Тема 7. Создание пользовательских элементов управления.

Цель занятия:

Сформировать у обучающихся понимание назначения пользовательских элементов управления в Windows Forms, изучить принципы их создания, настройки и повторного использования в приложениях.

Учебные вопросы:

1. Создание пользовательских элементов управления. Введение.
2. Архитектура класса Control.
3. Жизненный цикл и отрисовка (GDI+).
4. Интеграция с Visual Studio Designer.
5. Обработка событий и интерактивность.

1. Создание пользовательских элементов управления. Введение.

Стандартная библиотека Windows Forms предоставляет богатый набор элементов управления (Button, Label, TreeView и т.д.), которые являются обертками над стандартными компонентами Windows (WinAPI).

Однако в профессиональной разработке часто возникают задачи, выходящие за рамки их возможностей.

Зачем создавать свои элементы?

- Уникальный дизайн (UI/UX): Создание брендированного интерфейса, который невозможно реализовать стандартными средствами (например, круглые кнопки, анимированные переключатели или сложные графики).
- Специфическая бизнес-логика: Объединение нескольких функций в одном компоненте (например, поле ввода с автоматической валидацией и встроенным индикатором сложности пароля).
- Повторное использование кода: Чтобы не копировать одну и ту же логику оформления или поведения между разными формами и проектами.
- Оптимизация: Создание облегченного элемента, который отрисовывает только то, что нужно, без лишних ресурсов тяжелых стандартных контролов.

Три способа создания элементов в WinForms:

- UserControl: Композиция (сборка из готовых кнопок/полей).
- Расширение (Inheritance): Наследование от существующего контрола (например, class MyButton : Button).
- Custom Control (Наследование от Control): Создание элемента «с нуля».

Выбор способа зависит от того, насколько глубоко вы хотите изменить поведение и внешний вид элемента.

UserControl (Пользовательский элемент управления)

Это композиция. Вы создаете «контейнер», на который перетаскиваете уже существующие элементы (кнопки, текстовые поля, списки).

Наследование: от класса System.Windows.Forms.UserControl.

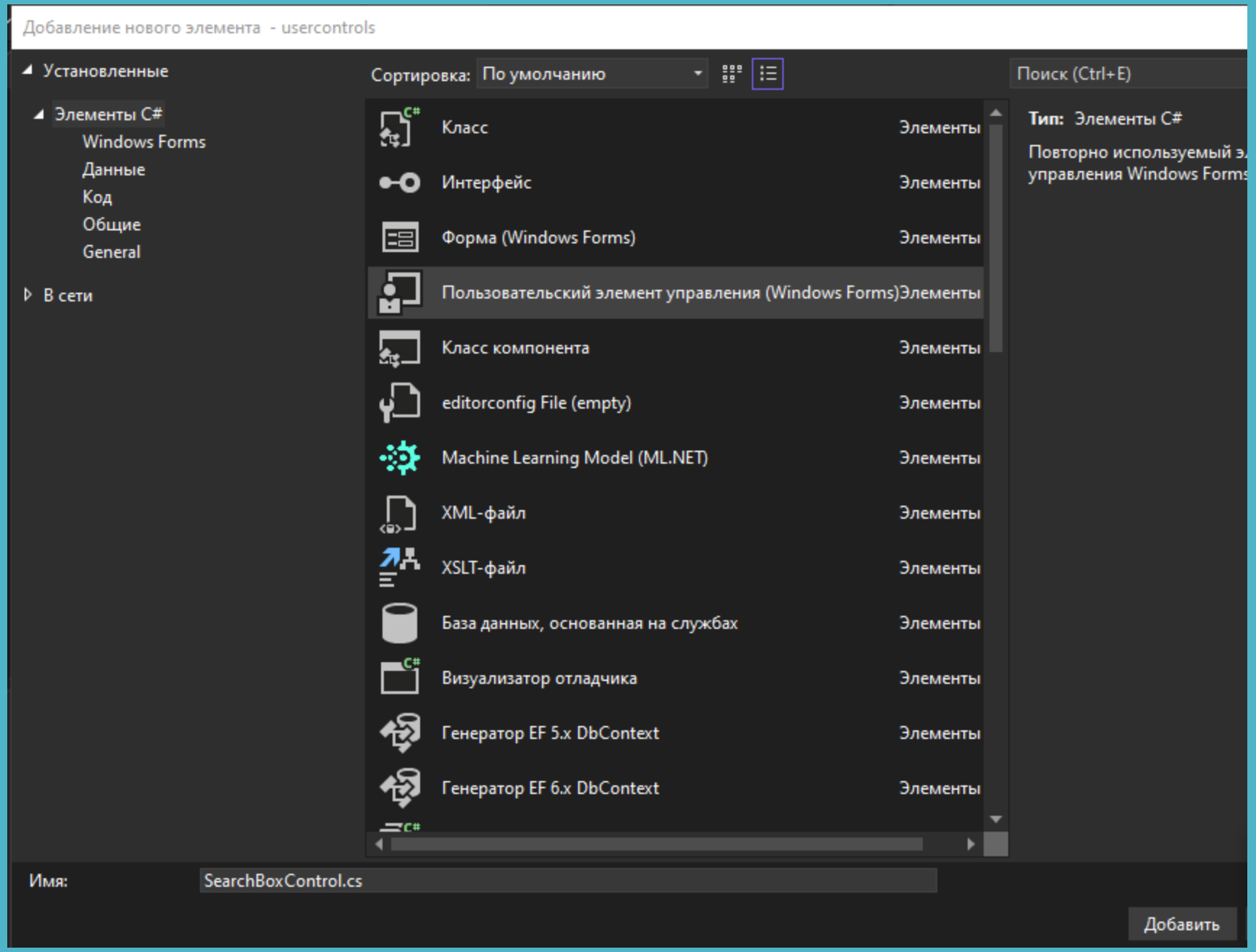
Плюсы: Быстрая визуальная разработка в дизайнере VS.

Пример: Карточка товара, содержащая PictureBox, Label (название) и Button (купить).

Пример. Представим, что нам нужен блок для поиска: текстовое поле и кнопка, которые всегда работают вместе.

Шаг 1: Создание файла

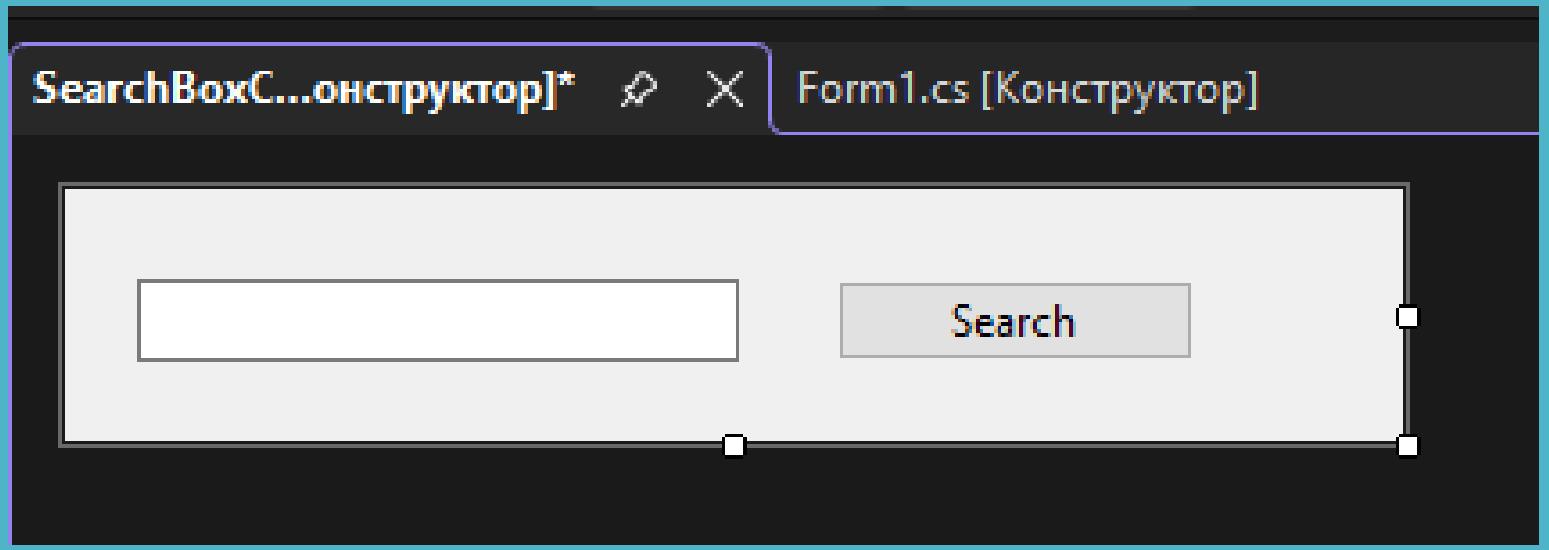
В Visual Studio вы нажимаете правой кнопкой мыши по проекту: Добавить → Пользовательский элемент управления (Windows Forms). Назовем его SearchBoxControl. Перед вами откроется пустое серое поле, похожее на форму, но без рамок и заголовка.


