Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерного проектирования

Кафедра инженерной психологии и эргономики

А. Л. Раднёнок, К. Д. Яшин

УПРАВЛЕНИЕ ИНФОРМАЦИОННЫМИ ПРОЕКТАМИ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве пособия для специальности 1-58 01 01 «Инженерно-психологическое обеспечение информационных технологий»

Минск БГУИР 2017

УДК 004.41(076)
ББК 32.973.26-018.2я73
Р15

Р е ц е н з е н т ы:

кафедра интеллектуальных систем
Белорусского национального технического университета
(протокол №6 от 08.02.2016);

главный научный сотрудник лаборатории логического проектирования
государственного научного учреждения «Объединенный институт проблем
информатики Национальной академии наук Беларуси»,
доктор технических наук, доцент Л. Д. Черемисинова

Раднёнок, А. Л.

Р15 Управление информационными проектами : пособие / А. Л. Раднёнок, К. Д. Яшин. – Минск : БГУИР, 2017. – 72 с. : ил.

ISBN 978-985-543-256-3.

Содержит материалы к лабораторным занятиям, включающие теоретическую часть и задания с вариантами.

УДК 004.41(076)
ББК 32.973.26-018.2я73

ISBN 978-985-543-256-3

Содержание

Введение 4

Лабораторная работа №1 Организация памяти и используемые типы данных в языке программирования C# при реализации информационных проектов 5

Лабораторная работа №2 Способы организации классов и методов в языке программирования C# при реализации информационных проектов 16

Лабораторная работа №3 Способы организации массивов в языке программирования C# при реализации информационных проектов 23

Лабораторная работа №4 Методы реализации интерфейсов в языке программирования C# и их применение при реализации информационных проектов 31

Лабораторная работа №5 Способы реализации обработки событий в языке программирования C# для организации рефлексии при организации информационных проектов 39

Лабораторная работа №6 Создание Windows-приложений в качестве основы графического интерфейса на языке C# при реализации информационных проектов 45

Лабораторная работа №7 Реализация главного меню, диалоговых окон, строк состояния Windows-приложения в языке программирования C# как основы взаимодействия приложения и пользователя при создании информационных проектов 52

Лабораторная работа №8 Создание и представление базы данных в Windows-приложении с помощью встроенных компонентов платформы .NET при реализации информационных проектов 55

Литература 72

Введение

Важную роль при создании коммерческих информационных продуктов играют проектирование информационных проектов и управление ими. Так как информационные проекты, как правило, разрабатываются и реализуются группами разработчиков, очень важным является эффективное взаимодействие между их участниками. Одним из средств такого взаимодействия является язык программирования, знание которого всеми членами команды позволяет добиться хороших результатов при реализации информационных проектов. Не менее важен выбор языка программирования, удовлетворяющего современным требованиям. В языке программирования, кроме присутствия сильного математического аппарата, должны также соблюдаться принципы объектно-ориентированного программирования. Одним из таких языков является Visual C#.

Данное пособие содержит лабораторные работы, позволяющие научиться работать с линейными, разветвленными алгоритмами, изучить и применить на практике принципы инкапсуляции, наследования и полиморфизма объектно-ориентированного программирования, а также освоить создание проектов различных типов, принципы работы с платформой .NET, научиться создавать базы данных и управлять ими с помощью компонент .NET.

Лабораторная работа №1

Организация памяти и используемые типы данных в языке программирования C# при реализации информационных проектов

Цель работы: изучить принципы организации памяти, различные типы памяти в языке программирования C#; освоить механизм преобразования и приведения типов с использованием математического аппарата в языке программирования C# при реализации информационных проектов.

Теоретические сведения

Типы данных имеют особенное значение в C#, поскольку это строго типизированный язык: все операции подвергаются строгому контролю со стороны компилятора на соответствие типов, причем недопустимые операции не компилируются. Следовательно, такой контроль типов позволяет исключить ошибки и повысить надежность программ. Для обеспечения контроля типов все переменные, выражения и значения должны принадлежать к определенному типу. Такого понятия, как «бестиповая» переменная, в данном языке программирования не существует. Более того, тип значения определяет те операции, которые разрешается выполнять над ним: операция, разрешенная для одного типа данных, может оказаться недопустимой для другого.

В C# имеются две общие категории встроенных типов данных: типы значений и ссылочные типы (рисунок 1.1). Они отличаются содержимым переменной. Концептуально разница между ними состоит в том, что **тип значения** (value type) хранит данные непосредственно, в то время как **ссылочный тип** (reference type) хранит ссылку на значение. Эти типы сохраняются в разных местах памяти: типы значений сохраняются в области, известной как **стек**, а ссылочные типы – в области, называемой **управляемой кучей.**

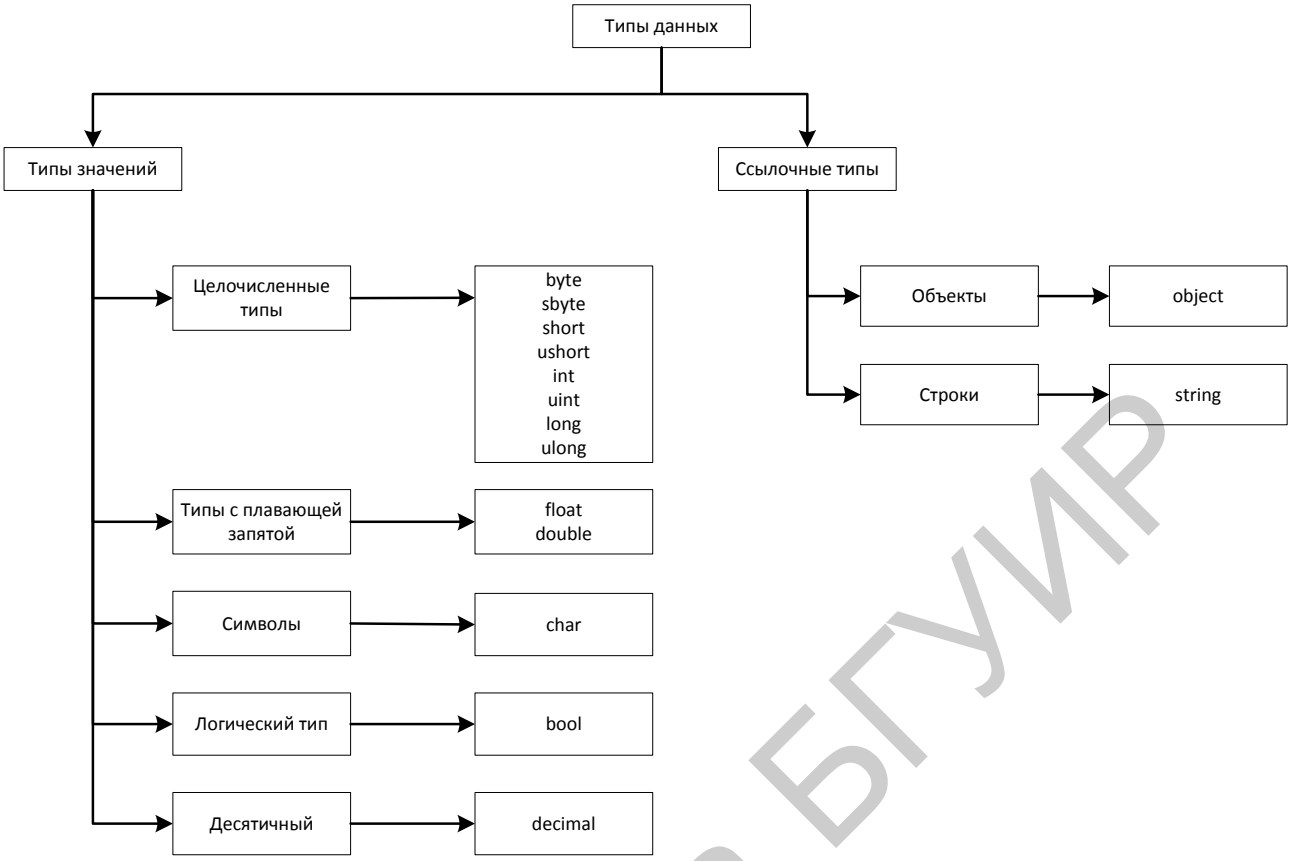


Рисунок 1.1 – Типы данных

В C# определены девять целочисленных типов: char, byte, sbyte, short, ushort, int, uint, long и ulong. Но тип char применяется главным образом для представления символов и поэтому рассматривается отдельно. Остальные восемь целочисленных типов предназначены для числовых расчетов. В таблице 1.1 представлены их диапазон представления чисел и разрядность в битах.

Таблица 1.1 – Типы данных, их диапазон и разрядность

Тип	Тип CTS	Разрядность в битах	Диапазон
1	**2**	**3**	**4**
byte	System.Byte	8	0 : 255
sbyte	System.SByte	8	–128 : 127
short	System.Int16	16	–32768 : 32767

Продолжение таблицы 1.1

1	2	3	4
ushort	System.UInt16	16	0 : 65535
int	System.Int32	32	–2147483648 : 2147483647
uint	System.UInt32	32	0 : 4294967295
long	System.Int64	64	–9223372036854775808 : 9223372036854775807
ulong	System.UInt64	64	0 : 18446744073709551615

Как следует из таблицы 1.1, в C# определены оба варианта различных целочисленных типов – со знаком и без знака, которые отличаются способом интерпретации старшего разряда целого числа. Так, если в программе указано целочисленное значение со знаком, то компилятор C# сгенерирует код, в котором старший разряд целого числа используется в качестве флага знака. Число считается положительным, если флаг знака равен 0, и отрицательным, если он равен 1. Отрицательные числа практически всегда представляются методом дополнения до двух, в соответствии с которым все двоичные разряды отрицательного числа сначала инвертируются, а затем к этому числу добавляется 1.

Самым распространенным в программировании целочисленным типом является int. Переменные типа int нередко используются для управления циклами, индексирования массивов и математических расчетов общего назначения. Когда же требуется целочисленное значение с большим диапазоном представления чисел, чем у типа int, то для этой цели используется целый ряд других целочисленных типов.

Типы с плавающей точкой позволяют представлять числа с дробной частью. В C# имеются две разновидности таких типов данных: float и double. Они представляют числовые значения с одинарной и двойной точностью соответственно. Так, разрядность типа float составляет 32 бита, что приближенно соответствует диапазону представления чисел от 5E – 45 до 3,4E + 38. Разрядность типа double составляет 64 бита, что приближенно соответствует диапазону представления чисел от 5E – 324 до 1,7E + 308. Тип данных float предназначен

для меньших значений с плавающей точкой, для которых требуется меньшая точность. Тип данных double больше, чем float, и предлагает более высокую степень точности (15 разрядов). Если нецелочисленное значение жестко кодируется в исходном тексте (например, 12,3), то обычно компилятор предполагает, что подразумевается значение типа double. Если значение необходимо специфицировать как float, потребуется добавить к нему символ F (или f):

```
float f = 12.3F;
```

Для представления чисел с плавающей точкой высокой точности предусмотрен также десятичный тип decimal, который применяется в финансовых расчетах. Этот тип имеет разрядность 128 бит для представления числовых значений в пределах от 1E – 28 до 7,9E + 28. Вам, вероятно, известно, что для обычных арифметических вычислений с плавающей точкой характерны ошибки округления десятичных значений. Эти ошибки исключаются при использовании типа decimal, который позволяет представить числа с точностью до 28 (а иногда и 29) десятичных разрядов. Благодаря тому, что этот тип данных способен представлять десятичные значения без ошибок округления, он особенно удобен для расчетов, связанных с финансами:

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

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         // *** расчет стоимости капиталовложения с ***
         // *** фиксированной нормой прибыли***
         decimal money, percent;
         int i;
         const byte years = 15;
         money = 1000.0m;
         percent = 0.045m;
         for (i = 1; i <= years; i++)
         {
            money *= 1 + percent;
```

```
            }
            Console.WriteLine("Общий доход за {0} лет: {1} $$",years,money);
            Console.ReadLine();
        }
    }
}
```

Тип bool представляет два логических значения: «истина» и «ложь». Эти логические значения обозначаются в C# зарезервированными словами true и false соответственно. Следовательно, переменная или выражение типа bool будет принимать одно из этих логических значений. Кроме того, в C# не определено взаимное преобразование логических и целых значений. Например, 1 не преобразуется в значение true, а 0 – в значение false.

Рассмотрим понятие **области видимости**. Каждая переменная доступна в рамках определенного контекста, или области видимости. Вне этого контекста переменная уже не существует. Существуют различные контексты:

– контекст класса – переменные, определенные на уровне класса, доступные в любом методе этого класса;

– контекст метода – переменные, определенные на уровне метода, являются локальными и доступны только в рамках данного метода;

– контекст блока кода – переменные, определенные на уровне блока кода, также являются локальными и доступны только в рамках данного блока.

Например, пусть класс Program определен следующим образом:

```
class Program // начало контекста класса
{
    static int a = 9; // переменная уровня класса

    static void Main(string[] args)        // начало контекста метода Main
    {
        int b = a - 1;                     // переменная уровня метода
        {                                  // начало контекста блока кода
            int c = b - 1;                 // переменная уровня блока кода
        }                                  // конец контекста блока кода
        // так нельзя, переменная c определена в блоке кода
        // Console.WriteLine(c);

        // так нельзя, переменная d определена в другом методе
        // Console.WriteLine(d);
        Console.Read();
```

```
    }           // конец контекста метода Main, переменная b уничтожается

    void Display()   // начало контекста метода Display
    {
        // переменная a определена в контексте класса, поэтому доступна
        int d = a + 1;
    }  // конец контекста метода Display, переменная d уничтожается
} // конец контекста класса, переменная a уничтожается
```

Здесь определены четыре переменные: a, b, c, d. Каждая из них существует в своем контексте. Переменная a существует в контексте всего класса Program и доступна в любом месте и блоке кода в методах Main и Display. Переменная b существует только в рамках метода Main. Переменная d существует в рамках метода Display, поэтому в методе Main мы не можем к ней обратиться. Переменная c существует только в блоке кода, границами которого являются открывающая и закрывающая фигурные скобки. Вне его границ переменная c не существует, и к ней нельзя обратиться. Нередко, как в данном случае, границы различных контекстов можно ассоциировать с открывающимися и закрывающимися фигурными скобками, которые задают пределы блока кода, метода, класса. При работе с переменными надо учитывать, что локальные переменные, определенные в методе или в блоке кода, скрывают переменные уровня класса, если их имена совпадают:

```
class Program
{
    static int a = 9; // переменная уровня класса

    static void Main(string[] args)
    {
        // скрывает переменную a, которая объявлена на уровне класса
        int a = 5;
        Console.WriteLine(a); // 5
    }
}
```

При объявлении переменных также надо учитывать, что в одном контексте нельзя определить несколько переменных с одним и тем же именем. В программировании нередко значения переменных одного типа присваиваются переменным другого типа. Например, в приведенном ниже фрагменте кода целое значение типа int присваивается переменной с плавающей точкой типа float:

```
int i;
```

```
float f;
i = 10;
f = i;          // присвоить целое значение переменной типа float
```

Если в одной операции присваивания смешиваются совместимые типы данных, то значение в правой части оператора присваивания автоматически преобразуется в тип, указанный в левой его части. Поэтому в приведенном выше фрагменте кода значение переменной i сначала преобразуется в тип float, а затем присваивается переменной f. Но вследствие строгого контроля типов далеко не все типы данных в C# оказываются полностью совместимыми, а следовательно, не все преобразования типов разрешены в неявном виде. Например, типы bool и int несовместимы. Однако может быть осуществлено явное преобразование несовместимых типов путем приведения.

Когда данные одного типа присваиваются переменной другого типа, неявное преобразование типов происходит автоматически при следующих условиях: оба типа совместимы; диапазон представления чисел целевого типа шире, чем у исходного типа. Если оба эти условия удовлетворяются, то происходит расширяющее преобразование. Например, тип int достаточно крупный, чтобы вмещать в себя все действительные значения типа byte, кроме того, оба типа являются совместимыми целочисленными типами, и поэтому для них вполне возможно неявное преобразование.

Рассмотрим приведение несовместимых типов. Несмотря на всю полезность неявных преобразований типов, они неспособны удовлетворить все потребности в программировании, поскольку допускают лишь расширяющие преобразования совместимых типов. Во всех остальных случаях приходится обращаться к приведению типов. **Приведение** – это команда компилятору преобразовать результат вычисления выражения в указанный тип, для чего требуется явное преобразование типов. Ниже приведена общая форма приведения типов:

```
(целевой_тип) выражение
```

Здесь целевой_тип обозначает тот тип, в который желательно преобразовать указанное выражение.

Если приведение типов приводит к сужающему преобразованию, то часть информации может быть утеряна. Например, в результате приведения типа long к типу int часть информации утратится, если значение типа long окажется больше диапазона представления чисел для типа int, поскольку старшие разряды этого числового значения отбрасываются. Когда же значение с плавающей точкой приводится к целочисленному, то в результате усечения теряется дробная часть этого числового значения. Так, если присвоить значение 1,23 целочисленной переменной, то в результате в ней останется лишь целая часть ис-

ходного числа (1), а дробная его часть (0,23) будет утеряна. Переменная i1 корректно преобразовалась в тип short, так как ее значение входит в диапазон этого типа данных. Преобразование переменной dec в тип int вернуло целую часть этого числа. Преобразование переменной i2 вернуло значение переполнения 18964 (то есть 84500 – 2 × 32767). Следует отметить, что в пространстве имен System имеется класс Convert, который тоже может применяться для расширения и сужения данных:

```
byte sum = Convert.ToByte(var1 + var2);
```

Одно из преимуществ подхода с применением класса System.Convert связано с тем, что он позволяет выполнять преобразования между типами данных нейтральным к языку образом (например, синтаксис приведения типов в Visual Basic полностью отличается от предлагаемого для этой цели в C#).

Индивидуальное задание по лабораторной работе

Используя среду разработки Microsoft Visual Studio, создать консольное приложение на языке программирования C#. Приложение должно вычислять значение выражения (см. варианты). Ввод данных должен производиться с клавиатуры. Некоторые функции класса Math представлены в таблице 1.2.

Варианты

1. $\dfrac{b+\sqrt{b^2+4ac}}{2a} - a^3c + b^{-2};$

2. $\dfrac{a}{c}\cdot\dfrac{b}{d} - \dfrac{ab-c}{cd};$

3. $\dfrac{\sin x + \cos y}{\sin x - \cos y}\operatorname{tg} xy;$

4. $\dfrac{x+y}{y+1} - \dfrac{xy-12}{34+x};$

5. $\dfrac{3+e^{y-1}}{1+x^2\lvert y-\operatorname{tg} x\rvert};$

6. $x - \dfrac{x^3}{3} + \dfrac{x^5}{5};$

7. $\ln\left\lvert\left(y-\sqrt{\lvert x\rvert}\right)\left(x-\dfrac{y}{x+\dfrac{x^2}{4}}\right)\right\rvert;$

8. $(1-\operatorname{tg} x)^{\operatorname{ctg} x} + \cos(x-y);$

9. $\dfrac{\ln\lvert\cos x\rvert}{\ln(1+x^2)};$

10. $\left(\dfrac{x+1}{x-1}\right)^x + 18xy^2;$

11. $\left(1+\dfrac{1}{x^2}\right)^x + 12x^2y;$

12. $\dfrac{x^2-7x+10}{x^2-8x+12};$

13. $\frac{\cos x}{\pi - 2x} + 16x \cos xy - 2;$

14. $2^{-x} - \cos x + \sin(2xy);$

15. $2\,\mathrm{ctg}(3x) - \frac{1}{12x^2 + 7x - 5};$

16. $|x^2 - x^3| - \frac{7x}{x^3 - 15x};$

17. $x \ln x + \frac{y}{\cos x - \frac{x}{3}};$

18. $\sin\sqrt{x+1} - \sin\sqrt{x-1};$

19. $e^x - \frac{y^2 + 12xy - 3x^2}{18y - 1};$

20. $\frac{1 + \sin\sqrt{x+1}}{\cos(12y - 4)};$

21. $2\,\mathrm{ctg}(3x) - \frac{\ln\cos x}{\ln(1 + x^2)};$

22. $e^x - x - 2 + (1 + x)^x;$

23. $3^x - 4x + \left(y - \sqrt{|x|}\right);$

24. $x - 10\sin x + |x^4 - x^5|;$

25. $x - 10^{\sin x} + \cos(x + y);$

26. $\frac{1 + \sin^2(x + y)}{2 + \left|x - \frac{2x}{1 + x^2y^2}\right|} + x;$

27. $\cos^2\left(\sin\frac{1}{z}\right);$

28. $\frac{\cos^2 x}{\sin x} - xyz + \frac{ax^2 + bx + c}{dx^3 - f}.$

Таблица 1.2 – Методы класса Math

Наименование метода	Описание метода
1	**2**
Abs(<Type>)	Возвращает абсолютное значение числа с типом <Type>
Acos(Double)	Возвращает угол, косинус которого равен указанному числу
Asin(Double)	Возвращает угол, синус которого равен указанному числу
Atan(Double)	Возвращает угол, тангенс которого равен указанному числу

Продолжение таблицы 1.2

1	2
Atan2(Double, Double)	Возвращает угол, тангенс которого равен отношению двух указанных чисел
BigMul(Int32, Int32)	Умножает два 32-битовых числа
Ceiling(Double)	Возвращает наименьшее целое число, которое больше или равно заданному числу с плавающей запятой двойной точности
Cos(Double)	Возвращает косинус указанного угла
Cosh(Double)	Возвращает гиперболический косинус указанного угла
Exp(Double)	Возвращает значение *e*, возведенное в указанную степень
Floor(Double)	Возвращает наибольшее целое число, которое меньше или равно заданному числу двойной точности с плавающей запятой
IEEERemainder(Double, Double)	Возвращает остаток от деления одного указанного числа на другое указанное число
Log(Double)	Возвращает натуральный логарифм (с основанием *e*) указанного числа
Log(Double, Double)	Возвращает логарифм указанного числа в системе счисления с указанным основанием
Log10(Double)	Возвращает логарифм с основанием 10 указанного числа
Max(<Type>, <Type>)	Возвращает большее из двух чисел типа <Type>
Min(<Type>, <Type>)	Возвращает меньшее из чисел типа <Type>
Pow(Double, Double)	Возвращает указанное число, возведенное в указанную степень
Round(<Type>)	Округляет значение до ближайшего целого
Round(<Type>, Int32)	Округляет значение до указанного числа дробных разрядов
Sign(<Type>)	Возвращает значение, определяющее знак десятичного числа

Продолжение таблицы 1.2

1	2
Sin(Double)	Возвращает синус указанного угла
Sinh(Double)	Возвращает гиперболический синус указанного угла
Sqrt(Double)	Возвращает квадратный корень из указанного числа
Tan(Double)	Возвращает тангенс указанного угла
Tanh(Double)	Возвращает гиперболический тангенс указанного угла
Truncate(Double)	Вычисляет целую часть заданного числа двойной точности с плавающей запятой

Лабораторная работа №2

Способы организации классов и методов в языке программирования C# при реализации информационных проектов

Цель работы: изучить способы организации классов в языке программирования C# при реализации информационных проектов; освоить механизм построения и работы классов и методов в языке программирования C#.

Теоретические сведения

Класс – это логическая структура, позволяющая создавать собственные пользовательские типы путем группирования переменных других типов, методов и событий. Класс подобен чертежу, поскольку он определяет данные и поведение типа. Если класс не объявлен статическим, то клиентский код может его использовать, создав объекты, или иначе экземпляры, приписанные переменной. Переменная остается в памяти, пока все ссылки на нее не выйдут из области видимости. В это время среда CLR помечает ее пригодной для сборщика мусора. Если класс объявляется статическим, то в памяти остается только одна копия, и клиентский код может получить к ней доступ только посредством самого класса, а не переменной экземпляра. В отличие от структур классы поддерживают наследование, то есть фундаментальную характеристику объектно-ориентированного программирования. Классы объявляются с помощью ключевого слова class, как показано в следующем примере.

```
public class Customer
{
   //Fields, properties, methods and events go here...
}
```

Ключевому слову class предшествует уровень доступа. Поскольку в данном случае используется public, любой пользователь может создавать объекты из этого класса. Имя класса указывается после ключевого слова class. Оставшаяся часть определения является телом класса, в котором задаются данные и поведение. Поля, свойства, методы и события в классе обозначаются термином **члены класса**.

Рассмотрим создание объектов. Класс и объект – это разные понятия, хотя в некоторых случаях они взаимозаменяемы. Класс определяет тип объекта, но не сам объект. **Объект** – это конкретная сущность, основанная на классе и иногда называемая **экземпляром класса**. Объекты можно создавать с помо-

щью ключевого слова new, за ним следует имя класса, на котором будет основан объект:

```
Customer object1 = new Customer();
```

При создании экземпляра класса ссылка на этот объект передается программисту. В предыдущем примере object1 является ссылкой на объект, основанный на Customer. Эта ссылка указывает на новый объект, но не содержит его данные. Фактически можно создать ссылку на объект без создания самого объекта:

```
Customer object2;
```

Создание ссылок, не указывающих на объект, не рекомендуется, так как попытка доступа к объекту по такой ссылке приведет к сбою во время выполнения. Однако ссылку можно сделать указывающей на объект, создав новый объект или назначив ее существующему объекту:

```
Customer object3 = new Customer();
Customer object4 = object3;
```

В данном коде создаются две ссылки на объекты, которые указывают на один объект, поэтому любые изменения объекта, выполненные посредством object3, будут видны при последующем использовании object4. Поскольку на объекты, основанные на классах, указывают ссылки, классы называют ссылочными типами. Наследование выполняется с помощью образования производных, то есть класс объявляется с помощью базового класса, от которого он наследует данные и поведение. Базовый класс задается добавлением после имени производного класса двоеточия и имени базового класса:

```
public class Manager : Employee
{
   // Employee fields, properties, methods and events are inherited
   // New Manager fields, properties, methods and events go here...
}
```

Когда класс объявляет базовый класс, он наследует все члены базового класса, за исключением конструкторов. В отличие от классов C++, класс в C# может только напрямую наследовать от одного базового класса. Однако, поскольку базовый класс может сам наследовать от другого класса, класс может косвенно наследовать несколько базовых классов. Кроме того, класс может напрямую реализовать несколько интерфейсов.

Класс может быть объявлен абстрактным. Абстрактный класс содержит абстрактные методы, которые имеют определение сигнатуры, но не имеют реализации. Нельзя создавать экземпляры абстрактных классов. Они могут использоваться только посредством производных классов, реализующих аб-

страктные методы. И наоборот, запечатанный класс не позволяет другим классам быть от него производными.

Рассмотрим **методы.** Если переменные хранят некоторые значения, то методы содержат набор операторов, которые выполняют определенные действия. Общее определение метода выглядит следующим образом:

[модификаторы] тип_возвращаемого_значения название_метода ([параметры])

```
{
   // тело метода
}
```

Модификаторы и параметры необязательны.

Рассмотрим на примере метода Main:

```
static void Main(string[] args)
{
   Console.WriteLine("привет мир!");
}
```

Ключевое слово static является модификатором. Далее идет тип возвращаемого значения. В данном случае ключевое слово void указывает на то, что метод ничего не возвращает. Такой метод также называется процедурой. Далее идет название метода Main и в скобках параметры (string[] args). В фигурные скобки заключено тело метода – все действия, которые он выполняет. Создадим две подобные процедуры:

```
static void Method1()
{
   Console.WriteLine("Method1");
}

// определение третьего метода
void Method2()
{
   Console.WriteLine("Method2");
}
```

В отличие от процедур функции возвращают определенное значение. Например, определим следующие функции:

```
int Factorial()
{
   return 1;
}
```

```
string Hello()
{
   return "Hell to World";
}
```

В функции в качестве типа возвращаемого значения вместо void используется любой другой тип (в данном случае int). Функции также отличаются тем, что мы обязательно должны использовать оператор return, после которого ставится возвращаемое значение. Также стоит отметить, что возвращаемое значение всегда должно иметь тот же тип, что значится в определении функции. То есть если функция возвращает значение типа int, то после оператора return стоит число 1, которое неявно является объектом типа int. Определив методы, мы можем использовать их в программе. Чтобы вызвать метод в программе, надо указать имя метода, далее – в скобках значения для его параметров:

```
static void Main(string[] args)
{
   string message = Hello(); // вызов первого метода
   Console.WriteLine(message);
   Sum(); // вызов второго метода
   Console.ReadLine();
}
static string Hello()
{
   return "Hell to World!";
}
static void Sum()
{
   int x = 2;
   int y = 3;
   Console.WriteLine("{0} + {1} = {2}", x, y, x+y);
}
```

Здесь определены два метода. Первый метод Hello возвращает значение типа string. Поэтому мы можем присвоить это значение какой-нибудь переменной типа string:

```
string message = Hello();
```

Второй метод – процедура Sum – складывает два числа и выводит результат на консоль.

Рассмотрим **конструкторы**. Когда создаются класс или структура, вызывается их конструктор, который имеет то же имя, что и класс или структура, и, как правило, инициализирует элементы данных нового объекта. В следующем примере класс Taxi определяется с помощью простого конструктора. После этого с помощью оператора new создается экземпляр класса. Конструктор Taxi вызывается оператором new сразу после выделения памяти для нового объекта.

```
public class Taxi
{
   public bool isInitialized;
   public Taxi()
   {
      isInitialized = true;
   }
}
class TestTaxi
{
   static void Main()
   {
      Taxi t = new Taxi();
      Console.WriteLine(t.isInitialized);
   }
}
```

Конструктор без параметров называется **конструктором по умолчанию**. Конструкторы по умолчанию вызываются при создании экземпляров объекта с помощью оператора new, при этом для оператора не указываются аргументы.

Статический конструктор используется для инициализации любых статических данных или для выполнения единовременного действия. Конструктор вызывается автоматически перед созданием первого экземпляра или перед обращением к любому статическому члену.

```
class SimpleClass
{
   // Static variable that must be initialized at run time.
   static readonly long baseline;
   // Static constructor is called at most one time, before any
   // instance constructor is invoked or member is accessed.
   static SimpleClass()
   {
      baseline = DateTime.Now.Ticks;
```

```
    }
}
```

Статические конструкторы обладают следующими свойствами:

– не принимают модификаторы доступа и не имеют параметров;

– вызываются автоматически для инициализации класса перед созданием первого экземпляра или ссылкой на какие-либо статические члены;

– его нельзя вызывать напрямую.

Пользователь не управляет временем выполнения статического конструктора в программе. Типичным использованием статических конструкторов является случай, когда класс использует файл журнала, и конструктор применяется для добавления записей в этот файл. Статические конструкторы также полезны при создании классов-оболочек для неуправляемого кода, когда конструктор может вызвать метод LoadLibrary.

Если статический конструктор инициирует исключение, среда выполнения не вызывает его во второй раз, и тип остается неинициализированным на время существования домена приложения, в котором выполняется программа.

Индивидуальное задание по лабораторной работе

Используя среду разработки Microsoft Visual Studio, создать консольное приложение на языке программирования C#. В приложении предусмотреть ввод и вывод информации об объектах соответствующих классов. Ввод данных должен производиться с клавиатуры.

Варианты

1. Построить систему классов для описания плоских геометрических фигур: круга, квадрата, прямоугольника. Предусмотреть методы для создания объектов, перемещения на плоскости, изменения размеров и поворота на заданный угол.

2. Построить описание класса, содержащего информацию о почтовом адресе организации. Предусмотреть возможность раздельного изменения составных частей адреса, создания и уничтожения объектов этого класса.

3. Составить описание класса для представления комплексных чисел с возможностью задания вещественной и мнимой частей как числами типов double, так и целыми числами. Обеспечить выполнение операций сложения, вычитания и умножения комплексных чисел.

4. Составить описание класса для работы с цепными списками строк (строки произвольной длины), используя операции включения в список, удале-

ния из списка элемента с заданным значением данного, удаления всего списка или конца списка, начиная с заданного элемента.

5. Составить описание класса для объектов-векторов, задаваемых координатами концов в трехмерном пространстве. Обеспечить операции сложения и вычитания векторов с получением нового вектора (суммы или разности), вычисления скалярного произведения двух векторов, длины вектора, косинуса угла между векторами.

6. Составить описание класса прямоугольников со сторонами, параллельными осям координат. Предусмотреть возможность перемещения прямоугольников на плоскости, изменения размеров, построения наименьшего прямоугольника, содержащего два заданных прямоугольника, и прямоугольника, являющегося общей частью (пересечением) двух прямоугольников.

7. Составить описание класса для определения одномерных массивов целых чисел (векторов). Предусмотреть: возможность обращения к отдельному элементу массива с контролем выхода за пределы индексов, возможность задания произвольных границ индексов при создании объекта и выполнения операций поэлементного сложения и вычитания массивов с одинаковыми границами индексов, умножения и деления всех элементов массива на скаляр, а также печати (вывода на экран) элементов массива по индексам и всего массива.

8. Составить описание класса для определения одномерных массивов строк фиксированной длины. Предусмотреть: возможность обращения к отдельным строкам массива по индексам; контроль выхода за пределы индексов, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов; печать (вывод на экран) элементов массива и всего массива.

9. Составить описание класса многочленов от одной переменной, задаваемых степенью многочлена и массивом коэффициентов. Предусмотреть методы для вычисления значения многочлена для заданного аргумента, операции сложения, вычитания и умножения многочленов с получением нового объекта-многочлена, печать (вывод на экран) описания многочлена.

10. Составить описание класса одномерных массивов строк, каждая строка которых задается длиной и указателем на выделенную для нее память. Предусмотреть: возможность обращения к отдельным строкам массива по индексам; контроль выхода за пределы индексов, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов; печать (вывод на экран) элементов массива и всего массива.

11. Составить описание объектного типа TMatr, обеспечивающего размещение матрицы произвольного размера с возможностью изменения числа строк и столбцов, вывода на экран подматрицы любого размера и всей матрицы.

Лабораторная работа №3

Способы организации массивов в языке программирования C# при реализации информационных проектов

Цель работы: изучить способы организации массивов в языке программирования C#; освоить принципы использования операторов управления и перехода, механизмы работы циклов при реализации информационных проектов.

Теоретические сведения

Существует четыре оператора перехода в языке программирования C#: goto, break, continue, return.

Оператор goto имеет простой синтаксис и семантику:

goto [метка|case константное_выражение|default];

Все операторы языка C# могут иметь метку – уникальный идентификатор, предшествующий оператору и отделенный от него символом двоеточия. Передача управления помеченному оператору – это классическое использование оператора goto. Применение данного оператора не рекомендуется концепцией структурного программирования, так как нарушает структурированность кода и затрудняет понимание последовательности выполнения операторов.

В структурном программировании признаются полезными «переходы вперед», позволяющие при выполнении некоторого условия выйти из цикла, оператора выбора, блока. Для этой цели можно использовать оператор goto, но лучше применять специально предназначенные для этих целей операторы break (прекратить выполнение текущего блока программы и выйти из него на 1 уровень вложенности блоков программы вверх) и continue (продолжить). Оператор break может стоять в теле цикла или завершать case-ветвь в операторе switch. При выполнении оператора break в теле цикла завершается выполнение самого внутреннего цикла. В теле цикла чаще всего оператор break помещается в одну из ветвей оператора if, проверяющего условие преждевременного завершения цикла:

```
public void Jumps() {
        int i = 1, j = 1;
        for (i = 1; i < 100; i++) { // for i
          for (j = 1; j < 10; j++) { // for j
                  if (j >= 3) break;
          } // for j
          Console.WriteLine("Выход из цикла j при j = {0}", j);
```

```
            if (i >= 3) break;
          } // for i
          Console.WriteLine("Выход из цикла i при i= {0}", i);
    } // Jumps
```

Оператор continue используется только в теле цикла. В отличие от оператора break, завершающего внутренний цикл, continue осуществляет переход к следующей итерации этого цикла. К группе операторов перехода также относится оператор return, позволяющий завершить выполнение процедуры или функции. Его синтаксис:

return [выражение];

Для функций его присутствие и аргумент обязательны, поскольку выражение в операторе return задает значение, возвращаемое функцией. Для организации условного ветвления язык C# унаследовал от C и C++ конструкцию if...else. Ее синтаксис должен быть интуитивно понятен для любого, кто программировал на процедурных языках:

```
if (условие)
  оператор (операторы)
else
  оператор (операторы)
```

Если по каждому из условий нужно выполнить более одного оператора, эти операторы должны быть объединены в блок с помощью фигурных скобок {...}. Стоит обратить внимание, что в отличие от языков C и C++, в C# условный оператор if может работать только с булевскими выражениями, но не с произвольными значениями вроде –1 и 0. В операторе if могут применяться сложные выражения, и он может содержать операторы else, обеспечивая выполнение более сложных проверок. Синтаксис похож на применяемый в аналогичных ситуациях в языках C (C++) и Java.

Одним из операторов выбора в C# является оператор switch, который обеспечивает многонаправленное ветвление программы. Следовательно, этот оператор позволяет сделать выбор среди нескольких альтернативных вариантов дальнейшего выполнения программы. Несмотря на то что многонаправленная проверка может быть организована с помощью последовательного ряда вложенных операторов if, во многих случаях более эффективным оказывается применение оператора switch. Этот оператор действует следующим образом. Значение выражения последовательно сравнивается с константами выбора из заданного списка. Как только будет обнаружено совпадение с одним из условий выбора, выполняется последовательность операторов, связанных с этим условием. Ниже приведена общая форма оператора switch:

```
switch(выражение) {
case константа1:
последовательность операторов
break;
case константа2:
последовательность операторов
break;
case константа3:
последовательность операторов
break;
...
default:
последовательность операторов
break;
}
```

Хотя оператор switch...case должен быть знаком программистам на языках C и C++, в C# он немного безопаснее. В частности, он запрещает «сквозные» условия почти во всех случаях. Это значит, что если часть case вызывается в начале блока, то фрагменты кода за последующими частями case не могут быть выполнены, если только не используется явно оператор goto для перехода к ним. Компилятор обеспечивает это ограничение за счет требования, чтобы за каждой частью case следовал оператор break, в противном случае он выдает ошибку. Следует отметить, что заданное выражение в операторе switch должно быть целочисленного типа (char, byte, short или int), перечислимого или же строкового. А выражения других типов, например с плавающей точкой, в операторе switch не допускаются. Зачастую выражение, управляющее оператором switch, сводится к одной переменной. Кроме того, константы выбора должны иметь тип, совместимый с типом выражения. В одном операторе switch не допускается наличие двух одинаковых по значению констант выбора.

Рассмотрим массивы. **Массив** представляет собой набор однотипных переменных. Объявление массива похоже на объявление переменной:

```
тип_переменной[] название_массива
int[] nums = new int[4];
nums[0] = 1;
nums[1] = 2;
nums[2] = 3;
nums[3] = 5;
Console.WriteLine(nums[3]);
```

Здесь вначале мы объявили массив nums, который будет хранить данные типа int. Далее, используя операцию new, мы выделили память для четырех элементов массива: new int[4]. Число четыре также называется размерностью массива. Отсчет элементов массива начинается с 0, поэтому в данном случае, чтобы обратиться к четвертому элементу в массиве, нам надо использовать выражение nums[3]. И так как массив определен только для четырех элементов, то мы не можем обратиться, например, к шестому элементу: nums[5] = 5;. В противном случае мы получим исключение IndexOutOfRangeException.

В предыдущем примере мы сначала создали массив, а потом определили значения для всех его элементов. Но есть и альтернативные пути инициализации массивов:

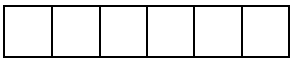
```
int[] nums2 = new int[] { 1, 2, 3, 5 };
int[] nums3 = { 1, 2, 3, 5 };
```

Здесь мы сразу указываем все элементы массива. Массивы бывают одномерными и многомерными. В предыдущих примерах мы создавали одномерные массивы, теперь создадим двухмерный:

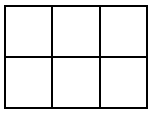
```
int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };
int[,] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };
```

Визуально оба массива можно представить следующим образом.

Одномерный массив nums1:



Двухмерный массив nums2:



Поскольку массив nums2 двухмерный, он представляет собой простую таблицу. Объявление трехмерного массива могло бы выглядеть так:

```
int[,,] nums3 = new int[2, 3, 4];
```

От многомерных массивов надо отличать **массив массивов**, или так называемый «зубчатый массив»:

```
int[][] nums = new int[3][];
nums[0] = new int[2];
nums[1] = new int[3];
nums[2] = new int[5];
```

Здесь две группы квадратных скобок указывают, что это массив массивов – такой массив, который, в свою очередь, содержит в себе другие массивы. В данном случае массив nums содержит три массива, причем размерность каждого из них может не совпадать.

Также мы можем использовать в качестве массивов и многомерные:

```
int[][,] nums = new int[3][,]
{
   new int[,] { {1,2}, {3,4} },
   new int[,] { {1,2}, {3,6} },
   new int[,] { {1,2}, {3,5}, {8, 13} }
};
```

Так, здесь показан пример массива из трех массивов, причем каждый из этих массивов представляет двухмерный массив.

Рассмотрим некоторые методы и свойства массивов. Свойство Length позволяет получить количество элементов массива, свойство Rank – размерность массива. Метод Array.Reverse изменяет порядок следования элементов массива на обратный, метод Array.Sort сортирует элементы массива. Примеры использования:

```
int[] nums1 =new int[] {8, 1, 5, 3, 4, 2};
int length = nums1.Length;
Console.WriteLine("количество элементов: {0}", length);
int rank = nums1.Rank;
Console.WriteLine("размерность массива: {0}", rank);
Array.Reverse(nums1);
Array.Sort(nums1);
Console.WriteLine(nums1[1]);
```

Рассмотрим **операторы цикла**. Цикл for в C# предоставляет механизм итерации, в котором определенное условие проверяется перед выполнением каждой итерации. Синтаксис этого оператора показан ниже:

```
for (инициализатор; условие; итератор)
  оператор (операторы)
```

Инициализатор – это выражение, вычисляемое перед первым выполнением тела цикла (обычно инициализация локальной переменной в качестве счетчика цикла). Инициализация, как правило, представлена оператором присваивания, задающим первоначальное значение переменной, которая выполняет роль счетчика и управляет циклом. **Условие** – это выражение, проверяемое перед каждой новой итерацией цикла (должно возвращать true, чтобы была выполнена следующая итерация). **Итератор** – выражение, вычисляемое после каждой итерации (обычно приращение значения счетчика цикла). Эти три основные части оператора цикла for должны быть разделены точкой с запятой. Выполнение цикла for будет продолжаться до тех пор, пока проверка условия дает истинный результат. Как только эта проверка даст ложный результат, цикл

завершится, а выполнение программы будет продолжено с оператора, следующего после цикла for.

Подобно for, while также является циклом с предварительной проверкой. Синтаксис его аналогичен, но циклы while включают только одно выражение:

while (условие)

оператор (операторы);

Оператор – это единственный оператор или же блок операторов, а условие означает конкретное условие управления циклом и может быть любым логическим выражением. В этом цикле оператор выполняется до тех пор, пока условие истинно. Как только условие становится ложным, управление программой передается строке кода, следующей непосредственно после цикла. Как и в цикле for, в цикле while проверяется условное выражение, указываемое в самом начале цикла. Это означает, что код в теле цикла может не выполняться, и не возникает необходимости осуществлять отдельную проверку перед самим циклом.

Цикл do...while в C# – это версия while с постпроверкой условия цикла после выполнения тела цикла. Следовательно, циклы do...while удобны в тех ситуациях, когда блок операторов должен быть выполнен как минимум однажды. Ниже приведена общая форма оператора цикла do...while:

do {

операторы;

} while (условие);

При наличии лишь одного оператора фигурные скобки в данной форме записи необязательны. Тем не менее они зачастую используются для того, чтобы сделать конструкцию do...while более удобочитаемой и не путать ее с конструкцией цикла while. Цикл do...while выполняется до тех пор, пока условное выражение истинно.

Цикл foreach служит для циклического обращения к элементам коллекции, представляющей собой группу объектов. В C# определено несколько видов коллекций, каждая из которых является массивом. Ниже приведена общая форма оператора цикла foreach:

foreach (тип имя_переменной_цикла in коллекция)

оператор;

Здесь тип имя_переменной_цикла обозначает тип и имя переменной управления циклом, которая получает значение следующего элемента коллекции на каждом шаге выполнения цикла foreach, а коллекция обозначает циклически опрашиваемую коллекцию, которая здесь и далее представляет собой массив. Следовательно, тип переменной цикла должен соответствовать типу элемента массива. Кроме того, тип может обозначаться ключевым словом var.

В этом случае компилятор определяет тип переменной цикла исходя из типа элемента массива, это может оказаться полезным для работы с определенного рода запросами. Но, как правило, тип указывается явным образом.

Оператор цикла foreach действует следующим образом. Когда цикл начинается, первый элемент массива выбирается и присваивается переменной цикла. На каждом последующем шаге итерации выбирается следующий элемент массива, который сохраняется в переменной цикла. Цикл завершается, когда все элементы массива окажутся выбранными. Цикл foreach позволяет проходить по каждому элементу коллекции (объекту, представляющему список других объектов). Формально, для того чтобы класс можно было рассматривать как коллекцию, эта группа должна поддерживать интерфейс IEnumerable. Примерами коллекций могут служить массивы C#, классы коллекций из пространства имен System.Collection, а также пользовательские классы коллекций.

Индивидуальное задание по лабораторной работе

Используя среду разработки Microsoft Visual Studio, создать консольное приложение на языке программирования C#. В приложении необходимо объявить и инициализировать массив и произвести действия согласно варианту, а также вывести полученный результат на экран. Ввод данных должен производиться с клавиатуры.

Варианты

1. В массив A[N] занесены натуральные числа. Найти сумму элементов, кратных данному K.

2. В целочисленной последовательности есть нулевые элементы. Создать массив из номеров этих элементов.

3. Дана последовательность целых чисел a1, a2, ..., an. Выяснить, какое число встречается раньше – положительное или отрицательное.

4. Дана последовательность действительных чисел a1, a2, ..., an. Выяснить, будет ли она возрастающей.

5. Дана последовательность натуральных чисел a1, a2, ..., an. Создать массив из четных чисел этой последовательности. Если таких чисел нет, то вывести сообщение об этом факте.

6. Дана последовательность чисел a1, a2, ..., an. Указать наименьшую длину числовой оси, содержащую все эти числа.

7. Дана последовательность действительных чисел a1, a2, ..., an. Заменить все ее члены, большие данного Z, этим числом. Подсчитать количество замен.

8. Последовательность действительных чисел оканчивается нулем. Найти количество членов этой последовательности.

9. Дан массив действительных чисел, размерность которого равна N. Подсчитать, сколько в нем отрицательных, положительных и нулевых элементов.

10. Даны действительные числа a1, a2, ..., an. Поменять местами наибольший и наименьший элементы.

11. Даны целые числа a1, a2, ..., an. Вывести на печать только те числа, для которых $ai \geq i$.

12. Даны натуральные числа a1, a2, ..., an. Указать те из них, у которых остаток от деления на M равен L ($0 \leq L \leq M - 1$).

13. В заданном одномерном массиве поменять местами соседние элементы, стоящие на четных местах, с элементами, стоящими на нечетных местах.

14. При поступлении в вуз абитуриенты, получившие двойку на первом экзамене, ко второму не допускаются. В массиве A[n] записаны оценки экзаменующихся, полученные на первом экзамене. Подсчитать, сколько человек не допущено ко второму экзамену.

15. Дана последовательность чисел, среди которых имеется один нуль. Вывести на печать все числа до нуля включительно.

16. В одномерном массиве размещены: в первых элементах – значения аргумента, в следующих – соответствующие им значения функции. Напечатать элементы этого массива в виде двух параллельных столбцов (аргумент и значения функции).

Лабораторная работа №4

Методы реализации интерфейсов в языке программирования C# и их применение при реализации информационных проектов

Цель работы: изучить принципы наследования и полиморфизма объектно-ориентированного программирования при реализации интерфейсов в языке программирования C#; освоить способы применения интерфейсов при реализации информационных проектов.

Теоретические сведения

Используя механизм наследования, можно дополнять и переопределять общий функционал базовых классов в классах-наследниках. Однако напрямую мы можем наследовать только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование. В языке C# подобную проблему позволяют решить **интерфейсы**, играющие важную роль в системе ООП. Интерфейсы позволяют определить некоторый функционал, не имеющий конкретной реализации. Затем этот функционал реализуют классы, применяющие данные интерфейсы. Для определения интерфейса используется ключевое слово interface. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а особенность стиля программирования. Интерфейсы, так же как и классы, могут содержать свойства, методы и события, только без конкретной реализации.

Определим следующий интерфейс IAccount, который будет содержать методы и свойства для работы со счетом клиента. Для добавления интефрейса в проект можно нажать правой кнопкой мыши на проект и в появившемся контекстном меню выбрать «Add» → «New Item» и в диалоговом окне добавления нового компонента выбрать «Interface».

Изменим пустой код интерфейса IAccount на следующий:

```
interface IAccount
{
    // текущая сумма на счете
    int CurrentSum { get; }
    // положить деньги на счет
    void Put(int sum);
    // взять со счета
    void Withdraw(int sum);
```

```
    // процент начислений
    int Percentage { get; }
}
```

У интерфейса IAccount методы и свойства не имеют реализации, в этом они сближаются с абстрактными методами абстрактных классов. Сущность данного интерфейса проста: он определяет два свойства (для текущей суммы денег на счете и ставки процента по вкладам) и два метода (для добавления денег на счет и изъятия денег). Еще одна особенность в объявлении интерфейса: все его члены – методы и свойства – не имеют модификаторов доступа, но фактически по умолчанию установлен доступ public, так как цель интерфейса – определение функционала для реализации его классом, поэтому весь функционал должен быть открыт для реализации. Применение интерфейса аналогично наследованию класса:

```
class Client : IAccount
{
// реализация методов и свойств интерфейса
}
```

В качестве примера описывается интерфейс в классе Client, так как клиент обладает счетом:

```
class Client : IAccount
{
    int _sum; // переменная для хранения суммы
    int _percentage; // переменная для хранения процента

    public string Name { get; set; }
    public Client(string name, int sum, int percentage)
    {
        Name = name;
        _sum = sum;
        _percentage = percentage;
    }

    public int CurrentSum
    {
        get { return _sum; }
    }

    public void Put(int sum)
```

```
        {
            _sum += sum;
        }

        public void Withdraw(int sum)
        {
            if (sum <= _sum)
            {
                _sum -= sum;
            }
        }
        public int Percentage
        {
            get { return _percentage; }
        }
        public void Display()
        {
            Console.WriteLine("Клиент " + Name + " имеет счет на сумму " +
_sum);
        }
    }
```

Как и в случае с абстрактными методами абстрактного класса, класс Client реализует все методы интерфейса. Однако, поскольку все методы и свойства интерфейса являются публичными, при их реализации к ним можно применять только модификатор public. Поэтому если класс должен иметь метод с каким-то другим модификатором, например, protected, то интерфейс не подходит для определения подобного метода. Применение класса в программе:

```
Client client = new Client("Tom", 200, 10);
client.Put(30);
Console.WriteLine(client.CurrentSum); //230
client.Withdraw(100);
Console.WriteLine(client.CurrentSum); //130
```

Интерфейсы, как и классы, могут наследоваться:

```
interface IDepositAccount : IAccount
{
    void GetIncome(); // начисление процентов
}
```

При применении этого интерфейса класс Client должен будет реализовать как методы и свойства интерфейса IDepositAccount, так и методы и свойства базового интерфейса IAccount. Рассмотрим интерфейсы в преобразованиях типов. Все сказанное в отношении преобразования типов характерно и для интерфейсов. Поскольку класс Client реализует интерфейс IAccount, то переменная типа IAccount может хранить ссылку на объект типа Client:

```
IAccount client1 = new Client("Том", 200, 10);
client1.Put(200);
Console.WriteLine(client1.CurrentSum); // 400
// интерфейс не имеет метода Display, необходимо явное приведение
((Client)client1).Display();
```

Если необходимо обратиться к методам класса Client, которые не определены в интерфейсе IAccount, а определены в самом классе Client или в его базовом классе, то следует явным образом выполнить преобразование типов: ((Client)client1).Display();.

Как и классы, интерфейсы могут быть обобщенными:

```
interface IAccount<T>
{
   void SetSum(T _sum);
   void Display();
}
class Client<T> : IAccount<T>
{
   T sum=default(T);
   public void SetSum(T _sum)
   {
      this.sum = _sum;
   }
   public void Display()
   {
      Console.WriteLine(sum);
   }
}
```

Интерфейс IAccount типизирован параметром T, который при реализации интерфейса используется в классе Client. В частности, переменная суммы определена как T, что позволяет нам использовать для суммы различные числовые типы. Определим две реализации: одна в качестве параметра будет использовать тип int, а другая – тип double:

```
IAccount<int> intClient = new Client<int>();
intClient.SetSum(300);
intClient.Display();

IAccount<double> doubleClient = new Client<double>();
doubleClient.SetSum(500.09);
doubleClient.Display();
```

Может возникнуть ситуация, когда класс применяет несколько интерфейсов, но они имеют один и тот же метод с одним и тем же возвращаемым результатом и одним и тем же набором параметров, например:

```
class Person : ISchool, IUniversity
{
   public void Study()
   {
      Console.WriteLine("Учеба в школе или в университете");
   }
}
interface ISchool
{
   void Study();
}
interface IUniversity
{
   void Study();
}
```

Класс Person определяет один метод Study(), создавая одну общую реализацию для обоих примененных интерфейсов. И вне зависимости от того, будем ли мы рассматривать объект Person как объект типа ISchool или IUniversity, результат метода будет один и тот же. Однако нередко возникает необходимость разграничить реализуемые интерфейсы. В этом случае надо явным образом применить интерфейс:

```
class Person : ISchool, IUniversity
{
   void ISchool.Study()
   {
      Console.WriteLine("Учеба в школе");
   }
   void  IUniversity.Study()
```

```
    {
        Console.WriteLine("Учеба в университете");
    }
}
```

При явной реализации указывается название метода после названия интерфейса, при этом мы не можем использовать модификатор public, то есть методы являются закрытыми. В этом случае при использовании метода Study в программе нам надо объект Person привести к типу соответствующего интерфейса:

```
static void Main(string[] args)
{
    Person p = new Person();
    ((ISchool)p).Study();
    ((IUniversity)p).Study();

    Console.Read();
}
```

В языке C# допускается объявлять переменные ссылочного интерфейсного типа, то есть переменные ссылки на интерфейс. Такая переменная может ссылаться на любой объект, реализующий ее интерфейс. При вызове метода для объекта посредством интерфейсной ссылки выполняется его вариант, реализованный в классе данного объекта. Этот процесс аналогичен применению ссылки на базовый класс для доступа к объекту производного класса. Переменной ссылки на интерфейс доступны только методы, объявленные в ее интерфейсе, поэтому такую ссылку нельзя использовать для доступа к любым другим переменным и методам, которые не поддерживаются объектом класса, реализующего данный интерфейс.

