# Лабораторная работа № 2 ВВедение в windows forms

**Цель лабораторной работы**

Познакомиться с основами программирования в Windows Forms. Научиться размещать и настраивать внешний вид элементов управления на форме.

**Теоретический материал**

**Делегаты**

Делегатыпредставляют объекты, которые указывают на методы. Методы, на которые ссылаются делегаты, должны иметь те же параметры и тот же тип возвращаемого значения.

Для объявления делегата используется ключевое слово *delegate*, после которого идет возвращаемый тип, название и параметры:

delegate int Operation(int x, int y);

//здесь делегат ссылается на функцию, которая в //качестве параметров принимает два значения типа int и //возвращает некоторое число

delegate void GetMessage();

Для использования делегата надо создать его объект с помощью конструктора, в который мы передаем адрес метода, вызываемого делегатом. Чтобы вызвать метод, на который указывает делегат, надо использовать его метод *Invoke* (или *BeginInvoke/EndInvoke* при асинхронном режиме выполнения делегатов).

Пример с использованием делегата:

class Program

{

// 1. Объявляем делегат

    delegate int Operation(int x, int y);

    static void Main(string[] args)

    {

        // присваивание адреса метода через конструктор

// 2. Создаем переменную делегата

        Operation del = new Operation(Add);

// 3. делегат указывает на метод Add

// 4. Вызываем метод

        int result = del.Invoke(4,5);

        Console.WriteLine(result);

        del = Multiply;

// теперь делегат указывает на метод Multiply

        result = del.Invoke(4, 5);

        Console.WriteLine(result);

        Console.Read();

    }

    private static int Add(int x, int y)

    {

        return x+y;

    }

    private static int Multiply (int x, int y)

    {

        return x \* y;

    }

}

Здесь объявляется делегат Operation, возвращающий целочисленное значение и принимающий два параметра, также целочисленные. Методы (Add и Multiply), на которые указывает созданный экземпляр делегата, определяются с использованием конструктора.

Метод Invoke() при вызове делегата можно опустить и использовать сокращенную форму:

del = Multiply;

// теперь делегат указывает на метод Multiply

result = del(4, 5);

Использование делегатов обусловлено тем, что не всегда есть доступ к коду классов. Например, часть классов может создаваться и компилироваться одним человеком, который не будет знать, как эти классы будут использоваться. А использовать эти классы будет другой разработчик.

**Анонимные методы**

Анонимные методы используются для создания экземпляров делегатов. Определение анонимных методов начинается с ключевого слова *delegate*, после которого идет в скобках список параметров и тело метода в фигурных скобках:

delegate(параметры)

{

    // инструкции

}

Пример:

class Program

{

    delegate void MessageHandler(string message);

    static void Main(string[] args)

    {

        MessageHandler handler = delegate(string mes)

        {

            Console.WriteLine(mes);

        };

        handler("hello world!");

        Console.Read();

    }

}

Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата, как в данном случае переменная handler представляет анонимный метод. И через эту переменную делегата можно вызвать данный анонимный метод.

В отличие от блока методов или условных и циклических конструкций, блок анонимных методов должен заканчиваться точкой с запятой после закрывающей фигурной скобки.

Если анонимный метод использует параметры, то они должны соответствовать параметрам делегата. Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются. При этом даже если делегат принимает несколько параметров, то в анонимном методе можно вовсе опустить параметры.

Анонимные методы используются, когда надо определить однократное действие, которое не имеет много инструкций и нигде больше не используется.

Также, как и обычные методы, анонимные могут возвращать результат:

delegate int Operation(int x, int y);

static void Main(string[] args)

{

    Operation operation = delegate (int x, int y)

    {

        return x + y;

    };

    int d = operation(4, 5);

    Console.WriteLine(d);       // 9

    Console.Read();

}

**Лямбда-выражения**

Лямбда-выражения представляют упрощенную запись анонимных методов. Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы.

Синтаксис лямбда-выражений:

слева от лямбда-оператора => определяется список параметров, а справа блок выражений, использующий эти параметры:

(список\_параметров) => выражение.

Например:

class Program

{

    delegate int Operation(int x, int y);

    static void Main(string[] args)

    {

        Operation operation = (x, y) => x + y;

        Console.WriteLine(operation(10, 20));  // 30

        Console.WriteLine(operation(40, 20));  // 60

        Console.Read();

    }

}

Здесь x, y => x + y; – лямбда-выражение, x и y – параметры, x + y – выражение.

При этом не надо указывать тип параметров, а при возвращении результата не надо использовать оператор return.

Возвращаемое значение лямбда-выражений должно быть тем же, что и у делегата. Если лямбда-выражение принимает один параметр, то скобки вокруг параметра можно опустить, если параметров нет – ставятся пустые скобки.

Также бывает, что лямбда-выражение не возвращает никакого значения, и вызывает другие методы:

class Program

{

    delegate void Hello(); // делегат без параметров

    static void Main(string[] args)

    {

        Hello message = () => Show\_Message();

        message();

    }

    private static void Show\_Message()

    {

        Console.WriteLine("Вызов сообщения через

делегат");

    }

}

Одним из частых примеров использования лямбда-выражений является обработка событий. Как и делегаты, лямбда-выражения можно передавать в качестве аргументов методу для тех параметров, которые представляют делегат, что довольно удобно.

**События**

События сигнализируют системе о том, что произошло определенное действие. События объявляются в классе с помощью ключевого слова event, после которого идет название делегата.

// Объявляем делегат

public delegate void AccountStateHandler(

string message);

// Событие, возникающее при выводе денег

public event AccountStateHandler Withdrawn;

Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры, что и делегат, и возвращать тот же тип, что и делегат. Рассмотрим пример:

class Account

{

    // Объявляем делегат

    public delegate void AccountStateHandler(string message);

    // Событие, возникающее при выводе денег

    public event AccountStateHandler Withdrawn;

    // Событие, возникающее при добавление на счет

    public event AccountStateHandler Added;

int \_sum; // Переменная для хранения суммы

public Account(int sum)

    {

        \_sum = sum;

    }

    public int CurrentSum

    {

        get { return \_sum; }

    }

    public void Put(int sum)

    {

        \_sum += sum;

        if (Added != null)

            Added($"На счет поступило {sum}");

    }

    public void Withdraw(int sum)

    {

        if (sum <= \_sum)

        {

            \_sum -= sum;

            if (Withdrawn != null)  Withdrawn($"Сумма {sum} снята со счета");

        }

        else

        {

            if (Withdrawn != null)  Withdrawn("Недостаточно денег на счете");

        }

    }

}

События Withdrawn и Added объявлены как экземпляры делегата AccountStateHandler, поэтому для их обработки потребуется метод, принимающий строку в качестве параметра. Затем в методах Put и Withdraw эти события вызываются. Перед вызовом мы проверяем, закреплены ли за этими событиями обработчики. Так как эти события представляют делегат AccountStateHandler, принимающий в качестве параметра строку, то и при вызове событий мы передаем в них строку.

Для прикрепления обработчика события к определенному событию используется операция += и, соответственно, для открепления – операция -=:

событие += метод\_обработчика\_события.

Метод обработчика должен иметь такие же параметры, как и делегат события, и возвращать тот же тип:

class Program

{

    static void Main(string[] args)

    {

        Account account = new Account(200);

        // Добавляем обработчики события

        account.Added += Show\_Message;

        account.Withdrawn += Show\_Message;

        account.Withdraw(100);

        // Удаляем обработчик события

        account.Withdrawn -= Show\_Message;

        account.Withdraw(50);

        account.Put(150);

        Console.ReadLine();

    }

    private static void Show\_Message(string message)

    {

        Console.WriteLine(message);

    }

}

Оба способа прикрепления обработчиков равноценны:

account.Added += Show\_Message;

account.Added += new

Account.AccountStateHandler(Show\_Message);

**Приложения Windows Forms**

Чтобы создать приложение с графическим интерфейсом необходимо:

Шаг 1: Создание нового проекта

- Откройте Visual Studio и выберите "Create a new project".

- Выберите тип проекта "Windows Forms App (Майкрософт)".

- Укажите имя проекта и расположение файлов.

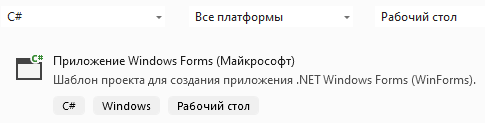


Рисунок 1 – Создание проекта

Шаг 2: Дизайн формы

- После создания проекта, вы увидите окно дизайнера формы.