Добавление нового элемента - usercontrols
Установленные
Элементы C#
Windows Forms
Данные
Код
Общие
General
В сети
Сортировка: По умолчанию
Поиск (Ctrl+E)
Класс Элементы
Интерфейс Элементы
Форма (Windows Forms) Элементы
Пользовательский элемент управления (Windows Forms) Элементы
Класс компонента Элементы
editorconfig File (empty) Элементы
Machine Learning Model (ML.NET) Элементы
XML-файл Элементы
XSLT-файл Элементы
База данных, основанная на службах Элементы
Визуализатор отладчика Элементы
Генератор EF 5.x DbContext Элементы
Генератор EF 6.x DbContext Элементы
Тип: Элементы C#
Повторно используемый э
управления Windows Form
Имя: SearchBoxControl.cs
Добавить

Шаг 2: Визуальное проектирование

Вы просто перетаскиваете элементы из Toolbox (Панели инструментов) на это поле:

- TextBox (назовем его txtQuery).
- Button (назовем его btnSearch).

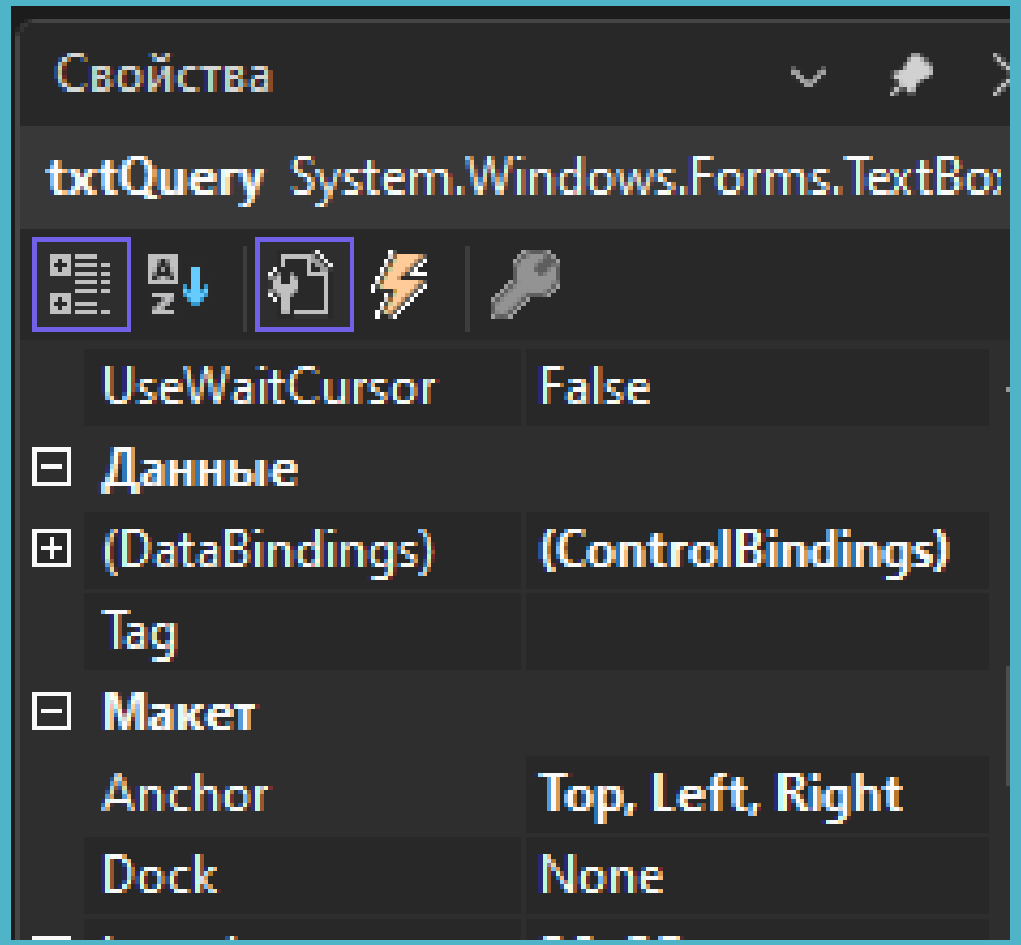


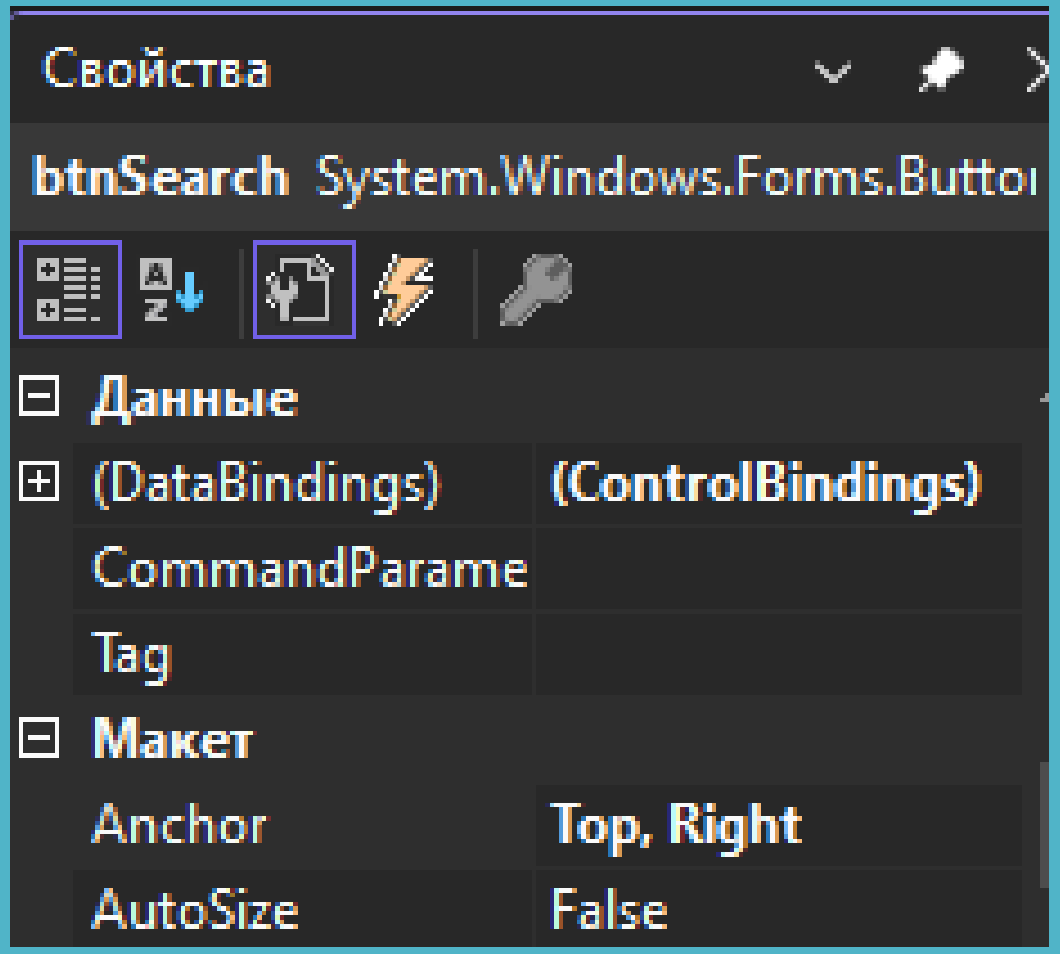
Шаг 3: Настройка внутренних свойств

Чтобы ваш контрол выглядел профессионально, можно настроить Anchor (привязки):

Для txtQuery установите Anchor = Top, Left, Right. Тогда при растягивании вашего компонента текстовое поле будет удлиняться.

