УП МДК 02

Практическая работа 1

1. Репозиторий
2. Знакомство с WPF

Теория:

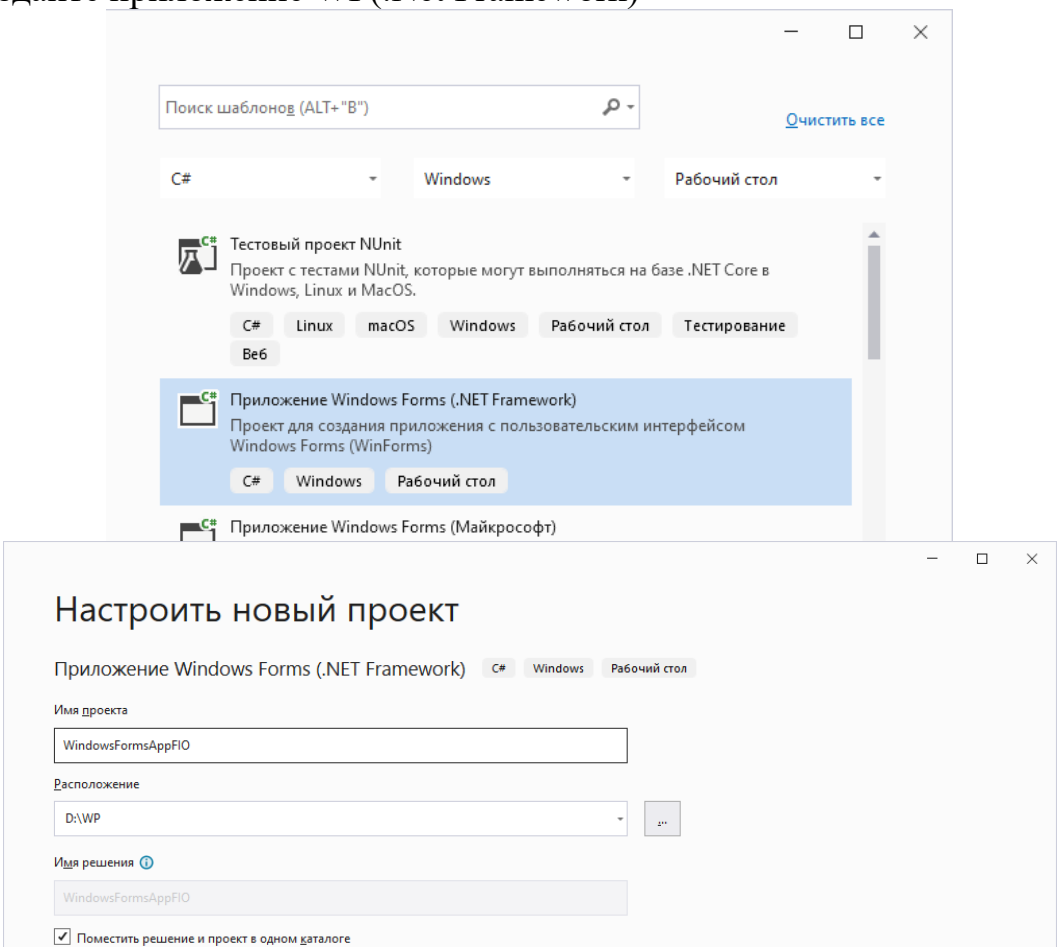
1. **Знакомство с репозиторием.**

https://smartiqa.ru/courses/git/lesson-1

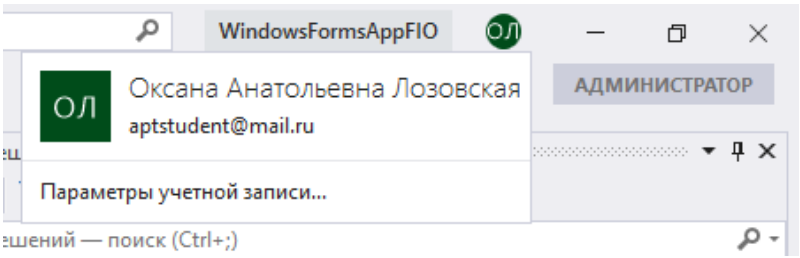
Задание 1. Создание репозитория

Создайте учетную запись на Github https://github.com/ . Все последующие практические работы нужно будет сохранять в этой учётной записи, в других репозиториях.

Создайте приложение WF(.Net Framework)

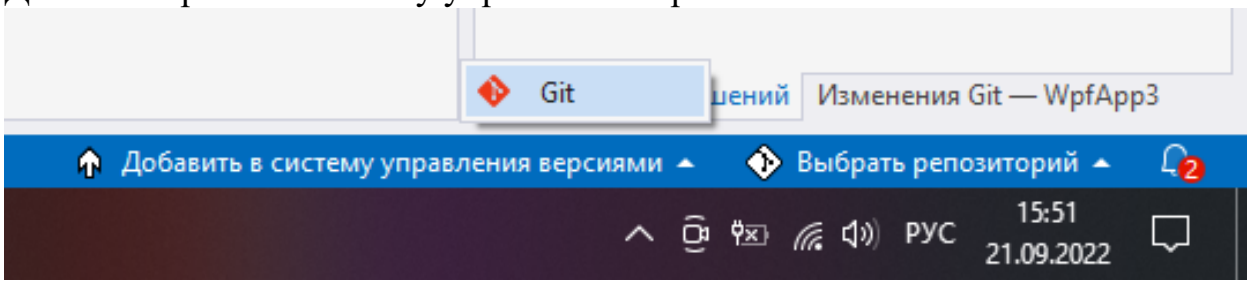


Выполните вход в **СВОЮ** учетную запись. Если ее нет – создайте и авторизируйтесь.

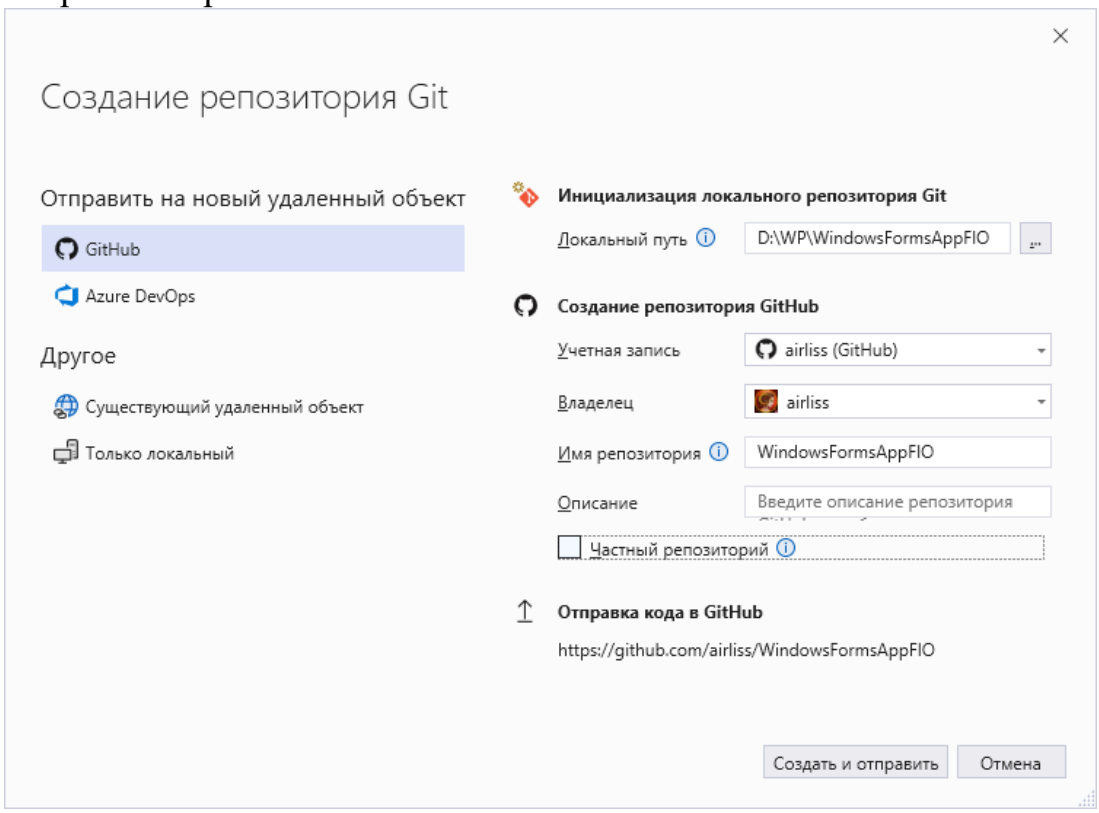


УП МДК 02

Добавьте проект в систему управления версиями - GIT

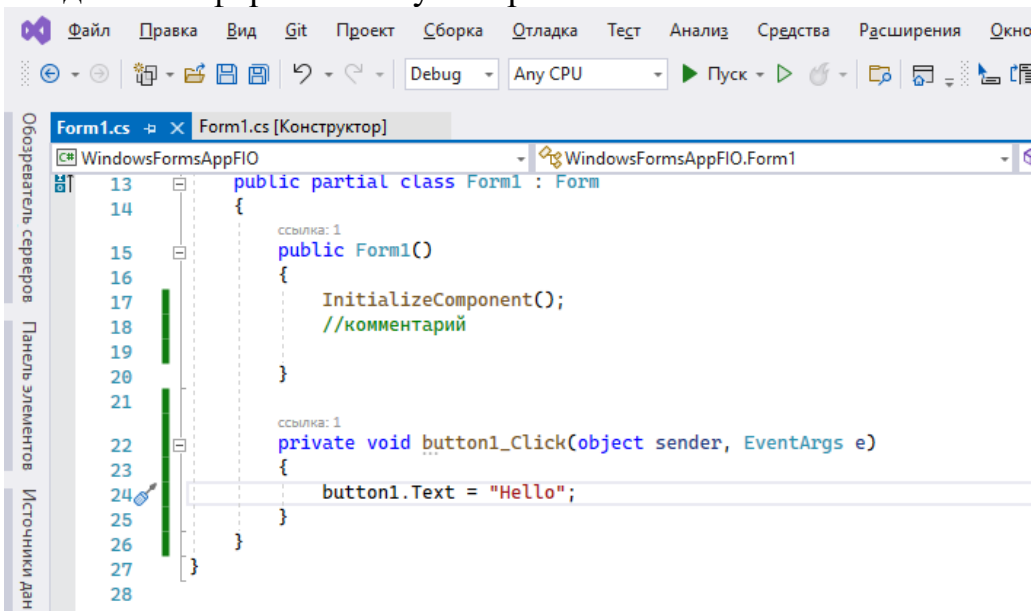


Создайте репозиторий в своей учетной записи. Уберите галочку частного репозитория

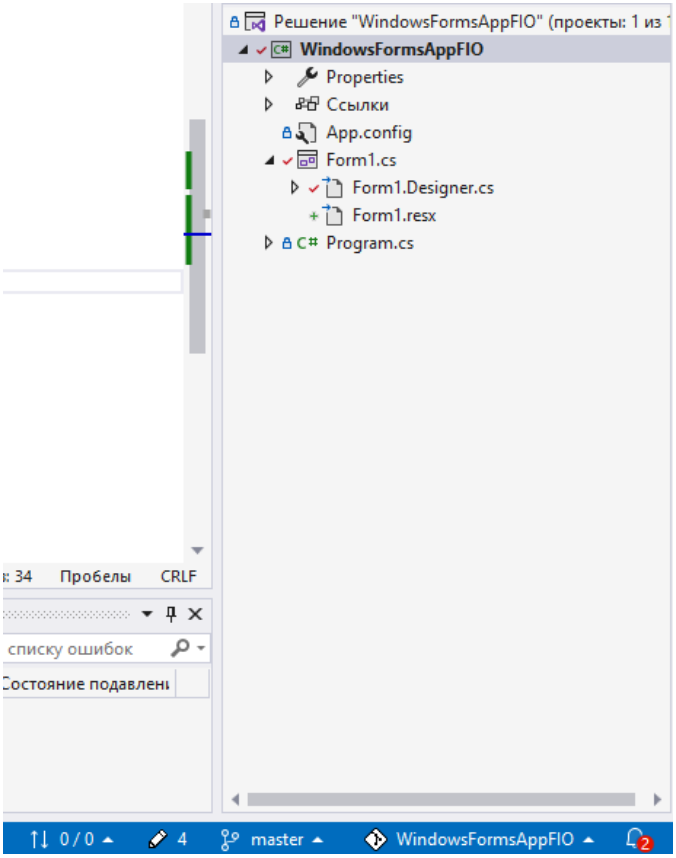


Создать и отправить

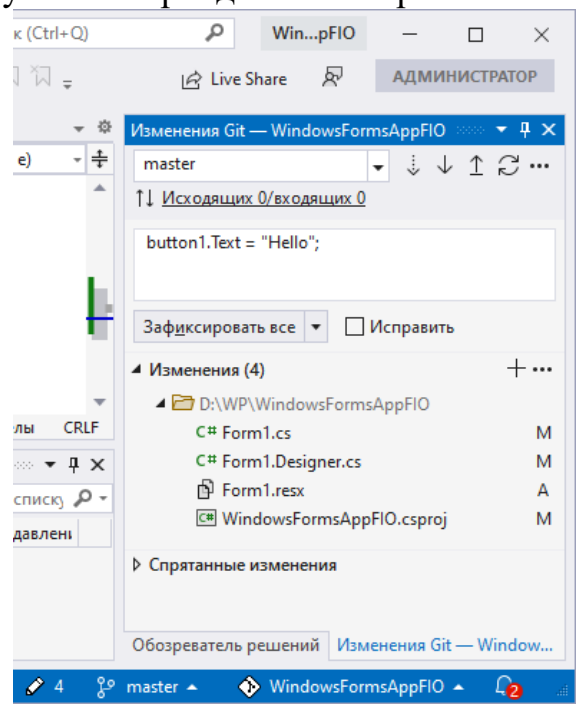
Создайте на форме кнопку и обработчик к ней.