- Добавьте на форму элементы управления (кнопки, текстовые поля, метки) из панели инструментов.

- Изменяйте свойства элементов управления на свой вкус.

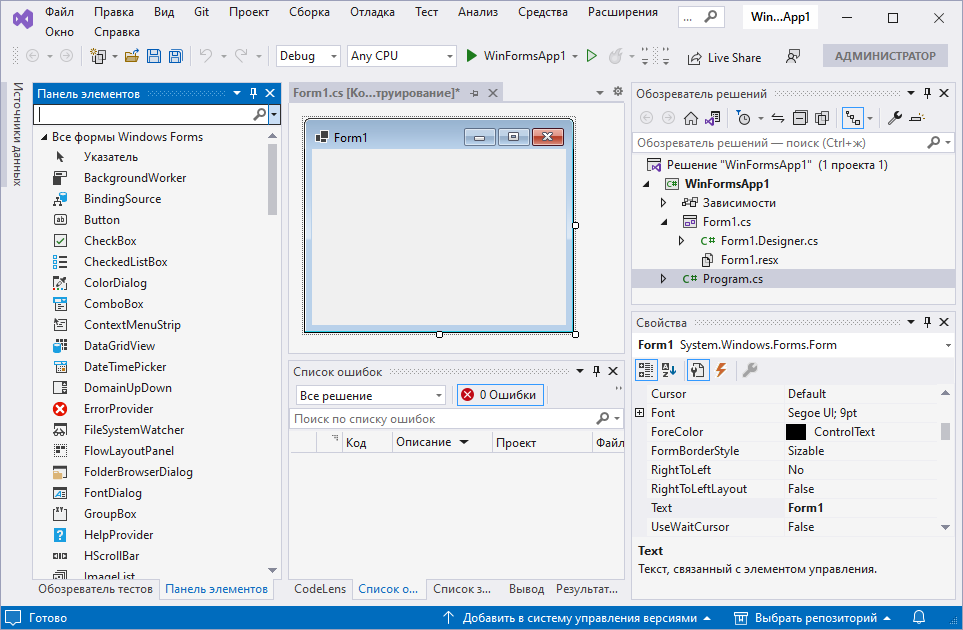


Рисунок 2 – Окно дизайнера проекта

Чтобы добавить элементы на форму необходимо их перетащить из панели элементов (слева).

При нажатии на «Свойства» в правом нижнем углу появится окно с параметрами данного объекта, также параметры отобразятся если просто нажать ЛКМ на объект (например, на форму).

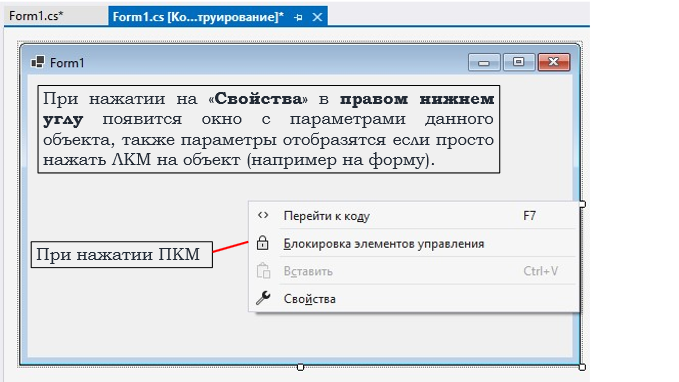


Рисунок 3 – Окно формы

Окно «Свойства» содержит параметры компонента и выглядит следующим образом:

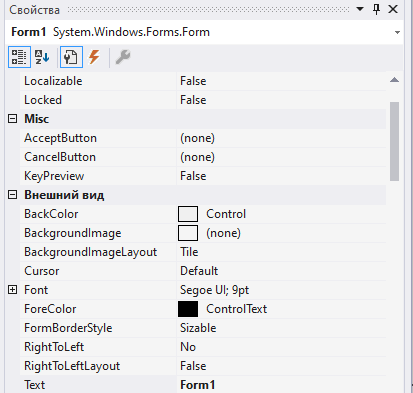


Рисунок 4 – Окно «Свойства»

Для каждого компонента есть общие и уникальные свойства, как правило, можно менять имя компонента, размеры, положение, привязки (Anchor) и др.

Рассмотрим основные компоненты такие как: кнопка, лейбл, поле для ввода текста и др.

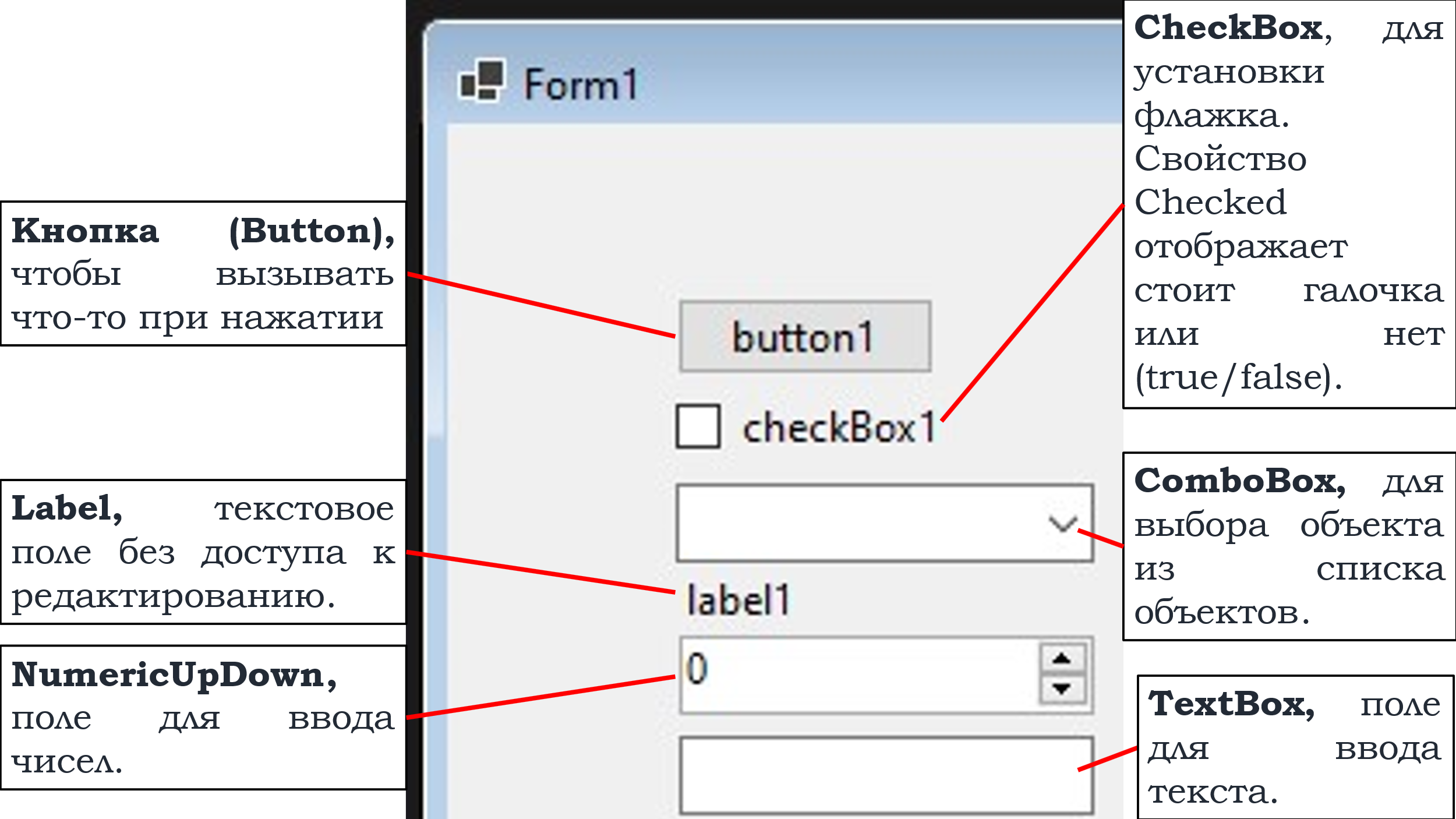


Рисунок 5 – Основные компоненты

Также существуют такие компоненты, как: многострочное текстовое поле (необходимо включить Multiline у обычного текстового поля), таблица, переключатель radioButton, прокручиваемый элемент TrackBar.