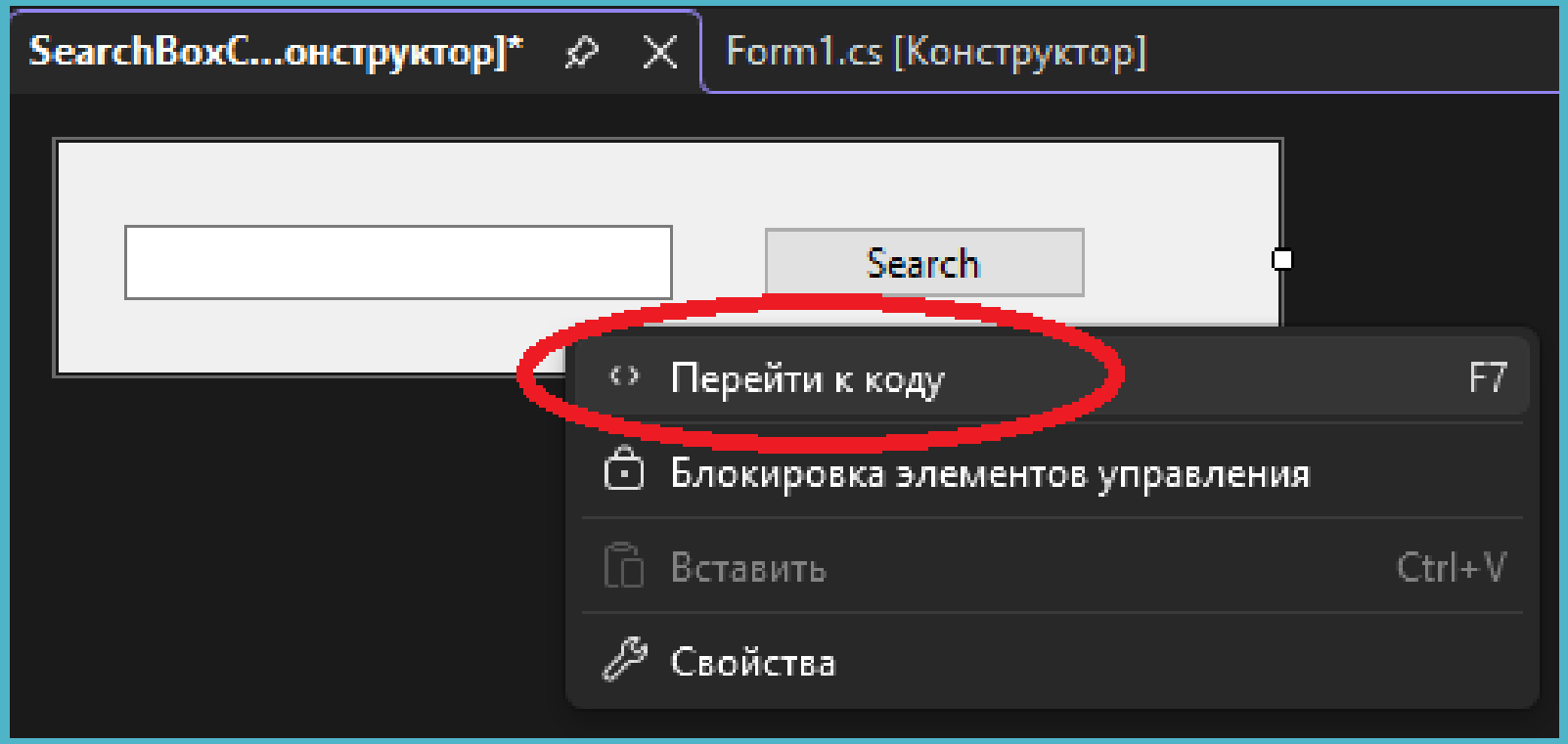
Для btnSearch установите Anchor = Top, Right. Кнопка всегда будет оставаться справа.

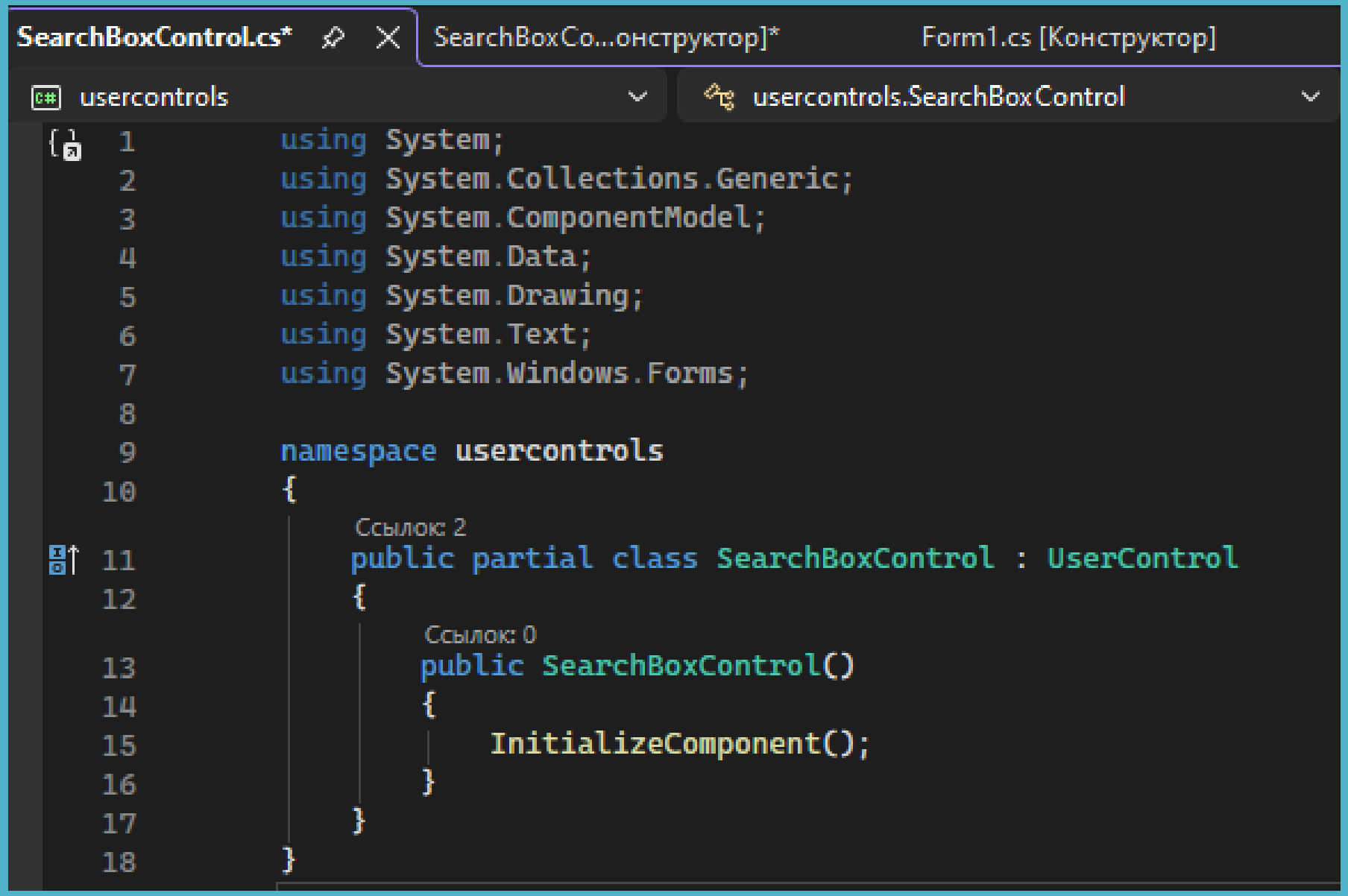

Свойства
txtQuery System.Windows.Forms.TextBo
UseWaitCursor
False
Данные
(DataBindings)
(ControlBindings)
Tag
Макет
Anchor
Top, Left, Right
Dock
None


Свойства
btnSearch System.Windows.Forms.Buttor
Данные
(DataBindings)
(ControlBindings)
CommandParame
Tag
Макет
Anchor
Top, Right
AutoSize
False

Шаг 4: Написание логики (Code-behind)

Теперь нужно сделать так, чтобы форма, на которую вы положите этот контрол, могла с ним взаимодействовать.