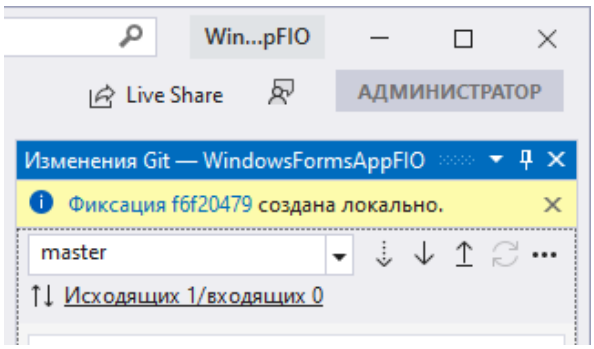
Обратите внимание на статус изменений в системе контроля версий



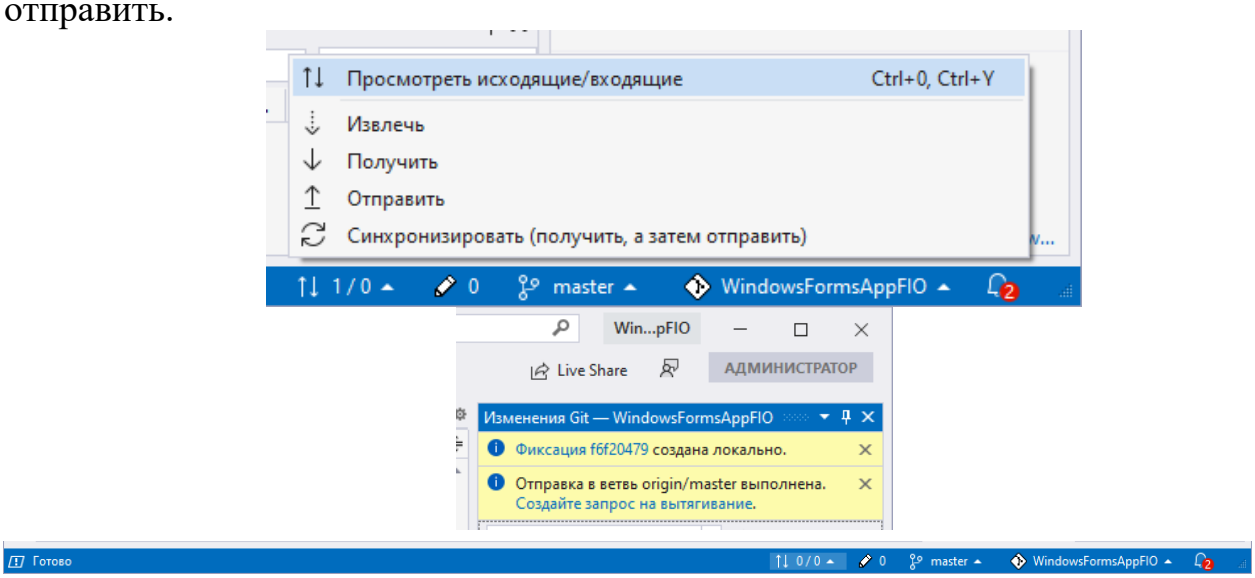
Нажмите на цифру возле карандаша. Отобразится список изменений.



Напишите комментарий к изменениям. Зафиксируйте их все. На данный момент изменения произошли локально.

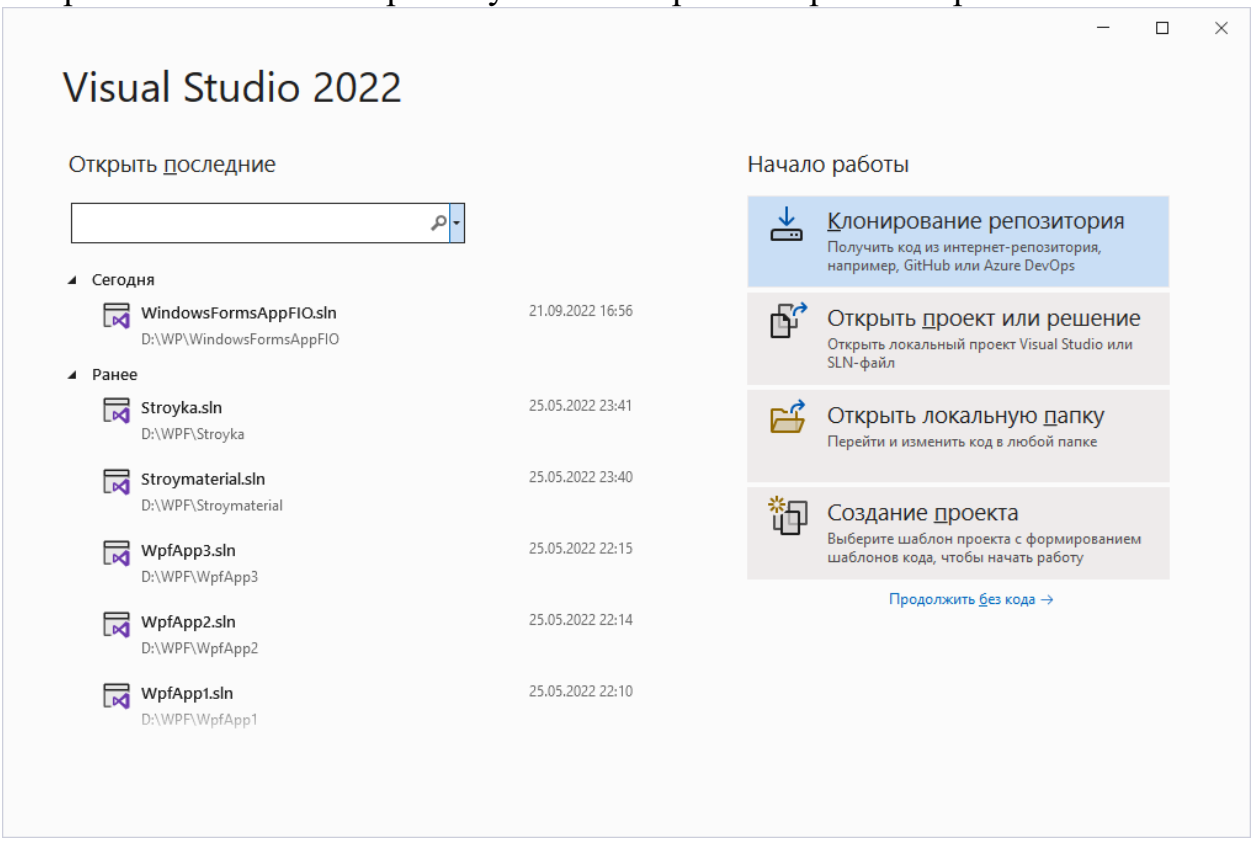


Что бы они произошли и в системе контроля версий(GIT), следует их отправить.



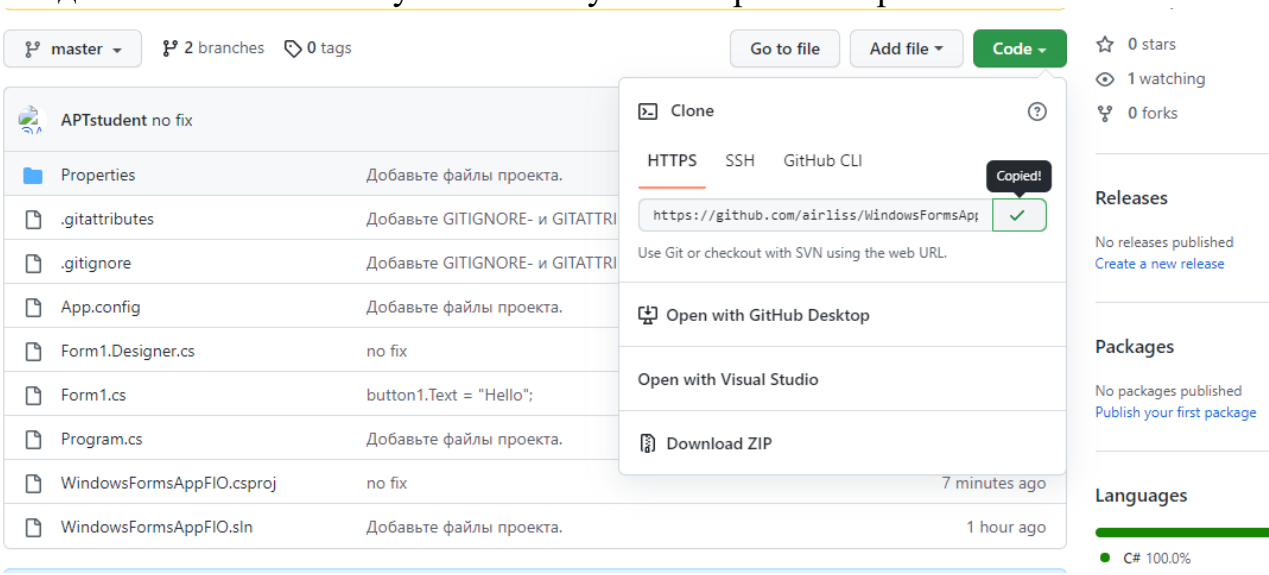
Закройте Visual Studio

Откройте заново и выберите пункт Клонирование репозитория:

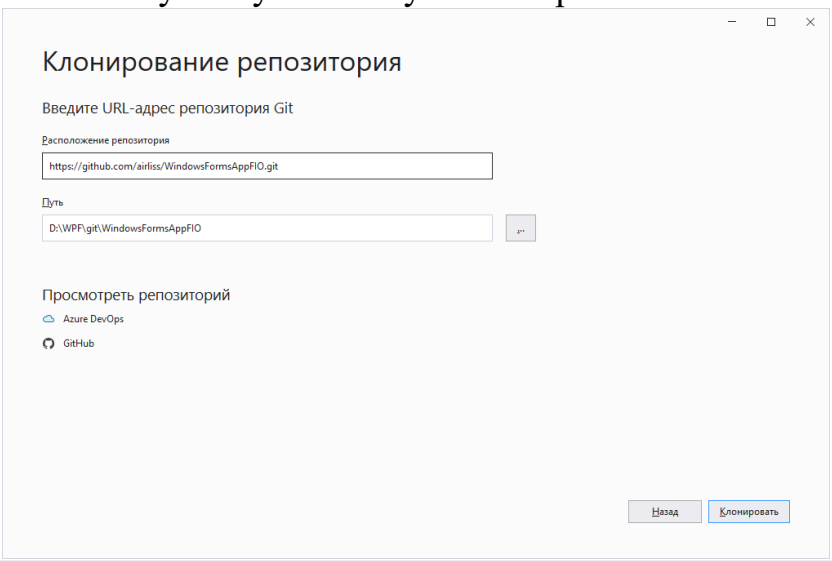


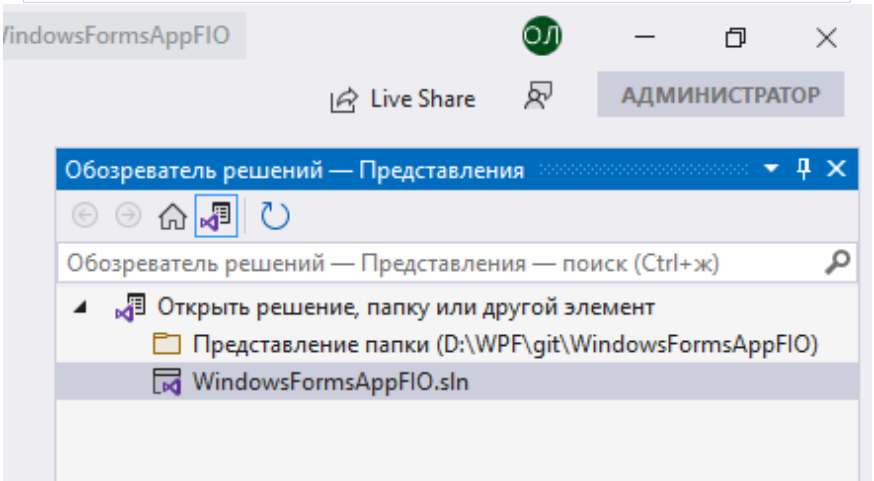
УП МДК 02

Зайдите на GitHub и получите ссылку на ваш репозиторий

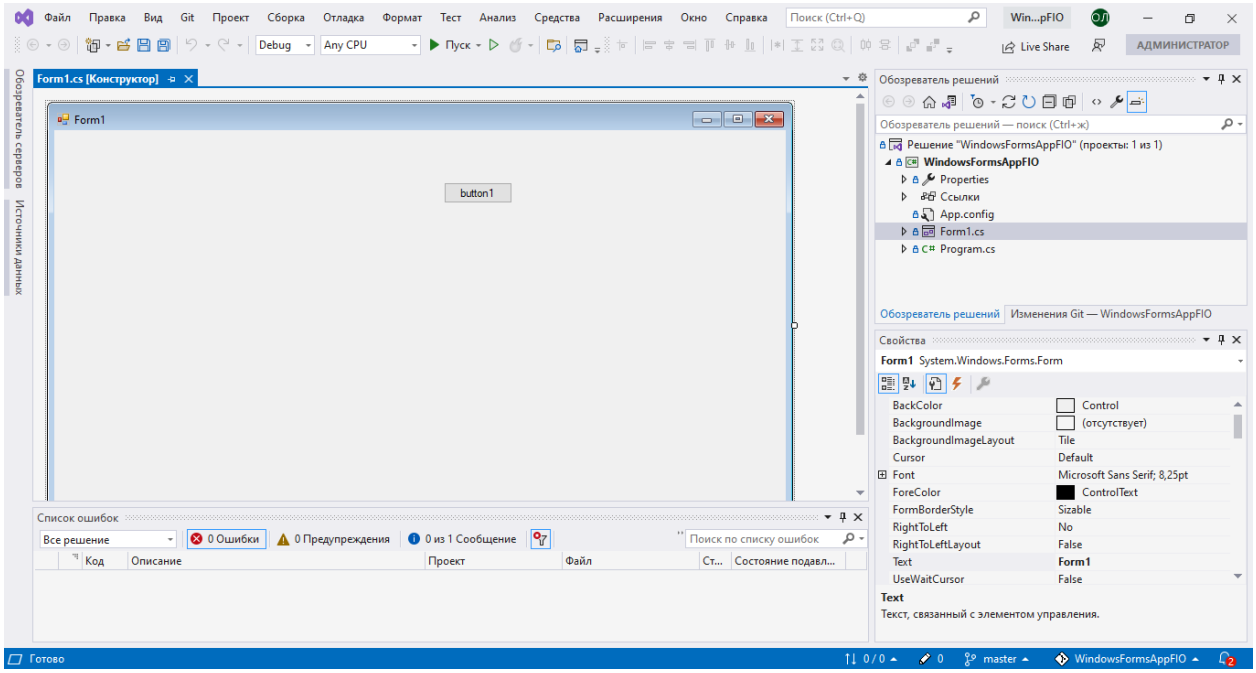


В окне вставьте полученную ссылку - Клонировать





Разверните ваш проект и проверьте что там находятся все ваши данные.



Поздравляю! Вы научились в самом простом виде работать с GitHub.

Далее, для сдачи ваших работ вы предоставляете Отчет, в котором расположены:

- Титульный лист
- Экранные формы
- Ссылка на GitHub с проектом

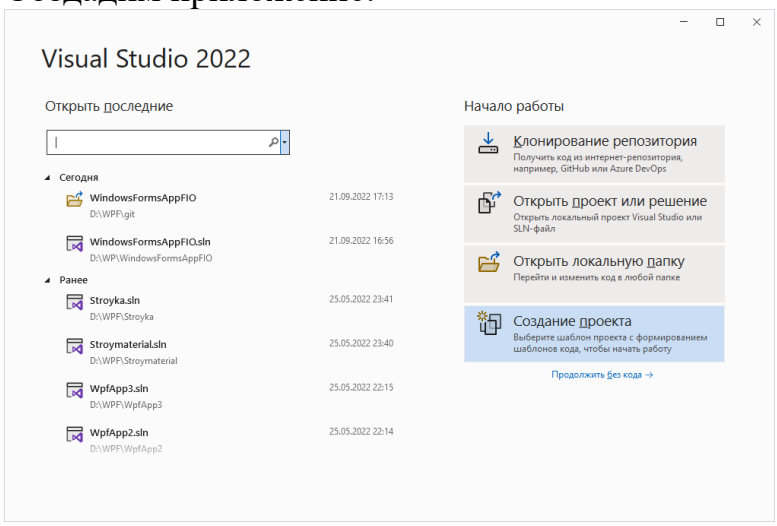
Все изменения в вашем проекте должны сохранятся на сервер не реже чем 1 раз в 15 минут!

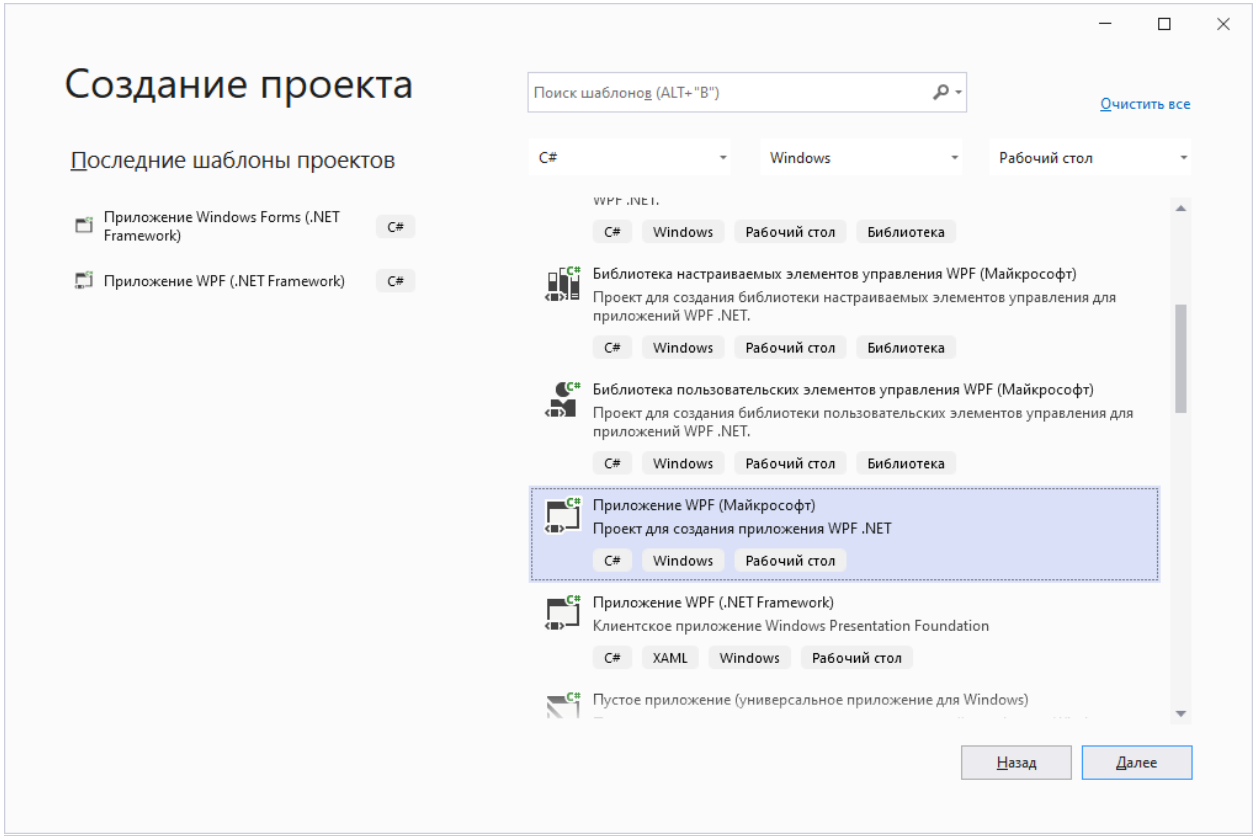
Отчет так же сохраняется на сервер GitHub

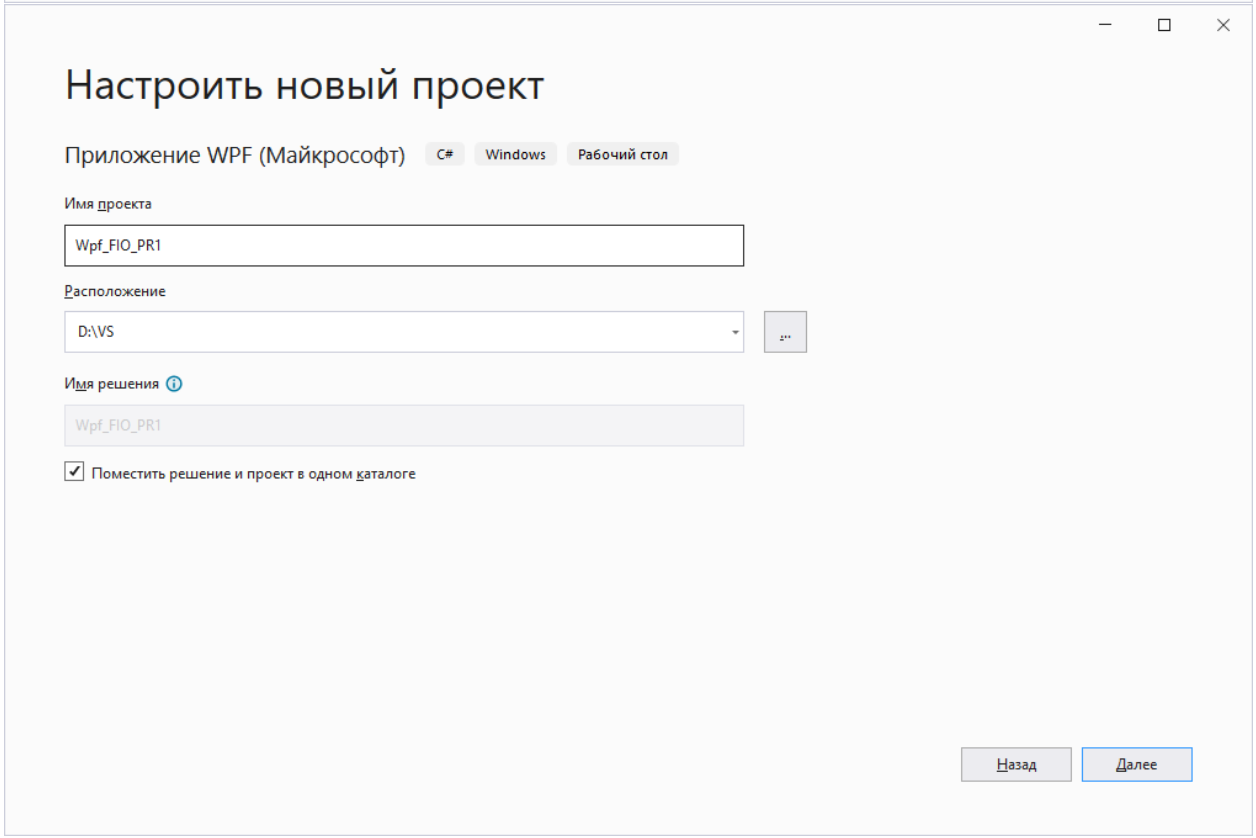
Задание 2.