```
using System;
namespace ConsoleApplication1
{
    public interface IInfo
    {
        void uiName();
        void uiFamily();
        void uiAge();
    }

    class UI : IInfo
```

```
{
    string Name, Family;
    int Age;

    public UI(string Name, string Family, int Age)
    {
        this.Name = Name;
        this.Family = Family;
        this.Age = Age;
    }

    // реализуем интерфейс
    public void uiName()
    {
        Console.WriteLine("Имя пользователя: " + Name);
    }

    public void uiFamily()
    {
        Console.WriteLine("Фамилия: " + Family);
    }

    public void uiAge()
    {
        Console.WriteLine("Возраст: " + Age);
    }

    // собственный метод класса UI
    public void allInfo()
    {
        Console.WriteLine(Name + " " + Family + " " + Age);
    }
}

class Program
{
    static void Main()
    {
```

```
            UI ui1 = new UI(Name: "Ivan", Family: "Ivanov", Age: 26);

            // создадим ссылку на интерфейс
            IInfo obj;
            // используем ссылку на объект ui1
            obj = ui1;
            obj.uiName();
            obj.uiFamily();
            obj.uiAge();
            // вызов собственного метода не разрешается:
            // obj.allInfo();
            Console.ReadLine();
        }
    }
}
```

Индивидуальное задание по лабораторной работе

Используя среду разработки Microsoft Visual Studio, создать консольное приложение на языке программирования C#. Реализовать интерфейсы и показать их работоспособность. Ввод данных должен производиться с клавиатуры.

Варианты

В качестве основы взять задания для лабораторной работы №2 «Способы организации классов и методов в языке программирования C# при реализации информационных проектов».

Лабораторная работа №5

Способы реализации обработки событий в языке программирования C# для организации рефлексии при организации информационных проектов

Цель работы: изучить способы реализации обработки событий в языке программирования C#; освоить методы организации событий в платформе .NET.

Теоретические сведения

Делегат представляет собой объект, содержащий ссылку на метод. Более того, делегат позволяет вызывать метод, на который он ссылается. Таким образом, **делегат** – это безопасный в отношении типов объект, указывающий на другой метод (или список методов) приложения, который может быть вызван позднее. В частности, объект делегата поддерживает три важных фрагмента информации: адрес метода, по которому он вызывается; аргументы (если есть) этого метода; возвращаемое значение (если есть) этого метода. Как только делегат создан и снабжен необходимой информацией, он может динамически вызывать методы, на которые указывает, во время выполнения. Каждый делегат в .NET Framework (включая специальные делегаты) автоматически снабжается способностью вызывать свои методы синхронно или асинхронно. Этот факт значительно упрощает задачи программирования, поскольку позволяет вызывать метод во вторичном потоке выполнения без ручного создания и управления объектом Thread. Тип делегата объявляется с помощью ключевого слова delegate. Ниже приведена общая форма объявления делегата:

delegate возвращаемый_тип имя (список_параметров);

Возвращаемый_тип обозначает тип значения, возвращаемого методами, которые будут вызываться делегатом; имя – конкретное имя делегата; список_параметров – параметры, необходимые для методов, вызываемых делегатом. Как только создан экземпляр делегата, он может вызывать и ссылаться на те методы, возвращаемый тип и параметры которых соответствуют указанным в объявлении делегата.

Суть делегата в том, что он может служить для вызова любого метода с соответствующей сигнатурой и возвращаемым типом. Более того, вызываемый метод может быть методом экземпляра, связанным с отдельным объектом, или же статическим методом, связанным с конкретным классом. При этом важно то, что возвращаемый тип и сигнатура метода должны быть согласованы с теми, которые указаны в объявлении делегата. В таблице 5.1 представлены свойства и методы для работы с делегатами.

Таблица 5.1 – Свойства и методы для работы с делегатами

Член	Назначение
Method	Это свойство возвращает объект System.Reflection.Method, который представляет детали статического метода, поддерживаемого делегатом
Target	Если метод, подлежащий вызову, определен на уровне объекта (то есть не является статическим), то Target возвращает объект, который представляет метод, поддерживаемый делегатом. Если возвращенное Target значение равно null, значит, подлежащий вызову метод является статическим
Combine()	Этот статический метод добавляет метод в список, поддерживаемый делегатом. В C# метод вызывается за счет использования перегруженной операции += в качестве сокращенной нотации
GetInvocationList()	Этот метод возвращает массив типов System.Delegate, каждый из которых представляет определенный метод, доступный для вызова
Remove() RemoveAll()	Эти статические методы удаляют метод (или все методы) из списка вызовов делегата. В C# метод Remove() может быть вызван неявно, посредством перегруженной операции =

Событие представляет собой автоматическое уведомление о том, что произошло некоторое действие. События действуют по следующему принципу. Объект, проявляющий интерес к событию, регистрирует обработчик этого события. Когда же событие происходит, вызываются все зарегистрированные обработчики этого события, которые обычно представлены делегатами. События являются членами класса и объявляются с помощью ключевого слова event. Чаще всего для этой цели используется следующая форма:

event делегат_события имя_события;

Делегат_события обозначает имя делегата, используемого для поддержки события, а имя_события – конкретный объект объявляемого события. События основаны на делегатах и предоставляют им механизм публикации/подписки. В каркасе .NET события присутствуют повсюду. В приложениях Windows класс Button поддерживает событие Click, которое является делегатом. Метод-обработчик, вызываемый с событием Click, должен быть определен с параметрами, заданными в типе делегата. Как и делегаты, события поддерживают групповую адресацию, это дает возможность нескольким объектам реагировать на уведомление о событии.

Методы экземпляра и статические методы могут быть использованы в качестве обработчиков событий, но между ними имеется одно существенное отличие. Когда статический метод используется в качестве обработчика, уведомление о событии распространяется на весь класс; когда в качестве обработчика

используется метод экземпляра, то события адресуются конкретным экземплярам объектов. Следовательно, каждый объект определенного класса, которому требуется получить уведомление о событии, должен быть зарегистрирован отдельно. На практике большинство обработчиков событий представляет собой методы экземпляра, однако это зависит от конкретного приложения.

Многоадресная передача – это способность создавать список автоматических вызовов методов, которые должны совершаться при вызове делегата. Создается экземпляр делегата, а затем для добавления методов в эту цепочку используется оператор «+=», а для удаления из цепочки вызовов используется оператор «–=». Делегат многоадресной передачи имеет одно ограничение: он должен возвращать тип void.

События – это особый тип многоадресных делегатов, их можно вызвать только из класса или структуры, в которой они объявлены (класс издателя). Если на событие подписаны другие классы или структуры, их методы обработчиков событий будут вызваны, когда класс издателя инициирует событие.

Предусмотрены две формы записи инструкций, связанных с событиями. Форма, используемая в предыдущих примерах, обеспечивает создание событий, которые автоматически управляют списком вызова обработчиков, включая такие операции, как добавление обработчиков в список и удаление их из списка. Таким образом, можно не беспокоиться о реализации операций по управлению этим списком, поэтому такие типы событий являются наиболее применимыми. Однако можно и самостоятельно организовать ведение списка обработчиков событий, чтобы, например, реализовать специализированный механизм хранения событий. Для управления списком обработчиков событий используется вторая форма event-инструкции, которая позволяет использовать средства доступа к событиям. Эти средства доступа дают возможность управлять реализацией списка обработчиков событий. Упомянутая форма имеет следующий вид:

```
event событийный_делегат имя_события{ add {
// код добавления события в цепочку событий
}
remove {
// код удаления события из цепочки событий
}
}
```

Эта форма включает в себя два средства доступа к событиям: add и remove. Средство доступа add вызывается в случае, когда с помощью оператора «+=» в цепочку событий добавляется новый обработчик; средство доступа remove вызывается, когда с помощью оператора «–=» из цепочки событий уда-

ляется новый обработчик. Средство доступа add (или remove) при вызове получает в качестве параметра обработчик, который необходимо добавить или удалить. Этот параметр, как и в случае использования других средств доступа, называется value. При реализации средств доступа add и remove можно задать собственную схему хранения обработчиков событий. Например, для этого можно использовать массив, стек или очередь. Приведем event-инструкцию, в которой используются событийные средства доступа:

```
public event MyEventHandler SomeEvent {
//добавляем обработчик события в список
add {
int i;
for(i=0; i < 3; i++) if(evnt[i] == null) {
evnt[i] = value; break;
}
if(i == 3) Console.WriteLine(
"Список обработчиков событий полон.");
}
//удаляем обработчик события из списка
remove { int i;
for(i=0; i < 3; i++) if(evnt[i] == value) {
evnt[i] = null; break;
}
if(i == 3)
Console.WriteLine("Обработчик события не найден.");
}
}
```

При добавлении в список обработчика событий вызывается add-средство, и ссылка на этот обработчик (содержащаяся в параметре value) помещается в первый встретившийся неиспользуемый элемент массива event. Если свободных элементов нет, выдается сообщение об ошибке. Поскольку массив event рассчитан на хранение трех элементов, он может принять только три обработчика событий. При удалении заданного обработчика событий вызывается remove-средство, и в массиве evnt выполняется поиск ссылки на обработчик, переданной в параметре value. Если ссылка найдена, в соответствующий элемент массива помещается значение null, что равнозначно удалению обработчика из списка.

При генерировании события вызывается метод OnSomeEvent(), который в цикле просматривает массив event, по очереди вызывая каждый обработчик со-

бытий. Как показано в предыдущих примерах, при необходимости нетрудно реализовать собственный механизм хранения обработчиков событий. Для большинства приложений все же лучше использовать стандартный механизм хранения, в котором не используются событийные средства доступа. Однако в определенных ситуациях форма event-инструкции, ориентированной на событийные средства доступа, может оказаться весьма полезной. Например, если в программе обработчики событий должны выполняться в порядке уменьшения приоритетов, а не в порядке их добавления в событийную цепочку, то для хранения таких обработчиков можно использовать очередь по приоритету.

События можно определять в интерфейсах. «Поставкой» событий должны заниматься соответствующие классы. События можно определять как абстрактные, и обеспечить реализацию таких событий должен производный класс. Однако события, реализованные с использованием средств доступа add и remove, абстрактными быть не могут. Любое событие можно определить с помощью ключевого слова sealed. Событие может быть виртуальным, то есть его можно переопределить в производном классе.

C# позволяет программисту создавать события любого типа. Однако в целях компонентной совместимости со средой .NET Framework необходимо следовать рекомендациям, подготовленным Microsoft специально для этих целей. Центральное место в этих рекомендациях занимает требование того, чтобы обработчики событий имели два параметра: первый должен быть ссылкой на объект, который будет генерировать событие; второй должен иметь тип EventArgs и содержать остальную информацию, необходимую обработчику. Таким образом, .NET-совместимые обработчики событий должны иметь следующую общую форму записи:

void handler(object *source*, EventArgs*arg*) { // ...
}

Обычно параметр source передается вызывающим кодом. Параметр типа EventArgs содержит дополнительную информацию, которую в случае ненадобности можно проигнорировать.

Класс EventArgs не содержит полей, которые используются при передаче дополнительных данных обработчику; он используется в качестве базового класса, из которого можно выводить класс, содержащий необходимые поля. Но, поскольку многие обработчики обходятся без дополнительных данных, в класс EventArgs включено статическое поле Empty, которое задает объект, не содержащий никаких данных.

Индивидуальное задание по лабораторной работе

Используя среду разработки Microsoft Visual Studio, создать консольное приложение на языке программирования C#. На основе результата лабораторной работы №4 реализовать обработку событий, реагирующую на изменение данных в реализованных структурах. Реакции на событие должны быть в виде вывода сообщения на экран и звукового сигнала. Ввод данных должен производиться с клавиатуры.

Лабораторная работа №6

Создание Windows-приложений в качестве основы графического интерфейса на языке C# при реализации информационных проектов

Цель работы: изучить принципы организации и разработки Windows-приложений в среде разработки Microsoft Visual Studio; освоить методы чтения и записи данных в файл с помощью платформы .NET; изучить способы реализации обработки исключительных ситуаций в языке программирования C#.

Теоретические сведения

Для создания новых проектов на основании заданных пользователем параметров в Microsoft Visual Studio используются шаблоны проектов. Каждый шаблон представляет определенный тип проекта. Отдельные файлы, добавляемые пользователями в проекты, создаются на основе шаблонов элементов.

Установленные шаблоны проекта можно найти в диалоговом окне «Создать проект», развернув список на левой панели в разделе «Установлено». Можно также перейти в раздел «Последние» и выбрать тип проекта, который был использован недавно, или в раздел «В сети» для поиска шаблонов проектов, которые можно загрузить с веб-сайта.

При создании проекта автоматически создается решение, если этот проект не является уже частью решения.

Создание нового проекта и содержащего его решения происходит следующим образом.

1. В меню «Файл» последовательно выберите пункты «Создать» и «Создать проект». Откроется диалоговое окно «Новый проект».

2. В левой области выберите «Установлено» и выделите категорию типов проектов из развернутого списка. Если необходим однотипный недавно созданный проект, выберите «Последние» для быстрого перехода.

3. В средней области в разделе «Шаблоны» выберите один из шаблонов проекта. В правой области отображается описание выбранного шаблона.

4. В поле «Имя» введите имя нового проекта.

5. В поле «Расположение» выберите место для сохранения проекта.

6. Если доступен, в списке «Решение» укажите, следует ли создать решение или добавить проект в решение, которое открыто в «Обозревателе решений».

7. В поле «Имя решения» введите имя решения. При возможности Visual Studio использует это имя для пространства имен завершенного проекта. Имя решения будет соответствовать имени продукта по умолчанию.

8. Убедитесь, что установлен флажок «Создать каталог для решения».

9. Нажмите кнопку «ОК».

Чтобы создать Windows-приложение, необходимо на этапах 2 и 3 выбрать соответствующие опции.

Рассмотрим методы чтения и записи в файл. Операции чтения и записи данных в файлы очень просты, однако в .NET Framework 4 они выполняются не через объекты DirectoryInfo и FileInfo, а через объект File. Позже будет показано, как можно их выполнять с помощью других классов, представляющих обобщенную концепцию потока (stream). До появления .NET Framework 2.0 по поводу того, как следует производить чтение и запись данных в файлы, велось много споров. Классы из .NET Framework можно было использовать, но такой подход не был простым. В версии .NET Framework 2.0 класс File был расширен, с его помощью стало возможным выполнение операций чтения и записи данных в файлы с помощью всего одной строки кода, что доступно и в .NET Framework 4.

Для открытия файлов и работы с ними предназначены также методы ReadAllBytes и ReadAllLines. Метод ReadAllBytes позволяет открывать двоичные файлы и считывать их содержимое в байтовый массив. Упомянутый выше метод ReadAllText возвращает все содержимое файла в виде одной строки, однако такое поведение не всегда подходит. Вместо этого может требоваться работа с содержимым файла в построчной манере, то есть строка за строкой, для чего служит метод ReadAlllLines, который возвращает массив строк с содержимым файла.

Помимо чтения, библиотека базовых классов (Base Class Library – BCL) в .NET Framework существенно упрощает также и запись данных в файлы. Вдобавок к методам для чтения файлов ReadAllText(), ReadAllLines() и ReadAllBytes() в ней предлагаются методы, предназначенные для записи данных в файлы – WriteAllText(), WriteAllBytes() и WriteAllLines().

Основным для потоков является класс System.IO.Stream, он представляет собой байтовый поток и является базовым для всех остальных классов потоков. Кроме того, он является абстрактным классом, значит, получить экземпляр объекта класса Stream нельзя. В абстрактном классе System.IO.Stream определен набор членов, которые обеспечивают поддержку синхронного и асинхронного взаимодействия с хранилищем (например, с файлом или областью памяти). Концепция потока не ограничена файловым вводом-выводом. Для точности

следует отметить, что библиотеки .NET предоставляют потоковый доступ к сетям, областям памяти и прочим абстракциям, связанным с потоками. Потомки класса Stream представляют данные как низкоуровневые потоки байт, а непосредственная работа с низкоуровневыми потоками может оказаться довольно специфичной. Некоторые типы, унаследованные от Stream, поддерживают поиск (seeking), что означает возможность получения и изменения текущей позиции в потоке.

Иногда при выполнении программы возникают ошибки, которые трудно или невозможно предусмотреть. Например, при передаче файла по сети может неожиданно оборваться сетевое подключение. Такие ситуации называются **исключениями**. Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция try...catch...finally. При возникновении исключения среда CLR ищет блок catch, который может обработать данное исключение. Если такого блока не найдено, то пользователю отображается сообщение о необработанном исключении, а дальнейшее выполнение программы останавливается. И чтобы подобной остановки не произошло, надо использовать блок try..catch.

```
static void Main(string[] args)
{
   int[] a = new int[4];
   try
   {
      a[5] = 4; // тут возникнет исключение, так как в массиве только 4 элемента
      Console.WriteLine("Завершение блока try");
   }
   catch (Exception ex)
   {
      Console.WriteLine("Ошибка: " + ex.Message);
   }
   finally
   {
      Console.WriteLine("Блок finally");
   }
   Console.ReadLine();
}
```

При использовании блока try...catch..finally вначале выполняются все инструкции между операторами try и catch. Если между этими операторами вдруг

возникает исключение, то обычный порядок выполнения останавливается и переходит к инструкции catch. Инструкция catch имеет следующий синтаксис:

catch (тип_исключения имя_переменной)

В данном случае объявляется переменная ex, которая имеет тип Exception. Но если возникшее исключение не является исключением типа, указанного в инструкции catch, то оно не обрабатывается, а программа зависает или выводит сообщение об ошибке. Однако так как тип Exception является базовым классом для всех исключений, то выражение catch (Exception ex) будет обрабатывать практически все исключения. Вся обработка исключения в таком случае сводится к выводу на консоль сообщения об исключении, которое находится в свойстве message класса Exception.

Далее в любом случае выполняется блок finally. Однако этот блок необязательный, и его можно при обработке исключений опускать. Если же в ходе программы исключений не возникнет, то программа не будет выполнять блок catch, а сразу перейдет к блоку finally (при его наличии). Блок finally является полезным для запуска любого кода, который должен выполняться, даже если есть исключение. Управление всегда передается блоку finally независимо от того, как был выполнен выход из блока try...catch. Код в блоке finally выполняется, даже если ваш код встречает оператор return в блоке try или catch. Элемент управления не передает из блока try или блока catch в соответствующий блок finally в двух случаях: оператор end обнаруживается внутри блока try или catch; StackOverflowException создается внутри блока try или catch. Недопустимо явно передавать управление в блок finally. Досрочное прерывание исполнения кода в блоке finally и выход из него возможны только посредством исключений; другие способы передачи управления наружу блока являются недопустимыми. Если инструкция try не содержит ни одного блока catch, то она должна содержать finally.

В рассматриваемых выше примерах перехватывались исключения, генерировавшиеся исполняющей системой автоматически. Но исключение может быть сгенерировано и вручную с помощью оператора throw. Ниже приведена общая форма такого генерирования:

throw exceptOb;

В качестве exceptOb должен быть обозначен объект класса исключений, производного от класса Exception. Ниже приведен пример программы, в которой демонстрируется применение оператора throw для генерирования исключения DivideByZeroException.