SearchBoxControl.cs*
SearchBoxCo...онструктор]*
Form1.cs [Конструктор]
usercontrols
usercontrols.SearchBoxControl
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace usercontrols
{
Ссылок: 2
public partial class SearchBoxControl : UserControl
{
Ссылок: 0
public SearchBoxControl()
{
InitializeComponent();
}
}
}

Изменим существующий класс:

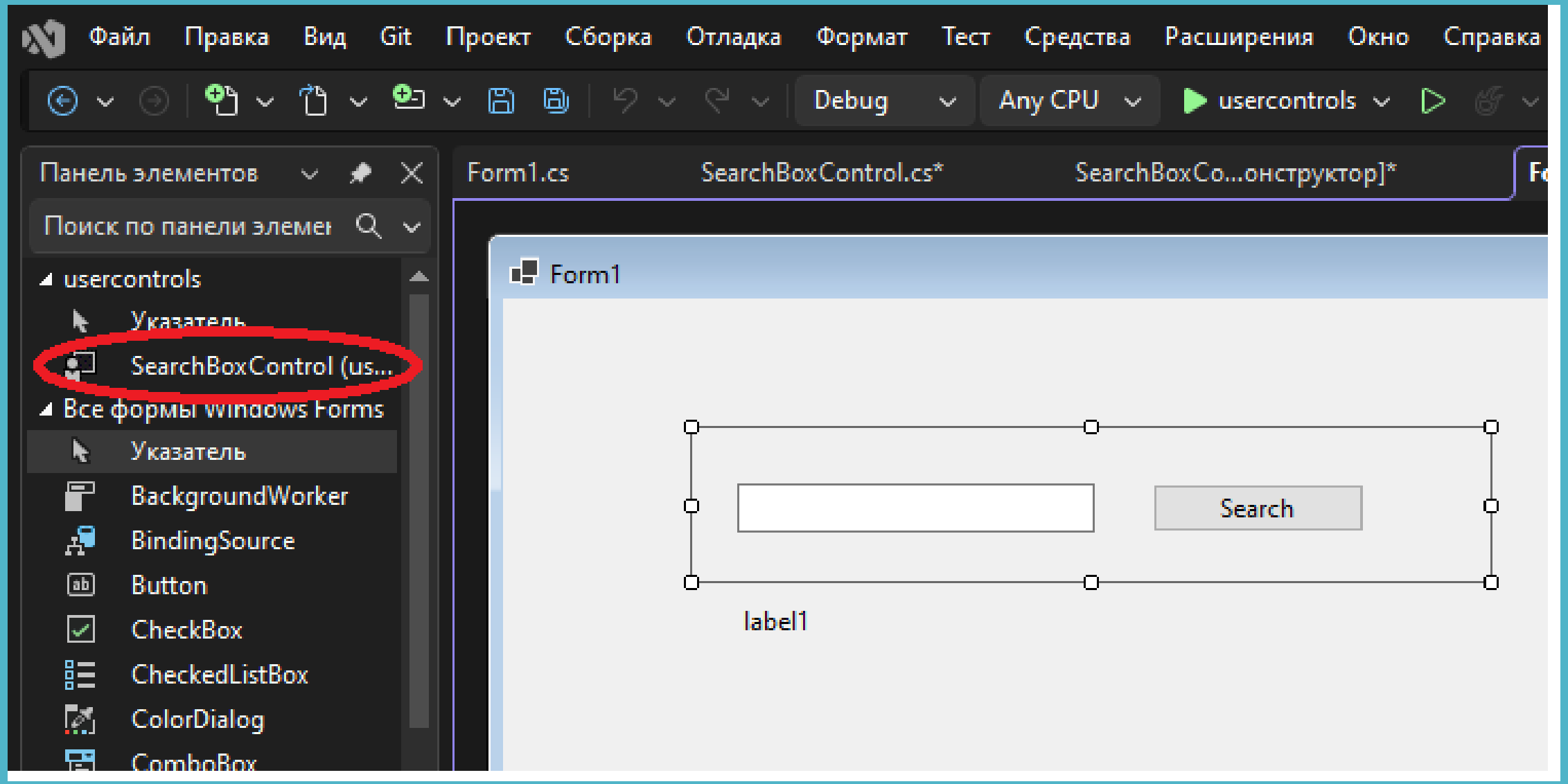
```
Ссылок: 4
public partial class SearchBoxControl : UserControl
{
    // Событие, которое мы выставим наружу для формы
    public event EventHandler SearchClicked;
    Ссылок: 1
    public SearchBoxControl()
    {
        InitializeComponent();
        // Подписываемся на клик внутренней кнопки
        btnSearch.Click += (s, e) => {
            // Генерируем внешнее событие
            SearchClicked?.Invoke(this, EventArgs.Empty);
        };
    }
    // Свойство для получения текста из внутреннего TextBox
    Ссылок: 1
    public string SearchText => txtQuery.Text;
}
```

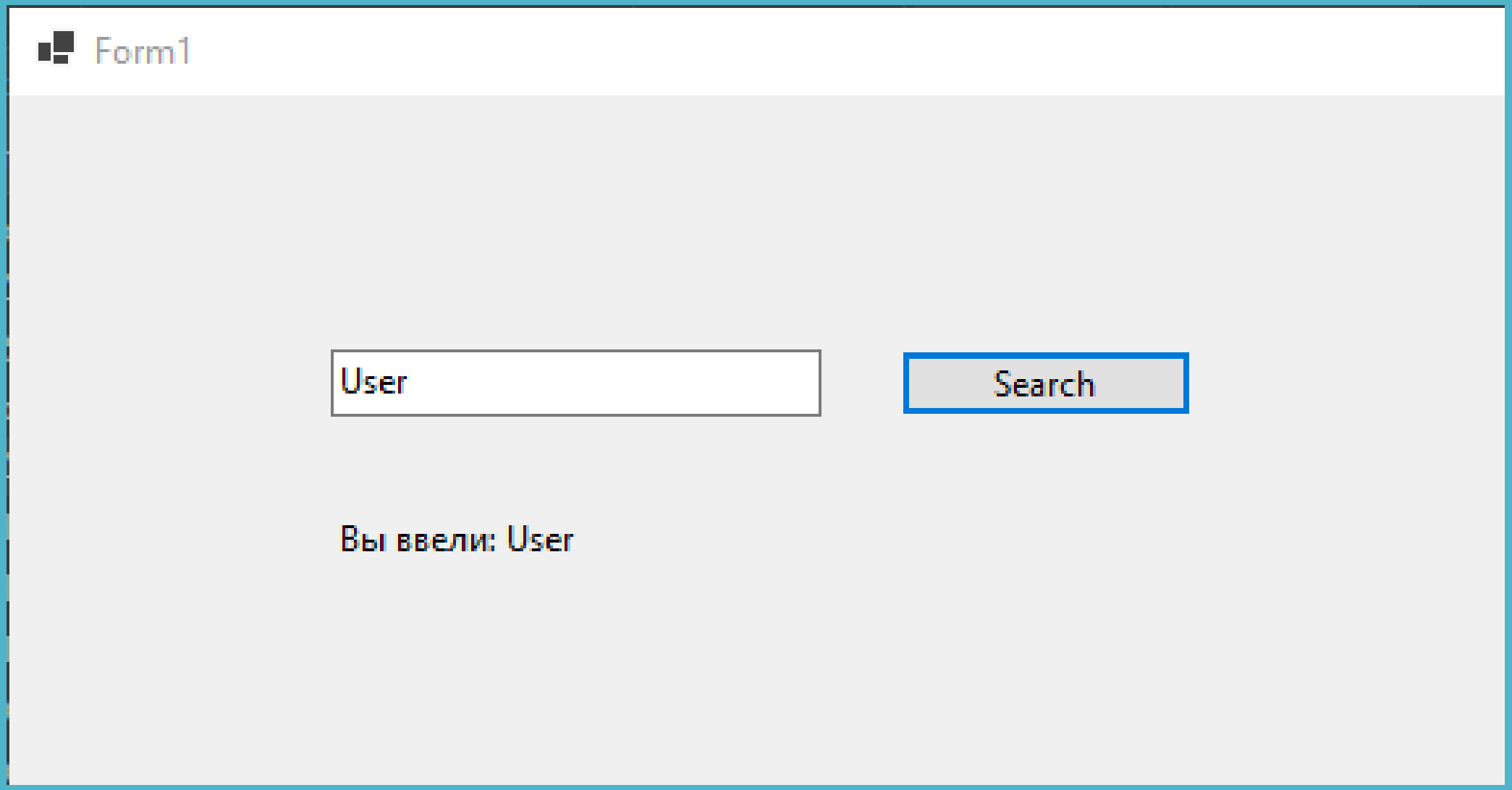
Шаг 5: Использование на форме

Соберите проект (Build -> Build Solution).

Ваш новый элемент SearchBoxControl появится в самом верху Toolbox.

Просто перетащите его на любую форму.


Файл Правка Вид Git Проект Сборка Отладка Формат Тест Средства Расширения Окно Справка
Debug
Any CPU
usercontrols
Панель элементов
Поиск по панели элемен
usercontrols
Указатель
SearchBoxControl (us...
Все формы Windows Forms
Указатель
BackgroundWorker
BindingSource
Button
CheckBox
CheckedListBox
ColorDialog
ComboBox
Form1.cs
SearchBoxControl.cs*
SearchBoxCo...онструктор]*
Form1
Search
label1


Form1
User
Search
Вы ввели: User

Расширение (Наследование от существующего элемента)

Это модификация. Вы берете готовый элемент (например, TextBox) и добавляете ему новую функциональность или немного меняете отрисовку.

Наследование: от конкретного класса (например, public class NumericTextBox : TextBox).

