обеспечивающие самые широкие возможности для разработчика. Так или иначе, но в каждом конкретном случае решение о способе реализации остается за программистом.

К формам мы еще вернемся, теперь перейдем к вариантам реализации поиска.

6.3. Поиск запросом к БД

В качестве примера рассмотрим поиск художников в базе данных, который будем выполнять по имени художника и/или его национальности. SQL запрос для такого отбора записей будет выглядеть следующим образом:

```
SELECT ArtistID, Name, BirthYear, DeceaseYear, Artist.NatID FROM ARTIST JOIN Nation on
Artist.NatID = Nation.NatID WHERE Name like @Name AND Value like @Nation
```

Остается дополнить класс `ArtistDao` соответствующим методом:

```
public IList<Artist> SearchArtists(string Name, string Nation)
{
    IList<Artist> artists = new List<Artist>();
    using (var conn = GetConnection())
    {
        conn.Open();
        using (var cmd = conn.CreateCommand())
        {
            cmd.CommandText = «SELECT ArtistID, Name, BirthYear, DeceaseYear,
Artist.NatID FROM ARTIST JOIN Nation on Artist.NatID = Nation.NatID WHERE Name like
@Name AND Value like @Nation»;
            cmd.Parameters.AddWithValue(«@Name»,»%»+Name+»%»);
            cmd.Parameters.AddWithValue(«@Nation», «%»+Nation);
            using (var dataReader = cmd.ExecuteReader())
            {
                while (dataReader.Read())
                {
                    artists.Add(LoadArtist(dataReader));
                }
            }
        }
    }
    return artists;
}
```

Этот метод очень похож на метод `GetList()`.

Также не забудем объявить метод в интерфейсе `IArtistDao`:

```
IEnumerable<Artist> SearchArtists(string Name, string Nation);
```

В соответствии с выбранным шаблоном проектирования на уровне “бизнес-логики” в классе `ArtistProcessDb` дописываем следующий метод:

```
public IList<ArtistDto> SearchArtist(string Name, string Nation)
{
    return DtoConverter.Convert(_artistDao.SearchArtists(Name,Nation));
}
```

В интерфейсе `IArtistProcess` декларируем его:

```
IList<ArtistDto> SearchArtist(string Name, string Nation);
```

Теперь ничего не мешает реализации формы поиска.

6.4. Реализация формы поиска

Как уже говорилось выше, вы можете реализовывать окна поиска, как пожелаете. Мы же опишем более удобную, на наш взгляд, реализацию через TabControl. Ниже приведен XAML код реализации TabItem для поиска художников:

```
<TabItem x:Name =«sArtist» Header=«Поиск Художника»>
    <Grid>
        <Grid.RowDefinitions>

            <RowDefinition Height=«auto» />
            <RowDefinition Height=«auto» />
            <RowDefinition Height=«auto» />
            <RowDefinition Height=«auto» />
            <RowDefinition Height=«auto» />
            <RowDefinition Height=«auto» />

        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>

            <ColumnDefinition Width=«auto» />
            <ColumnDefinition Width=«*» />

        </Grid.ColumnDefinitions>

        <TextBlock Grid.Row=«0» Grid.Column =«0» Margin=«3» Text=«Имя художника»/>
        <TextBlock Grid.Row=«1» Grid.Column =«0» Margin =«3» Text=«Национальность»/>

        <TextBox Name=«ArtistName» Grid.Row =«0» Grid.Column=«1» Margin=«3» Width=«200»
HorizontalAlignment=«Left»/>
        <ComboBox Name=«cbArtistCountry» ItemsSource=«{Binding}»
DisplayMemberPath=«Nationality» Grid.Row=«1» Grid.Column=«1» Margin=«3» Width=«200»
HorizontalAlignment=«Left»/>

        <StackPanel Grid.Row=«3» Grid.Column=«1» Orientation=«Horizontal»
HorizontalAlignment=«Right»>
            <Button x:Name =«btnSearchArtist» Content=«Принять» Margin=«3» Width=«70»
Click=«SearchArtist» />
            <Button x:Name =«btnCancelA» Content=«Отмена» Margin=«3» Width=«70»
Click=«CloseForm»/>
        </StackPanel>
    </Grid>
</TabItem>
```

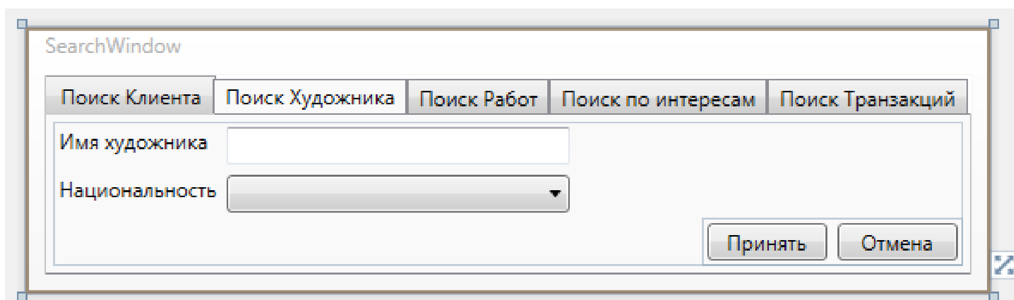

Рис.6.2. Вид формы поиска Художников

Теперь опишем реализацию кода управления формой.

Для начала получим список всех имеющихся художников и их национальностей, а также поле, в котором мы будем сохранять результат поиска (отбора) `AllowArtists`, `AllowNations` и `FindedArtists` соответственно:

```
private readonly IList<ArtistDto> AllowArtists =
ProcessFactory.GetArtistProcess().GetList();

private readonly IList<NationDto> AllowNations =
ProcessFactory.GetNationProcess().GetList();

public IList<ArtistDto> FindedArtists;
```

Создадим булеву переменную `exec`, по ней позже будем определять, выполнена ли команда поиска или нажата кнопка «Отмена».

```
public bool exec;
```

Перейдем к реализации загрузки формы (вида, в котором она будет отображаться). Для этого приведем метод `SearchWindow()` к виду:

```
public SearchWindow(string status)
{
    InitializeComponent();

    this.cbArtistCountry.ItemsSource = AllowNations;

    switch (status)
    {
        case «Artist»:
            this.SearchTab.SelectedIndex = 1;
            this.sCustomer.Visibility = Visibility.Collapsed;
            this.sWork.Visibility = Visibility.Collapsed;
            this.tiInterests.Visibility = Visibility.Collapsed;
            this.sTransaction.Visibility = Visibility.Collapsed;
            break;
    }
}
```

Переменную `status` мы будем передавать из главного окна при вызове окна поиска. Эта переменная содержит имя таблицы, в которой мы находимся в главной форме. Для реализации окна поиска для других объектов вам будет необходимо лишь добавить недостающие кейсы. Также вы можете заметить, что мы сразу заполняем комбо-бокс необходимыми данными (списком национальностей).

Осталось описать события для кнопок «Принять» и «Отмена».

При нажатии на «Отмена» просто закрываем форму:

```
private void CloseForm(object sender, RoutedEventArgs e)
{
    this.Close();
}
```

А при нажатии на кнопку «Принять» происходит несколько действий. Сначала выполняется отбор художников, удовлетворяющих заданному параметру, с помощью нашего метода `SearchArtist`, результат сохраняется в коллекции `FindedArtists`, а переменной `exec` присваивается значение `true`. После чего форма поиска закрывается.

```
private void SearchArtist(object sender, RoutedEventArgs e)
{
    this.FindedArtists =
ProcessFactory.GetArtistProcess().SearchArtist(this.ArtistName.Text,
this.cbArtistCountry.Text);
    this.exec = true;
    this.Close();
}
```

На этом реализация формы поиска художника завершена. Остается лишь описать взаимодействие ее с главной формой.

6.5. Обмен данными между формами

На главной форме добавим новую кнопку действия «Поиск»:

```
<Button Height=«52» Width=«70» HorizontalAlignment=«Left» Name=«btnSearch»
VerticalAlignment=«Top» Click =«btnSearchClick» >
    <Button.Content>
        <StackPanel>
            <Image Width=«30» Source=«/VRA;component/Images/Search.ico»></Image>
            <TextBlock HorizontalAlignment=«Center» Margin=«1»
FontSize=«10»>Поиск</TextBlock>
        </StackPanel>
    </Button.Content>
    <Button.ToolTip>Поиск данных</Button.ToolTip>
</Button>
```

Для нее мы реализуем событие `btnSearchClick()`:

```
private void btnSearchClick(object sender, RoutedEventArgs e)
{
    SearchWindow search = new SearchWindow(status);
    {
        switch (status)
        {
            case «Artist»:
                search.ShowDialog();
                if (search.exec)
                {
                    this.dgArtists.ItemsSource = search.FindedArtists;
                }
                break;

            default: MessageBox.Show(«Для поиска необходимо выбрать таблицу!»); break;
        }
    }
}
```

Для начала мы создаем переменную типа `SearchWindow` для общения с формой поиска, и сразу же передаем в форму ту самую переменную `status`:

```
SearchWindow search = new SearchWindow(status);
```

где `status` - обозначает текущее состояние главной формы, т.е. ту таблицу, в которой происходит поиск.