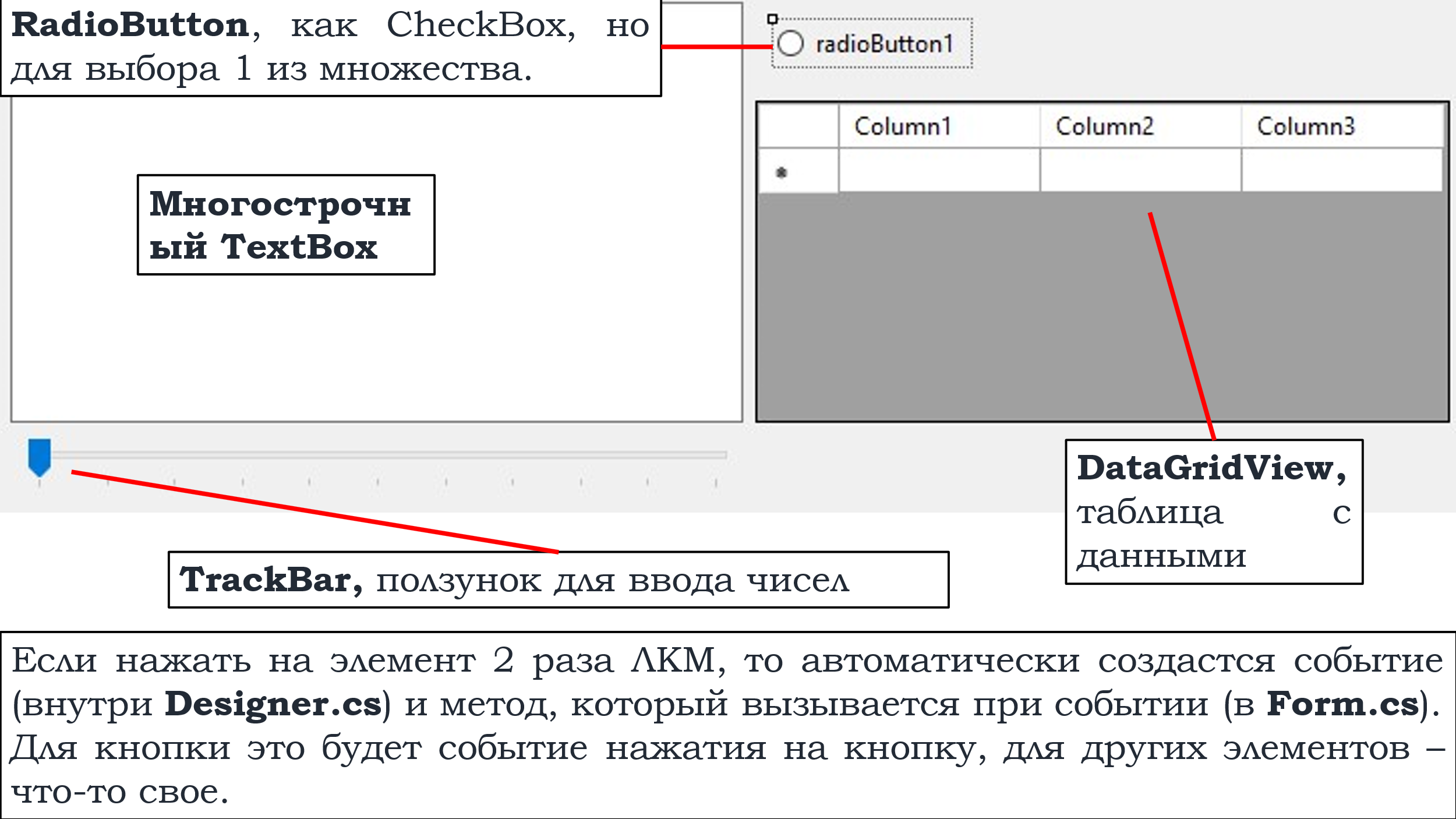


Рисунок 6 – Компоненты

Для написания кода необходимо перейти в файл с кодом. Для этого необходимо нажать «Перейти к коду» в дизайнере (рисунок 2) или на вкладке «Обозреватель решения».

Чтобы добавить в код обработчик события (например, нажатия на кнопку) необходимо перейти в окно «Свойства» соответствующего компонента и открыть вкладку «События» (обозначена знаком молнии visual-studio-events-button). Далее нужно найти событие, к которому нужно добавить обработчик, например событие Click и нажать 2 раза на пустое поле справа (рисунок 7).

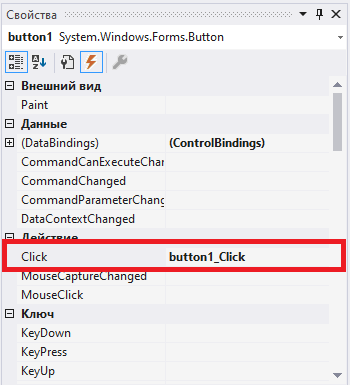


Рисунок 7 – Добавление обработчика события  
нажатия на кнопку Button1

После добавления события нас перенесет в окно с кодом, где добавится код обработчика:

private void button1\_Click(object sender, EventArgs e)

{

}

Чтобы удалить обработчик события нужно очистить поле справа (рисунок 7) и удалить код обработчика. Если удалить сначала код обработчика, то в дизайнере появится ошибка (рисунок 8).

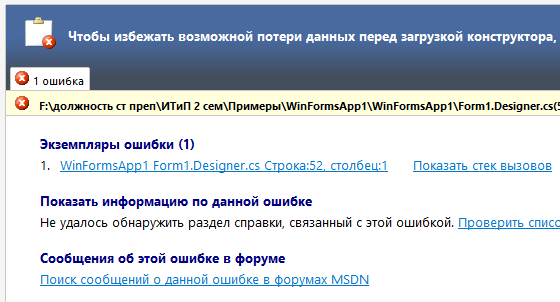


Рисунок 8 – Ошибка при неполном удалении обработчика

Для исправления ошибки необходимо перейти в код дизайнера формы, для этого необходимо нажать на текст ошибки:



или в обозревателе решений открыть файл с кодом дизайнера Form1.Designer.cs (рисунок 9).

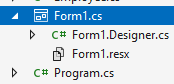


Рисунок 9 – Структура формы в обозревателе решений

Далее необходимо найти и удалить строку с добавлением обработчика события (будет подчеркнуто красным), код которого был удален:

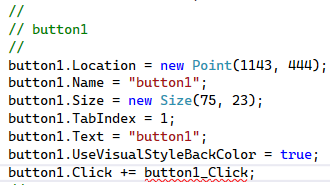


Рисунок 10 – Код компонента «button1»

После размещения компонентов на форме и добавления обработчиков необходимо перейти в файл с кодом для работы с логикой формы (рисунок 11).