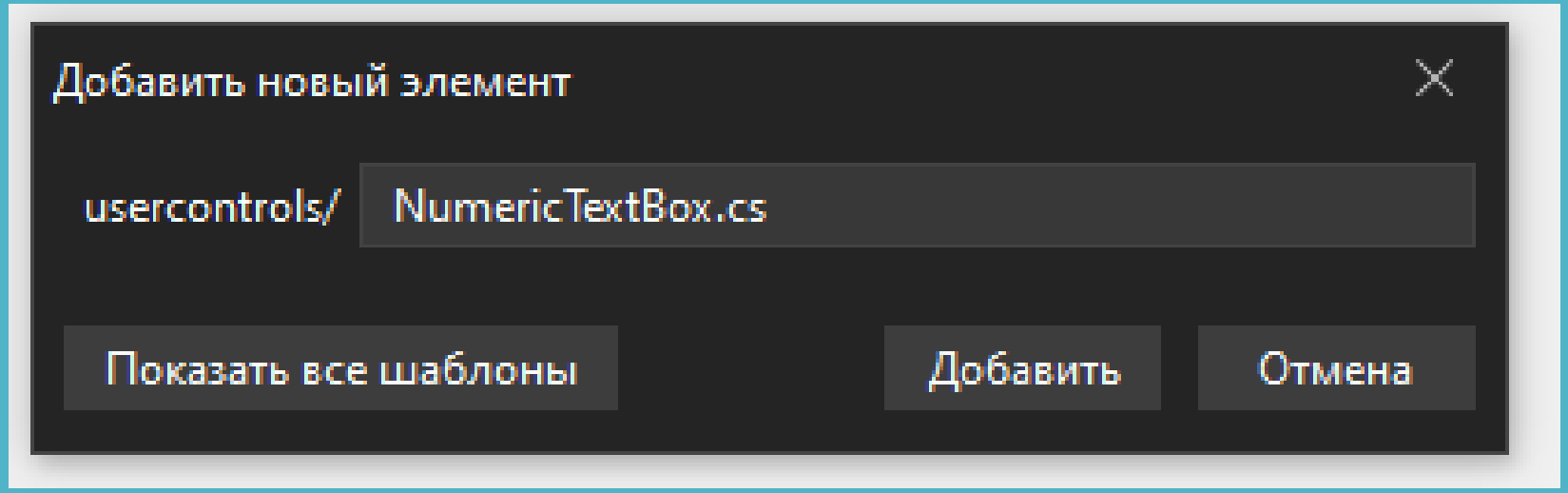
Плюсы: Сохранение всех базовых свойств и событий родителя.

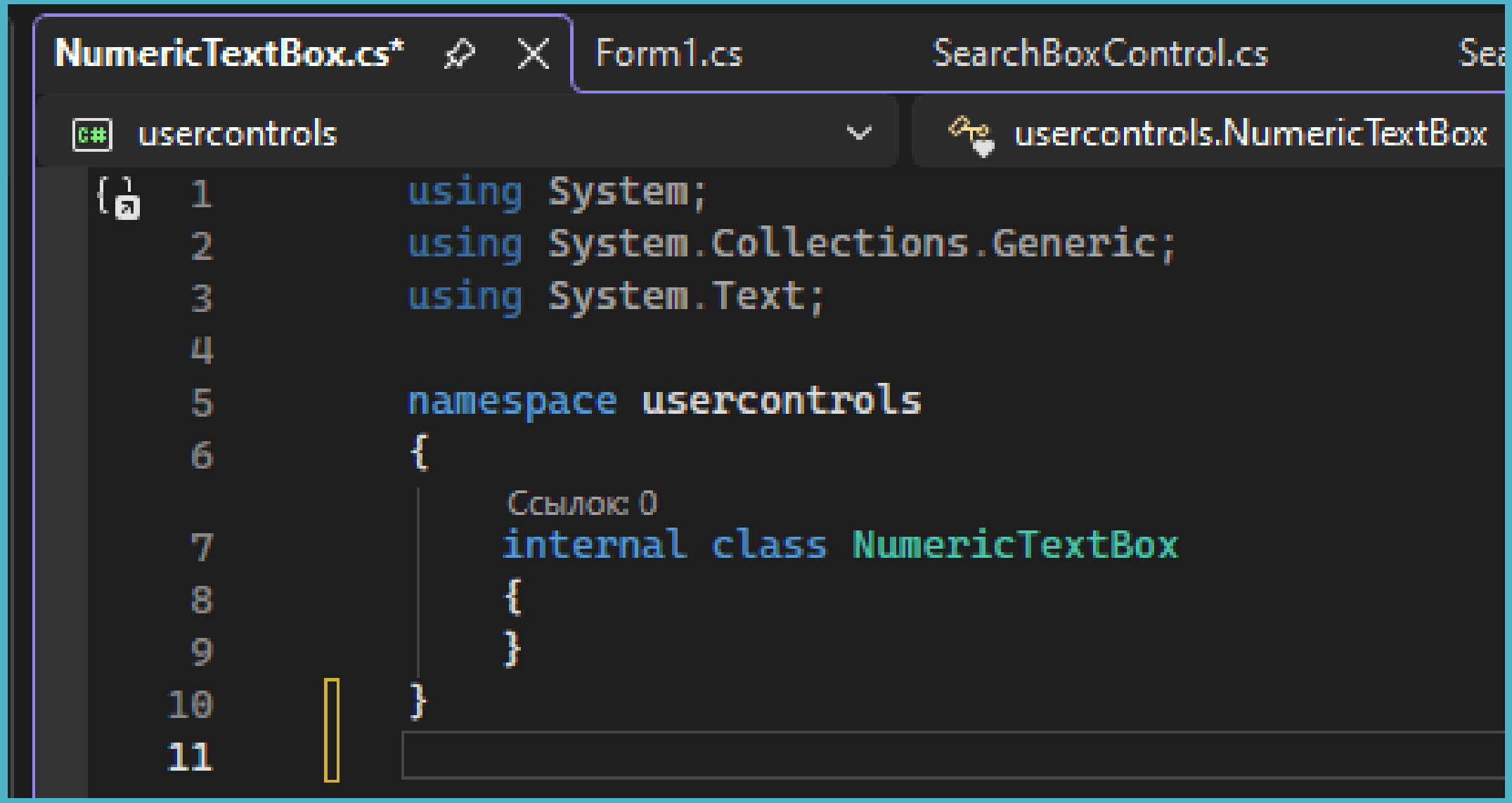
Пример: TextBox, который разрешает ввод только цифры.

Пример: TextBox, который разрешает ввод только цифры.

Шаг 1: Создание класса.

Вместо использования визуального дизайнера, мы создаем обычный класс .cs. Назовем его NumericTextBox.




NumericTextBox.cs*
Form1.cs
SearchBoxControl.cs
usercontrols
usercontrols.NumericTextBox
using System;
using System.Collections.Generic;
using System.Text;
namespace usercontrols
{
Ссылок: 0
internal class NumericTextBox
{
}
}