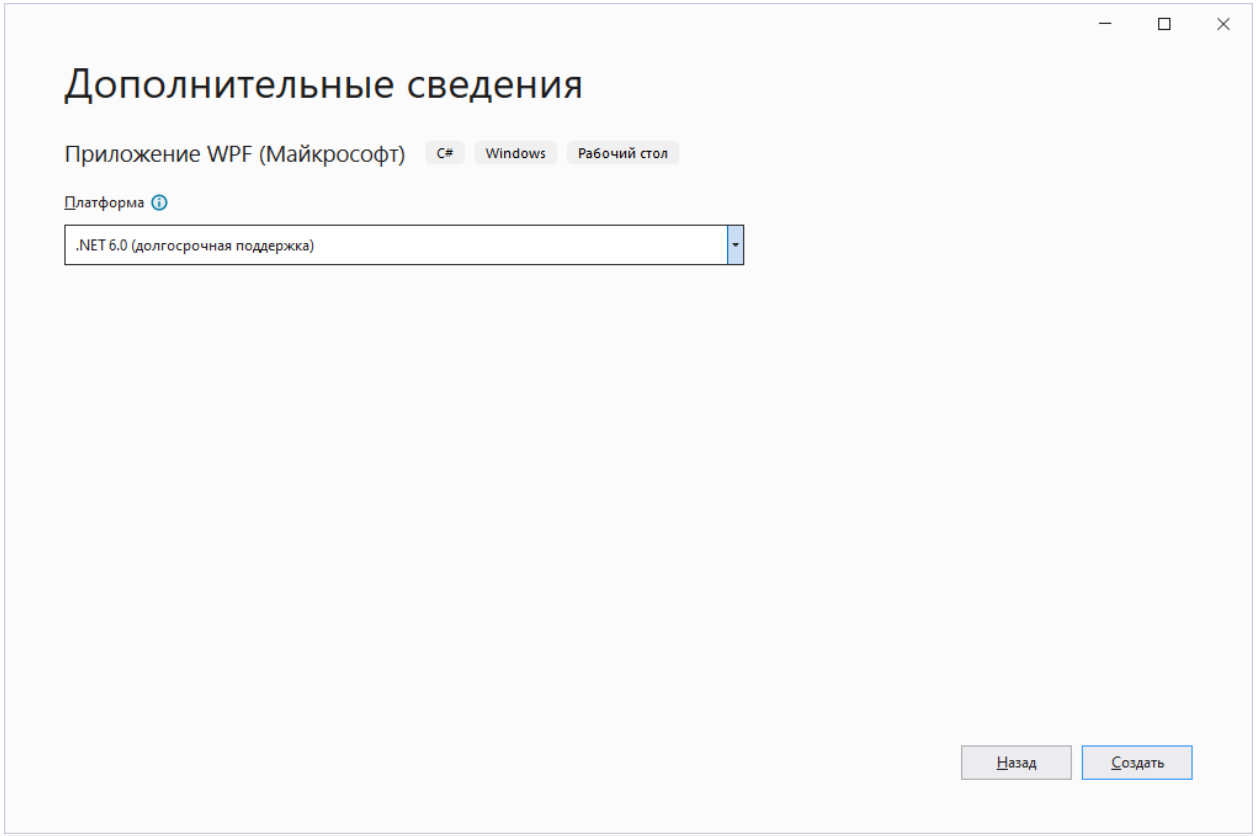
Знакомство с WPF.

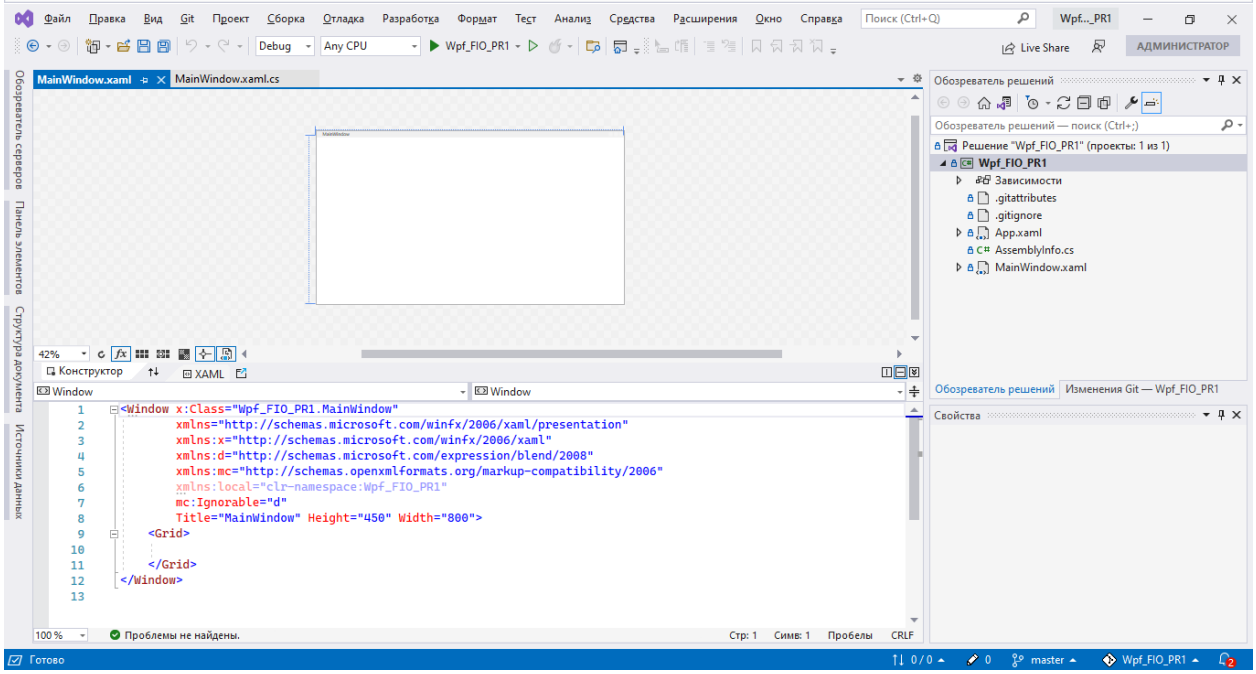
Создадим приложение:




Создание проекта
Последние шаблоны проектов
Приложение Windows Forms (.NET Framework)
C#
Приложение WPF (.NET Framework)
C#
Поиск шаблонов (ALT+"B")
Очистить все
C#
Windows
Рабочий стол
C# Windows Рабочий стол Библиотека
Библиотека настраиваемых элементов управления WPF (Майкрософт)
Проект для создания библиотеки настраиваемых элементов управления для приложений WPF .NET.
C# Windows Рабочий стол Библиотека
Библиотека пользовательских элементов управления WPF (Майкрософт)
Проект для создания библиотеки пользовательских элементов управления для приложений WPF .NET.
C# Windows Рабочий стол Библиотека
Приложение WPF (Майкрософт)
Проект для создания приложения WPF .NET
C# Windows Рабочий стол
Приложение WPF (.NET Framework)
Клиентское приложение Windows Presentation Foundation
C# XAML Windows Рабочий стол
Пустое приложение (универсальное приложение для Windows)
Назад
Далее


Настроить новый проект
Приложение WPF (Майкрософт)
C#
Windows
Рабочий стол
Имя проекта
Wpf_FIO_PR1
Расположение
D:\VS
...
Имя решения
Wpf_FIO_PR1
Поместить решение и проект в одном каталоге
Назад
Далее





Структура проекта

Справа находится окно Solution Explorer, в котором можно увидеть структуру нашего проекта. В данном случае у нас сгенерированная по умолчанию структура:

- **Dependencies/Зависимости** - это узел содержит сборки dll, которые добавлены в проект по умолчанию. Эти сборки как раз содержат классы библиотеки .NET, которые будет использовать C#
- **App.xaml** задает ресурсы приложения и ряд конфигурационных настроек в виде кода XAML. В частности, в файле **App.xaml** задается файл окна программы, которое будет открываться при запуске приложения. Если вы

откроете этот файл, то можете найти в нем строку **StartupUri="MainWindow.xaml"** - то есть в данном случае, когда мы запустим приложение, будет создаваться интерфейс из файла **MainWindow.xaml**.

App.xaml.cs - это файл кода на C#, связанный с файл **App.xaml**, который также позволяет задать ряд общих ресурсов и общую логику для приложения, но в вид кода C#.

- **AssemblyInfo.cs** содержит информацию о создаваемой в процессе компиляции сборке
- **MainWindow.xaml** представляет визуальный интерфейс окна приложения в виде кода XAML.

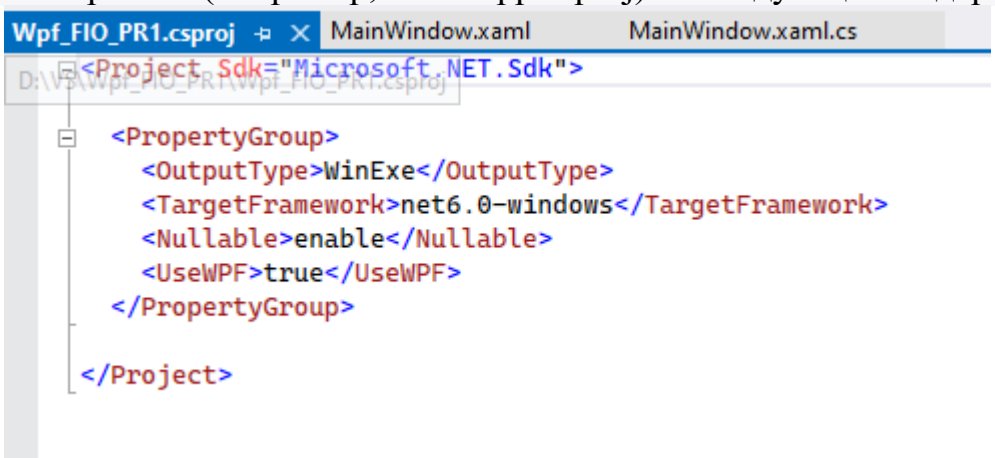
MainWindow.xaml.cs - это файл логики кода на C#, связанный с окном **MainWindow.xaml**.

По умолчанию эти файлы открыти в текстовом редакторе Visual Studio. Причем файл **MainWindow.xaml** имеет два представления: визуальное - в режиме WYSIWIG отображает весь графический интерфейс данного окна приложения, и под ним декларативное объявление интерфейса в XAML. Если мы изменим декларативную разметку, например, определим там кнопку, то эти изменения отображаться в визуальном представлении. Таким образом, мы сможем сразу же получить представление об интерфейсе окна приложения.

Настройка компиляции проекта

Кроме того, проект WPF имеет еще один важный файл, как и все проекты на языке C# - файл конфигурации проекта. Для его открытия нажмем двойным кликом левой кнопкой мыши на название проекта или нажмем правой кнопкой мыши на название проекта и появившемся контекстном меню выберем пункт **Edit Project File/Изменить файл проекта**

В итоге нам откроектся файл с расширением **csproj**, который называется по имени проекта (например, HelloApp.csproj) со следующим содержимым:



Для компиляции приложения WPF указаны следующие настройки:

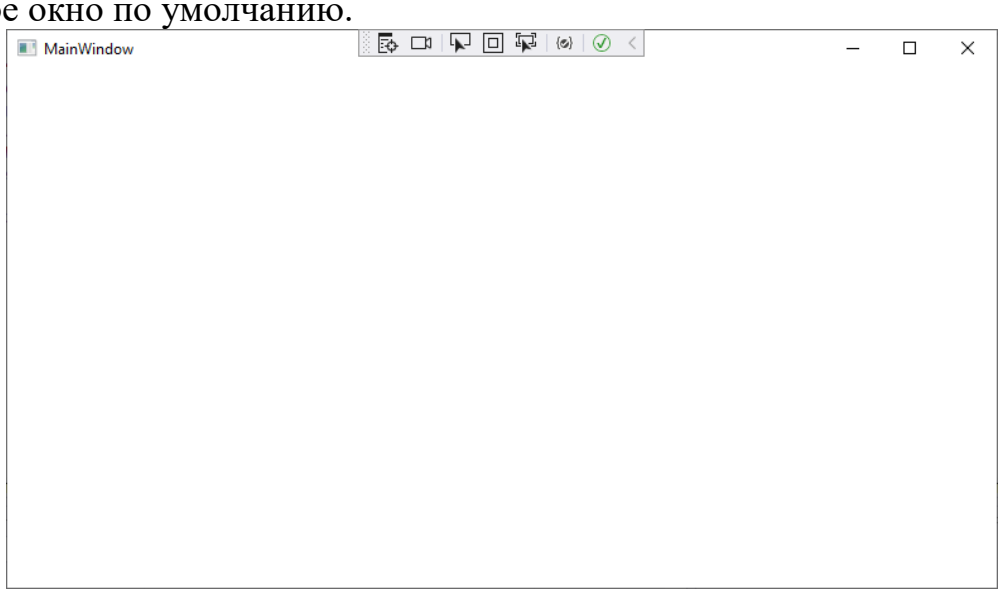
- **OutputType**: определяет выходной тип проекта. Должен иметь значение **WinExe** - то есть выполняемое приложение с расширением exe под Windows
- **TargetFramework**: определяет применяемую для компиляции версию фреймворка .NET. Поскольку при создании проекта была выбрана

версия .NET 6, а сам проект зависит от компонентов Windows, то здесь должно быть значение **net6.0-windows**

- **Nullable**: подключает в проект функциональность ссылочных nullable-типов
- **UseWPF**: указывает, будет ли проект использовать WPF. (при значении true проект использует WPF)

Запуск проекта

Чтобы запустить приложение в режиме отладки, нажмем на клавишу F5 или на зеленую стрелочку на панели Visual Studio. И после этого запустится пустое окно по умолчанию.