Статус передается конструктору формы поиска. Это необходимо для того, чтобы окно поиска «знало», какой `TabItem` ему отрисовывать. Далее открываем окно:

```
search.ShowDialog();
```

и ожидаем закрытия формы поиска. После ее закрытия формы поиска проверяем, был ли выполнен поиск. Это легко узнать, используя значение логической переменной exec.

```
if (search.exec)
{
    this.dgArtists.ItemsSource = search.FindedArtists;
}
```

Строкой `this.dgArtists.ItemsSource = search.FindedArtists;` мы заполняем дата-грид полученными в результате поиска данными.

Для работы с другими объектами вам также необходимо лишь добавить соответствующие кейсы.

На этом реализация поиска художника завершена. Запустите программу и проверьте ее работоспособность.

6.6. Поиск на уровне приложения

Как уже было сказано, кроме вышеописанного способа существует возможность производить поиск по коллекции объектов при помощи LINQ.

Мы реализуем запрос к коллекции полученных элементов прямо на форме поиска, хотя это конечно лучше вынести на уровень “бизнес-логики”. Так или иначе, общий принцип поиска останется неизменным.

На примере «Интересов» посмотрим, как это будет выглядеть.

Оговоримся, что мы опишем лишь сам метод, аналогичный SearchArtist(), реализация всего остального, связанного с поиском интересов, не представляет собой ничего нового и была описана выше на примере поиска художников.

Для начала нам необходимо получить полный список интересов покупателей. Делать такие вещи мы уже умеем.

```
IEnumerable<InterestsDto> AllInterests =ProcessFactory.GetInterestsProcess().GetList();
```

Теперь получим интересующие нас «Интересы» покупателей:

```
IEnumerable<InterestsDto> interests = from I in AllInterests where
(I.Artist.Contains(this.tbIArtistName.Text) &&
I.Customer.Contains(this.tbICustomerName.Text)) select I;
```

Так выглядит LINQ запрос к коллекции `AllInterests`, эквивалентный SQL запросу:

```
select *
from dbo.CustomerInterests
where Customer like '%@name0%' and Artist like '%@name1%'
```

Далее аналогично поиску художников мы присваиваем результат в `FindedInterests` (создайте это поле аналогично `FindedArtists`). Затем присваиваем переменной exec значение `true` и закрываем форму.

Полностью метод будет выглядеть так:

```
private void SearchInterests(object sender, RoutedEventArgs e)
{
    IEnumerable<InterestsDto> AllInterests =
ProcessFactory.GetInterestsProcess().GetList();

    IEnumerable<InterestsDto> interests = from I in AllInterests Where
```

```
(I.Artist.Contains(this.tbIArtistName.Text) &&
I.Customer.Contains(this.tbICustomerName.Text)) select I;

    this.FindedInterests = interests.ToList();
    this.exec = true;
    this.Close();
}
```

На этом реализация метода поиска «Интересов» на уровне приложения завершена. Завершите реализацию самого окна и свяжите его с главным окном, после чего запустите программу и проверьте ее работоспособность.

Контрольные вопросы и задания

1. Реализуйте окно поиска.
2. Реализуйте поиск для Клиента и Работы через запрос к БД.
3. Реализуйте поиск Транзакций на уровне приложения.
4. Что такое LINQ?
5. Перечислите способы организации поиска. В чем их отличие? В каких случаях более целесообразно применять тот или иной вариант организации поиска?

Лабораторная работа №7
Отчитайся. Выгрузка данных в формате Xlsx и Html

7.1. Постановка задачи

Согласно требованиям к нашему приложению, необходимо иметь возможность выгружать данные таблиц для дальнейшего их использования: составления списков, формирования отчетов заданной формы, выполнения аналитической работы и т.п. С этой целью довольно часто используется экспорт данных в виде таблицы Excel.

Также нам необходимо иметь возможность формировать список картин, выставленных на продажу, для его публикации на сайте галереи. Для реализации этой задачи создадим отчет «Прайс-лист» в html-формате. Это позволит сразу опубликовать его на сайте галереи.

7.2. Реализация экспорта данных в таблицу Excel

Для работы с отчетом создадим пункт «Отчеты» в меню главного приложения и добавим два подпункта. Возможно, вы уже добавили это меню при разработке главного окна приложения. Результат изображен в виде xaml кода:

```
<Menu Height=«25» HorizontalAlignment=«Stretch» Name=«mainMenu» VerticalAlignment=«Top»
Grid.ColumnSpan=«3»>
      <MenuItem Name=«Reports» Header=«Отчеты»>
            <MenuItem Name=«ExcelExporterButton» Header=«Экспорт таблицы Excel»
Click=«ExcelExporterButton_Click» />
            <MenuItem Name=«HtmlWorksInGalleryButton» Header=«Прайс-лист работ»
Click=«HtmlWorksInGalleryButton_Click» />
      </MenuItem>
</Menu>
```

Для реализации задачи экспорта данных в таблицу Excel нам потребуется установить пакет EPPlus. Самый простой способ сделать это, использовать менеджер пакетов NuGet. В версиях Visual Studio старше 2010 NuGet установлен по умолчанию. Для VS 2010 и ниже необходимо установить NuGet соответствующей версии VS, скачав его с официального сайта http://www.nuget.org.

После этого перейдите к проекту VRA.BusnessLayer и вызовите контекстное меню, нажав правой кнопкой мыши по папке «Ссылки» (References). В контекстном меню выберите пункт «Управление пакетами NuGet…» (Manage NuGet packages …). В появившемся окне менеджера пакетов перейдите в пункт «В сети» (Online), в строке поиска введите EPPlus и установите найденный пакет.

Оставаясь на уровне «бизнес-логики», создадим интерфейс `IReportGenerator` и класс `ReportGenerator`, реализующий задачу выгрузки данных в заданном формате (рис. 7.1).

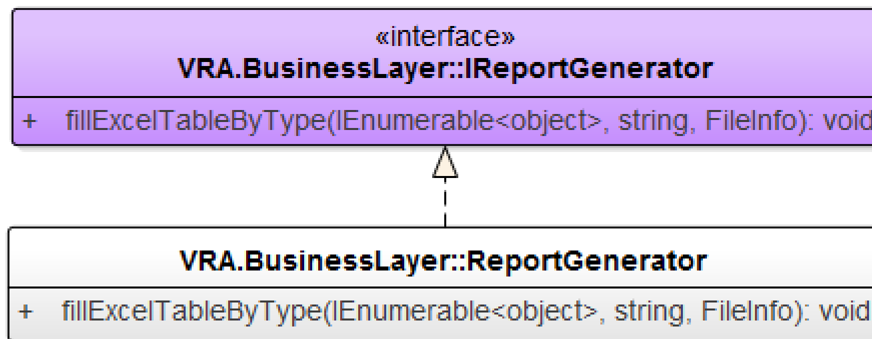

Рис. 7.1. Структура для реализации выгрузки данных в таблицу Excel

Добавим метод `fillExcelTableByType()`, который отвечает за выгрузку данных в виде Excel таблицы, и опишем его реализацию.