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

namespace usercontrols
{
    // Наследуемся от TextBox, чтобы получить все его свойства
    Ссылок: 0
    public class NumericTextBox : TextBox
    {
    }
}
```

Шаг 2: Перехват ввода (OnKeyPress)

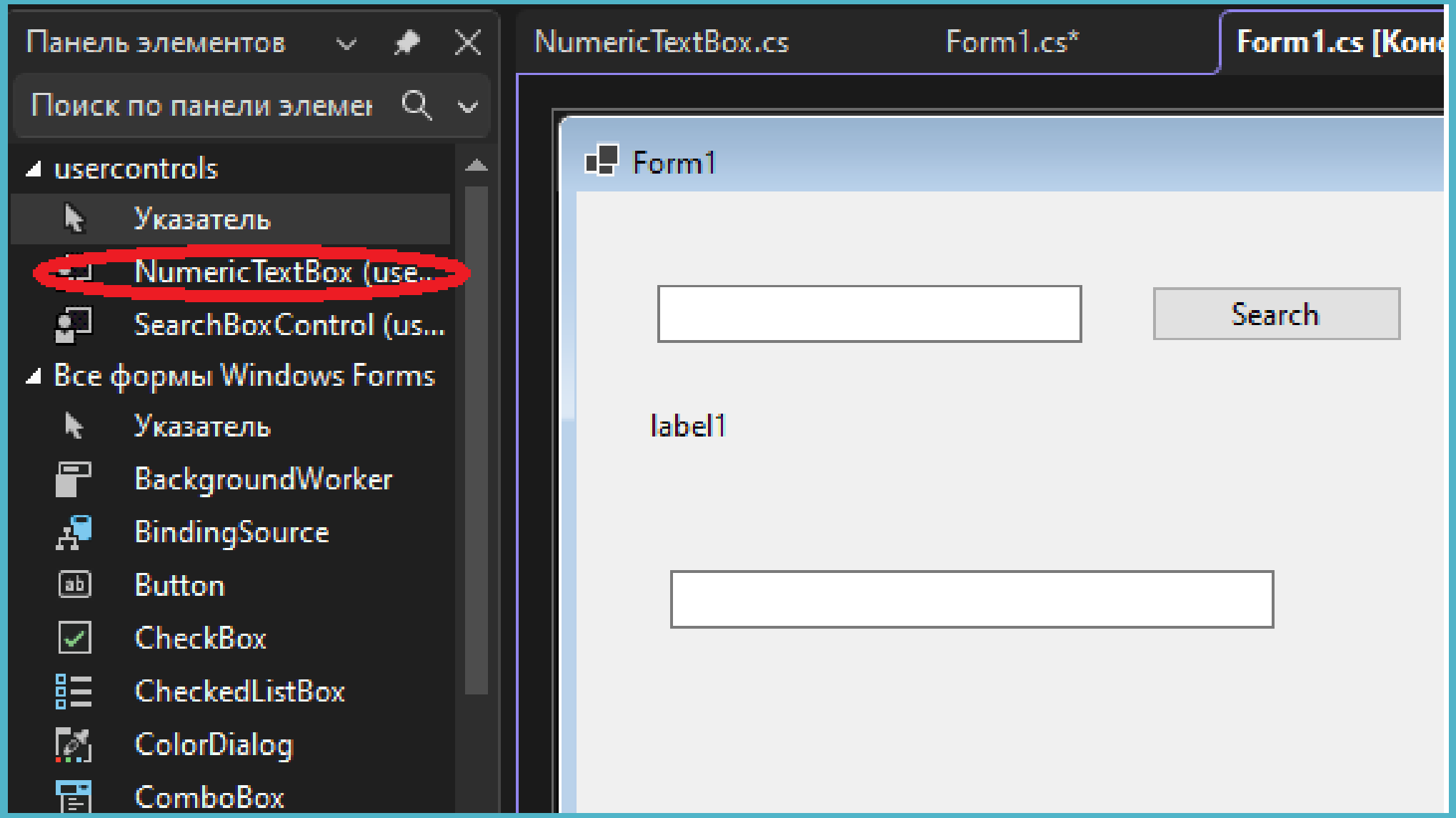
В Windows Forms за нажатие клавиш отвечает метод OnKeyPress. Нам нужно его переопределить (override). Это значит: «Когда нажата клавиша, сначала выполни мой код, а потом стандартный».

```
public class NumericTextBox : TextBox
{
    Ссылок: 0
    protected override void OnKeyPress(KeyPressEventArgs e)
    {
        // Вызываем базовый метод, чтобы стандартные функции работали
        base.OnKeyPress(e);

        // Проверяем: если символ НЕ цифра И НЕ клавиша Backspace (удаление)
        if (!char.IsDigit(e.KeyChar) && e.KeyChar != (char)Keys.Back)
        {
            // Устанавливаем флаг "Обработано".
            // Символ будет заблокирован и не появится в поле.
            e.Handled = true;
        }
    }
}
```

Шаг 3: Использование на форме

- Скомпилируйте проект (Ctrl + Shift + B).
- Откройте Toolbox (Панель инструментов).
- В самом верху вы увидите новый компонент: NumericTextBox.
- Просто перетащите его на форму, как обычное текстовое поле.


Панель элементов
Поиск по панели элемен
usercontrols
Указатель
NumericTextBox (use...
SearchBoxControl (us...
Все формы Windows Forms
Указатель
BackgroundWorker
BindingSource
Button
CheckBox
CheckedListBox
ColorDialog
ComboBox
NumericTextBox.cs
Form1.cs*
Form1.cs [Кон
Form1
Search
label1

Custom Control (Наследование от Control)

Это создание элемента «с чистого листа». Вы сами отвечаете за то, как элемент выглядит (рисуете его через GDI+) и как он реагирует на клики.

Наследование: напрямую от System.Windows.Forms.Control.

Плюсы: Максимальная гибкость и полный контроль над каждым пикселем. Минимальное потребление ресурсов (нет лишнего функционала родительских классов).

Пример: Круговая диаграмма, спидометр, игровой джойстик на форме.

2. Архитектура класса Control

Класс System.Windows.Forms.Control — это базовый класс для всех визуальных компонентов. Все, что вы видите на форме (и сама форма), унаследовано от него.

Иерархия наследования:

- Object: Базовый тип .NET.
- MarshalByRefObject: Позволяет объекту взаимодействовать через границы доменов приложений (важно для работы с памятью Windows).
- Component: Позволяет элементу размещаться в контейнерах и взаимодействовать с дизайнером Visual Studio.
- Control: Добавляет «физическое» воплощение: окно, координаты и способность рисовать.

Наследуясь от Control, ваш будущий элемент автоматически получает:

- Свойства геометрии: Location (позиция), Size (размер), Width, Height, Anchor, Dock.
- Свойства стиля: BackColor (цвет фона), ForeColor (цвет текста), Font, BackgroundImage.
- Состояние: Enabled (доступен ли), Visible (виден ли), Focused (есть ли на нем фокус).
- Дескриптор окна (Handle): Уникальный идентификатор IntPtr, через который операционная система Windows обращается к вашему контролу.

Основные группы свойств класса Control

Архитектурно свойства Control можно разделить на несколько логических групп.

1. Геометрия и расположение

- Size, Width, Height
- Location
- Bounds
- ClientRectangle

Данные свойства определяют прямоугольную область, занимаемую контролом в контейнере.

2. Внешний вид (визуальные свойства)

- BackColor
- ForeColor
- Font
- BackgroundImage

Эти свойства влияют на визуальное оформление, но не изменяют геометрию контрола.

3. Геометрическая форма контрола (Region)

Свойство Region определяет фактическую форму элемента управления.

public Region Region { get; set; }

В отличие от Size и Bounds, которые задают прямоугольник размещения,

Region позволяет задать произвольную геометрию.

Назначение Region:

- определение формы контрола;
- ограничение области отрисовки;
- определение области обработки событий мыши;
- корректное отображение нестандартных элементов в дизайнере.

Поведение по умолчанию:

если Region == null, контрол считается прямоугольным.

Использование свойства Region целесообразно, когда:

- требуется нестандартная форма контрола;
- необходимо исключить кликабельные области;
- визуальная форма должна соответствовать логической;
- создаются пользовательские элементы управления.

Поскольку Region не обновляется автоматически,

логика его изменения обычно выносится в отдельный метод, например:

void UpdateRegion()

Этот метод вызывается:

- при изменении размеров контрола;
- при изменении параметров формы;
- при необходимости принудительного пересчёта геометрии.

4. Отрисовка и визуальный цикл

- OnPaint
- OnPaintBackground
- Invalidate
- Refresh

Важно различать:

- OnPaint — рисует внешний вид;
- Region — задаёт геометрию контрола.

5. Ввод и взаимодействие с пользователем

- События мыши (OnMouseDown, OnMouseMove, OnMouseUp)
- События клавиатуры (OnKeyDown, OnKeyPress)
- Фокус (Focused, TabStop)

Обработка событий осуществляется только внутри области Region.

Наследуясь

Наследуясь

Ключевые методы, которые мы будем переопределять

При создании элемента «с нуля» мы чаще всего работаем с этими «точками входа»:

- OnPaint(PaintEventArgs e): Самый важный метод. Здесь мы рисуем внешний вид элемента.
- OnResize(EventArgs e): Срабатывает при изменении размера. Здесь мы обычно пересчитываем логику рисования.
- OnMouseEnter / OnMouseLeave: Для создания эффектов при наведении (например, подсвечивание кнопки).
- OnMouseDown / OnMouseUp: Для обработки кликов и реализации нажатий.

Важно помнить: Класс Control — это «чистый лист». В нем нет готового текста или рамок.

Если вы просто создадите класс-наследник и ничего в нем не напишете, на форме он будет выглядеть как невидимый прямоугольник.

3. Жизненный цикл и отрисовка (GDI+)

В Windows Forms отрисовка происходит не постоянно, а по запросу. Система вызывает специальный метод каждый раз, когда нужно обновить изображение элемента на экране (например, если вы свернули и развернули окно).

Метод OnPaint

Это главная точка входа для визуализации. Когда вы наследуетесь от Control, вы переопределяете этот метод, чтобы сказать программе, как именно должен выглядеть ваш элемент.

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e); // Вызываем базовую отрисовку фона
                     // Наш код рисования здесь...
}
```

Объект PaintEventArgs

Этот объект предоставляет два критически важных инструмента:

- e.Graphics (Объект GDI+): Это ваш «виртуальный холст». Он содержит методы для рисования:
  - DrawLine, DrawRectangle, DrawEllipse — для контуров.
  - FillRectangle, FillEllipse — для закрашенных фигур.
  - DrawString — для вывода текста.
  - DrawImage — для вставки картинок.
- e.ClipRectangle: Прямоугольная область, которую необходимо перерисовать в данный момент. Использование этого свойства помогает оптимизировать программу, чтобы не рисовать весь элемент, если изменился только его маленький кусочек.