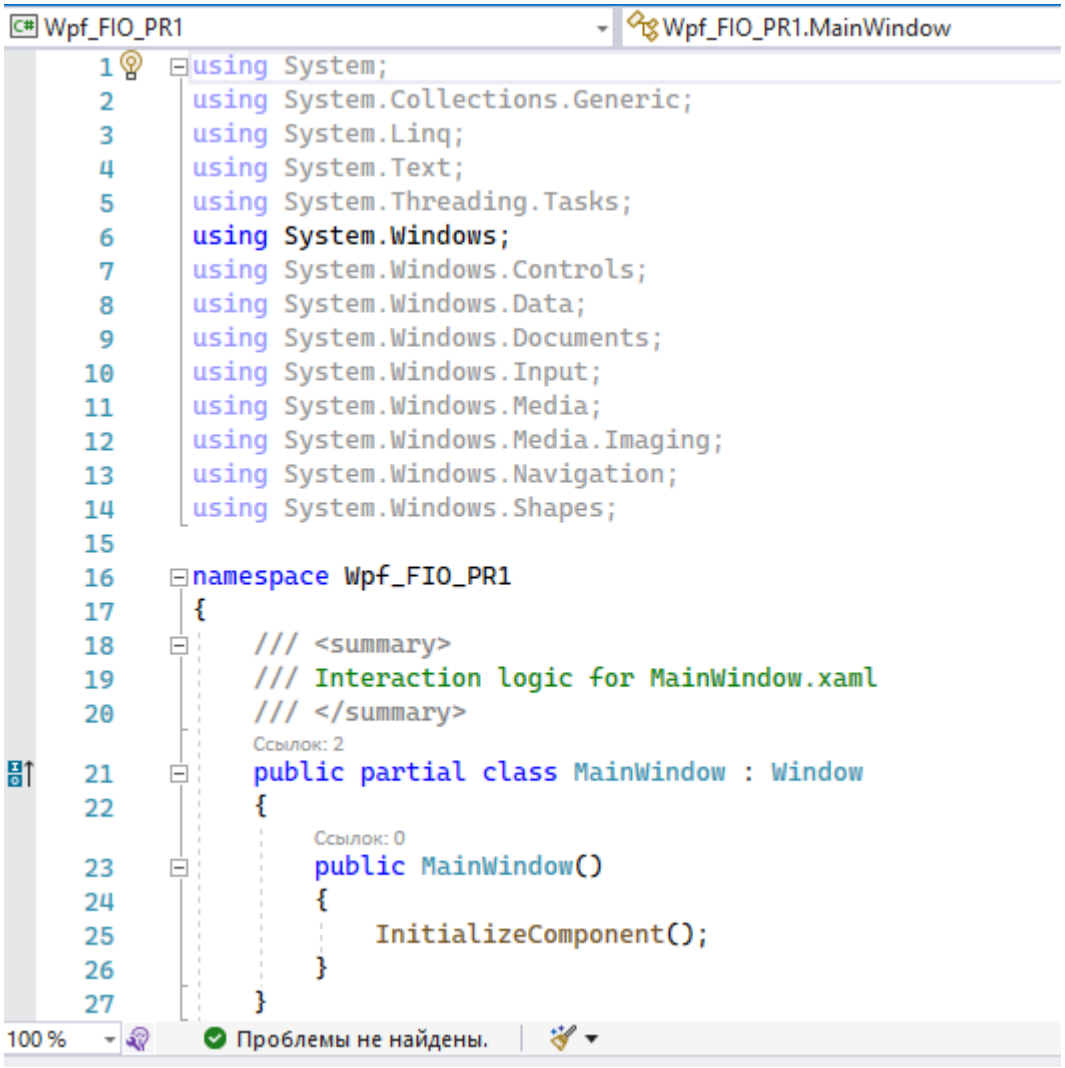


После запуска приложения студия компилирует его в файл с расширением exe. Найти данный файл можно, зайдя в папку проекта и далее в каталог \bin\Debug\net6.0-windows

Рассмотрев вкратце создание проекта графического приложения, мы можем перейти к обзору основных компонентов и начнем мы с форм.

Создание первого приложения

Однако приложение с пустым окном - не слишком показательный пример. Добавим в него чуть больше функционала. Для этого откроем файл логики кода окна приложения - MainWindow.xaml.cs. Сейчас он имеет следующий код:



Здесь определен класс MainWindow, который наследуется от класса Window и берет от него всю базовую функциональность окон. А в конструкторе этого класса вызывается метод InitializeComponent(), который позволяет применить интерфейс из файла **MainWindow.xaml**.

Теперь изменим файл **MainWindow.xaml.cs** следующим образом:

```
namespace Wpf_FIO_PR1
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    Ссылок: 2
    public partial class MainWindow : Window
    {
        Ссылок: 0
        public MainWindow()
        {
            InitializeComponent();
        }

        Ссылок: 0
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Привет, группа ИСП!");
        }
    }
}
```

Здесь добавлен метод Button_Click, который будет выступать в качестве обработчика нажатия кнопки. Обработчики нажатия кнопки должны принимать два параметра типов object и RoutedEventArgs. В самом обработчике вызывается метод **MessageBox.Show**, который отображает окно с сообщением. Отображаемое сообщение передается в качестве параметра.

Теперь определим саму кнопку. Для этого перейдем к файлу **MainWindow.xaml**, который содержит разметку визуального интерфейса в виде кода XAML. По умолчанию он имеете следующее содержимое:

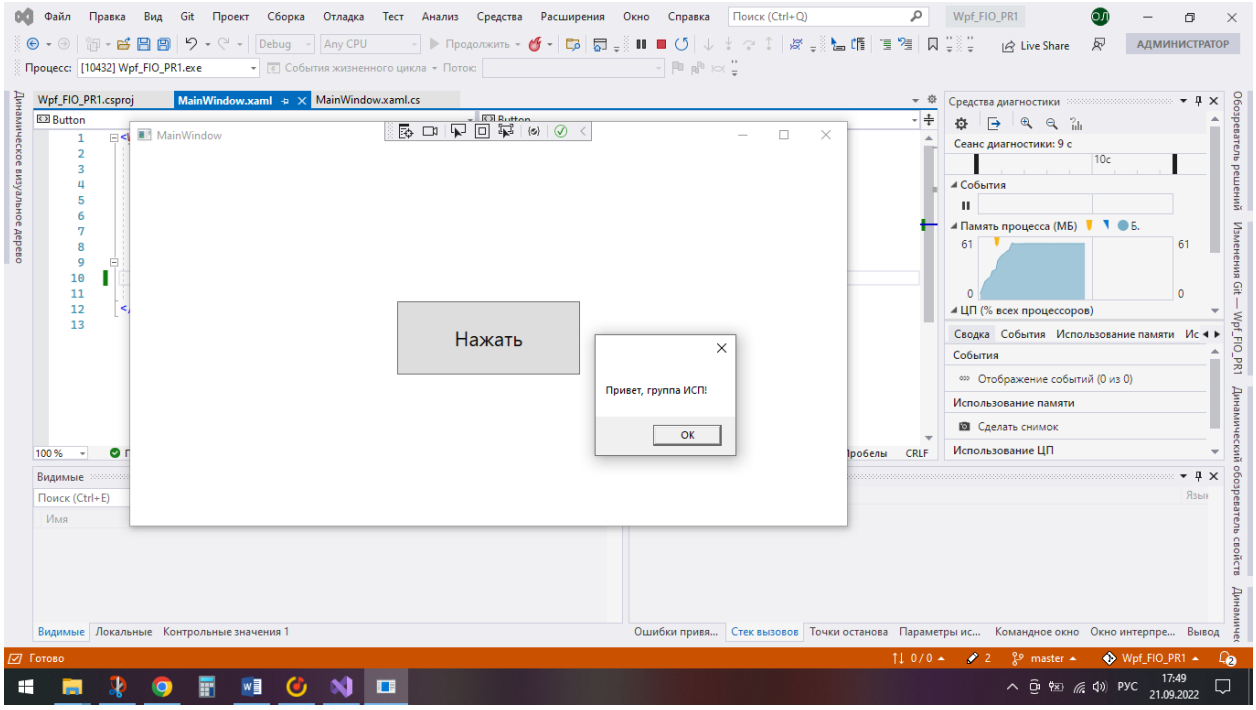
XAML в целом напоминает язык разметки HTML: здесь у нас сначала определен элемент верхнего уровня Window - окно приложения, в нем определен элемент Grid - контейнер верхнего уровня, в который мы можем добавлять другие элементы. Каждый элемент может иметь определенные атрибуты. Более подробно с языком XAML и элементами мы познакомимся позднее, а пока изменим эту разметку на следующую:

```
8           Title="MainWindow" Height="450" Width="800">
9       <Grid>
10          <Button Content="Нажать" FontSize="22" Width="200" Height="80" Click="Button_Click" />
11      </Grid>
```

Для определения кнопки внутри элемента Grid определен элемент **Button**. Для этого элемента с помощью атрибутов можно установать различные его характеристика. Так, в данном случае устанавливаются следующие атрибуты:

- **Content**: содержимое кнопки
- **FontSize**: высота шрифта
- **Width**: ширина кнопки
- **Height**: высота кнопки
- **Click**: обработчик нажатия кнопки. Здесь подключается созданные выше в файле кода C# метод Button_Click. В итоге по нажатию на эту кнопку сработает метод Button_Click

Запустим приложение и нажмем на кнопку, и нам должно отобразиться сообщение:



Сохраните результат в репозиторий!

Разместите в коде XAML в содержимом элемента Grid следующий код:

```
<Grid>
    <!--<Button Content="Нажать" FontSize="22" Width="200" Height="80" Click="Button_Click" />-->
    <Button x:Name="Btn1"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Width="150"
            Height="30"
            FontSize="17"
            Content="Обычная кнопка"
            Foreground="#006699"
            Background="#f0f0f0"
            BorderBrush="#303030" />
</Grid>
```

Обратите внимание на то, как выполнено комментирование кода (мы спрятали туда предыдущий пример. Далее каждый предыдущий пример прячем, для выполнения нового)

Запустите приложение и проверьте его поведение при изменении размеров окна.

В приведенном выше примере для элемента Button было задано простое значение атрибута Background. Для этого был использован синтаксис

<ЭЛЕМЕНТ АТРИБУТ=”ЗНАЧЕНИЕ” />

Для задания значения атрибуты может быть использован другой синтаксис:

<ЭЛЕМЕНТ>
<ЭЛЕМЕНТ.АТРИБУТ>
ЗНАЧЕНИЕ_АТРИБУТА
</ЭЛЕМЕНТ. АТРИБУТ>
</ЭЛЕМЕНТ>

Например, для задания того же значения атрибута Background можно записать:

<Button>
<Button.Background>
#f0f0f0
</Button.Background>
</Button>

Данный синтаксис используется для задания сложных значений атрибутов в виде дерева элементов.

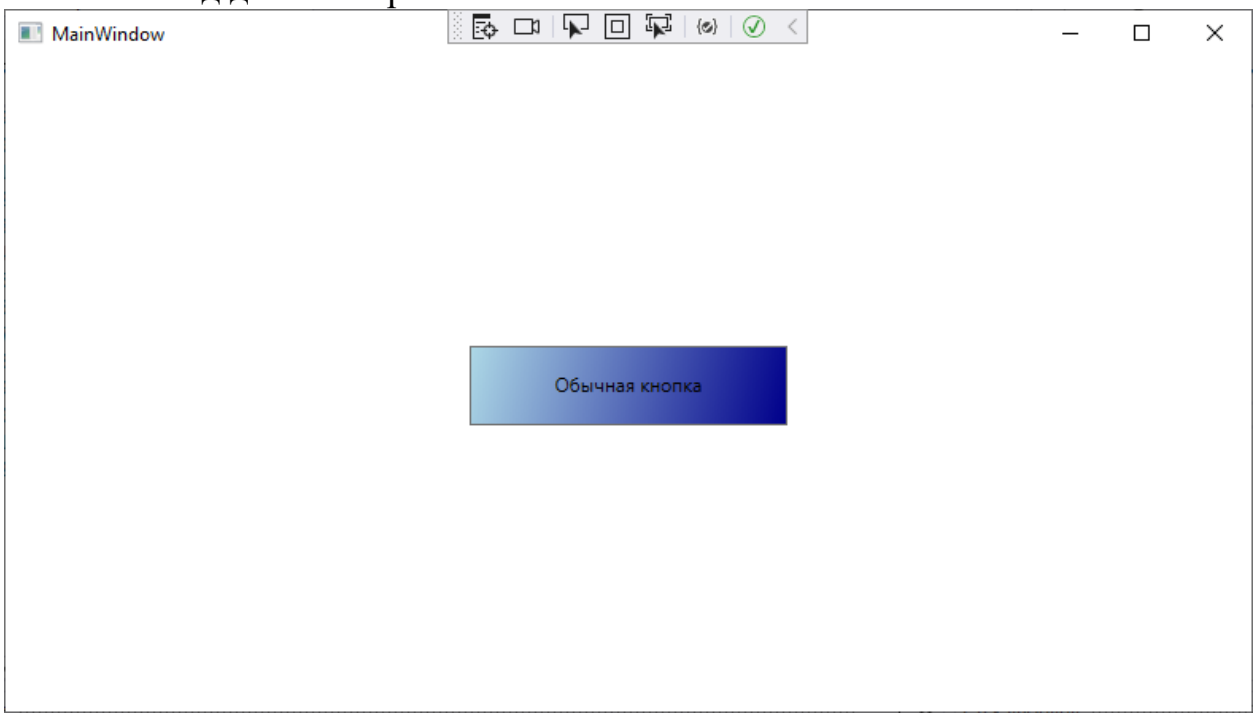
Пример задания для фона кнопки линейной градиентной заливки:

```
<Grid>
<Button x:Name="Btn1" Height="50" Width="200" Content="Обычная кнопка">
        <Button.Background>
            <LinearGradientBrush>
                <LinearGradientBrush.GradientStops>
                    <GradientStop Color="LightBlue" Offset="0" />
                    <GradientStop Color="DarkBlue" Offset="1" />
                </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
        </Button.Background>
    </Button>
</Grid>
```

Данное дерево элементов задает градиентную заливку с использованием двух цветов: LightBlue и DarkBlue. В атрибуте Offset указывается

относительное значение от 0 до 1, соответствующее положению цвета на отрезке от начальной точки до конечной.