Вначале задаем формат шрифта будущего отчета: жирный шрифт для заголовка, обычный для остальных ячеек. Задаем выравнивание текста в ячейках и разделитель для дробной части чисел.

```
public void fillExcelTableByType(IEnumerable<object> grid, string status, FileInfo
xlsxFile)
{
    if (grid != null)
    {
        ExcelPackage pck = new ExcelPackage(xlsxFile);
        var excel = pck.Workbook.Worksheets.Add(status);
        int x = 1;
        int y = 1;

        // Устанавливает фиксированный десятичный разделитель (нужно для верного
распознавания типа данных Excel'ем).
        CultureInfo cultureInfo = new
CultureInfo(Thread.CurrentThread.CurrentCulture.Name);
        Thread.CurrentThread.CurrentCulture = cultureInfo;
        cultureInfo.NumberFormat.NumberDecimalSeparator = «.»;
        // Первая строка (шапка таблицы) – жирным стилем.
        excel.Cells[«A1:Z1»].Style.Font.Bold = true;
        // Выравнивание текста в ячейках – по левому краю.
        excel.Cells.Style.HorizontalAlignment = ExcelHorizontalAlignment.Left;
        // Устанавливает формат ячеек.
        excel.Cells.Style.Numberformat.Format = «General»;
```

Далее получаем объект DTO, соответствующий загруженной в grid таблице. Свойства (Properties) объекта используются для формирования заголовка таблицы в результирующем файле и заполнения ее данными. При этом последовательность полей grid соответствует последовательности свойств (property) в соответствующем объекте DTO.

```
        // Пустой объект для получения списка property.
        Object dtObj = new Object();

        switch (status)
        {
            case «Customer»:
                dtObj = new CustomerDto();
                break;
```

```
            case «Artist»:
                dtObj = new ArtistDto();
                break;
            case «Work»:
                dtObj = new WorkDto();
                break;
            case «Trans»:
                dtObj = new TransactionDto();
                break;
            case «Interests»:
                dtObj = new InterestsDto();
                break;
            case «Nations»:
                dtObj = new NationDto();
                break;
        }
        // Генерация шапки таблицы.
        foreach (var prop in dtObj.GetType().GetProperties())
        {
            excel.Cells[y, x].Value = prop.Name.Trim();
            x++;
        }
        // Генерация строк-записей таблицы.
        foreach (var item in grid)
        {
            y++;
            // Объект-контейнер для текущего читаемого элемента.
            Object itemObj = item;
            x = 1;
            foreach (var prop in itemObj.GetType().GetProperties())
            {
                object t = prop.GetValue(itemObj, null);
                object val;

                if (t == null)
                    val = ««;
                else
                {
                    val = t.ToString();
                    // Если тип сложный, то вытаскиваем нужное поле.
                    if (t is NationDto)
                        val = ((NationDto) t).Nationality;
                    if (t is ArtistDto)
                        val = ((ArtistDto) t).Name;
                    if (t is CustomerDto)
                        val = ((CustomerDto) t).Name;
                    if (t is WorkDto)
                        val = ((WorkDto) t).Title;
                }
                excel.Cells[y, x].Value = val;
                x++;
            }
        }
        // Устанавливаем размер колонок по ширине содержимого.
        excel.Cells.AutoFitColumns();
        // Сохраняем файл.
        pck.Save();
    }
    else MessageBox.Show(«Данные не загружены!»);
}
```

Завершая реализацию метода, не забудьте добавить в `using` несколько строчек:

```
using System.IO;
using System.Globalization;
using System.Threading;
using System.Windows;
using OfficeOpenXml;
using OfficeOpenXml.Style;
using VRA.Dto;
```

В фабрику процессов `ProcessFactory` следует добавить метод:

```
public static IReportGenerator GetReport()
{
    return new ReportGenerator();
}
```

Перейдем на уровень логики представления и в обработчике главного окна приложения MainWindow.xaml.cs добавим метод `ExcelExporterButton_Click()`, который отвечает за получение данных из активной сетки данных (grid), создание результирующего файла и передачу их методу `fillExcelTableByType()`.

Источником данных нам послужат сетки данных (таблицы) главного окна. Благодаря этому нам не придется опускаться на уровень доступа к данным, достаточно лишь получить данные из активной таблицы. Существует множество способов это сделать. Мы предлагаем сделать это традиционно, используя блок `switch`, передавая в него `status` активной сетки данных.

```
private void ExcelExporterButton_Click(object sender, RoutedEventArgs e)
{
    List<object> grid = null;

    switch (status)
    {
        case «Customer»:
            grid = this.dgCustomers.ItemsSource.Cast<object>().ToList();
            break;
        case «Artist»:
            grid = this.dgArtists.ItemsSource.Cast<object>().ToList();
            break;
        case «Work»:
            grid = this.dgWork.ItemsSource.Cast<object>().ToList();
            break;
        case «Trans»:
            grid = this.dgTrans.ItemsSource.Cast<object>().ToList();
            break;
        case «Interests»:
            grid = this.dgInterests.ItemsSource.Cast<object>().ToList();
            break;
        case «Nations»:
            grid = this.dgNations.ItemsSource.Cast<object>().ToList();
            break;
    }
```

Далее создаем объект, который отвечает за создание результирующего файла.

```
    SaveFileDialog saveXlsxDialog = new SaveFileDialog
```

```
    {
        DefaultExt = «.xlsx»,
        Filter = «Excel Files (.xlsx)|*.xlsx»,
        AddExtension = true,
        FileName = status
    };
```

Запускаем окно для выбора директории сохранения результирующего файла и после нажатия кнопки «Сохранить» заполняем его данными, выполняя метод `fillExcelTableByType()`.

```
    bool? result = saveXlsxDialog.ShowDialog();

    if (result == true)
    {
        FileInfo xlsxFile = new FileInfo(saveXlsxDialog.FileName);

        ProcessFactory.GetReport().fillExcelTableByType(grid, status, xlsxFile);
    }
}
```

Реализация первой задачи закончена. Остается запустить приложение, выбрать таблицу и убедиться, что экспорт данных работает правильно.

7.3. Реализация отчета «Прайс-лист»

Напомним, что необходимо иметь возможность формировать список картин, выставленных на продажу, для его публикации на сайте галереи.

Вначале подготовим шаблон для отчета на языке html. Для этого создайте текстовый файл, скопируйте в него указанный ниже код и назовите файл, например, vra.vrt.

```
<!-- THIS IS VIEW RIDGE ASSISTANT REPORT HTML TEMPLATE --!>
<meta http-equiv=«Content-Type» content=«text/html;charset=utf-8»>
<html><head>
<style type=«text/css»>
td { font-family: Verdana, Arial; font-size: 12; font-weight: normal; color: #303030;
border:1px solid gray; padding: 5px 5px 5px 5px;}
td.light { font-family: Verdana, Arial; font-size: 12; font-weight: normal; color:
#303030; border:1px solid gray;padding: 5px 5px 5px 5px;}
html,body{ height:100%; background-color: #FFFFFF; }
</style>
</head>

<body>
<br>
      <div style=«text-align: center;»>
            <table align=«center» style=«position:absolute;border:1px solid
            black;width:80%;»>
            [VRA_TABLE_REPORT]
            </table>
      </div>
</body>
</html>
```

Этот шаблон мы будем подгружать в процессе созданий прайс-листа.

При создании прайс-листа нам потребуется получить список картин, доступных для продажи. Как вы помните, мы уже решали эту задачу в лабораторной работе №5, когда организовывали процесс продажи

художественных произведений. В принципе можно использовать этот метод для создания отчета «Прайс-лист». Единственная проблема: этот метод возвращает список объектов типа Work, но цена у нас хранится в объектах типа Transaction.

Самый простой способ решения этой проблемы: создать в базе данных специальное представление и использовать принятый в данном проекте шаблон проектирования.

Итак, добавьте в базу данных новое представление. Пусть оно будет иметь следующее имя: WorksInGallery. Это представление должно содержать картины, доступные для продажи. Нахождение картины в галереи определяется истинностью условия `CustomerID Is Null`. Его структура представлена на рис. 7.2

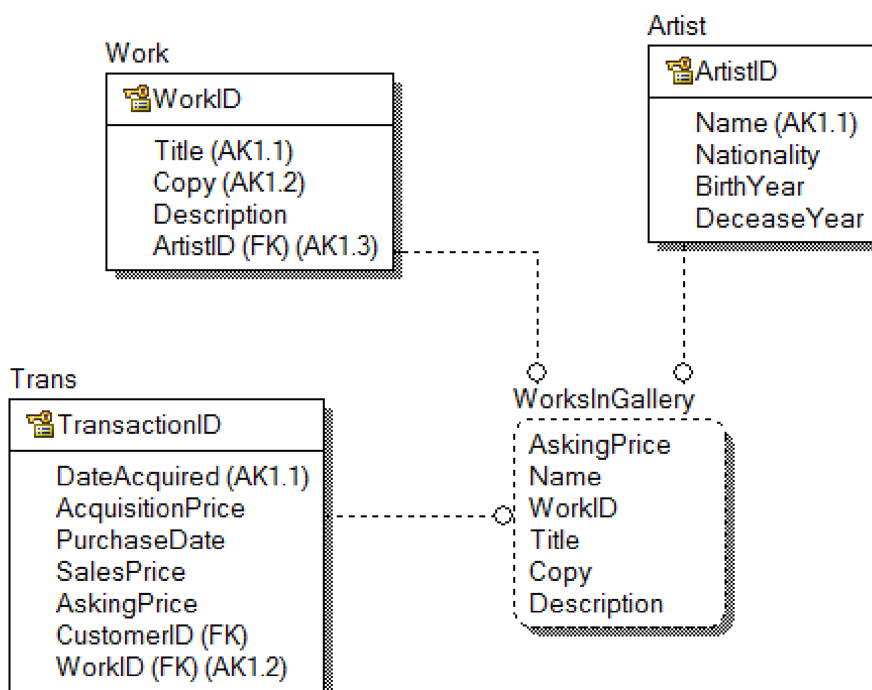

Рис. 7.2. Структура представления WorksInGallery

Теперь нам необходимо организовать работу с этим представлением в приложении. Общая схема реализации работы с представлением показана на рис. 7.3.