```
// сгенерировать исключение вручную
using System;
```

```
class ThrowDemo {
   static void Main() {
      try {
         Console.WriteLine("До генерирования исключения.");
         throw new DivideByZeroException();
      }
      catch (DivideByZeroException) {
         Console.WriteLine("Исключение перехвачено.");
      }
      Console.WriteLine("После пары операторов try/catch.");
   }
}
```

Результат выполнения этой программы можно увидеть на рисунке 6.1.

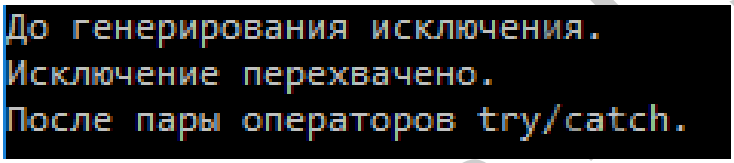


Рисунок 6.1 – Результат работы блоков try…catch

Исключение DivideByZeroException было сгенерировано с использованием ключевого слова new в операторе throw. Не следует забывать, что в данном случае генерируется конкретный объект, а следовательно, он должен быть создан перед генерированием исключения, и сгенерировать исключение только по его типу нельзя. В данном примере для создания объекта DivideByZeroException был автоматически вызван конструктор, используемый по умолчанию, хотя для генерирования исключений доступны и другие конструкторы.

Необходимо предоставить пользователю возможность на программном уровне различать условия возникновения ряда ошибок, например, создать свои пользовательские исключения. Платформа .NET Framework предоставляет иерархию классов исключений, производных от базового класса Exception. Каждый из этих классов самостоятельно определяет конкретное исключение, так что во многих случаях требуется только выполнить его перехват. Также можно создавать собственные классы исключений, производные от класса Exception.

При создании собственных исключений рекомендуется завершать имя класса пользовательского исключения словом «Exception». Также рекомендуется реализовать три общих конструктора.

У System.Exception имеются три общедоступных конструктора и один защищенный. Общедоступные конструкторы включают: конструктор по умолчанию, который не играет важной роли; конструктор, принимающий ссылку на строковый объект (строка представляет собой общее, определяемое программистом сообщение, которое можно рассматривать как более дружественное к пользователю описание исключения); третий конструктор также принимает строку сообщения и вдобавок ссылку на другой объект Exception.

Ссылка на другое исключение позволяет отслеживать исходные исключения, когда внутри блока try одно исключение транслируется в другое. Хорошим примером может служить ситуация, когда исключение не обрабатывается, а просачивается вверх, во фрейм стека статического конструктора. В этом случае исполняющая система генерирует исключение TypelnitializationException, но только после установки внутреннего исключения в исходное исключение, чтобы пользователю, перехватывающему TypelnitializationException, была известна причина исключения.

Защищенный конструктор, в свою очередь, позволяет создавать исключение из объекта SerializationInfo. Сериализуемые исключения создаются, чтобы их можно было использовать через границы контекстов, например, с .NET Remoting. Это значит, что пользовательские классы исключений также понадобится пометить атрибутом SerializableAttribute.

Благодаря трем рассмотренным общедоступным конструкторам, класс System.Exception очень полезен. Однако простую генерацию объектов типа System.Exception при каждом случае сбоя в программе следует считать плохим дизайном. Вместо этого целесообразно создать новый специфичный тип исключения, унаследовав его от System.Exception. Такой тип должен быть более выразительным при описании вызвавшей его проблемы, особенно если производный класс может содержать данные, которые соответствуют причине генерации данного исключения. И следует помнить, что в C# все исключения должны наследоваться от System.Exception.

```
using System;
public class EmployeeListNotFoundException: Exception
{
   public EmployeeListNotFoundException()
   {   }
   public EmployeeListNotFoundException(string message)
```