Внешний вид данного приложения:



Объект Applicaton

Любое приложение использует класс Application, который организует его подключение к модели событий операционной системы с помощью метода Run(). Объект Application отвечает за управление временем жизни приложения, отслеживает видимые окна, освобождает ресурсы и контролирует глобальное состояние приложения. Метод Run() запускает диспетчер среды исполнения, который начинает посылать события и сообщения компонентам приложения.

В каждый момент времени может быть активен только один объект Application и он будет работать до тех пор, пока приложение не завершится. К исполняемому объекту Application можно получить доступ из любого места приложения через статическое свойство Application.Current. Одна из основных задач объекта Application состоит в том, чтобы контролировать время жизни процесса. Конструирование объекта Application знаменует начало жизни приложения, а возврат из его метода Run() - завершение приложения.

Время жизни приложения WPF и объекта Application состоит из следующих этапов:

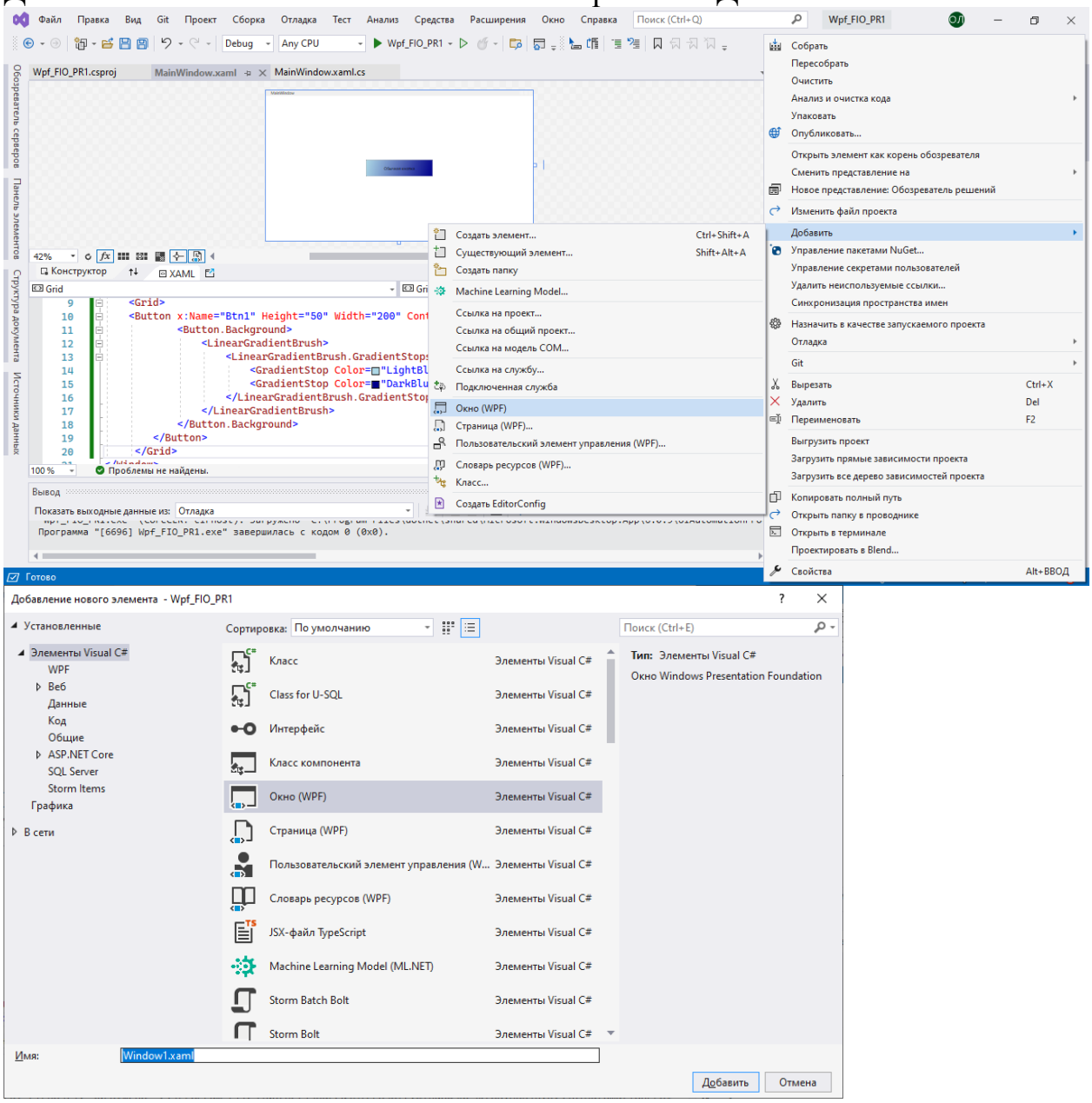
1. Конструируется объект Application
2. Вызывается его метод Run()
3. Выполняется событие Application.Startup
4. Пользовательский код конструирует один или несколько объектов Window (или Page) и приложение выполняет работу

5. Вызывается метод Application.Shutdown()
6. Вызывается метод Application.Exit()

При создании проекта среда разработки поместила в проект два файла, связанные с объектом Application: App.xaml и App.xaml.cs. В этих файлах нет кода, создающего объекты Application и Windows и вызывающего метод Run() – это происходит неявно. В файле App.xaml для элемента Window задается атрибут StartupUri, в котором определяется имя XAML-файла с окном, которое открывается при запуске приложения.

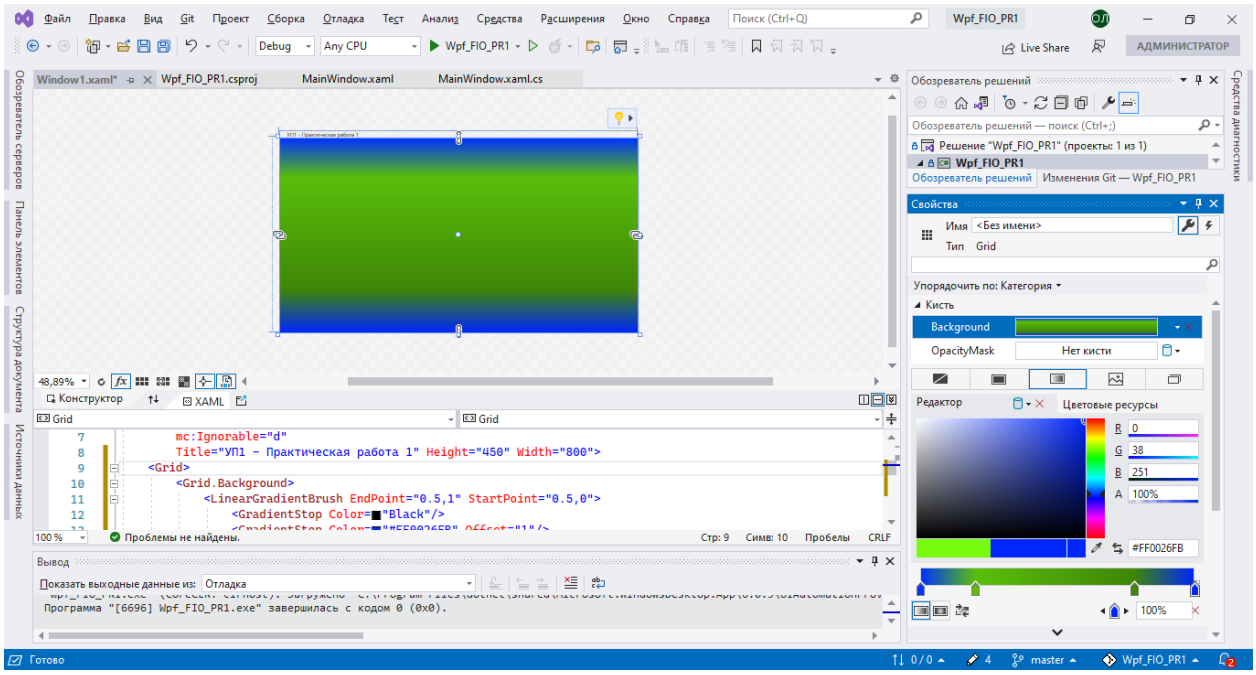
Задание 3:

Добавьте новое окно. ПКМ по названию проекта – Добавить - Окно



В XAML-коде для элемента Windows определите линейную градиентную заливку фона в соответствии с рисунком (можно свою, но подобную):

Используйте окно свойства для настройки цвета.



Для перехода с одного окна в другое(что бы посмотреть результат) выполните следующее - перед завершающим Grid наберите:

```
    <Canvas>
        <Button x:Name="Win1" Canvas.Left="10" Canvas.Top="10" Height="50" Width="100" Content="Win1" Click="Win1_Click"></Button>
        <Button x:Name="Win2" Canvas.Left="150" Canvas.Top="10" Height="50" Width="100" Content="Win2" Click="Win2_Click"></Button>
    </Canvas>
</Grid>
```

(Генерацию второй кнопки отложите до выполнения задания 4, а потом добавьте)

В файле найдите созданный обработчик и пропишите:

```
ссылка: 1
private void Win1_Click(object sender, RoutedEventArgs e)
{
    Window1 window1 = new Window1();
    window1.Show();
}
```

Где Window1 – имя вашего нового окна, которое вы создали в прошлом примере! А window1 – экземпляр вашего окна.

Запустите. Проверьте что по нажатию на кнопку появляется ваше выполненное задание 3.

Теория.

Диспетчеры компоновки

Окно WPF-приложения обычно представлено корневым элементом Window. Дочерним элементом корневого элемента является диспетчер компоновки, который в свою очередь содержит любое количество элементов (в том числе, вложенных диспетчеров компоновки), определяющих пользовательский интерфейс. Диспетчер компоновки является объектом класса, унаследованного от абстрактного класса System.Windows.Controls.Panel.

Основные панели (диспетчеры компоновки, контейнерные элементы управления) WPF:

Canvas Элементы остаются в точности там, где были размещены во время проектирования

DockPanel Привязывает содержимое к определенной стороне панели (Top (верхняя), Bottom (нижняя), Left (левая) или Right (правая))

Grid Располагает содержимое внутри серии ячеек, расположенных в табличной сетке

StackPanel Выводит содержимое по вертикали или горизонтали, в зависимости от значения свойства Orientation

WrapPanel Позиционирует содержимое слева направо, перенося на следующую строку по достижении границы панели. Последовательность размещения происходит сначала сверху вниз или сначала слева направо, в зависимости от значения свойства Orientation

Диспетчер компоновки Canvas Панель Canvas поддерживает абсолютное позиционирование содержимого пользовательского интерфейса. Если пользователь изменяет размер окна, делая его меньше, чем компоновка, обслуживаемая панелью Canvas, ее внутреннее содержимое становится невидимым до тех пор, пока контейнер вновь не увеличится до размера, равного или больше начального размера области Canvas. Панель Canvas обладает следующим недостатком: элементы внутри Canvas не изменяются динамически при применении стилей или шаблонов.

Задание 4.

Создайте новое окно и проделывайте следующие примеры.