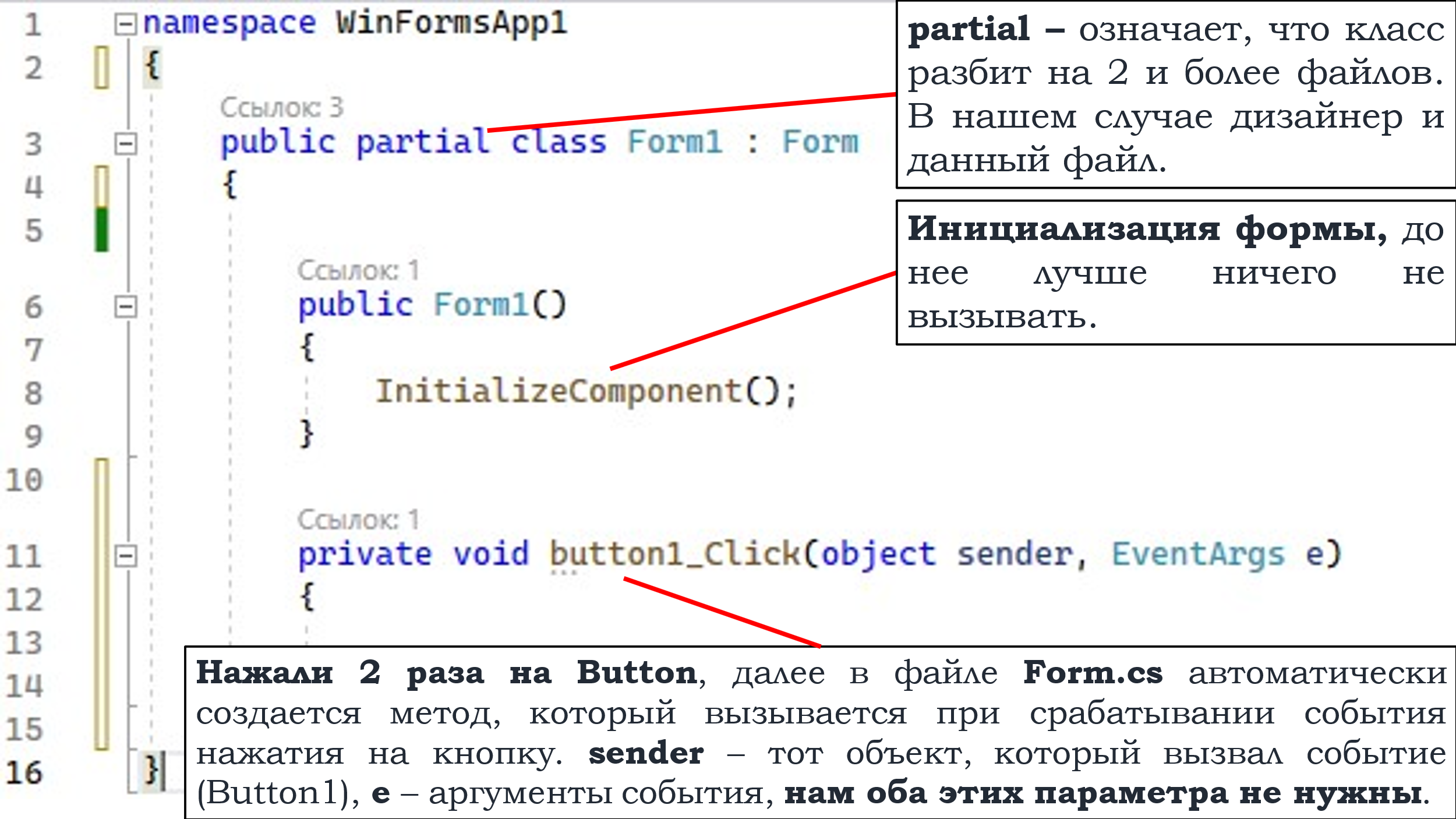


Рисунок 11 – Структура файла с формой Form1

В файл с формой не принято писать сложную логику программы. Необходимо разделять программу на **интерфейсную часть** (для взаимодействия с пользователем, например, форма или консоль), на **часть** **управления данными** и на **модельную часть** (описание данных и выполнение вычисления).

Каждый такой блок должен оформляться в виде отдельного файла (или библиотеки) с классом (классами).

Добавим в обработчик события код для вызова окна с сообзением, в котором отобразим содержимое текстового файла. Файл выбирается с помощью диалогового окна OpenFileDialog (рисунок 12).

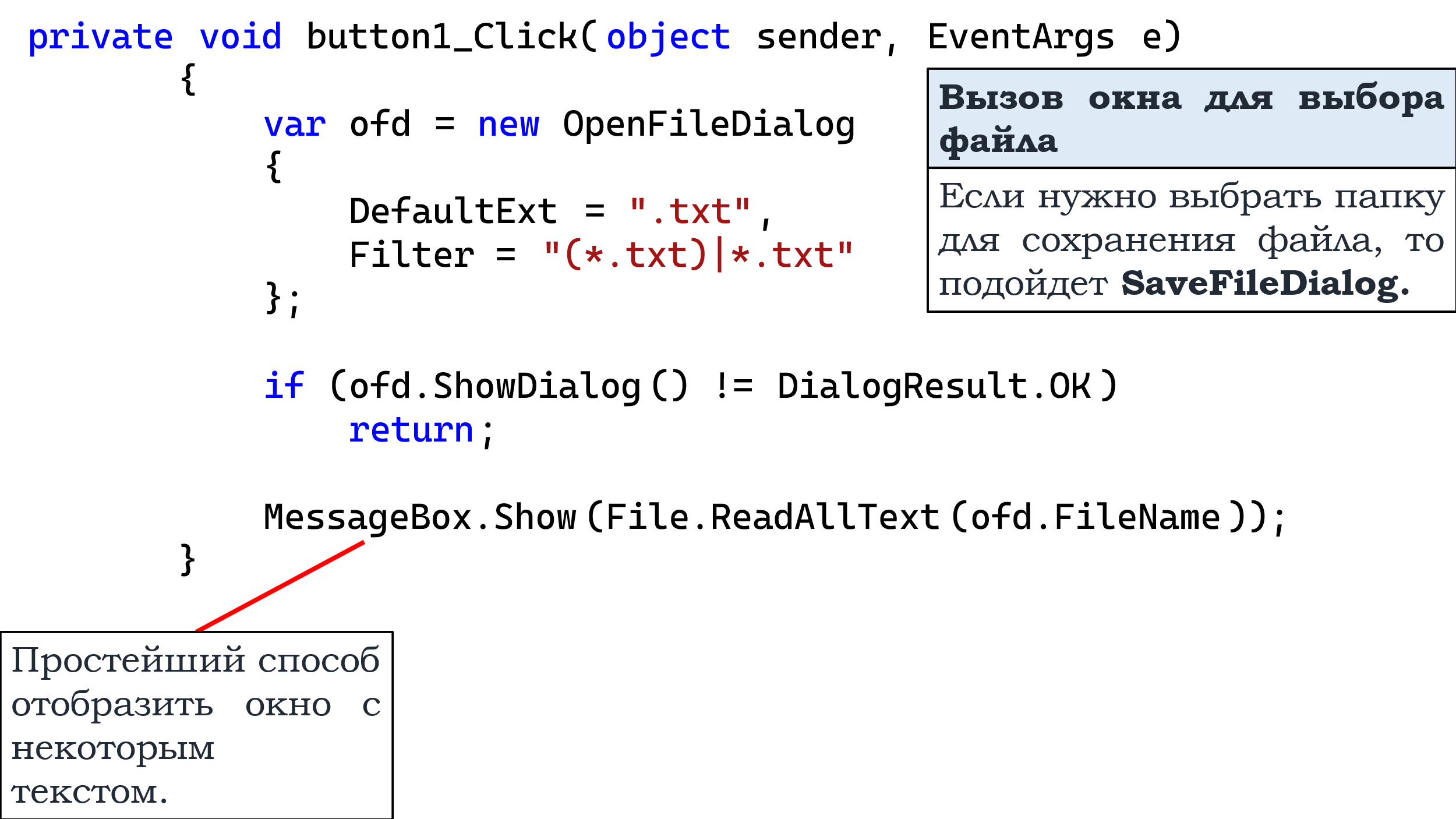


Рисунок 12 – Отображение содержимого файла в отдельном окне

Рассмотрим некоторые важные свойства:

**Name** – имя элемента,

**Text** – надпись, что отображается на элементе,

**Font** – шрифт,

**Anchor** – привязка граней элемента,

**Dock** – похож на Anchor, позволяет, например, заполнить элементом низ/верх/левую/правую часть родительского блока или весь блок,

**TextAlign** – выравнивание текста (по центру, слева справа и т.п.),

**FormBorderStyle** – стиль формы (с изменением размера, фиксированная и др).

**Некоторые события:**

**Click** – нажатие на элемент (не важно каким образом),

**KeyDown** – нажатие на кнопку клавиатуры, когда элемент в фокусе,

**MouseHover** – наведение курсора мыши на элемент,

**MouseMove** – перемещение мыши, когда элемент в фокусе,

**Resize** – изменение размеров элемента.

Рассмотрим ещё некоторые компоненты формы (рисунок 13).