Если следовать диаграмме, необходимо создать классы `WorkInGallery`, `WorkInGalleryDto`, `WorkInGalleryDao`, `WorkInGalleryProcessDb` и интерфейсы `IWorkInGalleryDao` и `IWorkInGalleryProcess`.

С `WorkInGallery`, `WorkInGalleryDto` все очень просто. С интерфейсом `IWorkInGalleryDao` также не должно быть никаких проблем. В классе `WorkInGalleryDao` будут реализованы всего два уже привычных метода `LoadWork()` и `GetAll()`. Запрос к БД в методе `GetAll()` будет выглядеть так:

```
«SELECT WorkID, Title, Copy, Name, AskingPrice, Description FROM WorksInGallery»
```

В классе `DaoFactory` не забудьте добавить метод, отвечающий за создание объектов класса `WorkInGalleryDao,` а в классе `ProcessFactory –` за создание

объектов класса `WorkInGalleryProcessDb`.

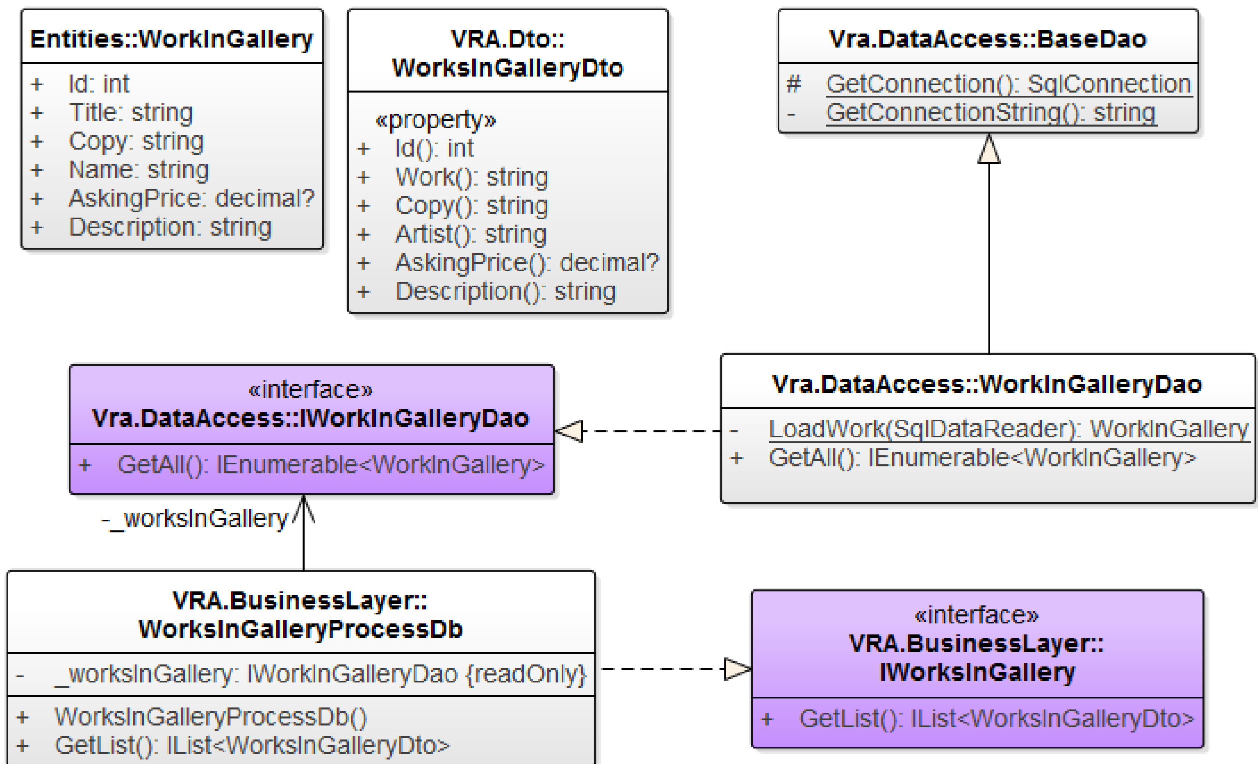

Рис. 7.3. Схема реализации работы с представлением WorksInGallery

Также не забудьте добавить методы-конвертеры для `WorkInGallery` и `WorkInGalleryDto`:

```
private static WorksInGalleryDto Convert(WorkInGallery worksInGallery){}
public static IList<WorksInGalleryDto> Convert(IEnumerable<WorkInGallery> worksInGallery){}
```

На этом реализация объекта WorksInGallery завершена. Попробуйте разобраться и применить этот объект в работе №5, а мы переходим к организации работы по формированию прайс-листа.

В класс `ReportGenerator` добавляем метод, отвечающий за формирование прайс-листа (рис. 7.4).

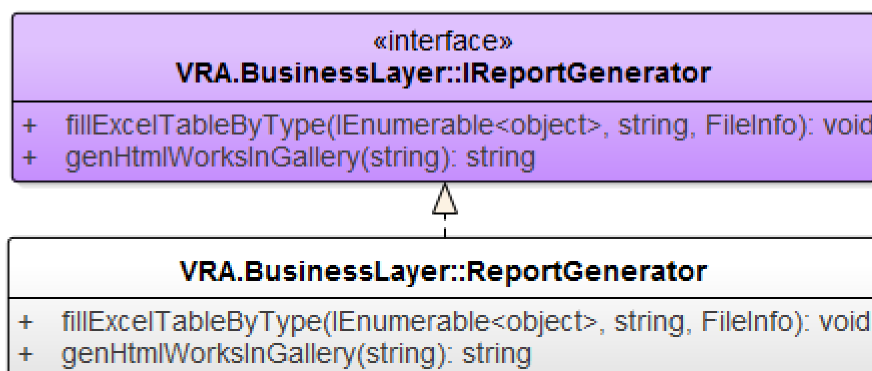

Рис. 7.4. Структура для реализации прайс-листа

Ниже представлена реализация метода `genHtmlWorksInGallery()`.

```
public string genHtmlWorksInGallery(string rep)
{
    // Получаем список работ на продажу.
    List<object> works =
ProcessFactory.GerWorksInGalleryProcess().GetList().Cast<object>().ToList();
    // Начинаем заполнять строку html кода. Для начала строка с заголовками таблицы.
    string res_html =
«<tr><td><b>Код</b></td><td><b>Название</b></td><td><b>Художник</b></td><td><b>Цена</b>
</td><td><b>Описание</b></td></tr>«;

    // Заполняет таблицу объектами.
    foreach (var work in works)
    {
        WorksInGalleryDto WorkItem = (WorksInGalleryDto) work;
        res_html += «<tr><td><p>« + WorkItem.Id + «</p></td>«;
        // Если заполнено поле «Копия», то дописываем его к имени в скобках.
        res_html += WorkItem.Copy != string.Empty
            ? «<td><p>« + WorkItem.Work + « (« + WorkItem.Copy + «)» + «</p></td>«
            : «<td><p>« + WorkItem.Work + «</p></td>«;

        res_html += «<td><p>« + WorkItem.Artist + «</p></td>«;
        res_html += «<td><p>« + WorkItem.AskingPrice + «</p></td>«;
        res_html += «<td><p>« + (WorkItem.Description ?? ««) + «</p></td></tr>«;
    }
    // Применяем наш подгруженный шаблон.
    res_html = rep.Replace(«[VRA_TABLE_REPORT]», res_html);

    // Возвращаем заполненный html файл.
    return res_html;
}
```

Ранее мы уже добавили пункт меню, отвечающий за вызов метода `HtmlWorksInGalleryButton_Click()`, формирующего данный отчет. Если вы не сделали этого, самое время его добавить.