```
        : base(message)
    {   }
    public EmployeeListNotFoundException(string message, Exception inner)
        : base(message, inner)
    {   }
}
```

Индивидуальное задание по лабораторной работе

Используя среду разработки Microsoft Visual Studio, создать Windows-приложение на языке программирования C#. На основе результата лабораторной работы №5 реализовать возможность чтения и записи информации об объектах в файлы, корректный вывод информации об объектах на окно приложения, обработку событий различных объектов формы, обработку исключений. Информация об объектах должна вводиться через специальную форму.

Лабораторная работа №7

Реализация главного меню, диалоговых окон, строк состояния Windows-приложения в языке программирования C# как основы взаимодействия приложения и пользователя при создании информационных проектов

Цель работы: освоить способы реализации главного меню в Windows-приложении, изучить принципы отображения диалоговых окон с помощью компонентов платформы .NET; освоить основные принципы практического применения строк состояния в языке программирования C#.

Теоретические сведения

Диалоговые окна используются для взаимодействия с пользователем и получения информации. Иначе говоря, диалоговое окно представляет собой форму со значением ее свойства перечисления FormBorderStyle, установленным в FixedDialog. С помощью конструктора Windows Forms в Visual Studio можно создавать собственные пользовательские диалоговые окна. Для настройки диалоговых окон под конкретные потребности можно добавить элементы управления, такие как Label, Textbox и Button. Платформа .NET также включает стандартные диалоговые окна, например, «Открыть файл», и окна сообщений, которые можно использовать для собственных приложений.

Строка состояния – это специальный элемент окна, состоящий из нескольких панелей для отображения текущей информации о состоянии и режиме работы приложения. На рисунке 7.1 показана строка состояния, отображающая состояние приложения, дату и текущее системное время. Этот элемент интерфейса обычно размещается в нижней части родительского окна приложения, если не требуется специально установить его в другом месте окна. Такое положение строки состояния является стандартным.

Рисунок 7.1 – Строка состояния приложения

Для добавления строки состояния в форму используется элемент управления statusBar. Чтобы этот объект можно было использовать в приложении, необходимо в окне «Components» («Компоненты») подключить к выбранному проекту библиотеку «Microsoft Window Common Control 6.0». После подключе-

ния библиотеки элемент управления «StatusBar» появляется на панели элементов управления среды проектирования, и его можно добавить в форму стандартным способом, как и все другие элементы управления.

Строка состояния состоит из набора панелей, каждая из которых является объектом и имеет следующие основные свойства. При организации работы с приложением в диалоговом режиме часто бывает необходимым подать определенную команду работающей программе, выбрать необходимый режим работы или осуществить какое-либо стандартное действие. В интерфейсе командной строки эта команда набиралась бы с помощью клавиатуры, но, поскольку это не удобно, для передачи команд приложению в графическом интерфейсе используется система меню.

Меню может быть горизонтальным, в котором пункты расположены один за другим в одну строку, и вертикальным, у которого пункты расположены друг над другом.Также меню бывает системным (system) и всплывающим (popup). Системное меню расположено сразу под заголовком окна и отображается постоянно. Всплывающее же меню (являющееся, как правило, вертикальным) появляется, когда пользователь или выбрал какой-либо пункт системного меню, или нажал кнопку вызова меню (например, кнопку контрольного меню), или вызвал контекстное меню с помощью правой кнопки мыши. Затем закрывается сразу же, как только выбран его пункт или когда оно теряет «фокус ввода» (при щелчке любой кнопкой мыши вне области меню или нажатием клавиши Esc).

Текст меню содержит краткое описание команды (в одно-два слова). Выделенный символ позволяет быстро выбрать пункт меню с помощью клавиатуры. Для этого надо активизировать меню и нажать клавишу, соответствующую выделенному символу. При этом команда сразу же выполняется. Для активизации нужного пункта системного меню с помощью клавиатуры необходимо нажать клавишу Alt, а затем – выделенный символ (hotkey). Клавиши-акселераторы указывают, каким образом можно выполнить команды с помощью клавиатуры, не активизируя меню. При работе с клавиатурой это значительно ускоряет вызов команды. Специальные знаки указывают на тип пункта меню.

Иконка, указанная слева от текста, является обозначением той кнопки с панели инструментов приложения, которая может использоваться для ускорения ввода команды с помощью мыши. Этот элемент может присутствовать только во всплывающем меню. Некоторая величина, стоящая справа от названия пункта, является значением некоторой переменной, управляющей выражением, а текст пункта является ее именем. При выборе пункта происходит изменение значения этой переменной.

Индивидуальное задание по лабораторной работе

Используя среду разработки Microsoft Visual Studio, создать Windows-приложение на языке программирования C#. На основе результата лабораторной работы №6 «Создание Windows-приложений в качестве основы графического интерфейса на языке C# при реализации информационных проектов» реализовать главное меню, предусматривающее различные действия с приложением, с помощью компонентов .NET, настроить диалоговые окна для открытия и сохранения информации об объектах в файлы, предусмотреть вывод служебной информации в строку состояния. Используя SQL Server Management Studio, разработать структуру базы данных.

Лабораторная работа №8

Создание и представление базы данных в Windows-приложении с помощью встроенных компонентов платформы .NET при реализации информационных проектов

Цель работы: освоить методику создания баз данных с помощью Microsoft SQL Server Management Studio; изучить способ подключения баз данных посредством компоненты ADO.NET Entity Data Model платформы .NET; освоить принципы отображения структурированной информации в Windows-приложении.

Теоретические сведения

Для создания базы данных воспользуемся SQL Server Management Studio. При запуске данной среды появится запрос на соединении с сервером (рисунок 8.1).

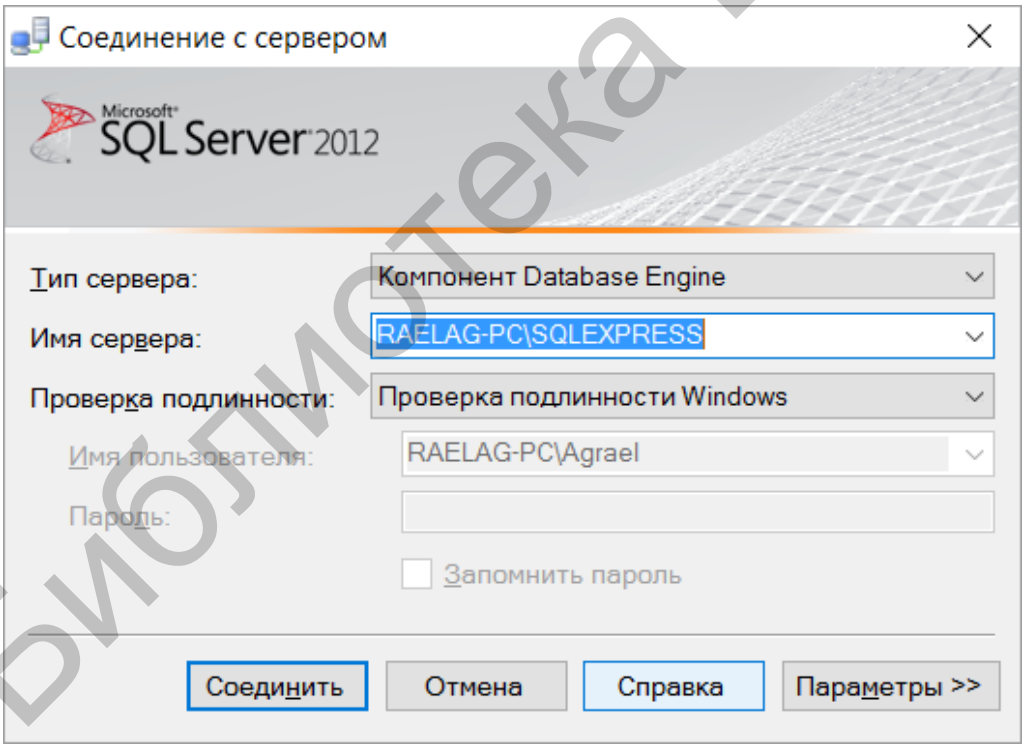


Рисунок 8.1 – Окно запроса на соединение с сервером

После соединения с сервером в главном окне среды слева отобразится список содержимого сервера, в частности «Базы данных» (рисунок 8.2).

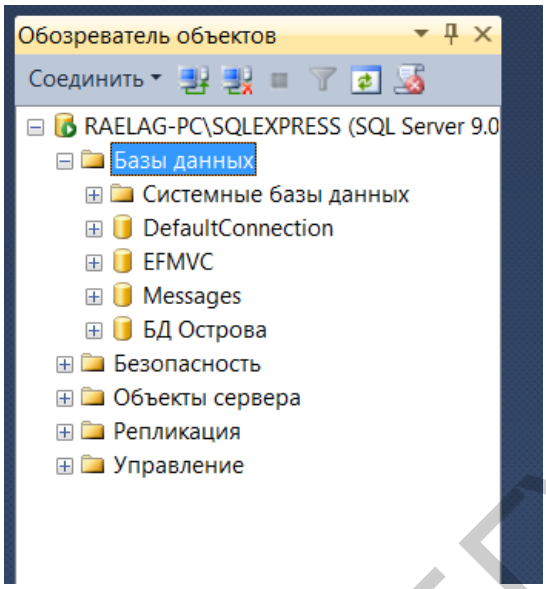


Рисунок 8.2 – Список содержимого SQL сервера

Для создания новой базы данных необходимо вызвать контекстное меню содержимого баз данных и выбрать пункт меню «Создать базу данных…», после чего отобразится окно создания новой базы данных, в которое необходимо ввести новое имя базы данных (рисунок 8.3).

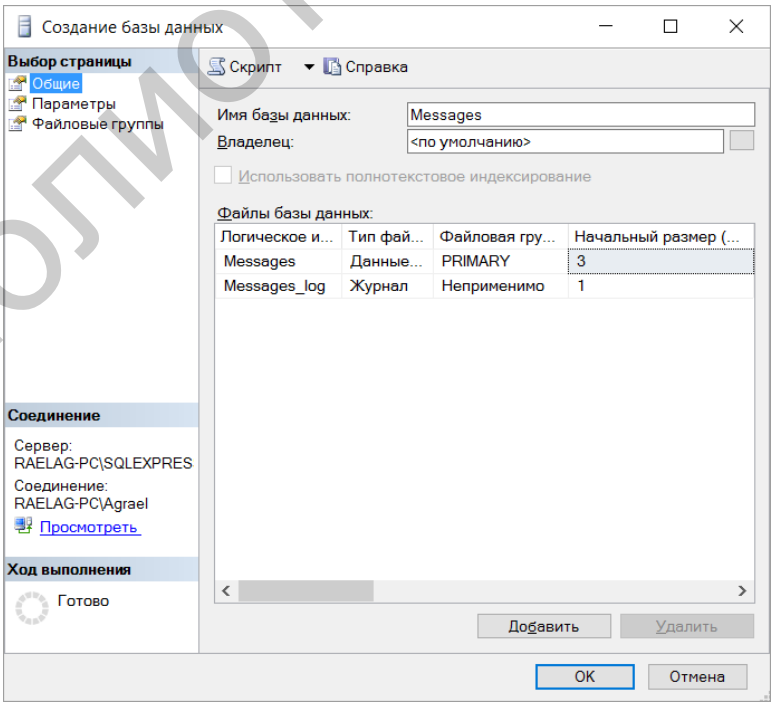


Рисунок 8.3 – Окно создания новой базы данных

После создания базы данных в главном окне слева отображается содержимое базы данных. Сюда входят диаграммы, таблицы, представления и др. (рисунок 8.4).

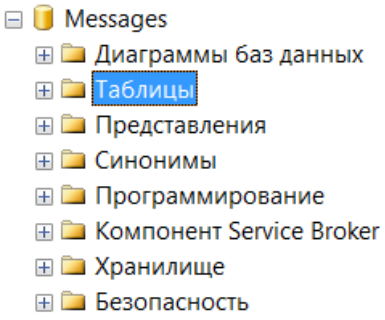


Рисунок 8.4 – Содержимое базы данных

Для создания новой таблицы (сущности) необходимо вызвать контекстное меню таблиц и выбрать пункт «Создать таблицу…», после чего в главном окне отобразится форма создания полей таблицы (рисунок 8.5), где необходимо ввести имя поля, выбрать тип поля (рисунок 8.6) и установить разрешение на значения поля, равные NULL.

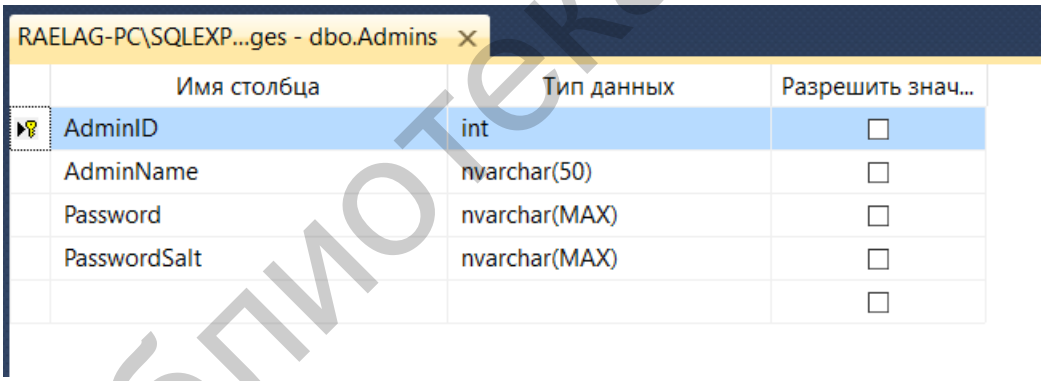


Рисунок 8.5 – Окно создания полей таблицы

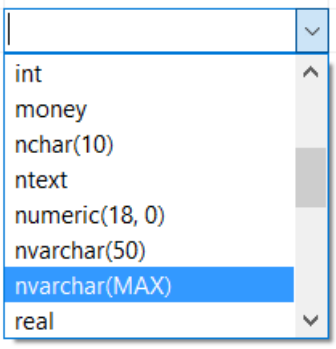


Рисунок 8.6 – Выпадающее меню возможных типов данных

В правой части главного окна находятся свойства текущей таблицы (рисунок 8.7).

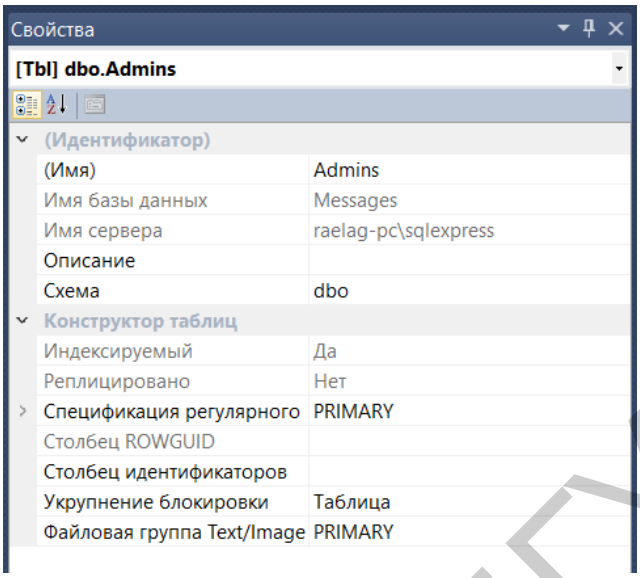


Рисунок 8.7 – Свойства таблицы

В нижней части главного окна отображаются свойства конкретного столбца (рисунок 8.8).

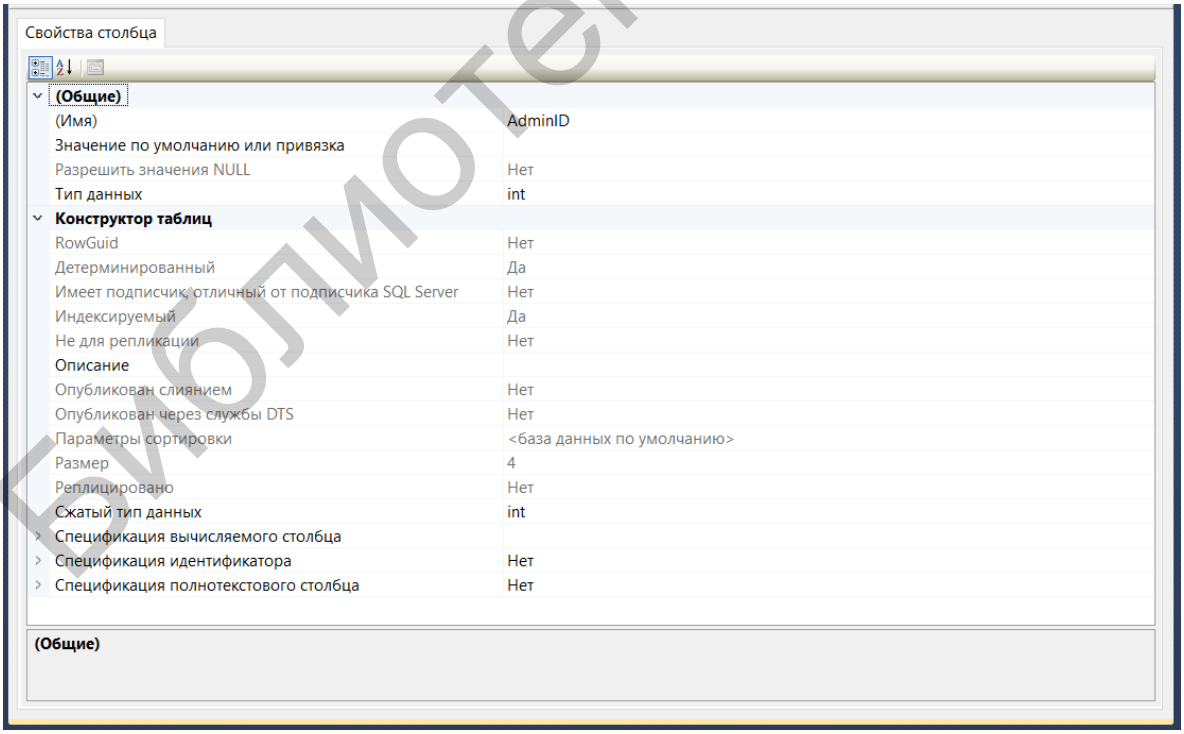


Рисунок 8.8 – Свойства отдельного столбца

Чтобы из поля сделать первичный ключ, необходимо вызвать контекстное меню столбца и выбрать пункт меню «Задать первичный ключ» (рисунок 8.9).

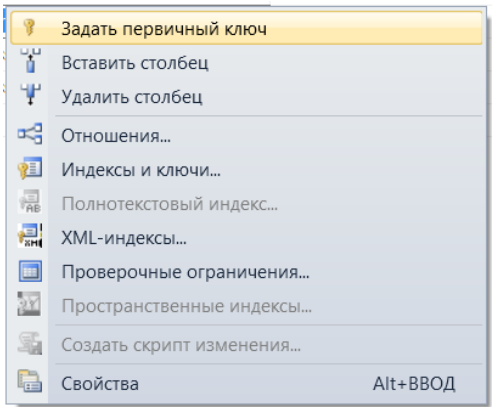


Рисунок 8.9 – Контекстное меню столбца таблицы

Для связывания полей различных таблиц воспользуемся диаграммой базы данных. Для создания диаграммы необходимо вызвать контекстное меню диаграмм баз данных в содержимом базы данных (в левой части главного окна среды) и выбрать пункт меню «Создать диаграмму базы данных». На экране отобразится окно с существующими таблицами (рисунок 8.10).

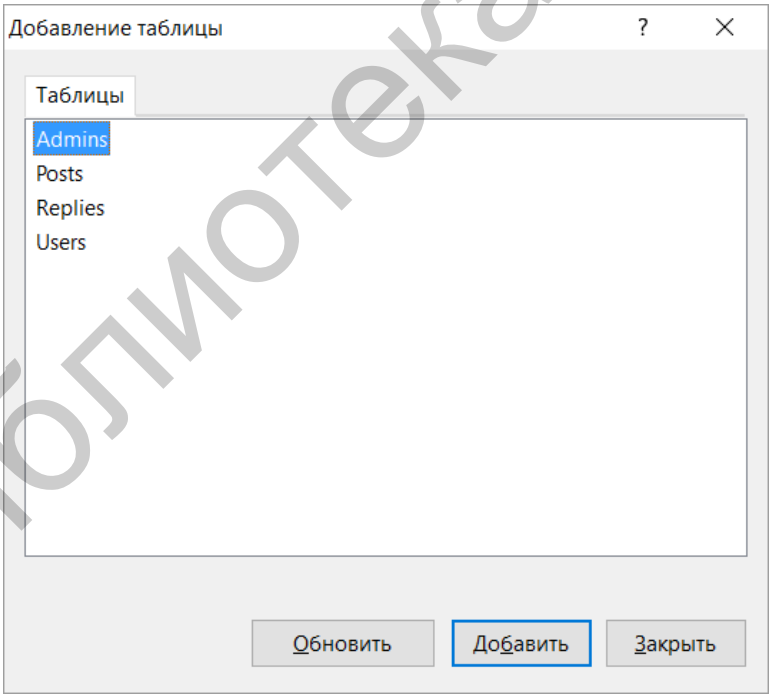


Рисунок 8.10 – Окно добавления таблиц в диаграмму

После добавления всех необходимых таблиц отобразится окно диаграммы, в котором располагаются схематические изображения таблиц (рисунок 8.11).

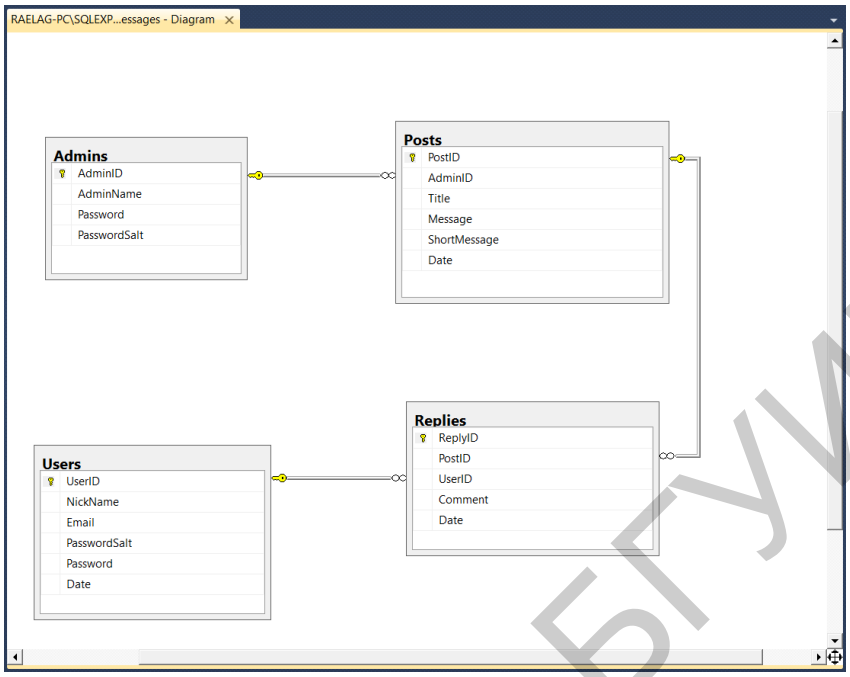


Рисунок 8.11 – Диаграмма базы данных

Чтобы создать связь между таблицами с помощью диаграммы базы данных, необходимо перетащить поле первичного ключа одной таблицы на другую таблицу с полем, которое будет составлять внешний ключ. Отобразится окно с созданием связи между таблицами (рисунок 8.12).

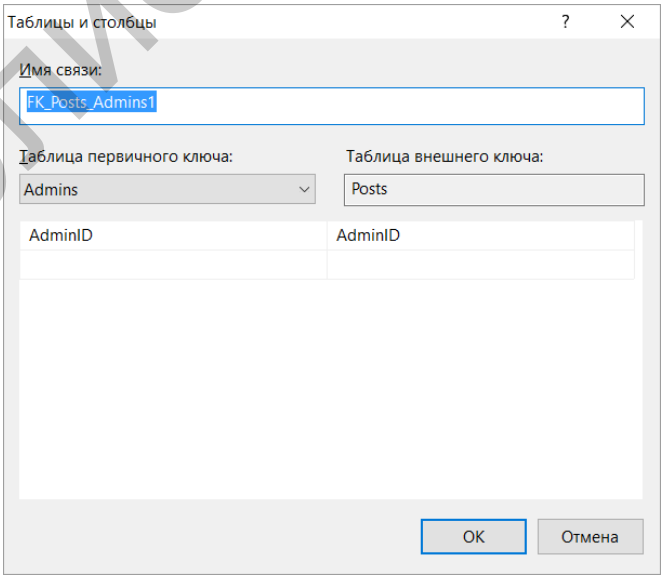


Рисунок 8.12 – Окно создания связи между таблицами

Затем необходимо задать определенные свойства данной связи между таблицами (рисунок 8.13).

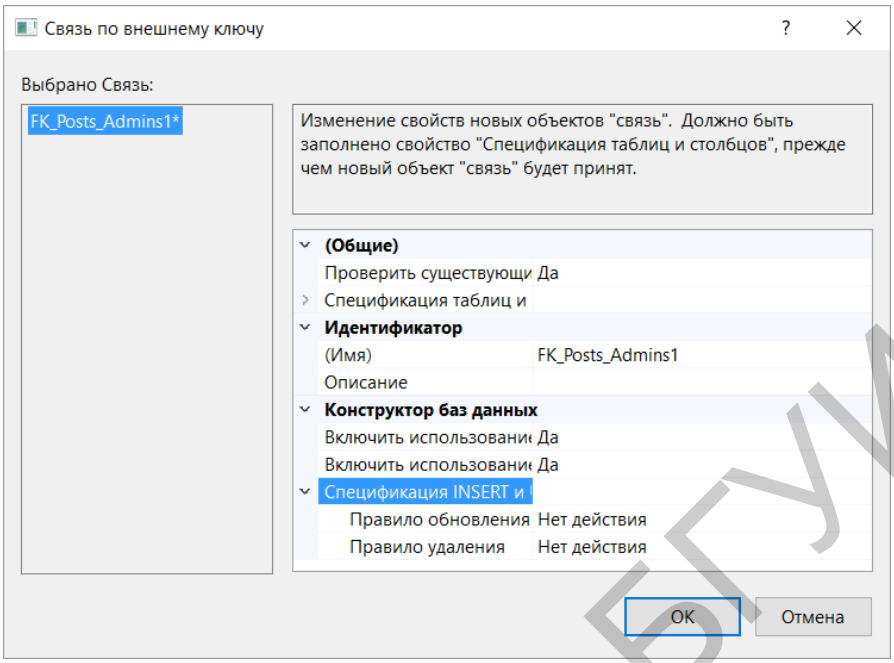


Рисунок 8.13 – Окно свойств созданной связи

Для редактирования данных таблицы необходимо вызвать контекстное меню таблицы и выбрать пункт меню «Изменить первые 200 строк» (рисунок 8.14).

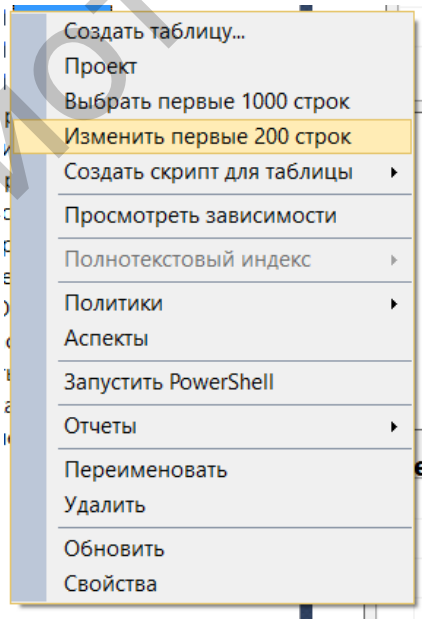


Рисунок 8.14 – Контекстное меню таблицы

В главном окне отобразится форма для заполнения данными (рисунок 8.15).

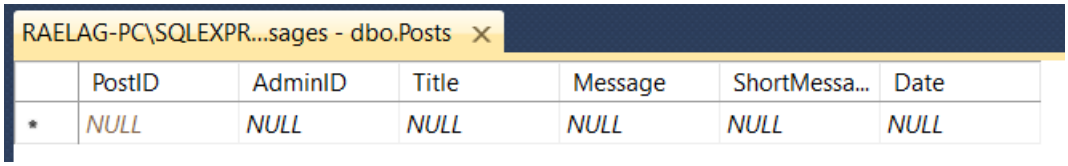


Рисунок 8.15 – Форма для заполнения таблицы данными

В случае если необходимо изменить структуру базы данных, изменить связи между таблицами, создать новую таблицу, создать или удалить столбец в таблице, необходимо в параметрах среды убрать галочку с соответствующей опции (рисунок 8.16). В противном случае при сохранении среда разработки выдаст сообщение об ошибке.

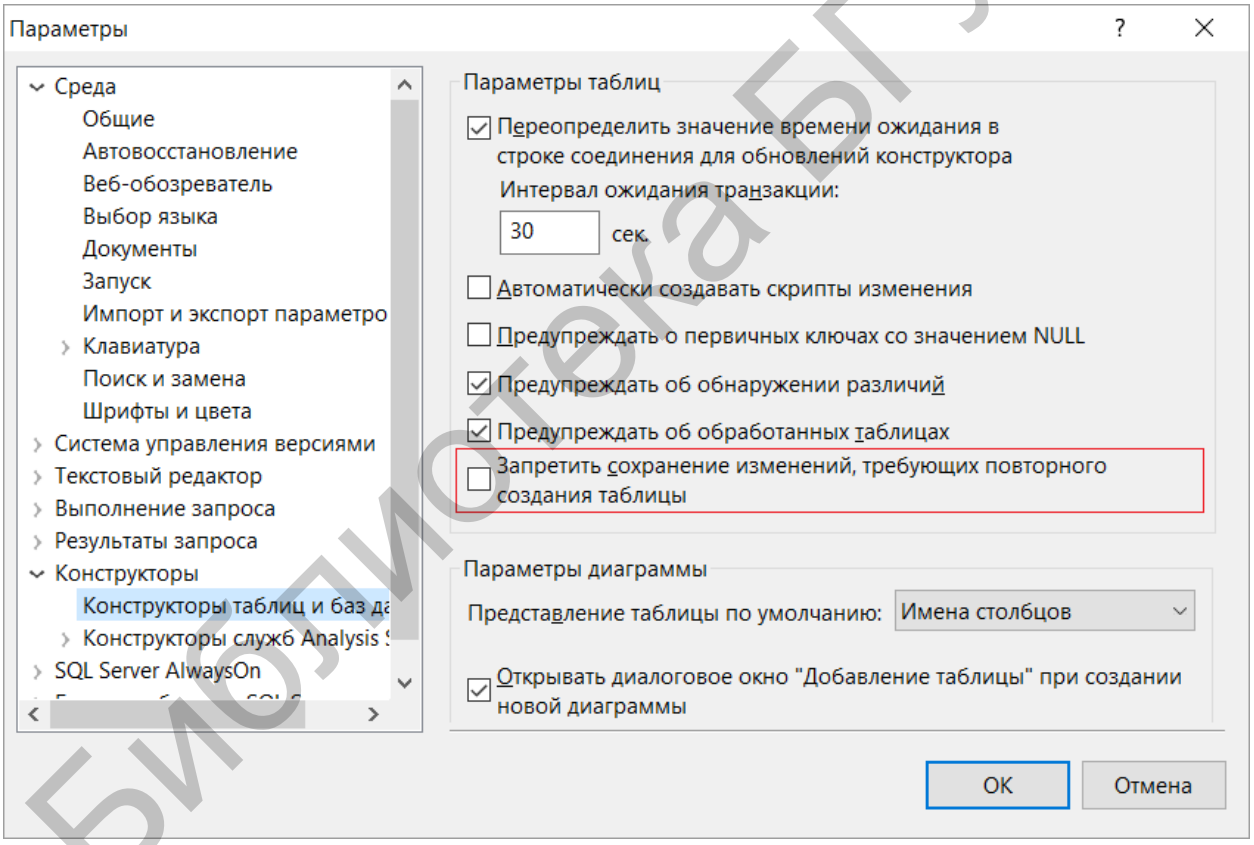


Рисунок 8.16 – Окно параметров среды

Подключение базы данных в проект происходит через добавление элемента под названием ADO.NET Entity Data Model. Для создания данного компонента необходимо вызвать контекстное меню проекта и выбрать пункт меню «Добавить» → «Новый элемент» («Add» → «New Item») (рисунок 8.17).

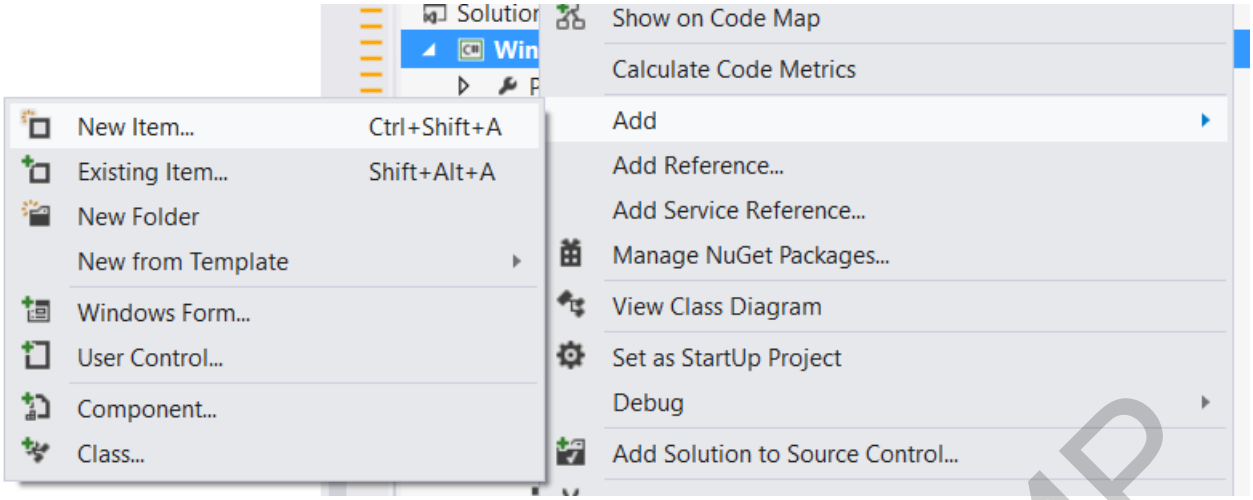


Рисунок 8.17 – Создание нового элемента в проекте

Затем необходимо выбрать в списке компонентов ADO.NET Entity Data Model и присвоить имя компоненту (рисунок 8.18).

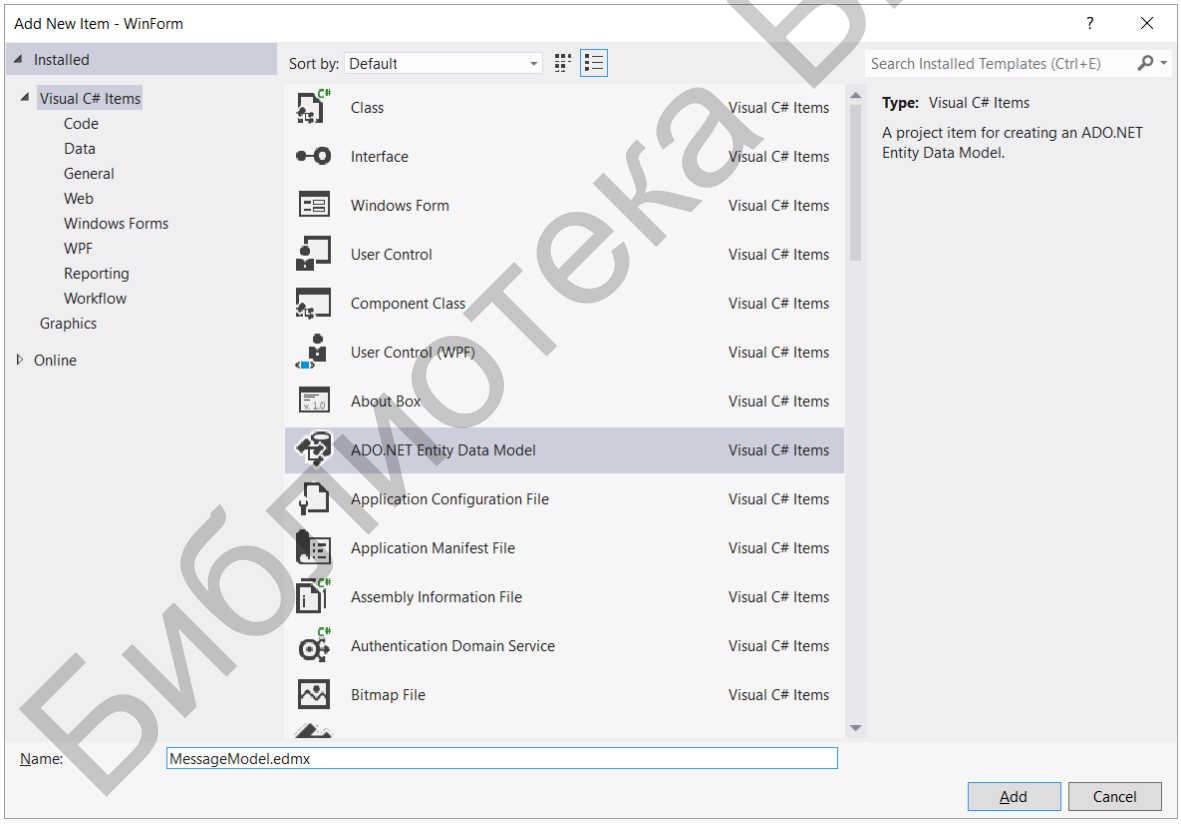


Рисунок 8.18 – Окно выбора нового компонента

На экране отобразится мастер создания подключения к базе данных, для подключения к существующей базе данных необходимо выбрать опцию «Generate from database» (рисунок 8.19).

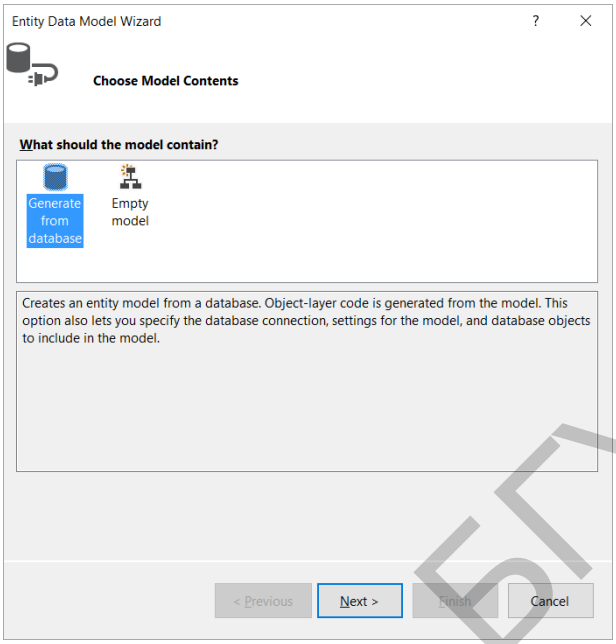


Рисунок 8.19 – Окно мастера создания подключения к базе данных, шаг выбора типа подключения

Следующий шаг – создание подключения к базе данных. На отобразившемся окне (рисунок 8.20) необходимо нажать кнопку «New Connection».

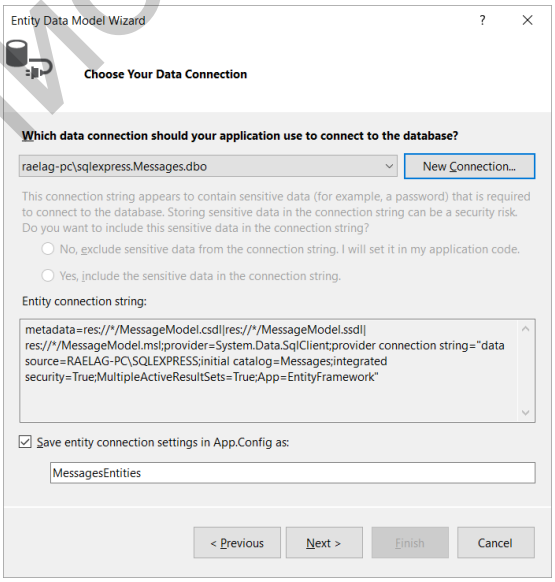


Рисунок 8.20 – Окно мастера создания нового подключения, шаг создания подключения

В результате откроется окно свойств нового подключения (рисунок 8.21). В окне свойств нового подключения следует указать имя сервера и имя базы данных, которая находится на сервере. Для корректности подключения необходимо нажать кнопку «Test connection».

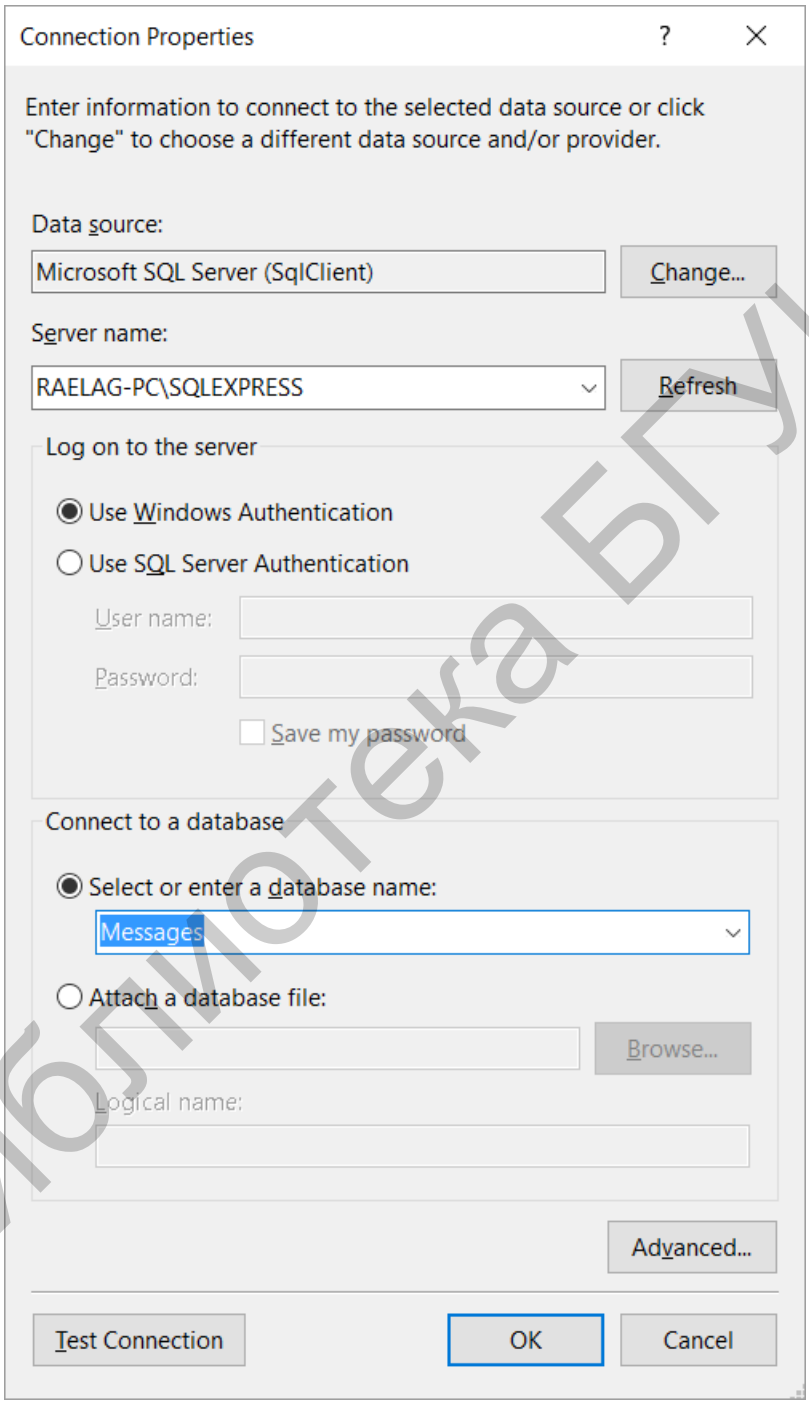


Рисунок 8.21 – Окно свойств подключения к базе данных

Если предыдущие действия были верны, то следующим шагом является выбор таблиц базы данных, с которыми будет происходить взаимодействие в проекте. Для этого в открывшемся окне достаточно поставить галочки на тех таблицах, с которыми будет вестись работа (рисунок 8.22).

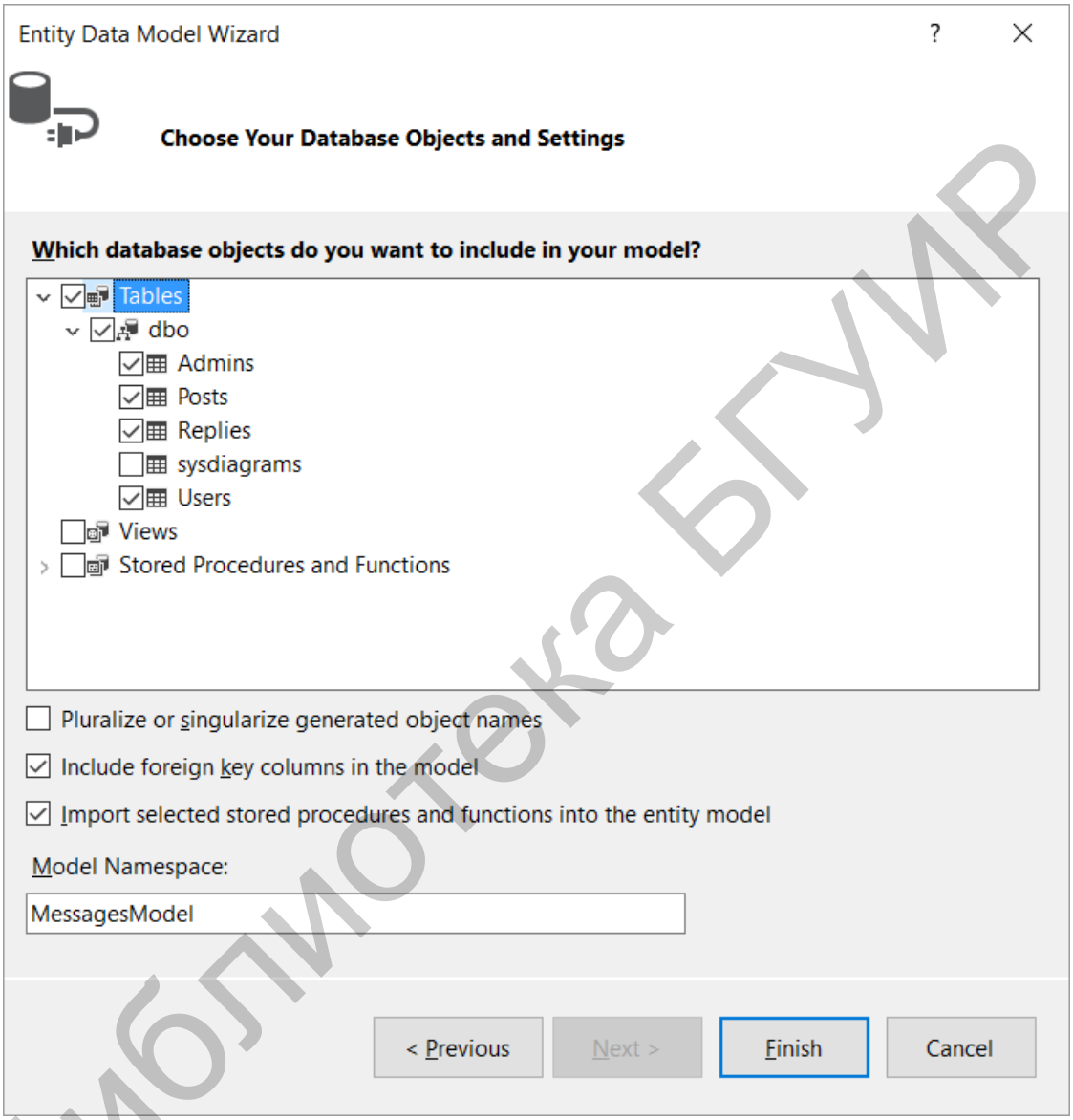


Рисунок 8.22 – Окно мастера создания нового подключения, шаг выбора рабочих таблиц

После завершения настройки подключения к базе данных таблицы отобразятся схематически в среде разработки в виде диаграммы с соответствующими таблицами со связями между ними; в таблицах находятся соответствующие поля и навигационные свойства (рисунок 8.23).

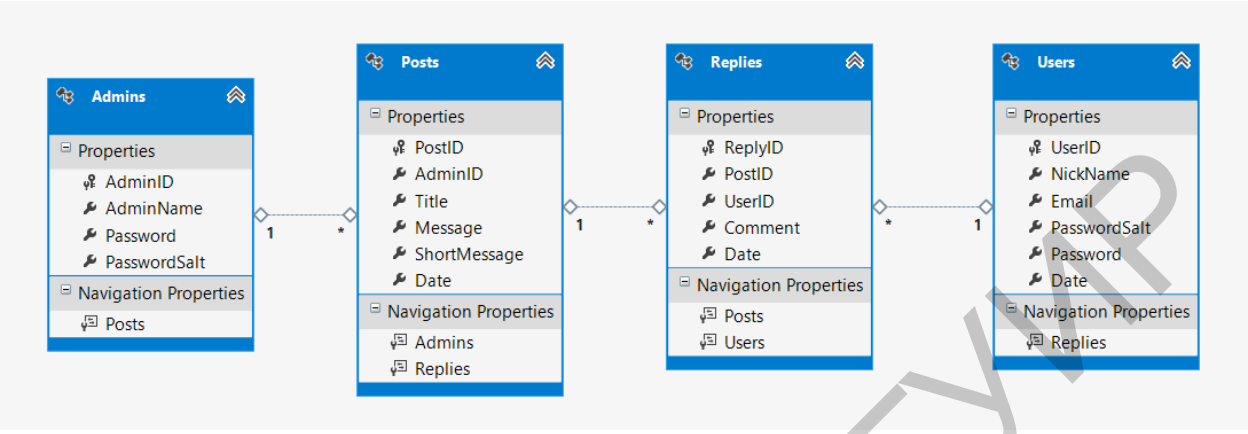


Рисунок 8.23 – Диаграмма таблиц подключенной базы данных

Для внесения различий названия таблиц в контексте базы данных с созданным подключением необходимо дать различные имена свойствам таблиц «Entity Set Name» и «Name» (рисунок 8.24).

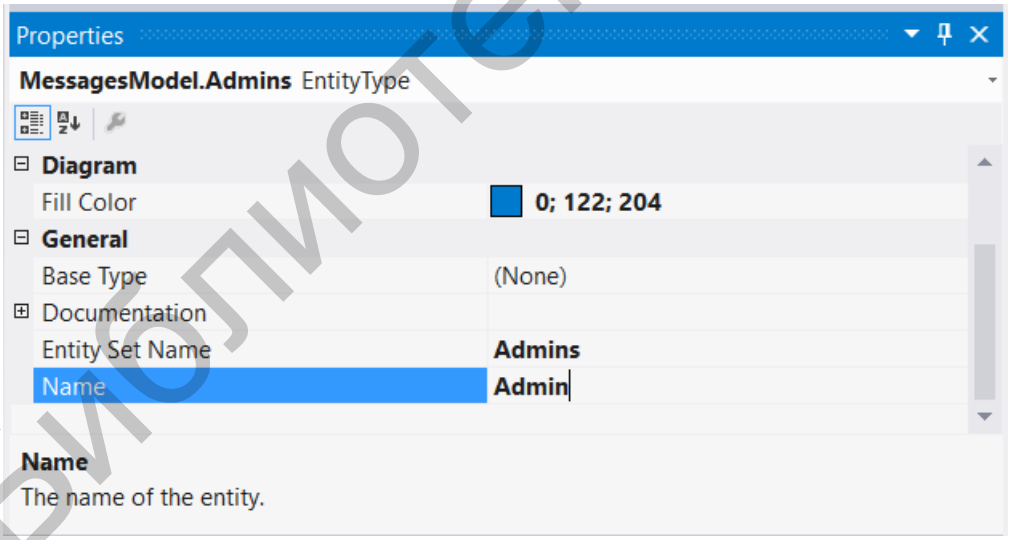


Рисунок 8.24 – Свойства таблицы

Данные из таблиц базы данных можно отображать различными способами, например, путем вывода структурированных данных с помощью .NET-компонента DataGridView. Для этого сначала необходимо поместить данную компонента на форме, в результате вид форм можно увидеть на рисунке 8.25.

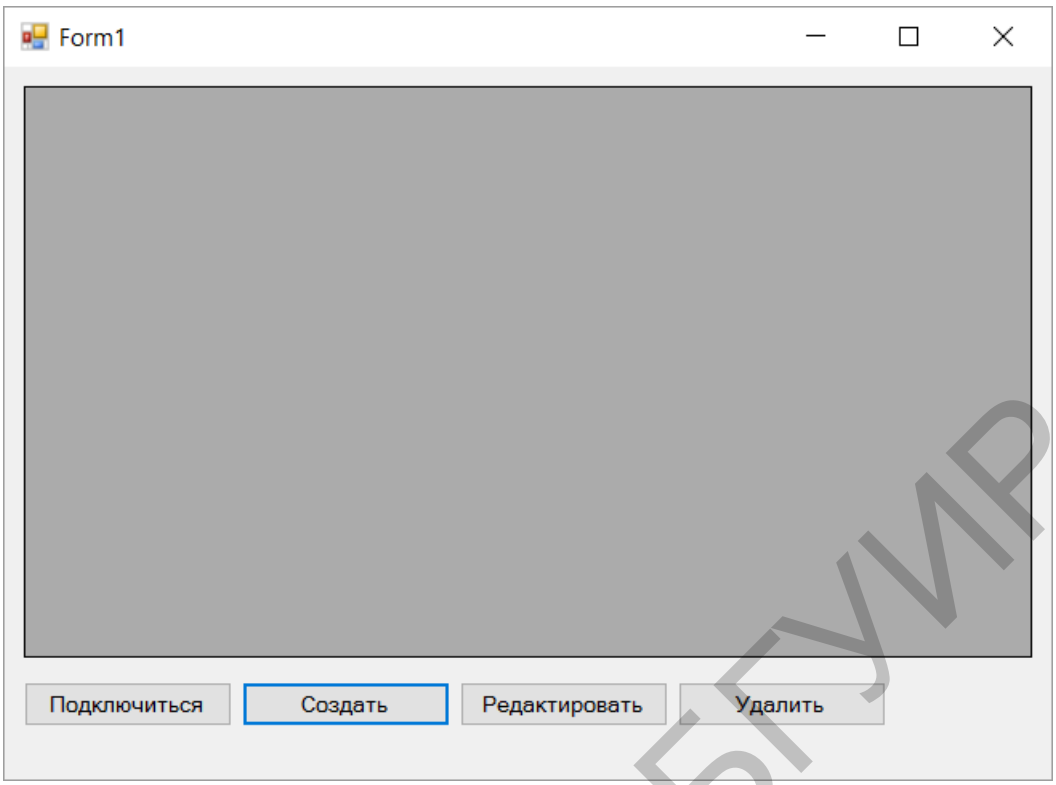


Рисунок 8.25 – Вид формы с компонентом DataGridView

Для отображения данных в компоненте DataGridView реализуем обработчик события нажатия кнопки «Подключиться», в котором будет создаваться контекст базы данных, и связывание источника данных DataGridView с данными из таблицы:

```
private void button1_Click(object sender, EventArgs e)
 {
    MessagesEntities me = new MessagesEntities();
    dataGridView1.DataSource = me.Posts.ToList();
 }
```

Результат выполнения данного кода можно увидеть на рисунке 8.26.

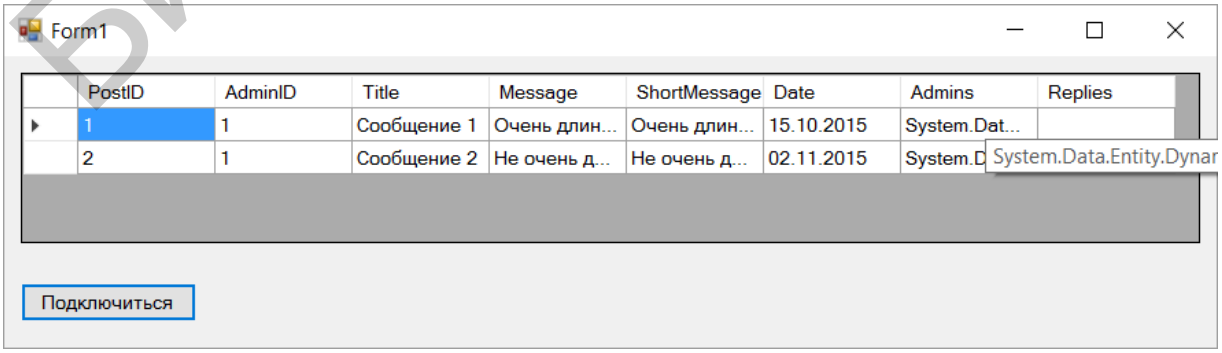


Рисунок 8.26 – Результат нажатия кнопки «Подключиться»

Для добавления новой записи реализуем обработчик события нажатия кнопки «Создать», открывающей форму добавления записи, через которую будет происходить добавление новых данных (рисунок 8.27).

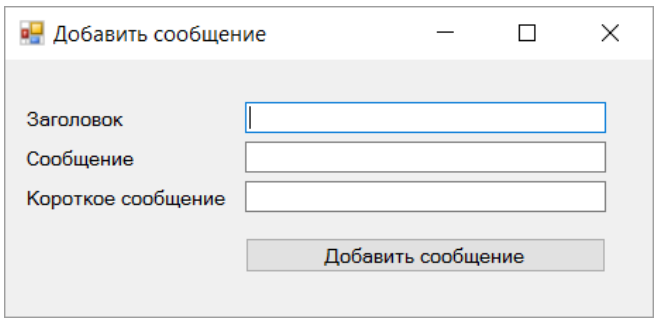


Рисунок 8.27 – Форма добавления новой записи

Заполнив все поля и нажав кнопку, реализуем следующий код:

```
MessagesEntities me = new MessagesEntities();
Post post = new Post();
post.AdminID = 1;
post.Date = DateTime.Now;
post.Title = textBox1.Text;
post.Message = textBox2.Text;
post.ShortMessage = textBox3.Text;
me.Posts.Add(post);
me.SaveChanges();
me.Dispose();
```

После выполнения добавляется новая запись в базу данных, результат добавления можно увидеть на рисунке 8.28.

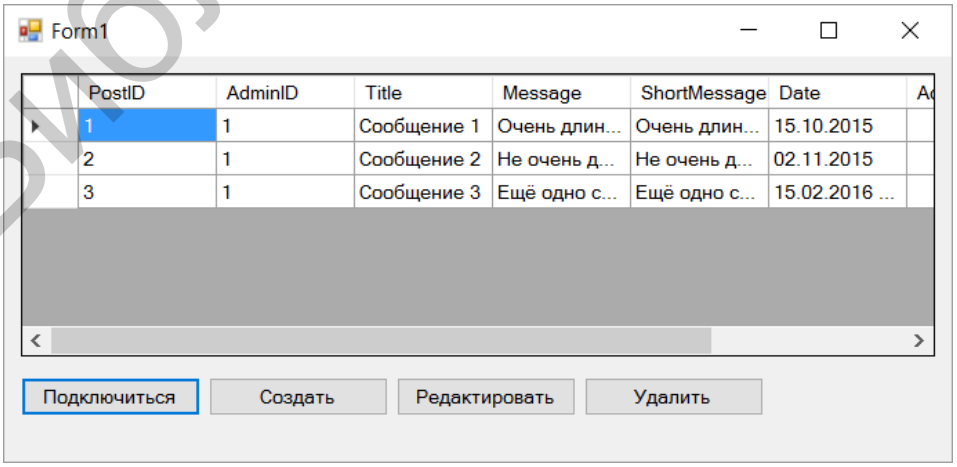


Рисунок 8.28 – Окно с результатом выполнения добавления

Для редактирования записи реализован код, который вначале определяем запись с заданным идентификационным кодом, затем заполняет форму для редактирования, само добавление происходит уже через созданную форму (рисунок 8.29).

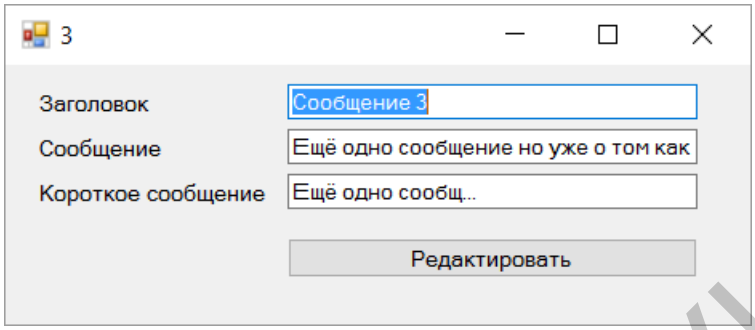


Рисунок 8.29 – Окно редактирования выбранной записи

Сам код изменения существующей записи будет выглядеть следующим образом:

```
MessagesEntities me = new MessagesEntities();
Post post = me.Posts.Find(Convert.ToInt32(Text));
post.Title = textBox1.Text;
post.Message = textBox2.Text;
post.ShortMessage = textBox3.Text;
me.SaveChanges();
me.Dispose();
```

На рисунке 8.30 можно увидеть результат выполнения операции изменения выбранной записи.

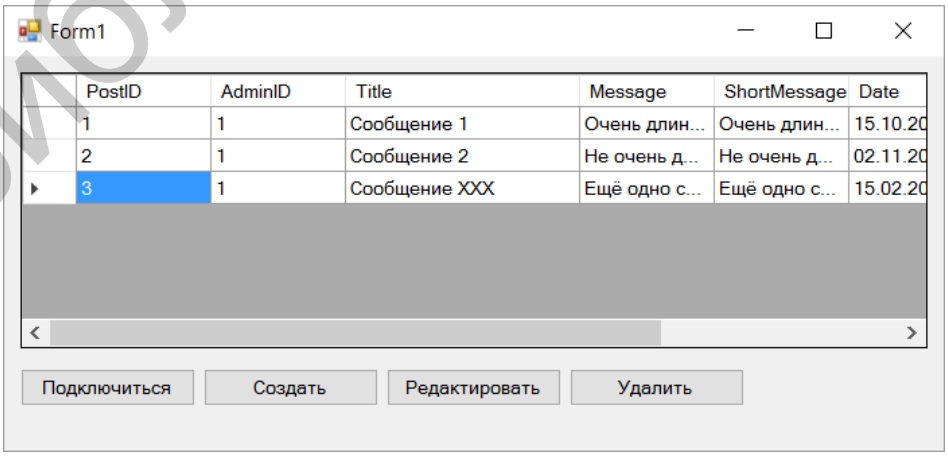


Рисунок 8.30 – Результат выполнения изменения существующей записи

Если необходимо удалить записи из таблицы, сначала определяем эту запись, затем удаляем из определенной таблицы, для чего создаем обработчик события нажатия кнопки «Удалить»:

```
MessagesEntities me = new MessagesEntities();
Post post = me.Posts.Find(Convert.ToInt32(textBox1.Text));
me.Posts.Remove(post);
me.SaveChanges();
me.Dispose();
```

Результат выполнения операции удаления можно увидеть на рисунке 8.31.

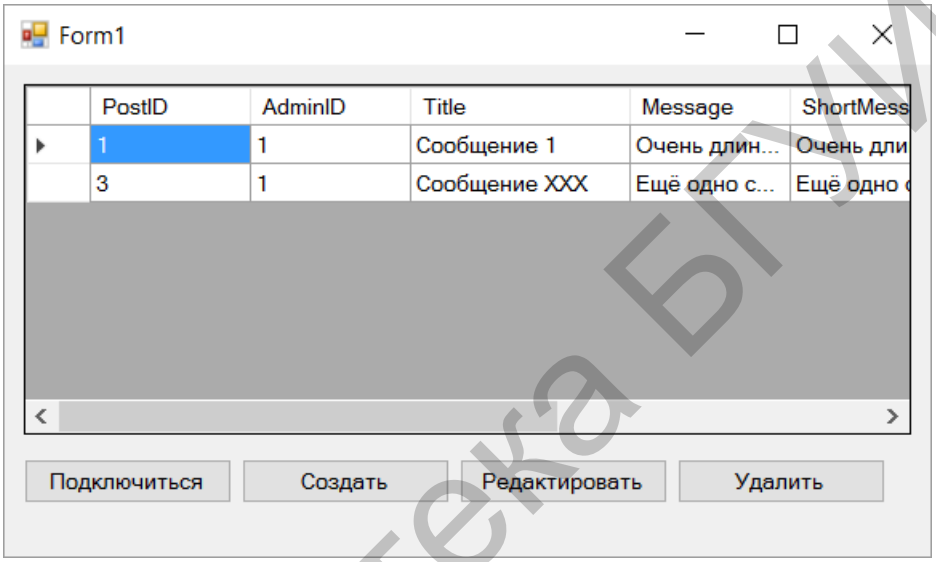


Рисунок 8.31 – Результат выполнения операции удаления

Индивидуальное задание по лабораторной работе

На основе результата выполнения лабораторной работы №7 выполнить следующие задания.

1. Используя SQL Server Management Studio, разработать структуру базы данных.
2. С помощью компонента ADO.NET Entity Data Model создать подключение к разработанной базе данных.
3. Реализовать добавление, удаление и изменение записей таблиц базы данных.

Литература

1. Шилдг, Г. Полный справочник по C# / Г. Шилдг ; пер. с англ. – М. : Издательский дом «Вильямс», 2004.

2. Троелсен, Э. C# и платформа .NET. Библиотека программиста / Э. Троелсен ; пер. с англ. – СПб. : Питер, 2004.

3. Рихтер, Дж. Программирование на платформе Microsoft .NET Framework 2.0 / Дж. Рихтер ; пер. с англ. – СПб. : Питер, 2007.

4. Бишоп, Дж. C# в кратком изложении / Дж. Бишоп, Н. Хорспул ; пер. с англ. – М. : Бином, 2005.

5. Microsoft Visual C# 2008. Базовый курс / К. Уотсон [и др.] ; пер. с англ. – Киев: Диалектика, 2009.

6. Фаронов, В. В. Программирование на языке C#. Учебный курс / В. В. Фаронов. – СПб. : Питер, 2007.

7. Павловская, Т. А. C#. Программирование на языке высокого уровня: учебник для вузов / Т. А. Павловская. – СПб. : Питер, 2009.

8. Макдональд, М. WPF в .NET 3.5 с примерами на C# 2008 для профессионалов / М. Макдональд ; пер. с англ. – М. : Издательский дом «Вильямс», 2004.

9. Петцольд, Ч. Программирование для Microsoft Windows на C#. В 2 т. / Ч. Петцольд ; пер. с англ. – М. : Русская редакция, 2002.

10. Лабор, В. В. Си Шарп. Создание приложений для Windows / В. В. Лабор. – Минск : Харвест, 2003.

11. Культин, Н. Microsoft Visual C# в задачах и примерах / Н. Культин. – СПб. : БХВ-Петербург, 2009.

12. Трей, Н. C# 2010. Ускоренный курс для профессионалов / Н. Трей ; пер. с англ. – М. : Издательский дом «Вильямс», 2010.

13. C# для профессионалов. В 2 т. / С. Робинсон [и др.] ; пер. с англ. – М. : Лори, 2003.

14. Либерти, Дж. Программирование на C# / Дж. Либерти ; пер. с англ. – СПб. Символ-плюс, 2002.

15. Марченко, А. Л. C#. Введение в программирование: учеб. пособие / А. Л. Марченко. – М. : МГУ, 2005.

16. Абрамян, М. Visual C# на примерах / М. Абрамян. – СПб. : БХВ-Петербург, 2008.

17. Петцольд, Ч. Программирование с использованием Microsoft Windows Forms / Ч. Петцольд ; пер. с англ. – СПб. : Питер, 2007.

Св. план 2016, поз. 21

Учебное издание

Раднёнок Антон Леонидович
Яшин Константин Дмитриевич

УПРАВЛЕНИЕ ИНФОРМАЦИОННЫМИ ПРОЕКТАМИ

ПОСОБИЕ

Редактор *А. К. Петрашкевич*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *Е. Д. Степусь*

Подписано в печать. 09.02.2017. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 4,42. Уч.-изд. л. 4,5. Тираж 50 экз. Заказ 77.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
ЛП №02330/264 от 14.04.2014.
220013, Минск, П. Бровки, 6