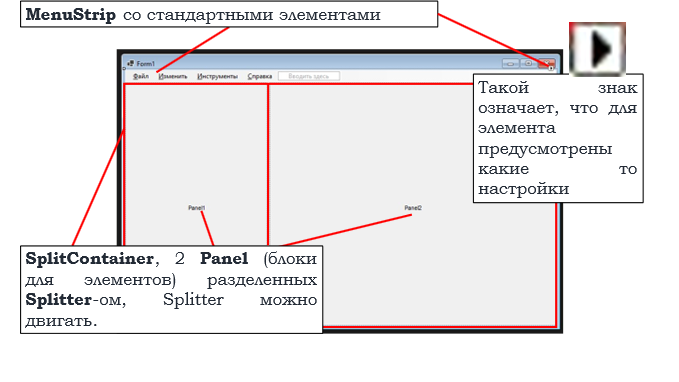


Рисунок 13 – Компоненты формы

Дополнительные возможности по позиционировании элемента позволяет определить свойство **Anchor** (рисунок 14). Это свойство определяет расстояние между одной из сторон элемента и стороной контейнера. И если при работе с контейнером мы будем его растягивать, то вместе с ним будет растягиваться и вложенный элемент.

По умолчанию у каждого добавляемого элемента это свойство равно Top, Left:

Это значит, что если мы будем растягивать форму влево или вверх, то элемент сохранит расстояние от левой и верхней границы элемента до границ контейнера, в качестве которого выступает форма.

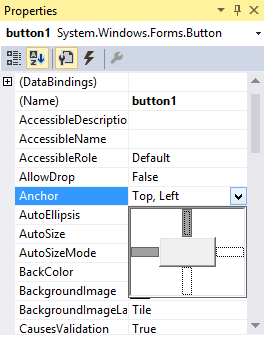


Рисунок 14 – Свойство Anchor

Например, если мы хотим, чтобы кнопка сохраняла расстояние до правой и нижней границы, когда мы растягиваем форму, то необходимо выставить Right и Bottom (рисунок 15).

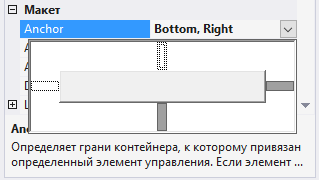


Рисунок 15 – Изменение привязок в свойстве Anchor

При изменении размеров окна кнопка будет следовать за правой и нижней сторонами (рисунок 16).

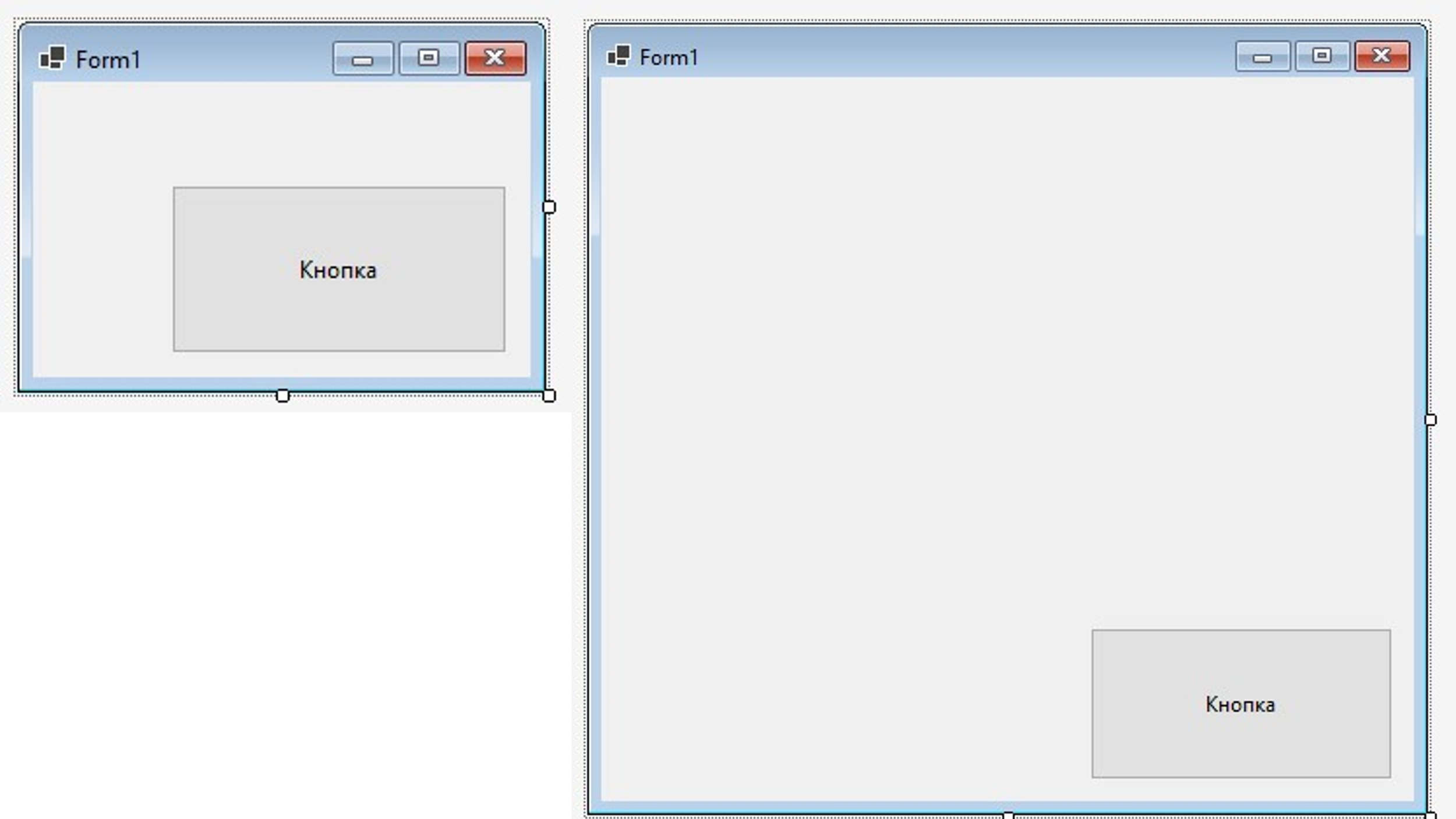


Рисунок 16 – Позиционирование кнопки  
после привязки к правой и нижней сторонам

Если выставить сразу Left, Top, Right, Bottom, то кнопка станет растягиваться во всех направлениях вместе с формой (рисунок 17):

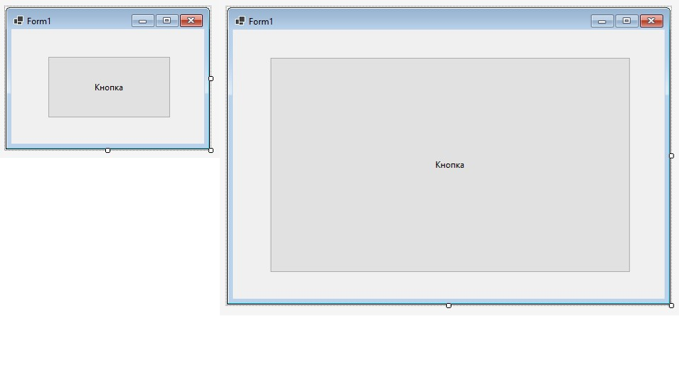


Рисунок 17 – Позиционирование кнопки  
после привязки ко всем сторонам

Свойство **Dock** (рисунок 18) позволяет прикрепить элемент к определенной стороне контейнера. По умолчанию оно имеет значение **None**, но также позволяет задать еще пять значений:

**Top**: элемент прижимается к верхней границе контейнера

**Bottom**: элемент прижимается к нижней границе контейнера

**Left**: элемент прижимается к левой стороне контейнера

**Right**: элемент прикрепляется к правой стороне контейнера

**Fill**: элемент заполняет все пространство контейнера

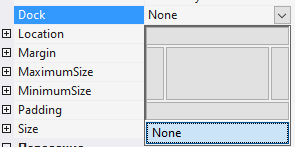


Рисунок 18 – Свойство Dock

Поставим значение Fill (центральный блок, рисунок 19):

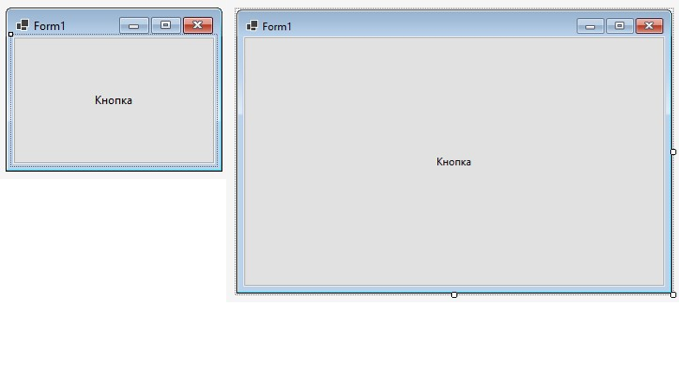


Рисунок 19 – Позиционирование кнопки  
после установки Dock = Fill

В Windows Form содержится огромное количество разнообразных компонентов, а также есть возможность создания своих компонентов. Доскональное изучение формы не укладывается в рамках данной работы и требует **самостоятельного изучения**.