Теперь рассмотрим реализацию метода на уровне главного окна приложения. Логика метода следующая: выбираем и загружаем подготовленный ранее шаблон прайс-листа, заполняем шаблон, вызывая метод `genHtmlWorksInGallery()`, и сохраняем полученный результат в файл формата html.

```
private void HtmlWorksInGalleryButton_Click(object sender, RoutedEventArgs e)
{
    // Объект для хранения шаблона.
    String rep;
    // Создаем объект для подгрузки нашего шаблона.
    OpenFileDialog dlg = new OpenFileDialog
    {
        DefaultExt = «.vrt»,
        Filter = «View Ridge Assistant Template files|*.vrt»
    };
    // Подгружаем наш шаблон.
    bool? result = dlg.ShowDialog();
    // Если шаблон погрузился, то сохраняем его в rep.
    if (result == true)
    {
        StreamReader sr = new StreamReader(dlg.FileName);
        rep = sr.ReadToEnd();
```

```
        sr.Close();
    }
    else
    {
        return;
    }
    // Получаем итоговый html код файла.
    string full_rep = ProcessFactory.GetReport().genHtmlWorksInGallery(rep);
    // Сохраняем html файл в указанную директорию.
    SaveFileDialog sdlg = new SaveFileDialog {DefaultExt = «.html», Filter = «Html
Documents (.html)|*.html»};
    if (sdlg.ShowDialog() == true)
    {
        string filename = sdlg.FileName;
        StreamWriter sw = new StreamWriter(filename);
        sw.WriteLine(full_rep);
        sw.Close();
    }
}
```

На этом реализация простых отчетов завершена. Запустите приложение и убедитесь, что все работает правильно.

Контрольные вопросы и задания

1. Доработайте задачу выгрузки данных в формат xsls так, чтобы после сохранения файла, автоматически отображалось его содержимое.
2. Доработайте задачу формирования списка картин на продажу так, чтобы после сохранения файла его содержимое отображалось автоматически в браузере по умолчанию.
3. Какую задачу выполняет менеджер пакетов NuGet?
4. Какой пакет используется в работе для решения задачи выгрузки данных в формате xlsx?
5. С какой целью создают и используют представления (view)?
6. Опишите алгоритм формирования прайс-листа продаваемых картин.

Лабораторная работа №8
Сложная отчетность. Визуализация данных

8.1. Постановка задачи

В предыдущей работе мы рассмотрели способ формирования простой отчетности путем выгрузки данных в виде файлов Excel без какой-то специальной обработки данных. Просто выгружали их как есть. Также мы рассмотрели способ создания html-отчета заданного вида, хотя и в самой простой форме.

Как вы понимаете, представление информации не ограничивается текстом или таблицами. Часто более удобно для дальнейшего анализа иметь возможность представлять информацию в графическом виде: в виде графиков, диаграмм, гистограмм и т.п.

Как всегда, существует масса возможностей сделать это. Мы для решения этой задачи будем использовать WF (Windows Forms) компонент.

8.2. Создаем окно визуальных отчетов

Для работы с визуальным отчетом создадим отдельное окно ReportWindow. Открывать его будем через меню главного окна приложения, для этого добавим в меню «Отчеты» еще один элемент:

```
<MenuItem Name=«GraphReportButton» Header=«Отчет по продажам»
Click=«GraphReportButton_Click»/>
```

В реализации `Click=«GraphReportButton_Click»` мы будем открывать окно визуальных отчетов:

```
private void GraphReportButton_Click(object sender, RoutedEventArgs e)
{
    var window = new ReportWindow();
    window.Show();
}
```

Теперь перейдем к реализации формы окна.

Для построения графиков мы будем использовать компонент `Chart`. Для этого добавим в раздел системных ссылок (References) проекта VRA ссылки на следующие компоненты .NET:

```
System.Windows.Forms
System.Windows.Forms.DataVisualization
WindowsFormsIntegration
```

Мы не будем давать полное описание xaml-кода окна. У вас уже имеется отличный опыт разработки диалоговых окон, поэтому предполагаем, что вы сумеете реализовать окно для визуального отчета самостоятельно. Должно получиться примерно то, что изображено на рис. 8.1. А мы лишь остановимся на основных и важных моментах.

К форме надо подключить наш WF компонент, добавив в начале определения окна строку:

```
xmlns:wf=«clr-namespace:System.Windows.Forms.DataVisualization.Charting;assembly=
System.Windows.Forms.DataVisualization»
```

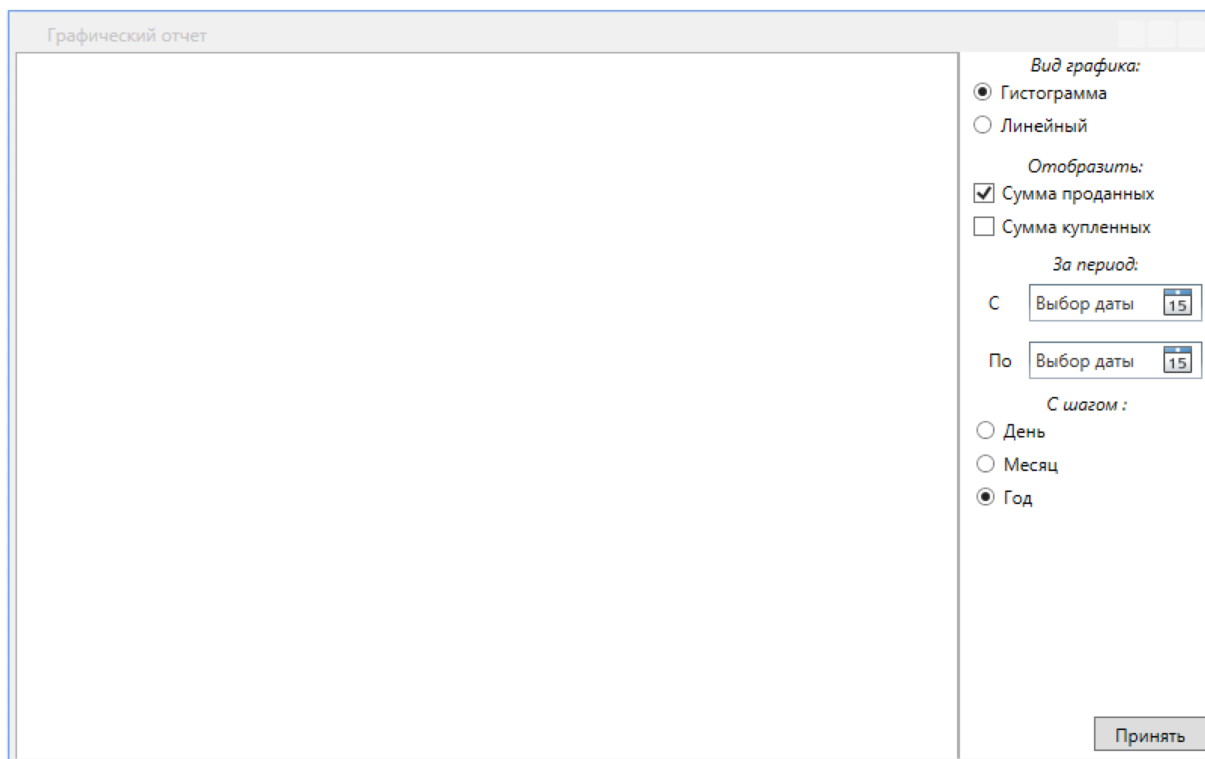

Рис. 8.1. Окно визуального отчета

Теперь опишем основные элементы управления окна, всего их одиннадцать.