Диспетчер компоновки Canvas

Панель Canvas поддерживает абсолютное позиционирование содержимого пользовательского интерфейса. Если пользователь изменяет размер окна, делая его меньше, чем компоновка, обслуживаемая панелью Canvas, ее внутреннее содержимое становится невидимым до тех пор, пока контейнер вновь не увеличится до размера, равного или больше начального размера области Canvas. Панель Canvas обладает следующим недостатком: элементы внутри Canvas не изменяются динамически при применении стилей или шаблонов.

4.1 Рассмотрим панель Canvas со следующим содержимым:

```
Grid                                                    Grid
 8          Title="Window2" Height="450" Width="800">
 9      <Grid>
10          <Canvas>
11              <Label Canvas.Left="10" Canvas.Top="10" Content="Регистрация пользователя" />
12              <Label Canvas.Left="10" Canvas.Top="40" Content="ФИО" />
13              <TextBox Canvas.Left="70" Canvas.Top="40" Width="200" />
14              <Label Canvas.Left="10" Canvas.Top="70" Content="Email" />
15              <TextBox Canvas.Left="70" Canvas.Top="70" Width="200" />
16              <Button Canvas.Left="50" Canvas.Top="100" Content="Зарегистрироваться" />
17          </Canvas>
18
19      </Grid>
20  </Window>
```

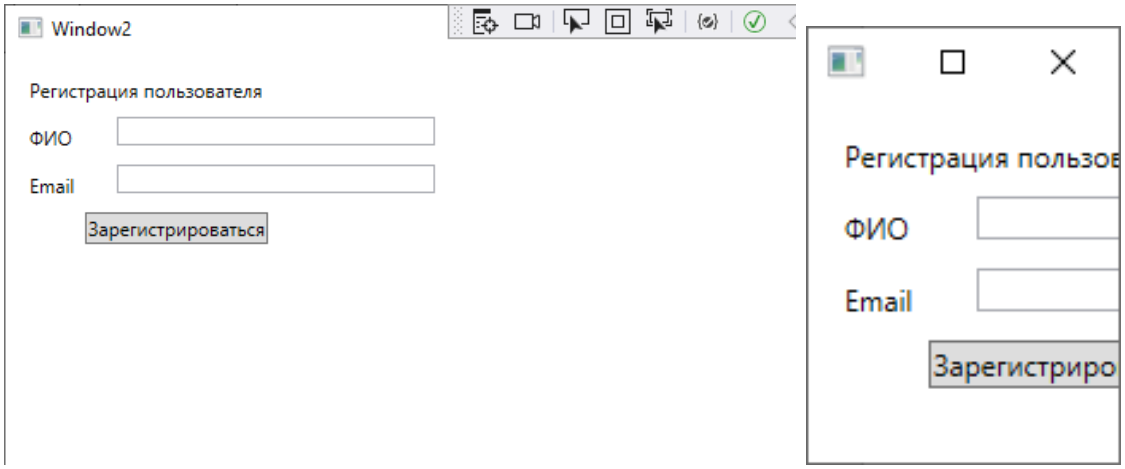
У элементов управления Label, TextBox, Button отсутствуют атрибуты Left и Top, поэтому для определения положения элементов на панели используется синтаксис присоединяемых свойств.

Присоединяемые свойства XAML (attached properties)

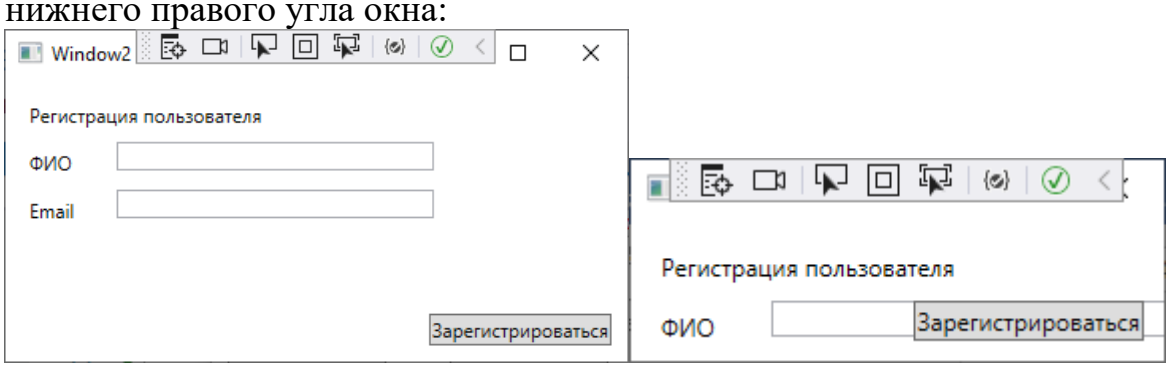
В XAML поддерживается специальный синтаксис, используемый для определения значения присоединяемого свойства. Присоединяемые свойства позволяют дочернему элементу устанавливать значение какого-то свойства, которое в действительности определено в родительском элементе.

Общий шаблон: С помощью присоединяемых свойств можно определить значения лишь ограниченного набора свойств родительских элементов, которые определены специальным образом в классе родительского элемента.

Дополнительное задание: определите, каким образом присоединяемые свойства объявляются в классе родительского элемента.



Для дочернего элемента необходимо указать привязку по вертикали (Canvas.Top или Canvas.Bottom) и привязку по горизонтали (Canvas.Left или Canvas.Right). Также можно (не обязательно) задать ширину и высоту элемента с помощью атрибутов Width и Height. Таким образом, положение дочернего элемента управления можно задать относительно любого угла окна. Например, в следующем примере положение кнопки задано относительно нижнего правого угла окна:



Порядок объявления дочерних элементов управления определяет порядок их вывода на экран. В приведенном выше примере кнопка выводится перед текстовыми полями, т.к. она была объявлена в файле XAML последней.

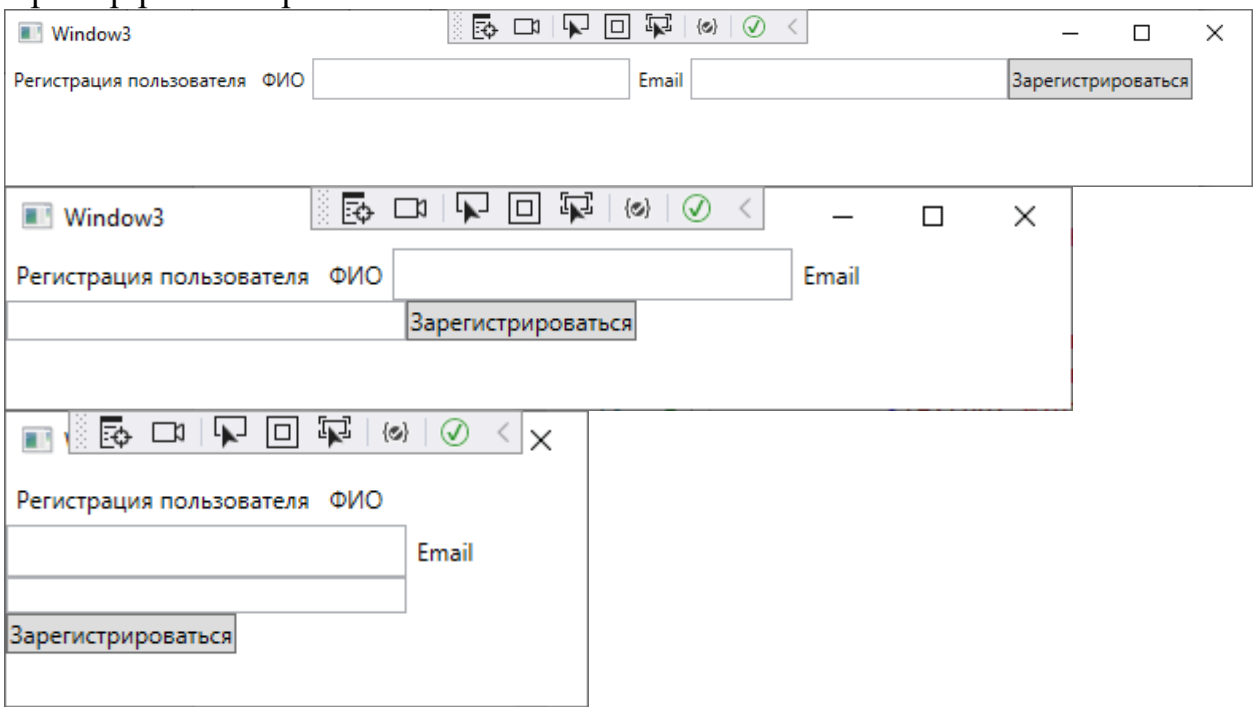
4.2 Создайте новое окно

Диспетчер компоновки WrapPanel

Панель WrapPanel выводит дочерние элементы последовательно слева направо (либо сверху вниз, если для атрибута Orientation установлено значение “Vertical”) и при достижении границы окна переходит на новую строку (столбец). При изменении размеров окна панель перераспределяет компоненты таким образом, чтобы они находились в окне. Рассмотрим панель WrapPanel со следующим содержимым:

```
WrapPanel                                                    WrapPanel
5           xmlns:mc="http://schemas.openxmlformats.org/markup-co
6           xmlns:local="clr-namespace:Wpf_FIO_PR1"
7           mc:Ignorable="d"
8           Title="Window3" Height="450" Width="800">
9       <Grid>
10          <WrapPanel>
11              <Label Content="Регистрация пользователя" />
12              <Label Content="ФИО" />
13              <TextBox Width="200" />
14              <Label Content="Email" />
15              <TextBox Width="200" />
16              <Button Content="Зарегистрироваться" />
17          </WrapPanel>
18      </Grid>
19  </Window>
20
```

Пример работы приложения:



4.3 Создайте новое окно

Диспетчер компоновки StackPanel

Панель StackPanel располагает содержащиеся в нем элементы управления либо в вертикальном столбце (по умолчанию), либо в горизонтальной строке (если в атрибут Orientation записано значение

“Vertical”). Если в панель StackPanel добавлено больше элементов управления, чем может быть отображено по ширине/высоте StackPanel, лишние элементы обрезаются и не отображаются. При выводе элементов сверху вниз элементы по умолчанию растягиваются по горизонтали. Это поведение можно изменить с помощью свойств HorizontalAlignment и VerticalAlignment. Рассмотрим панель StackPanel со следующим содержимым:

```
<Grid>
    <StackPanel HorizontalAlignment="Center">
        <Label Content="Регистрация пользователя" />
        <Label Content="ФИО" />
        <TextBox Width="200" />
        <Label Content="Email" />
        <TextBox Width="200" />
        <Button Content="Зарегистрироваться" />
    </StackPanel>
</Grid>
```

Посмотрите и отразите в отчет поведение данной панели(примеры работы).

4.4 Создайте новое окно

Диспетчер компоновки DockPanel

Панель DockPanel пристыковывает дочерние элементы к различным сторонам панели: Top, Bottom, Left, Right. Атрибут LastChildFill по умолчанию имеет значение True, что означает, что последний дочерний элемент управления будет занимать всё оставшееся пространство панели. Рассмотрим панель DockPanel со следующим содержимым:

```
<Grid>
    <DockPanel LastChildFill="False">
        <Label DockPanel.Dock="Top" Content="Регистрация пользователя" />
        <Label DockPanel.Dock="Left" Content="ФИО" />
        <TextBox DockPanel.Dock="Left" Width="200" />
        <Label DockPanel.Dock="Right" Content="Email" />
        <TextBox DockPanel.Dock="Right" Width="200" />
        <Button DockPanel.Dock="Bottom" Content="Зарегистрироваться" />
    </DockPanel>
</Grid>
```

В отчете приведите пример поведения данного диспетчера

4.5. Создайте новое окно

Диспетчер компоновки Grid

Подобно HTML-таблице, панель Grid может состоять из набора ячеек, каждая из которых имеет свое содержимое.

При определении панели Grid выполняются следующие шаги:

1. Определение и конфигурирование каждого столбца.
2. Определение и конфигурирование каждой строки.
3. Назначение содержимого каждой ячейке сетки с использованием синтаксиса присоединяемых свойств.