Шаг 3: Написание кода

- Дважды кликните на элемент управления (например, кнопку) для перехода к обработчику события (например, нажатие кнопки).

- Напишите код в обработчиках событий для обеспечения функциональности вашего приложения.

Шаг 4: Тестирование

- Запустите приложение, нажав F5 или кнопку "Start" в Visual Studio.

- Протестируйте функциональность вашего приложения.

Пример.

Создадим WindowsForms (Майкрософт) приложение. Код для главного окна имеет следующий вид. Чтобы данный код работал необходимо добавить соответствующие элементы на форму, а также подключить обработчики событий нажатия на кнопки.

public partial class MainForm : Form

{

// Создание экземпляра диалогового окна для ввода параметров фигуры

private FigureForm \_figureForm = new FigureForm();

// Переменная для работы с графикой на элементе управления pictureBox

private Graphics \_graphics;

// Конструктор класса MainForm

public MainForm()

{

InitializeComponent();

// Инициализация объекта Graphics для рисования на pictureBox

\_graphics = pictureBox.CreateGraphics();

}

// Обработчик события нажатия кнопки "Выход"

private void buttonExit\_Click(object sender, EventArgs e)

{

// Закрытие приложения

Application.Exit();

}

// Обработчик события нажатия кнопки "Ввод"

private void buttonInput\_Click(object sender, EventArgs e)

{

// Установка максимальных координат и размеров фигуры

\_figureForm.MaxFigureCoordinate = new Point(pictureBox.Size.Width - 1, pictureBox.Size.Height - 1);

\_figureForm.MaxFigureSize = new Size(pictureBox.Size.Width, pictureBox.Size.Height);

// Отображение диалогового окна для ввода параметров фигуры

\_figureForm.ShowDialog();

// Создание кисти с выбранным цветом фигуры

var brush = new SolidBrush(\_figureForm.FigureColor);

// Рисование прямоугольника или эллипса на pictureBox в зависимости от выбранной формы

switch (\_figureForm.GeometryForm)

{

case FigureGeometryForm.Rectangle:

\_graphics.FillRectangle(brush, new Rectan-gle(\_figureForm.FigurePoint, \_figureForm.FigureSize));

break;

case FigureGeometryForm.Ellipse:

\_graphics.FillEllipse(brush, new Rectan-gle(\_figureForm.FigurePoint, \_figureForm.FigureSize));

break;

}

}

// Обработчик события нажатия кнопки "Очистить"

private void buttonClear\_Click(object sender, EventArgs e)

{

// Очистка pictureBox, установка цвета на белый

\_graphics.Clear(Color.White);

}

}

1. **Класс MainForm**:
   * Этот класс представляет главную форму приложения, где пользователь может взаимодействовать с элементами управления.
2. **Переменные**:
   * \_figureForm: Экземпляр диалогового окна FigureForm, которое используется для ввода параметров фигуры.
   * \_graphics: Переменная для работы с графикой на элементе управления pictureBox.
3. **Конструктор MainForm()**:
   * В конструкторе происходит инициализация формы через вызов InitializeComponent() и создание объекта \_graphics для рисования на pictureBox.
4. **Метод buttonExit\_Click**:
   * Этот метод обрабатывает событие нажатия кнопки "Выход" и закрывает приложение.
5. **Метод buttonInput\_Click**:
   * Этот метод обрабатывает событие нажатия кнопки "Ввод".
   * Устанавливает максимальные координаты и размеры фигуры на основе размеров pictureBox.
   * Отображает диалоговое окно FigureForm, где пользователь выбирает параметры фигуры и ее цвет.
   * В зависимости от выбранной формы (прямоугольник или эллипс) рисуется фигура выбранного цвета на pictureBox.
6. **Метод buttonClear\_Click**:
   * Этот метод обрабатывает событие нажатия кнопки "Очистить".
   * Очищает pictureBox, устанавливая его цвет на белый, чтобы удалить все нарисованные фигуры.

Этот код позволяет пользователю создавать прямоугольники или эллипсы на форме с помощью дополнительного диалогового окна, выбирать цвет фигуры и управлять отображением и очисткой рисунков на элементе pictureBox. (рисунок 20)

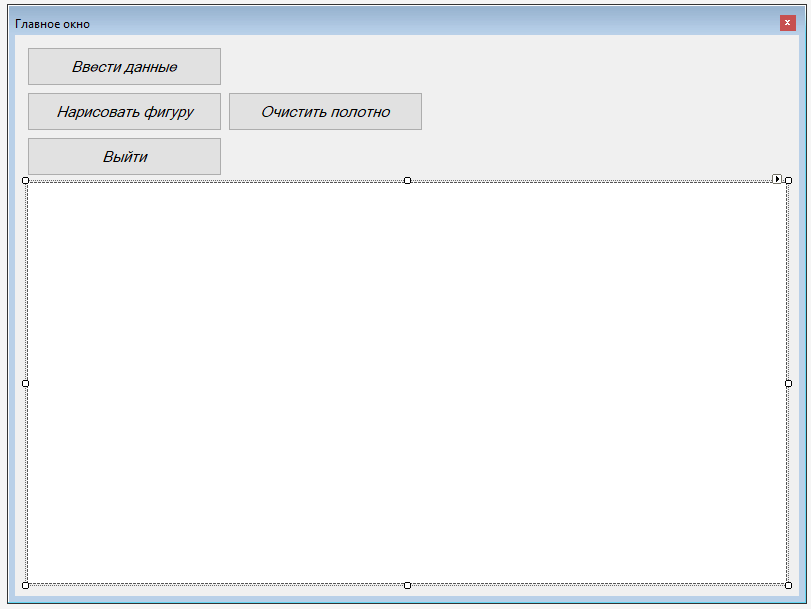


Рисунок 20 – Главное окно формы

Далее рассмотрим код окна, которое появляется при нажатии на кнопку «Ввести данные» (рисунок 21).

using System;

using System.Drawing;

using System.Windows.Forms;

namespace WinFormsAppExample

{

// Перечисление для определения формы фигуры

public enum FigureGeometryForm { Rectangle, Ellipse };

// Класс для создания диалогового окна с параметрами фигуры

public partial class FigureForm : Form

{

// Цвет фигуры

private Color figureColor;

// Свойство для доступа к цвету фигуры с обновлением цвета на панели

public Color FigureColor

{

get => figureColor;

set

{

// Обновляем цвет на панели

panelColor.BackColor = value;

figureColor = value;

}

}

// Свойства для формы, размеров и координат фигуры

public FigureGeometryForm GeometryForm { get; private set; }

public Size FigureSize { get; private set; }

public Point FigurePoint { get; private set; }

// Максимальные размеры фигуры

private Size maxFigureSize;

public Size MaxFigureSize

{

get => maxFigureSize;

set

{

// Устанавливаем максимальные значения для размеров фигуры

numericUpDownWidth.Maximum = value.Width;

numericUpDownHeight.Maximum = value.Height;

maxFigureSize = value;

}

}

// Максимальные координаты фигуры

private Point maxFigureCoordinate;

public Point MaxFigureCoordinate

{

get => maxFigureCoordinate;

set

{

// Устанавливаем максимальные значения для координат фигуры

numericUpDownX.Maximum = value.X;

numericUpDownY.Maximum = value.Y;

maxFigureCoordinate = value;

}

}

// Конструктор класса

public FigureForm()

{

InitializeComponent();

// Инициализация значений по умолчанию

FigureColor = Color.Black;

MaxFigureCoordinate = new Point(10\_000, 10\_000);

MaxFigureSize = new Size(10\_000, 10\_000);

}

// Обработчик события нажатия кнопки выбора цвета

private void buttonColor\_Click(object sender, EventArgs e)

{

// Создаем диалог выбора цвета

var cd = new ColorDialog();

cd.Color = FigureColor;