Первая группа переключателей (RadioButton) обеспечивает выбор вида выводимого графика (гистограмма или линейный график):

```
<RadioButton Content=«Гистограмма» Height=«16» Name=«radioGist» Margin=«3»
IsChecked=«True» />
<RadioButton Content=«Линейный» Height=«16» Name=«radioSpline» Margin=«3» />
```

Группа флажков (CheckBox) позволяет указать, какая сумма будет отображаться на графике. При этом возможно указать отображение как каждой суммы в отдельности, так и вместе на одном графике:

```
<CheckBox Content=«Сумма проданных» Height=«16» Name=«radioSales» IsChecked=«True»
Margin=«3»/>
<CheckBox Content=«Сумма купленных» Height=«16» Name=«radioPurchase» Margin=«3»/>
```

Группа полей «Календарь» (DatePicker) позволяет задать начало и конец отчетного периода:

```
<DatePicker Grid.Column=«1» Grid.Row=«2» Height=«25» HorizontalAlignment=«Left»
Margin=«46,25,0,0» Name=«datePicker1» VerticalAlignment=«Top» Width=«115» />
<DatePicker Height=«25» HorizontalAlignment=«Left» Margin=«46,63,0,0»
Name=«datePicker2» VerticalAlignment=«Top» Width=«115» Grid.Column=«1» Grid.Row=«2» />
```

Последняя группа переключателей обеспечивает выбор размера периода, т.е. с каким шагом должна быть построена временная шкала графика:

```
<RadioButton Content=«День» Height=«16» Name=«radioDay» Margin=«3» />
<RadioButton Content=«Месяц» Height=«16» Name=«radioMounth» Margin=«3» />
<RadioButton Content=«Год» Height=«16» Name=«radioYear» Margin=«3» IsChecked=«True» />
```

Кнопка (Button) служит для построения или обновления графика:

```
<Button Content=«Принять» Height=«23» Name=«btn_accept» Width=«75»
FlowDirection=«LeftToRight» Click=«btn_accept_Click» />
```

И, наконец, сам компонент графической области (`Chart`) предназначен для прорисовки различных рисованных объектов, в нашем случае построения графика:

```
<WindowsFormsHost Grid.Row=«0» Grid.RowSpan=«4» Grid.Column=«0»>
     <wf:Chart x:Name=«chart» />
</WindowsFormsHost>
```

Реализацию окна опишем немного позже, а пока перейдем к созданию объектов для работы с базой данных и реализации бизнес-логики.

8.3. Создание структуры для работы со сложными отчетами

Общая схема реализации сущности «Отчет» представлена на рис. 8.2 и основана на принятом в данном приложении шаблоне проектирования.

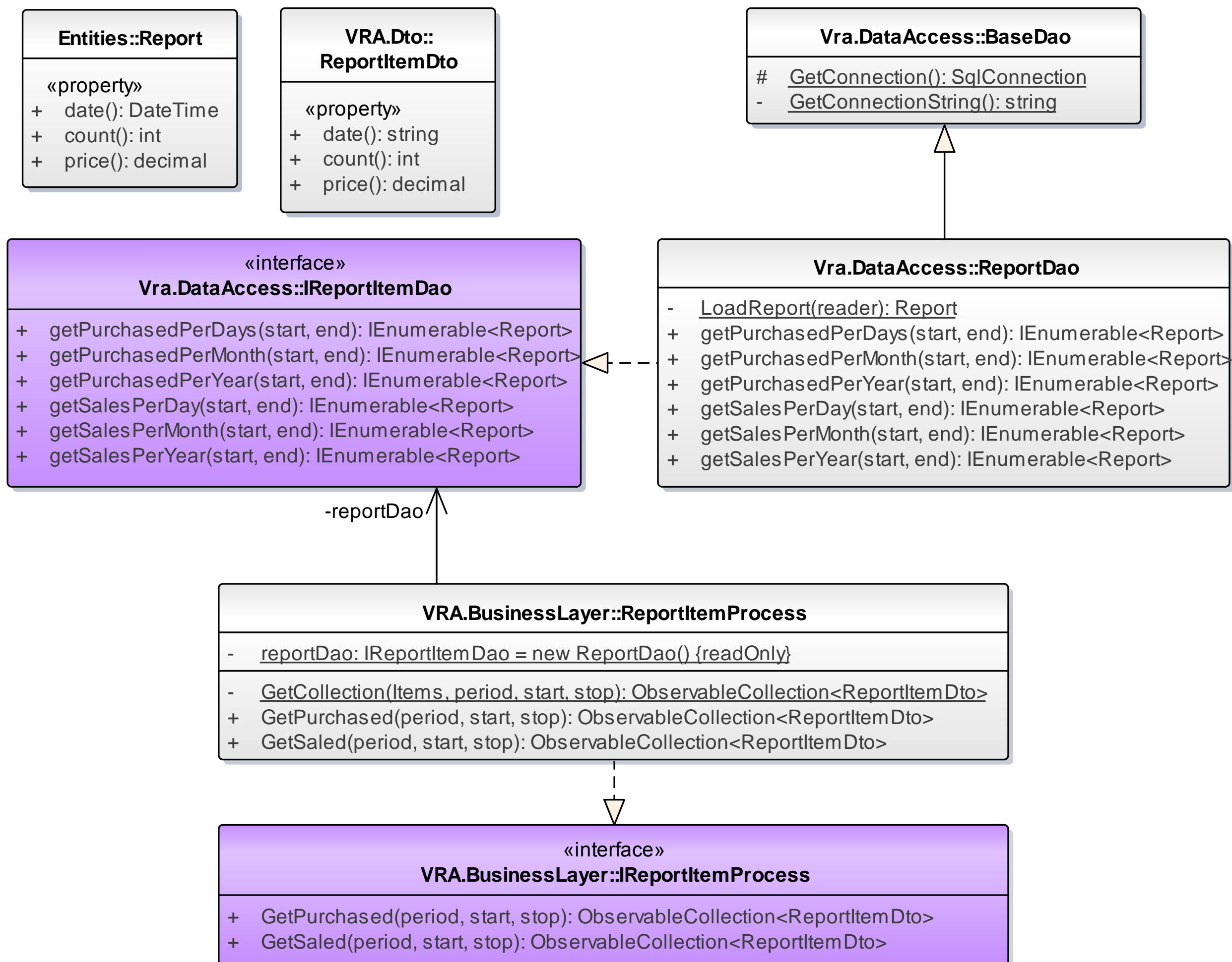

Рис. 8.2. Схема реализации сущности «Отчет»

Начнем с реализации класса `ReportDao`:

```
using Vra.DataAccess.Entities;
using System.Data.SqlClient;
```

```
namespace Vra.DataAccess
{
    public class ReportDao : BaseDao, IReportItemDao
    {
        private static Report LoadReport(SqlDataReader reader)
        {
            //Создаём пустой объект
            Report report = new Report();
            //Заполняем поля объекта в соответствии с названиями полей результирующего
            // набора данных
            try
            {
                report.date = reader.GetDateTime(reader.GetOrdinal(«mydate»));
            }
            catch (Exception ex)
            {
                report.date = DateTime.Now.Date;
            }
            report.price = reader.GetDecimal(reader.GetOrdinal(«mysum»));
            report.count = reader.GetInt32(reader.GetOrdinal(«mycount»));

            return report;
        }

        public IList<Report> getPurchasedPerDays(DateTime start, DateTime end)
        {
            IList<Report> reports = new List<Report>();

            //Получаем объект подключения к базе
            using (var conn = GetConnection())
            {
                //Открываем соединение
                conn.Open();
                //Создаем sql команду
                using (var cmd = conn.CreateCommand())
                {
                    //Задаём текст команды
                    cmd.CommandText = «select CONVERT(date, DateAcquired, 105) as
mydate, isnull(SUM(AcquisitionPrice), 0.0) as mysum, ISNULL(count(AcquisitionPrice),
0.0) as mycount from TRANS where DateAcquired between @start and @stop group by
CONVERT(date, DateAcquired, 105)»;
                    //Добавляем значение параметра
                    cmd.Parameters.AddWithValue(«@start», start);
                    cmd.Parameters.AddWithValue(«@stop», end);
                    //Открываем SqlDataReader для чтения полученных в результате
                    // выполнения запроса данных
                    using (var dataReader = cmd.ExecuteReader())
                    {
                        while (dataReader.Read())
                        {
                            reports.Add(LoadReport(dataReader));
                        }
                    }
                }
            }
            return reports;
        }
    }
}
```

Реализация остальных методов класса ReportDao будет однотипна. Вам необходимо лишь изменить текст запроса в соответствии с каждым конкретным методом:

```
getPurchasedPerMonth:
select CONVERT(date, DateAcquired, 105) as mydate, isnull(SUM(AcquisitionPrice), 0.0)
as mysum, ISNULL(count(AcquisitionPrice), 0.0) as mycount from TRANS where DateAcquired
between @start and @stop group by CONVERT(date, DateAcquired, 105)

getPurchasedPerYear:
select CONVERT(date, DateAcquired, 105) as mydate, isnull(SUM(AcquisitionPrice), 0.0)
as mysum, ISNULL(count(AcquisitionPrice), 0.0)as mycount from TRANS where
DATEPART(Year, DateAcquired) between DATEPART(Year, @start) and DATEPART(Year, @stop)
group by CONVERT(date, DateAcquired, 105)

getSalesPerDay:
select CONVERT(date, PurchaseDate, 105) as mydate, isnull(SUM(SalesPrice), 0.0) as
mysum, ISNULL(count(SalesPrice), 0.0) as mycount from TRANS where PurchaseDate between
@start and @stop group by CONVERT(date, PurchaseDate, 105)

getSalesPerMonth:
select CONVERT(date, PurchaseDate, 105) as mydate, isnull(SUM(SalesPrice), 0.0) as
mysum, ISNULL(count(SalesPrice), 0.0) as mycount from TRANS where PurchaseDate between
@start and @stop group by CONVERT(date, PurchaseDate, 105)

getSalesPerYear:
select CONVERT(date, PurchaseDate, 105) as mydate, isnull(SUM(SalesPrice), 0.0) as
mysum, ISNULL(count(SalesPrice), 0.0) as mycount from TRANS where DATEPART(Year,
DateAcquired) between DATEPART(Year, @start) and DATEPART(Year, @stop) group by
CONVERT(date, PurchaseDate, 105)
```

Продолжим описанием реализации класса ReportItemProcess на уровне бизнес-логике.