Если не определить никаких строк и столбцов, то по умолчанию панель Grid будет состоять из одной ячейки, занимающей всю поверхность окна. Кроме того, если не указать ячейку для дочернего элемента, то он разместится в

столбце 0 и строке 0. Определение столбцов и строк выполняются за счет использования элементов `<Grid.RowDefinitions>` и `<Grid.ColumnDefinitions>`, которые содержат коллекции элементов `ColumnDefinition` и `RowDefinitions`, соответственно. Каждый дочерний элемент прикрепляется к ячейке сетки, используя присоединяемые свойства Grid.Row и Grid.Column. Левая верхняя ячейка определяется с помощью Grid.Column="0" и Grid.Row="0". Рассмотрим панель Grid со следующим содержимым:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Label Grid.Row="0" Grid.Column="0" Content="Регистрация пользователя" />
    <Label Grid.Row="1" Grid.Column="0" Content="ФИО" />
    <TextBox Grid.Row="1" Grid.Column="1" Width="200" />
    <Label Grid.Row="2" Grid.Column="0" Content="Email" />
    <TextBox Grid.Row="2" Grid.Column="1" Width="200" />
    <Button Grid.Row="3" Grid.Column="0" Content="Зарегистрироваться" />
</Grid>
```

В первой части мы определяем сетку из колонок и столбцов, а во второй части распределяем позиции элементов – где в какой ячейке находятся.

В отчете отразите различные состояния окна

Объединение ячеек осуществляется с помощью присоединяемых свойств Grid.ColumnSpan и Grid.RowSpan аналогично объединению ячеек в HTML-таблицах. Рассмотрим панель Grid со следующим содержимым:

```
9     <Grid>
10        <Grid.RowDefinitions>
11            <RowDefinition/>
12            <RowDefinition/>
13            <RowDefinition/>
14            <RowDefinition/>
15        </Grid.RowDefinitions>
16        <Grid.ColumnDefinitions>
17            <ColumnDefinition/>
18            <ColumnDefinition/>
19        </Grid.ColumnDefinitions>
20
21        <Label Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2" HorizontalAlignment="Center"
22  Content="Регистрация пользователя" />
23        <Label Grid.Row="1" Grid.Column="0" Content="ФИО" />
24        <TextBox Grid.Row="1" Grid.Column="1" Width="200" />
25        <Label Grid.Row="2" Grid.Column="0" Content="Email" />
26        <TextBox Grid.Row="2" Grid.Column="1" Width="200" />
27        <Button Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2" HorizontalAlignment="Center"
28  Content="Зарегистрироваться" />
29    </Grid>
```

Отразите результат работы в отчете

При определении ряда можно задать его высоту с помощью атрибута Height, а при определении столбца можно задать его ширину с помощью атрибута Width. Значение этих атрибутов может быть следующим:

– "Auto" - высота строчки (или ширина колонки) определяется её содержимым;

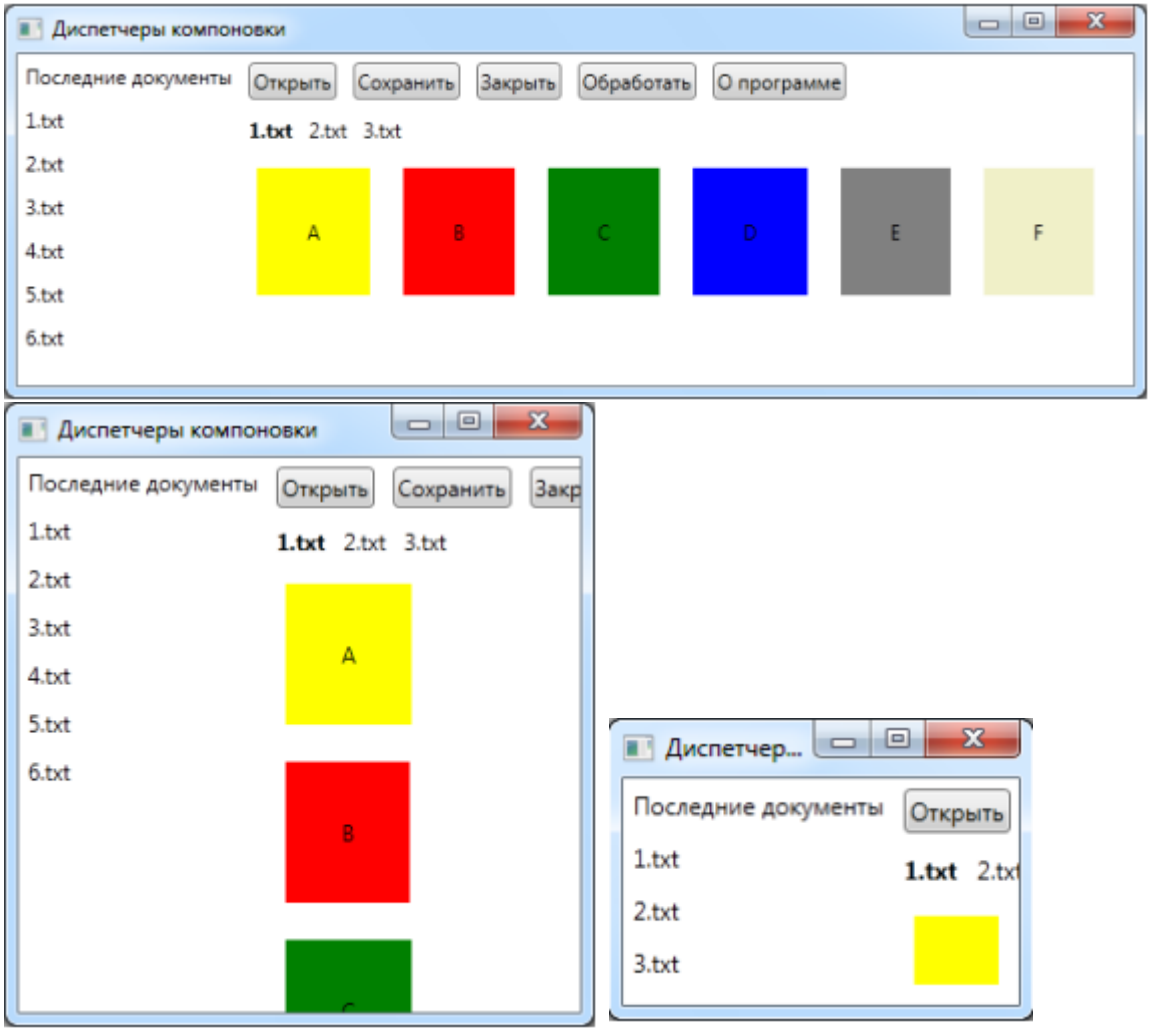
– "Число" - высота строчки (или ширина колонки) равна указанному числу точек;

– "*" - высота строчки (или ширина колонки) занимает всё свободное пространство. Если строчек (колонок) с таким значением атрибута несколько, то свободное пространство перераспределяется между ними.

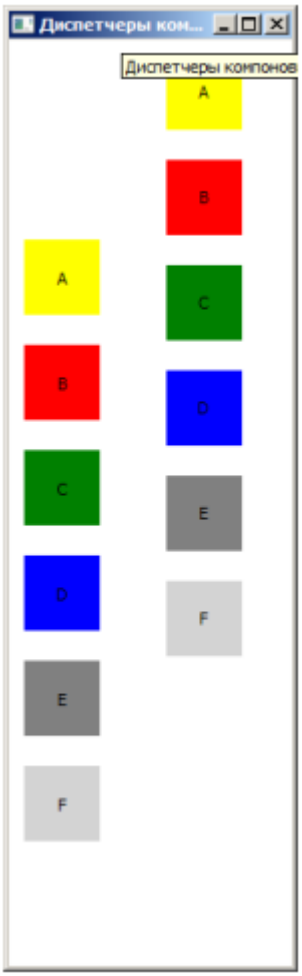
Задание 5. Самостоятельная работа

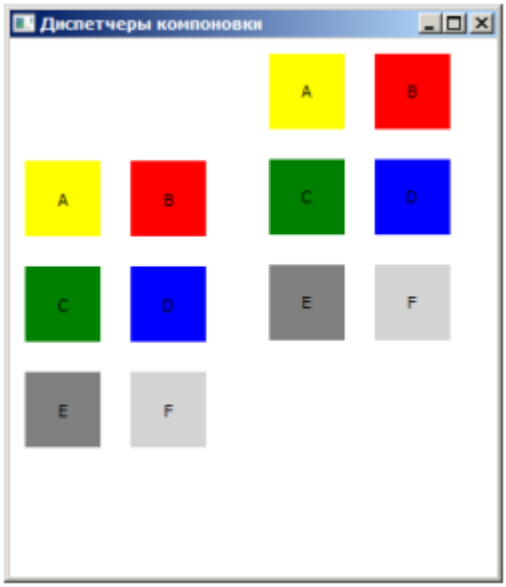
Разработать приложение WPF со следующим графическим интерфейсом в новом окне:

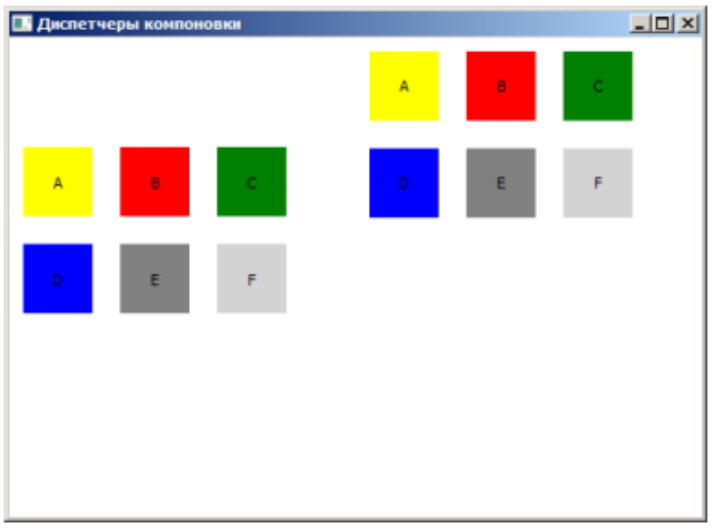
5.1



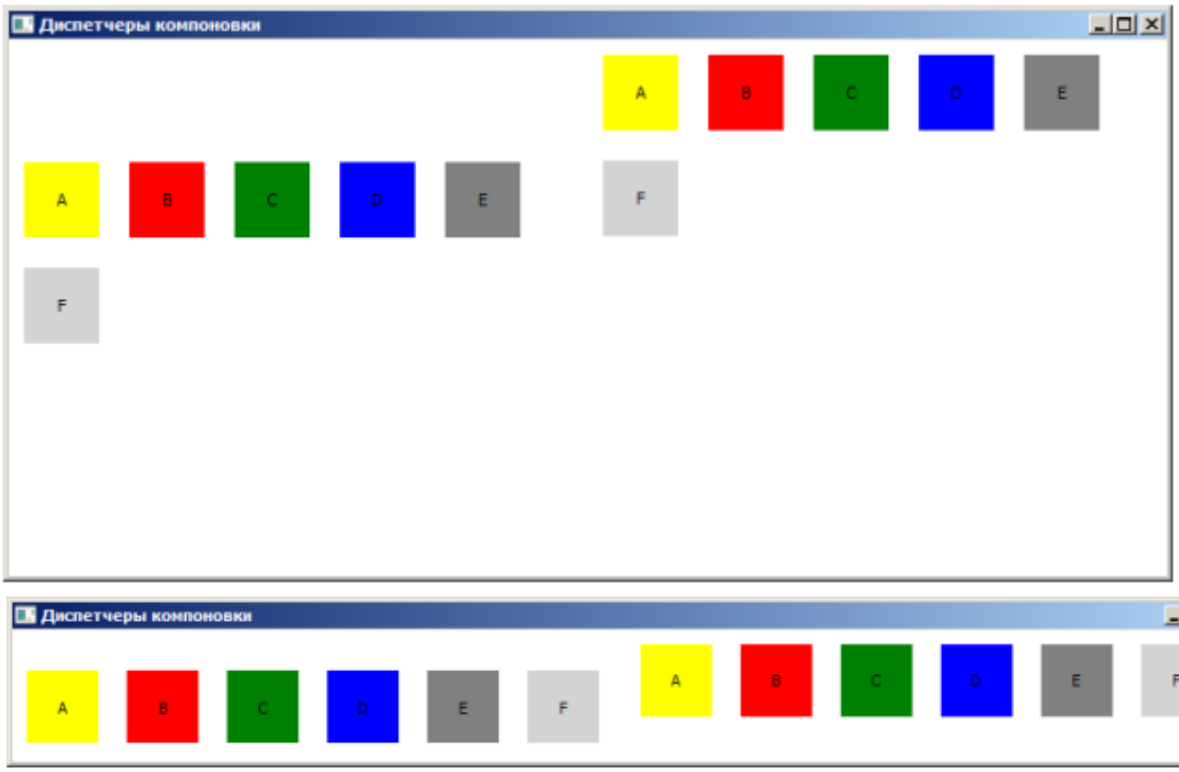
5.2


Диспетчеры ком...
Диспетчеры компонов
A
B
C
D
E
F
A
B
C
D
E
F


Диспетчеры компоновки
A
B
C
D
E
F
A
B
C
D
E
F


Диспетчеры компоновки
A
B
C
D
E
F
A
B
C
D
E
F

УП МДК 02



Проект расположить на GitHub

В отчет:
Титульный лист
Скриншоты всех заданий
Ссылка на гитхаб