// Если цвет выбран, обновляем цвет фигуры и на панели

if (cd.ShowDialog() == DialogResult.OK)

{

FigureColor = cd.Color;

panelColor.BackColor = FigureColor;

}

}

// Обработчик события нажатия кнопки "Принять"

private void buttonAccept\_Click(object sender, EventArgs e)

{

// Определяем выбранную форму фигуры

GeometryForm = getFigureGeometryForm();

// Получаем значения размеров и координат фигуры

var width = (int)numericUpDownWidth.Value;

var height = (int)numericUpDownHeight.Value;

var x = (int)numericUpDownX.Value;

var y = (int)numericUpDownY.Value;

// Устанавливаем размеры и координаты фигуры

FigureSize = new Size(width, height);

FigurePoint = new Point(x, y);

// Устанавливаем результат DialogResult.OK и закрываем окно

DialogResult = DialogResult.OK;

Close();

}

// Метод для определения выбранной формы фигуры

private FigureGeometryForm getFigureGeometryForm()

{

// Проверяем, какая форма фигуры выбрана

if (radioButtonRectangle.Checked)

return FigureGeometryForm.Rectangle;

else if (radioButtonEllipse.Checked)

return FigureGeometryForm.Ellipse;

// Если ни одна форма не выбрана, выбрасываем исключение

throw new ArgumentException("Не выбрана форма фигу-ры");

}

}

}

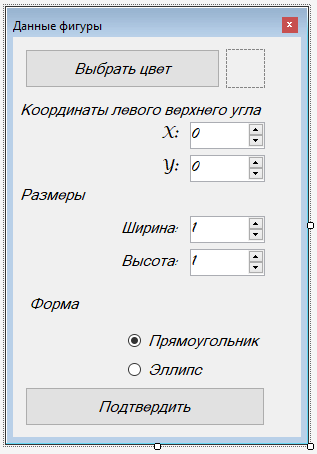


Рисунок 21 – Окно формы для ввода данных

Структура кода:

1. **Перечисление FigureGeometryForm**:
   * Определяет два вида фигур: прямоугольник и эллипс. Это позволяет выбирать форму, которую будет иметь фигура.
2. **Класс FigureForm**:
   * Этот класс представляет диалоговое окно, где пользователь может выбирать параметры для рисования фигуры.
3. **Свойства:**
   * FigureColor: Цвет фигуры, который пользователь может выбрать.
   * GeometryForm: Выбранная форма фигуры (прямоугольник или эллипс).
   * FigureSize: Размеры фигуры, которые вводит пользователь.
   * FigurePoint: Координаты фигуры, также вводимые пользователем.
   * MaxFigureSize: Максимальные размеры фигуры, чтобы ограничить ввод пользователя.
   * MaxFigureCoordinate: Максимальные координаты для фигуры.
4. **Конструктор FigureForm()**:
   * Инициализирует начальные значения для цвета, максимальных размеров и координат фигуры.
5. **Метод buttonColor\_Click**:
   * При нажатии кнопки выбора цвета открывается окно выбора цвета.
   * Выбранный цвет устанавливается для фигуры и отображается на панели.
6. **Метод buttonAccept\_Click**:
   * При нажатии кнопки "Принять" сохраняются выбранные пользователем параметры фигуры.
   * Эти параметры включают форму фигуры, размеры и координаты.
   * Диалоговое окно закрывается с результатом DialogResult.OK.
7. **Метод getFigureGeometryForm()**:
   * Определяет выбранную форму фигуры на основе выбора пользователя (прямоугольник или эллипс).

Этот класс позволяет пользователю выбирать цвет, форму, размеры и координаты фигуры для рисования. После ввода всех параметров и нажатия кнопки "Принять", диалоговое окно закрывается с сохраненными данными, которые могут быть использованы для рисования выбранной фигуры на главной форме (рисунок 22).

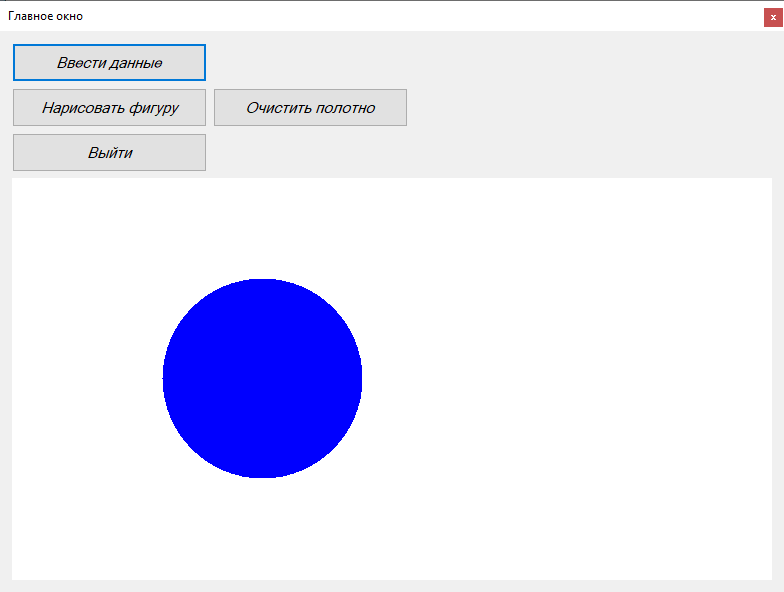
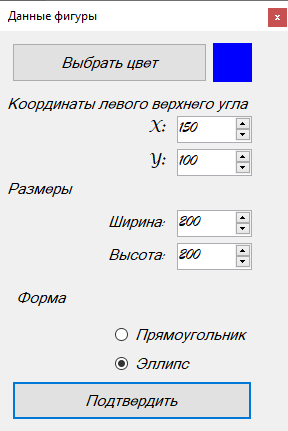


Рисунок 22 – Главное окно программы с отображением фигуры  
 и окно ввода данных

**Дополнительные материалы:**

1. Введение в Windows Forms. Первое приложение в Visual Studio. <https://metanit.com/sharp/windowsforms/1.1.php>
2. Событийная модель. Windows Forms. <https://ulearn.me/course/basicprogramming2/Sobytiynaya_model__73d69c1f-e3ac-4568-9473-8b0d0a522225>

**Задание на лабораторную работу**

1. Разработать графический интерфейс к лабораторной 1.
2. Подключить лабораторную 1 как библиотеку.
3. В соответствии с принципами ООП написать Windows-приложение, разместив на форме элементы управления.
4. Заголовок окна приложения должен содержать ФИО исполнителя, группу и номер варианта.
5. В программе должна быть предусмотрена обработка исключений, возникающих из-за ошибочного ввода пользователя.
6. Вспомогательные окна должны быть независимыми от главного окна.
7. Написать XML комментарии (с помощью ///) к классам и методам.

**Пример разработки приложения**

Пусть требуется разработать графический интерфейс для системы управления заказами в интернет-магазине.

Описание: Эта система предназначена для управления заказами в интернет-магазине, включая управление товарами, клиентами и статусами заказов.

Подключим как библиотеку dll лабораторную работу 5 с помощью вкладки «Зависимости», далее кнопка «Добавить ссылку на проект», далее «Обзор» необходимо выбрать dll разработанной ранее библиотеки.

Структура проекста представлена на рисунке 6.23.

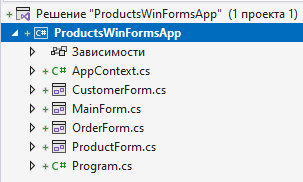


Рисунок 6.23 – Структура проекста

Внешний вид главного окна программы представлен на рисунке 6.24.

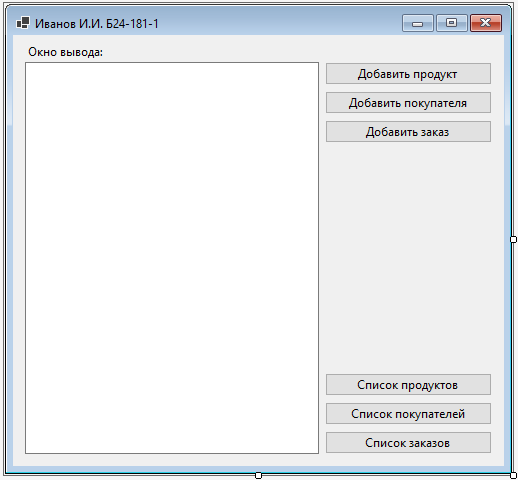


Рисунок 6.24 – Главное окно программы

Для корректного позиционирования элементов при изменении размера окна установим верхним кнопкам свойства Anchor: Right, Top. Нижним кнопкам: Right, Bottom. Окну вывода: Left, Right, Top, Bottom.