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using VRA.Dto;
using Vra.DataAccess;
using VRA.BusinessLayer.Converters;

namespace VRA.BusinessLayer
{
    public class ReportItemProcess : IReportItemProcess
    {
```

Объявляем поле для работы с объектом типа ReportDao:

```
        private static readonly IReportItemDao reportDao = new ReportDao();
```

Для начала реализуем метод для получения коллекции данных по картинам. Он будет общим и для купленных, и для проданных картин.

```
        private static ObservableCollection<ReportItemDto>
GetCollection(IList<ReportItemDto> Items, string period, DateTime start, DateTime stop)
        {
```

Создаем пустой объект для хранения коллекции:

```
            ObservableCollection<ReportItemDto> Collection = new
ObservableCollection<ReportItemDto>();
```

```
            // Условие проверки наличия принятых данных.
            if (Items == null) { return null; }
```

Теперь опишем алгоритм обработки коллекции перед ее выводом на график. Для организации обработки в зависимости от выбранного размера периода (день, месяц или год) традиционно используем оператор выбора `switch`. Реализация каждого варианта практически одинаковая, поэтому мы приведем реализацию обработки варианта «day» полностью, а для остальных вариантов лишь различающиеся строки (создания объекта и условия для его перебора). Вам необходимо их доработать по образцу.

```
            switch (period)
            {
                case «day»:
                    {
                        DateTime d = start;
                        while (d <= stop)
                        {
                            ReportItemDto repItem = new ReportItemDto { date =
d.Date.ToString(«dd.MM.yyyy»), count = 0, price = 0 };

                            foreach (var item in Items)
                            {
                                if (Convert.ToDateTime(item.date).Date == d)
                                {
                                    repItem.count += item.count;
                                    repItem.price += item.price;
                                }
                            }
                            Collection.Add(repItem);

                            d = d.AddDays(1);
                        }
                        break;
                    }

                case «month»:
ReportItemDto repItem = new ReportItemDto { date = d.Date.ToString(«Y») , count = 0,
price = 0 };

if ((Convert.ToDateTime(item.date).Month == d.Month) &&
(Convert.ToDateTime(item.date).Year == d.Year))

                case «year»:
ReportItemDto repItem = new ReportItemDto {date = d.Date.Year.ToString(), count = 0,
price = 0};

if (Convert.ToDateTime(item.date).Year == d.Year)

            }
            // Возвращаем коллекцию.
            return Collection;
        }
```

Переходим к моменту с типом картин. Все купленные и проданные картины будем получать в методах `GetPurchased()` и `GetSaled()` соответственно, а результат будем возвращать через общий метод, описанный выше. Метод `GetSaled()` реализуйте самостоятельно! Он аналогичен

GetPurchased(), изменится лишь аргумент .getPurchasedDay на .getSaledDay, как и аналогичные им Month и Year.

```
        public ObservableCollection<ReportItemDto> GetPurchased(string period, DateTime
start, DateTime stop)
        {
            IList<ReportItemDto> ReportList;

            switch (period)
            {
                case «day»:
                    {
                        ReportList =
DtoConverter.Convert(reportDao.getPurchasedPerDays(start, stop));
                        break;
                    }
                case «month»:
                    {
                        ReportList =
DtoConverter.Convert(reportDao.getPurchasedPerMonth(start, stop));
                        break;
                    }

                case «year»:
                    {
                        ReportList =
DtoConverter.Convert(reportDao.getPurchasedPerYear(start, stop));
                        break;
                    }
                default:
                    {
                        ReportList = null;
                        break;
                    }
            }

            return GetCollection(ReportList, period, start, stop);
        }
    }
}
```

Перейдем в фабрику процессов, где централизованно запускаются все наши процессы. Добавим фабрике метод:

```
public static IReportItemProcess GetReportProcess()
{
    return new ReportItemProcess();
}
```

В класс DtoConverter добавим пару методов:

```
private static ReportItemDto Convert(Report report)
{
    if (report == null) { return null; }

    ReportItemDto reportdto = new ReportItemDto {date = report.date.ToString(), count =
report.count, price = report.price};

    return reportdto;
}
```

```
public static IList<ReportItemDto> Convert(IEnumerable<Report> reports)
{
    if (reports == null) { return null; }

    IList<ReportItemDto> ReportsDto = new List<ReportItemDto>();

    foreach (var r in reports)
    {
        ReportsDto.Add(Convert(r));
    }

    return ReportsDto;
}
```

На этом все, переходим к реализации окна.

8.4. Реализация окна визуальных отчетов

Код окна будет иметь вид:

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Collections.ObjectModel;
using VRA.Dto;
using VRA.BusinessLayer;
using System.Windows.Forms.DataVisualization.Charting;

namespace VRA
{
    /// <summary>
    /// Логика взаимодействия для ReportWindow.xaml
    /// </summary>
    public partial class ReportWindow
    {
```

Объявляем необходимые для работы поля:

```
        private ObservableCollection<ReportItemDto> collection = new
ObservableCollection<ReportItemDto>();

        private readonly List<decimal> axisYDataSales = new List<decimal>();
        private readonly List<decimal> axisYDataPurchase = new List<decimal>();
        private readonly List<string> axisXData = new List<string>();
```

Инициализация формы не изменится, а вот загрузка и заполнение ее данными будет реализовываться в методе `Window_Loaded()`. Мы описывали способ применения этого метода в предыдущей работе. Этот метод будет выводить на форму начальный график, все значения в котором заданы по умолчанию.

```
        public ReportWindow()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            // Т.к. все графики находятся в пределах области построения, создадим ее.
            chart.ChartAreas.Add(new ChartArea(«Default»));
```

```
            // Добавим график, и назначим его в ранее созданную область «Default».
            chart.Series.Add(new Series(«Проданные»));
            chart.Series[«Проданные»].ChartArea = «Default»;
            chart.Series.Add(new Series(«Купленные»));
            chart.Series[«Купленные»].ChartArea = «Default»;

            // Определяем легенду.
            chart.Legends.Add(new Legend(«Legend»));
            chart.Legends[«Legend»].DockedToChartArea = «Default»;
            chart.Series[«Проданные»].Legend = «Legend»;
            chart.Series[«Купленные»].Legend = «Legend»;
            chart.Series[«Купленные»].IsVisibleInLegend = false;
            chart.Series[«Проданные»].IsVisibleInLegend = false;

            // Загрузка данных по умолчанию.
            IList<TransactionDto> transaction =
ProcessFactory.GetTransactionProcess().GetList();
            datePicker1.Text = transaction[0].DateAcquired.ToString();
            datePicker2.Text = transaction[transaction.Count -
1].DateAcquired.ToString();
            btn_accept_Click(sender, e);
        }
```

Переходим к реализации процесса работы. Он организован следующим образом: после заполнения всех необходимых полей пользователь нажимает на единственную, расположенную на форме, кнопку. Код обработки события для кнопки приведен ниже. При реализации события последовательно вызываются четыре метода: для проверки корректности заданной даты, формирования коллекции, выбора типа графика и его построения соответственно.

```
        private void btn_accept_Click(object sender, RoutedEventArgs e)
        {
            DateCompare();
            FillCollection();
            GraphType();
            DrawGraph();
        }

        private void DateCompare()
        {
            if ((Convert.ToDateTime(datePicker1.Text)) >=
Convert.ToDateTime(datePicker2.Text))
            {
                MessageBox.Show(«Дата окончания интервала запроса \n меньше либо равна
дате начала»);
            }
        }
```