Инструменты рисования: Кисти и Перья

Чтобы что-то изобразить, GDI+ требует инструменты:

- Pen (Перо): Используется для рисования линий и границ. Имеет свойства Color и Width.
- Brush (Кисть): Используется для заливки областей.
- SolidBrush — однотонный цвет.
- LinearGradientBrush — плавный переход цветов.

Важно: Инструменты Pen и Brush потребляют системные ресурсы. В профессиональном коде их принято создавать внутри блока using, чтобы они сразу удалялись из памяти.

Принудительная перерисовка: Invalidate() vs Refresh()

Если вы изменили какое-то свойство (например, цвет индикатора), элемент не перекрасится сам собой. Вам нужно «попросить» Windows перерисовать его.

- Invalidate() (Рекомендуется): Помечает элемент как «грязный» и ставит задачу на перерисовку в очередь сообщений Windows. Это происходит плавно и не тормозит интерфейс.
- Refresh(): Заставляет элемент перерисоваться немедленно. Это может вызвать «фризы» (зависания), если вызывать его слишком часто (например, в циклах).

Типичный алгоритм внутри OnPaint:

- Получаем холст из e.Graphics.
- Настраиваем сглаживание: e.Graphics.SmoothingMode = SmoothingMode.AntiAlias; (чтобы края не были «зубчатыми»).
- Рисуем фон (Кистью).
- Рисуем границы (Пером).
- Рисуем текст или иконки.

Пример. Круглая кнопка-индикатор, которая меняет цвет с серого на красный при клике.

Создадим в проекте новый класс LedIndicator.cs и вставим следующий код:

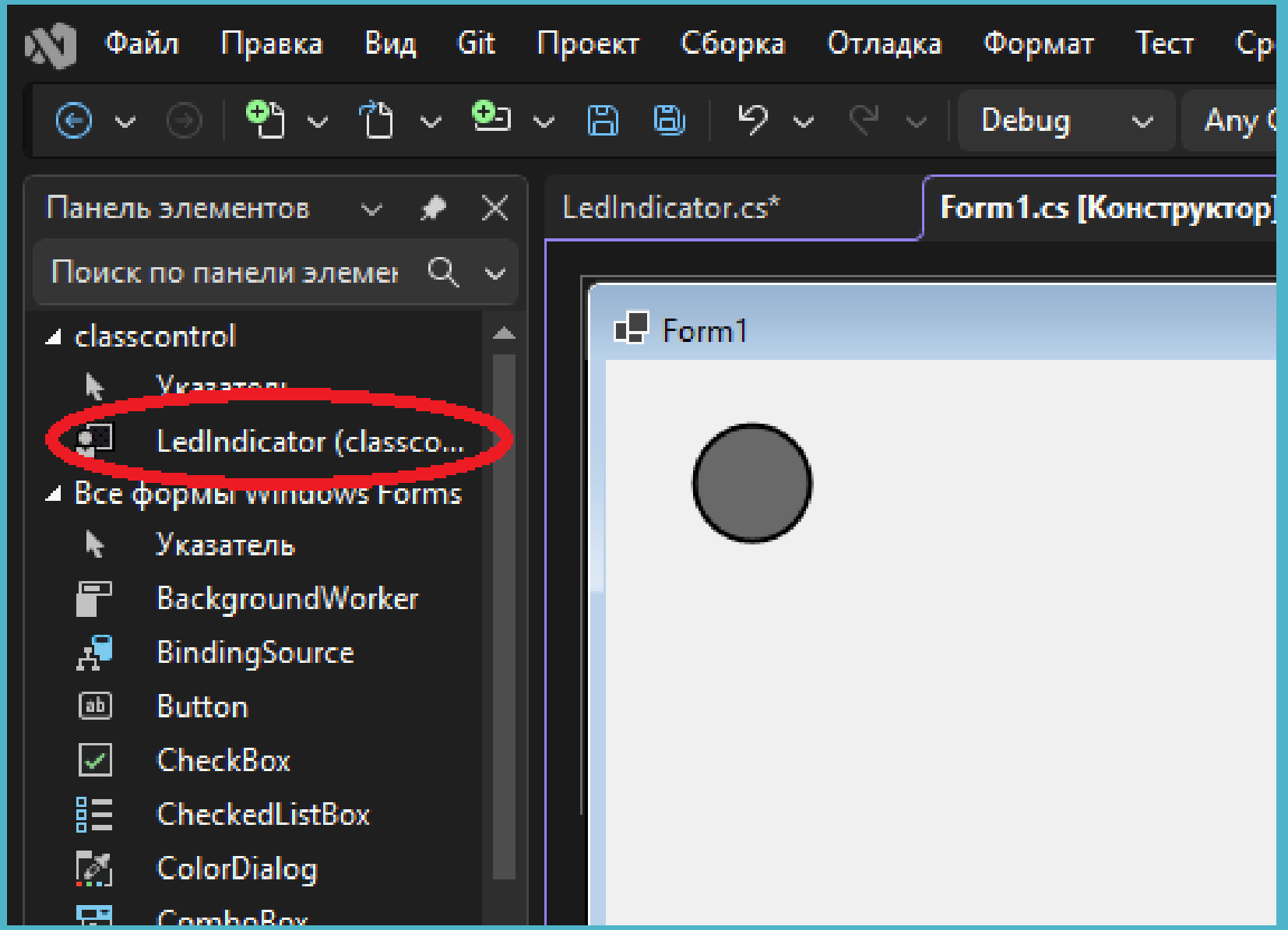
```
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing.Drawing2D; // Для сглаживания
using System.ComponentModel;
```

```
namespace classcontrol
{
    Ссылок: 3
    public class LedIndicator : Control
    {
        private bool _isOn = false;

        // Свойство: включен или выключен свет
        [Category("Appearance")] // свойство - Внешний вид
        [DefaultValue(false)] // Сообщаем дизайнеру значение по умолчанию
        Ссылок: 2
        public bool IsOn
        {
            get => _isOn;
            set
            {
                _isOn = value;
                Invalidate(); // Важно! Говорим системе перерисовать элемент
            }
        }
```

```
Ссылок: 1
public LedIndicator()
{
    // Включаем двойную буферизацию, чтобы не было мерцания
    this.DoubleBuffered = true;
    this.Size = new Size(50, 50); // Размер по умолчанию
}
```

```
// Главный метод отрисовки
Ссылок: 0
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics g = e.Graphics;
    // Включаем сглаживание, чтобы круг был ровным, а не "лесенкой"
    g.SmoothingMode = SmoothingMode.AntiAlias;
    // Определяем цвет: если включен — красный, если нет — темно-серый
    Color ledColor = _isOn ? Color.Red : Color.DimGray;
    // Рисуем тело индикатора (заливка круга)
    using (Brush brush = new SolidBrush(ledColor))
    {
        // Рисуем круг, вписанный в границы контрола (с небольшим отступом)
        g.FillEllipse(brush, 2, 2, Width - 5, Height - 5);
    }
    // Рисуем контур (черная рамка)
    using (Pen pen = new Pen(Color.Black, 2))
    {
        g.DrawEllipse(pen, 2, 2, Width - 5, Height - 5);
    }
}
```


Файл
Правка
Вид
Git
Проект
Сборка
Отладка
Формат
Тест
Debug
Панель элементов
Поиск по панели элеме
classcontrol
LedIndicator (classco...
Все формы Windows Forms
Указатель
BackgroundWorker
BindingSource
Button
CheckBox
CheckedListBox
ColorDialog
LedIndicator.cs*
Form1.cs [Конструктор]
Form1

Разбор кода по шагам:

1. Сглаживание (Anti-aliasing)

Без строки:

g.SmoothingMode = SmoothingMode.AntiAlias;

края вашего круга будут состоять из грубых пикселей ("квадратиков"). Это критически важно для любого Custom Control.

2. Динамика через Invalidate()

В свойстве IsOn мы вызываем Invalidate().

Если этого не сделать, вы измените значение переменной в памяти, но на экране индикатор останется прежним.

Invalidate() посылает сигнал: "Этот элемент устарел, вызови для него OnPaint как можно скорее".

3. Использование using

Объекты Brush и Pen — это ресурсы GDI+, которые напрямую обращаются к графическому движку Windows. Если их не удалять (через using), память приложения будет быстро забиваться.

4. Координаты

В GDI+ точка (0, 0) — это верхний левый угол вашего контрола, а не формы. Поэтому мы рисуем относительно Width и Height самого элемента.

4. Интеграция с Visual Studio Designer

Когда вы перетаскиваете свой элемент на форму, Visual Studio «заглядывает» внутрь вашего кода и составляет список всех доступных настроек (цвета, текст, размеры), чтобы показать их в окне Properties (Свойства).