Создадим класс контекста данных, он представляет собой упрощённую базу данных и позволяет хранить и предоставлять товары, покупателей и заказы.

using ProductsLibrary;

namespace ProductsWinFormsApp;

public class AppContext

{

public static IList<Product> Products { get; } = new List<Product>();

public static IList<Customer> Customers { get; } = new List<Customer>();

public static IList<Order> Orders { get; } = new List<Order>();

}

Создадим форму для добавления нового товара (рисунок 6.25).

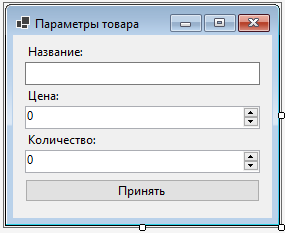


Рисунок 6.25 – Окно добавления товара

Запишем код для окна товара:

using ProductsLibrary;

namespace ProductsWinFormsApp

{

public partial class ProductForm : Form

{

public int NAME\_MAX\_LENGTH = 200;

public Product? Product { get; private set; }

public ProductForm()

{

InitializeComponent();

}

private void confirmButton\_Click(object sender, EventArgs e)

{

var name = nameTextBox.Text;

if (name.Length > NAME\_MAX\_LENGTH)

{

MessageBox.Show($"Имя не должно быть больше {NAME\_MAX\_LENGTH} символов");

return;

}

var price = (double)priceNumeric.Value;

var quantity = (int)quantityNumeric.Value;

Product = new Product(Guid.NewGuid().ToString(), name, price, quantity);

DialogResult = DialogResult.OK;

Close();

}

}

}

Создадим форму для добавления нового покупателя (рисунок 6.26).

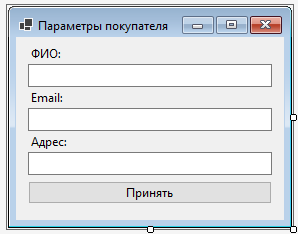


Рисунок 6.26 – Окно добавления покупателя

Запишем код для окна покупателя:

using ProductsLibrary;

namespace ProductsWinFormsApp

{

public partial class CustomerForm : Form

{

public int NAME\_MAX\_LENGTH = 200;

public Customer? Customer { get; private set; }

public CustomerForm()

{

InitializeComponent();

}

private void confirmButton\_Click(object sender, EventArgs e)

{

var name = nameTextBox.Text;

if (name.Length > NAME\_MAX\_LENGTH)

{

MessageBox.Show($"Имя не должно быть больше {NAME\_MAX\_LENGTH} символов");

return;

}

var email = emailTextBox.Text;

if (!email.Contains("@"))

{

MessageBox.Show($"Неверный адрес эл. почты");

return;

}

var address = addressTextBox.Text;

if (string.IsNullOrWhiteSpace(address))

{

MessageBox.Show($"Неверный адрес");

return;

}

Customer = new Customer(Guid.NewGuid().ToString(), name, email, address);

DialogResult = DialogResult.OK;

Close();

}

}

}

Создадим форму для оформления заказа (рисунок 6.27).

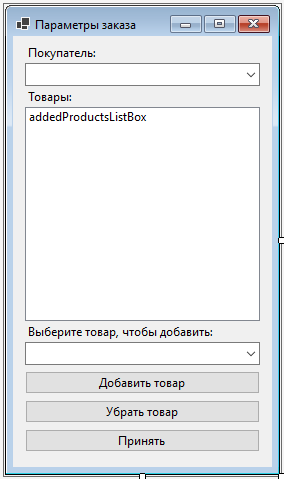


Рисунок 6.27 – Окно оформления заказа

Запишем код для окна покупателя:

using ProductsLibrary;

namespace ProductsWinFormsApp

{

public partial class OrderForm : Form

{

public int NAME\_MAX\_LENGTH = 200;

public Order? Order { get; private set; }

public OrderForm()

{

InitializeComponent();

}

private void confirmButton\_Click(object sender, EventArgs e)

{

var customer = customersComboBox.SelectedItem as Customer;

if (customer == null)

{

MessageBox.Show("Необходимо выбрать покупателя!");

return;

}

Order = new Order(Guid.NewGuid().ToString(), customer);

var products = addedProductsListBox.Items.Cast<Product>().ToArray();

foreach (var product in products)

{

Order.AddProduct(product);

}

DialogResult = DialogResult.OK;

Close();

}

private void OrderForm\_Load(object sender, EventArgs e)

{

customersComboBox.Items.AddRange(AppContext.Customers.ToArray());

productsComboBox.Items.AddRange(AppContext.Products.ToArray());

}

private void addProductButton\_Click(object sender, EventArgs e)

{

var product = productsComboBox.SelectedItem as Product;

addedProductsListBox.Items.Add(product);

}

private void removeProductButton\_Click(object sender, EventArgs e)

{

addedProductsListBox.Items.Remove(addedProductsListBox.SelectedItem);

}

}

}

Запишем код главного окна, который открывает записанные ранее окна и отображает информацию:

using ProductsLibrary;

namespace ProductsWinFormsApp

{

public partial class OrderForm : Form

{

public int NAME\_MAX\_LENGTH = 200;

public Order? Order { get; private set; }

public OrderForm()

{

InitializeComponent();

}

private void confirmButton\_Click(object sender, EventArgs e)

{

var customer = customersComboBox.SelectedItem as Customer;

if (customer == null)

{

MessageBox.Show("Необходимо выбрать покупателя!");

return;

}

Order = new Order(Guid.NewGuid().ToString(), customer);

var products = addedProductsListBox.Items.Cast<Product>().ToArray();

foreach (var product in products)

{

Order.AddProduct(product);

}

DialogResult = DialogResult.OK;

Close();

}

private void OrderForm\_Load(object sender, EventArgs e)

{

customersComboBox.Items.AddRange(AppContext.Customers.ToArray());

productsComboBox.Items.AddRange(AppContext.Products.ToArray());

}

private void addProductButton\_Click(object sender, EventArgs e)

{

var product = productsComboBox.SelectedItem as Product;

addedProductsListBox.Items.Add(product);

}

private void removeProductButton\_Click(object sender, EventArgs e)

{

addedProductsListBox.Items.Remove(addedProductsListBox.SelectedItem);

}

}

}

**Контрольные вопросы**

1. Что такое делегаты в C# и какова их основная цель?
2. Как объявить делегат и как его использовать?
3. В чем разница между многократным делегатом и делегатом с одним методом?
4. Что такое анонимные методы и как они отличаются от обычных методов?
5. Как создать анонимный метод и в каких случаях его следует использовать?
6. Что такое лямбда-выражения в C# и как они связаны с делегатами?
7. Какова синтаксическая структура лямбда-выражения?
8. В чем преимущества использования лямбда-выражений по сравнению с анонимными методами?
9. Как можно использовать лямбда-выражения для работы с коллекциями, например, с методом Where?
10. Что такое события в C# и как они связаны с делегатами?
11. Как объявить и реализовать событие в классе?
12. Как подписаться на событие и отписаться от него?
13. Что такое EventHandler и как он используется в событиях?
14. В чем разница между стандартным событием и событием с пользовательским делегатом?
15. Как можно передать параметры в обработчик события?
16. Как работает механизм уведомления об изменениях в событиях?
17. Что такое Windows Forms и каковы его основные компоненты?
18. Как создать простую форму в Windows Forms?
19. Как добавить элементы управления на форму?
20. Как обработать событие нажатия кнопки в Windows Forms?
21. Как можно использовать делегаты и события для обработки пользовательских действий в Windows Forms?
22. Как реализовать простую форму с несколькими вкладками в Windows Forms?
23. Как использовать DataGridView для отображения и редактирования данных в Windows Forms?
24. Что такое BindingSource и как он упрощает работу с данными в Windows Forms?
25. Как обрабатывать исключения, возникающие при работе с событиями?
26. Как можно использовать лямбда-выражения для обработки событий в Windows Forms?
27. Каковы основные принципы работы с жизненным циклом формы в Windows Forms?