Немного подробней о следующем методе. В нем мы будем не только формировать коллекцию, но и учитывать выбранные переключатели и флажки. Метод отдельно обрабатывает данные по проданным и купленным картинам.

```
        private void FillCollection()
        {
            axisYDataSales.Clear();
            axisYDataPurchase.Clear();

            // Если запрошена статистика по проданным.
            if (radioSales.IsChecked != null && radioSales.IsChecked.Value)
```

```
            {
                if (radioDay.IsChecked != null && radioDay.IsChecked.Value)
                {
                    TimeSpan ts =
(Convert.ToDateTime(datePicker2.Text)).Subtract(Convert.ToDateTime(datePicker1.Text));

                    if (ts.Days > 30)
                    {
                        MessageBox.Show(
                            «Выбранный Вами период времени слишком велик! \n
Максимальная длина периода - 30 дней «);
                        datePicker2.Text =
Convert.ToDateTime(datePicker1.Text).Date.AddDays(30).ToString();
                    }

                    collection.Clear();
                    collection = ProcessFactory.GetReportProcess()
                        .GetSaled(«day», Convert.ToDateTime(datePicker1.Text),
Convert.ToDateTime(datePicker2.Text));
                }

                if (radioMounth.IsChecked != null && radioMounth.IsChecked.Value)
                {
                    TimeSpan ts =
(Convert.ToDateTime(datePicker2.Text)).Subtract(Convert.ToDateTime(datePicker1.Text));

                    if (ts.Days/30 > 12)
                    {
                        MessageBox.Show(
                            «Выбранный Вами период времени слишком велик! \n
Максимальная длина периода - 12 месяцев «);
                        datePicker2.Text =
Convert.ToDateTime(datePicker1.Text).Date.AddMonths(12).ToString();
                    }

                    collection.Clear();
                    collection = ProcessFactory.GetReportProcess()
                        .GetSaled(«month», Convert.ToDateTime(datePicker1.Text),
Convert.ToDateTime(datePicker2.Text));
                }

                if (radioYear.IsChecked != null && radioYear.IsChecked.Value)
                {
                    TimeSpan ts =
(Convert.ToDateTime(datePicker2.Text)).Subtract(Convert.ToDateTime(datePicker1.Text));

                    if (ts.Days/(30*12) > 10)
                    {
                        MessageBox.Show(
                            «Выбранный Вами период времени слишком велик! \n
Максимальная длина периода - 10 лет «);
                        datePicker2.Text =
Convert.ToDateTime(datePicker1.Text).Date.AddYears(10).ToString();
                    }
                    collection.Clear();
                    collection = ProcessFactory.GetReportProcess()
                        .GetSaled(«year», Convert.ToDateTime(datePicker1.Text),
Convert.ToDateTime(datePicker2.Text));

                }
```

```
            // Заполнение коллекции проданных.
            foreach (var item in collection)
            {
                axisYDataSales.Add(item.price);
            }
        }
```

Для купленных картин все действия будут аналогичны, поэтому допишите метод самостоятельно. Мы приведем лишь условие:

```
        // Если запрошена статистика по купленным.
        if (radioPurchase.IsChecked != null && radioPurchase.IsChecked.Value)
        {
        // Ваша реализация...
        }
    }
```

И наконец, последняя пара методов. `GraphType()` будет проверять какой тип графика был выбран пользователем для построения, и задавать его в легенду.

```
    private void GraphType()
    {
        if (radioGist.IsChecked != null && radioGist.IsChecked.Value)
        {
            // Определяем вид графиков.
            chart.Series[«Проданные»].ChartType = SeriesChartType.Column;
            chart.Series[«Купленные»].ChartType = SeriesChartType.Column;
        }

        if (radioSpline.IsChecked != null && radioSpline.IsChecked.Value)
        {
            // Определяем вид графиков.
            chart.Series[«Проданные»].ChartType = SeriesChartType.Line;
            chart.Series[«Купленные»].ChartType = SeriesChartType.Line;
        }
    }
```

Осталось лишь построить и вывести график, для этого будет служить `DrawGraph()`:

```
    private void DrawGraph()
    {
        // Очищаем старые данные.
        axisXData.Clear();
        chart.Series[«Проданные»].Points.Clear();
        chart.Series[«Купленные»].Points.Clear();

        // Добавляем подписи по оси X.
        foreach (var item in collection)
        {
            axisXData.Add(item.date);
        }

        // Настраиваем легенду.
        if ((axisYDataSales.Count != 0) & (axisYDataPurchase.Count != 0))
        {
            chart.Series[«Купленные»].IsVisibleInLegend = true;
            chart.Series[«Проданные»].IsVisibleInLegend = true;
        }
        else
```

```
            {
                chart.Series[«Купленные»].IsVisibleInLegend = false;
                chart.Series[«Проданные»].IsVisibleInLegend = false;
            }

            // Строим графики.
            if (axisYDataSales.Count != 0)
                chart.Series[«Проданные»].Points.DataBindXY(axisXData, axisYDataSales);

            if (axisYDataPurchase.Count != 0)
                chart.Series[«Купленные»].Points.DataBindXY(axisXData,
axisYDataPurchase);
        }
    }
}
```

На этом реализация визуального отчета завершена. Запустите приложение и проверьте его работу.

Контрольные вопросы и задания

1. В чем отличие технологии Windows Forms от технологии Windows Presentation Foundation?
2. Каким образом можно в проекте WPF использовать компоненты WF?
3. Какой компонент используется для построения графиков и диаграмм?
4. Опишите алгоритм обработки коллекции перед ее выводом на график?
5. Какую задачу выполняет компонент DatePicker?

Библиографический список

1. Галиаскаров, Э.Г. Проектирование баз данных: лабораторный практикум / Э.Г. Галиаскаров, А.Ю. Крылов; Иван. гос. хим.-технол. ун-т.- Иваново, 2012.- 96 с.
2. Кренке, Д. Теория и практика построения баз данных / Д. Кренке; пер. с англ. – 9-е изд. – СПб.: Питер, 2005.
3. Хейлсберг, А. Язык программирования C#. Классика Computers Science / А. Хейлсберг и др. – 4-е изд. – СПб.: Питер, 2011. – 784 с.
4. Бурков, А.В. Проектирование информационных систем в Microsoft SQL Server 2008 и Visual Studio 2008 / А.В. Бурков. – http://www.intuit.ru/department/se/pisqlvs2008/.
5. Новиков, Ф.А. Моделирование на UML. Теория, практика, видеокурс / Ф.А. Новиков, Д.Ю. Иванов. – СПб.: Профессиональная литература; Наука и техника, 2010. – 640 с. (http://book.uml3.ru/about).
6. Архитектура информационных систем: учебник для студ. учреждений высш. проф. образования / Б.Я. Советов и др. - М.: Издательский центр «Академия», 2012. - 228 с.
7. Фаулер, М. Шаблоны корпоративных приложений / М. Фаулер; пер. с англ. – М.: ООО «И.Д. Вильямс», 2014. – 544 с.
8. Мартин, Р. Принципы, паттерны и методика гибкой разработки на языке C# / Р. Мартин, М. Мартин; пер. с англ. – СПб.: Символ-Плюс, 2013. – 768 с.
9. Мэтью, М.-Д. WPF 4: Windows Presentation Foundation в .NET 4.0 с примерами на C# 2010 для профессионалов / М.-Д. Мэтью; пер. с англ. – М.: ООО «И.Д. Вильямс», 2011. – 1024 с.
10. Снетков, В.М. Практикум прикладного программирования на MFC и C++ в среде VS.NET / В.М. Снетков – М.: Интуит, 2010 (http://www.intuit.ru/studies/courses/594/450/info)
11. Раскин, Дж. Интерфейс: новые направления в проектировании компьютерных систем / Дж. Раскин; пер. с англ. – М.: Символ-Плюс, 2005. – 272 с.

Учебное издание

Галиаскаров Эдуард Геннадьевич, **Хоченков** Алексей Евгеньевич,
Кострома Алексей Михайлович, **Мицык** Андрей Павлович

РАЗРАБОТКА ПРИЛОЖЕНИЙ БАЗ ДАННЫХ

Лабораторный практикум

Редактор О. А. Соловьева

Подписано в печать 25.09.2015. Формат 60x84 1/16. Бумага писчая.
Усл. п.л. 6,51. Тираж 50 экз. Заказ

ФГБОУ ВПО «Ивановский государственный химико-технологический университет»

Отпечатано на полиграфическом оборудовании кафедры экономики и финансов ФГБОУ ВПО «ИГХТУ»

153000, г. Иваново, Шереметевский пр., 7