Чтобы этот список не превратился в свалку, вы используете атрибуты — это специальные пометки в квадратных скобках.

Они служат инструкцией для Visual Studio: как правильно подписать настройку, в какую папку её положить и нужно ли её вообще показывать.

Основные атрибуты свойств

Атрибуты пишутся в квадратных скобках прямо над свойством. Они импортируются из пространства имен System.ComponentModel.

- [Category("Имя группы")]
  - Определяет, в какой вкладке окна свойств появится настройка.
  - Стандартные категории: Appearance (внешний вид), Behavior (поведение), Data (данные), Layout (расположение).
- [Description("Текст подсказки")]
  - Задает описание, которое отображается в нижней части окна Properties при выборе свойства. Это «мини-документация» для разработчика.
- [DefaultValue(true)]
  - Указывает значение по умолчанию. Если текущее значение свойства совпадает с дефолтным, оно отображается в окне свойств обычным шрифтом. Если значения различаются — жирным.
- [Browsable(false)]
  - Если установить false, свойство скроется из окна Properties, но останется доступным в коде. Полезно для технических свойств.

Сериализация свойств

Сериализация — это процесс сохранения настроек, которые вы выставили в дизайнере, в код (файл Form1.Designer.cs).

- [DesignerSerializationVisibility]: управляет тем, как значение сохраняется.
  - Visible: (по умолчанию) записывает значение в код.
  - Hidden: не сохраняет значение. Используется для свойств, которые вычисляются динамически или могут вызвать ошибки при попытке дизайнера их «запомнить».

Двойная буферизация (Double Buffering)

Одна из важнейших настроек при создании своих элементов — борьба с мерцанием (flickering).

- Проблема: По умолчанию WinForms сначала стирает фон элемента, а затем рисует на нем фигуры. При частой перерисовке глаз успевает заметить этот «пустой» момент, и элемент неприятно дрожит.
- Решение: Свойство DoubleBuffered = true.
  - При включении этого режима вся отрисовка происходит сначала в памяти (на невидимом холсте), а затем готовый результат мгновенно копируется на экран.

```
public MyControl()
{
    // Включаем в конструкторе для плавной графики
    this.DoubleBuffered = true;

    // Также можно использовать SetStyle для более тонкой настройки
    this.SetStyle(ControlStyles.AllPaintingInWmPaint |
                  ControlStyles.UserPaint |
                  ControlStyles.OptimizedDoubleBuffer, true);
}
```

1. ControlStyles.UserPaint

Что делает: Говорит Windows: «Не рисуй этот элемент сам, я всё нарисую вручную в методе OnPaint».

Зачем это нужно: Если это не включить, операционная система может попытаться наложить стандартный серый фон поверх ваших рисунков, что приведет к конфликтам и мерцанию.

2. ControlStyles.AllPaintingInWmPaint

Что делает: Запрещает операционной системе рисовать фон отдельным сообщением (WM_ERASEBKGND). Все рисование (и фон, и сам элемент) происходит за один присест в одном методе.

Зачем это нужно: Это главный убийца мерцания. Обычно мерцание возникает из-за того, что Windows сначала быстро заливает элемент белым цветом (очистка фона), а потом вы рисуете сверху. Глаз успевает заметить эту белую вспышку. Этот флаг убирает этот этап.

3. ControlStyles.OptimizedDoubleBuffer

Что делает: Включает продвинутую буферизацию. Элемент сначала рисуется на невидимой картинке в памяти (буфере), и только когда рисунок полностью готов, он мгновенно «шлепается» на экран.

Зачем это нужно: Чтобы пользователь не видел, как вы по очереди рисуете сначала круг, потом текст, потом рамку. Все появляется одновременно.

5. Обработка событий и интерактивность.

Здесь мы учим «безжизненную» картинку, которую нарисовали в OnPaint, реагировать на действия пользователя и общаться с внешним миром (формой).

В классе Control уже заложены механизмы для работы с устройствами ввода.

Нам нужно лишь «подключиться» к ним.

Перехват событий мыши

Вместо того чтобы подписываться на события самого себя (например, this.MouseDown += ...), внутри класса контрола принято переопределять защищенные методы.

Основные методы для переопределения:

- OnMouseEnter(EventArgs e): Мышь вошла в область контрола. Идеально для «подсветки» элемента.
- OnMouseLeave(EventArgs e): Мышь покинула область. Возвращаем обычный вид.
- OnMouseDown(MouseEventArgs e): Нажата кнопка мыши. Через e.Button можно узнать, какая именно (левая/правая).
- OnMouseMove(MouseEventArgs e): Мышь движется над контролом. e.Location дает точные координаты курсора.

Важное правило: Всегда вызывайте base.OnНазваниеМетода(e), иначе внешние пользователи вашего контрола не смогут поймать эти события на форме.

```
protected override void OnMouseEnter(EventArgs e)
{
    base.OnMouseEnter(e);
    _isHovered = true;
    Invalidate(); // Перерисовываем, чтобы показать эффект наведения
}
```

Создание собственных событий

Форма не должна лезть внутрь логики вашего контрола. Если в контроле что-то изменилось (например, наш светодиод переключился), контрол должен сам оповестить об этом форму через событие.

Алгоритм создания события:

1. Объявление события: Используем стандартный делегат EventHandler.

```
[Category("Behavior")]
[Description("Событие возникает при изменении состояния индикатора")]
public event EventHandler StateChanged;
```

2. Вызов события (Метод-инициатор):

Создаем защищенный метод, который проверяет, есть ли подписчики, и запускает событие.

```
protected virtual void OnStatusChanged()
{
    // ?.Invoke проверяет, подписался ли кто-то на событие на форме
    StatusChanged?.Invoke(this, EventArgs.Empty);
}
```

3. Триггер: Вызываем этот метод там, где меняются данные.

```
public bool IsOn
{
    get => _isOn;
    set
    {
        _isOn = value;
        OnStatusChanged(); // Сообщаем миру об изменениях
        Invalidate();
    }
}
```

```
public class InteractiveButton : Control
{
    private Color _currentColor = Color.Gray;
    public event EventHandler StatusChanged;
    Ссылок: 0
    protected override void OnMouseEnter(EventArgs e)
    {
        _currentColor = Color.LightGray;
        Invalidate();
    }
    Ссылок: 0
    protected override void OnMouseLeave(EventArgs e)
    {
        _currentColor = Color.Gray;
        Invalidate();
    }
    Ссылок: 0
    protected override void OnMouseDown(MouseEventArgs e)
    {
        StatusChanged?.Invoke(this, EventArgs.Empty);
        base.OnMouseDown(e);
    }
    Ссылок: 0
    protected override void OnPaint(PaintEventArgs e)
    {
        e.Graphics.FillRectangle(new SolidBrush(_currentColor), ClientRectangle);
    }
}
```

Схема взаимодействия:

- Форма видит только событие StateChanged.
- Контрол сам решает, когда это событие "выстрелит".
- В итоге, если вы решите изменить форму индикатора с круга на квадрат, код на форме Form1 не изменится ни на одну букву.

Контрольные вопросы:

- В чем ключевое различие между UserControl и Custom Control (наследование от Control)?
- От какого базового класса нужно наследоваться, чтобы создать текстовое поле с измененным поведением?
- Что такое «Дескриптор окна» (Handle) и почему он важен для класса Control?
- Какое пространство имен (namespace) необходимо подключить, чтобы использовать классы Control, TextBox или UserControl?
- В чем разница между инструментами Pen (Перо) и Brush (Кисть) в GDI+?
- Для чего используется свойство e.ClipRectangle в аргументах метода OnPaint?
- Что такое «Сглаживание» (Anti-aliasing) и с помощью какого свойства объекта Graphics оно включается?
- Метод Invalidate() и метод Refresh(): какой из них предпочтительнее для производительности и почему?
- Зачем включать DoubleBuffered = true в конструкторе пользовательского элемента?
- Как работает флаг ControlStyles.AllPaintingInWmPaint и какую проблему он решает?
- Для чего используется атрибут [Category("Appearance")] над свойством класса?
- Что делает атрибут [DesignerSerializationVisibility(DesignerSerializationVisibility.Hidden)] и в каких случаях он необходим?
- Как сделать так, чтобы свойство было доступно в коде, но не отображалось в окне Properties в Visual Studio?
- Как «пробросить» событие клика внутренней кнопки UserControl наружу, чтобы его могла обработать родительская форма?
- Зачем в переопределенных методах (например, OnMouseDown) первым делом вызывать base.OnMouseDown(e)?

Список литературы:

1. https://metanit.com/sharp/windowsforms/1.3.php

Материалы лекций:

https://github.com/ShViktor72/Education2025