

# Урок 1 - Инженерный подход к разработке ПО

---

## Оглавление

- Главная задача курса
- Цель урока
- Подготовка рабочего места
  - Установка Visual Studio
    - Выбор компонентов
    - Платформа .NET Core
  - Начало - Новый проект
  - Первые шаги для нового проекта
    - Добавление в систему контроля версий
    - Тестовый консольный проект
  - Прототип бизнес-функции
  - Структура проекта WPF-приложения
    - Файлы проекта WPF
      - AssemblyInfo.cs
      - App.xaml
      - App.xaml.cs
      - MainWindow.xaml
      - MainWindow.xaml.cs
    - Визуальный дизайнер интерфейса
      - Свойства визуальных элементов
      - События визуальных элементов
  - Различие WinForms и WPF
- Проектирование приложения
  - Главные этапы
  - Архитектура
  - Сервисы
  - Эскиз интерфейса
  - Обзор контейнеров компоновки
    1. Таблица (**Grid**)
    2. Стек-панель (**StackPanel**)
    3. Док-панель (**DockPanel**)
    4. Панель одинаковых размеров (**UniformGrid**)
    5. Панель с переносом (**WrapPanel**)
    6. Панель-холст/канва (**Canvas**)
    7. Виртуализованная стек-панель (**VirtualizedStackPanel**)
- Домашнее задание
- Ссылки

## Главная задача курса

Разработка программного обеспечения - сложный комплексный процесс, требующий постановки задачи, её анализа и принятия комплекса решений, позволяющих выбрать пути развития. Для иллюстрации этого процесса выберем задачу разработки программного средства, которое бы обеспечивало возможность автоматизированной рассылки сообщений электронной почты.

## Цель урока

1. Рассмотреть принципы и подходы разработки и развития ПО
2. Рассмотреть технологию *WPF* и её отличие от *WinForms*
3. Показать пример простейшего приложения на *WPF*, реализующего главную функцию бизнес-логики
4. Обзор основных структурных элементов компоновки

И так, целью нашей работы будет создание программы, которая будет отправлять сообщения по электронной почте нужным нам адресатам.

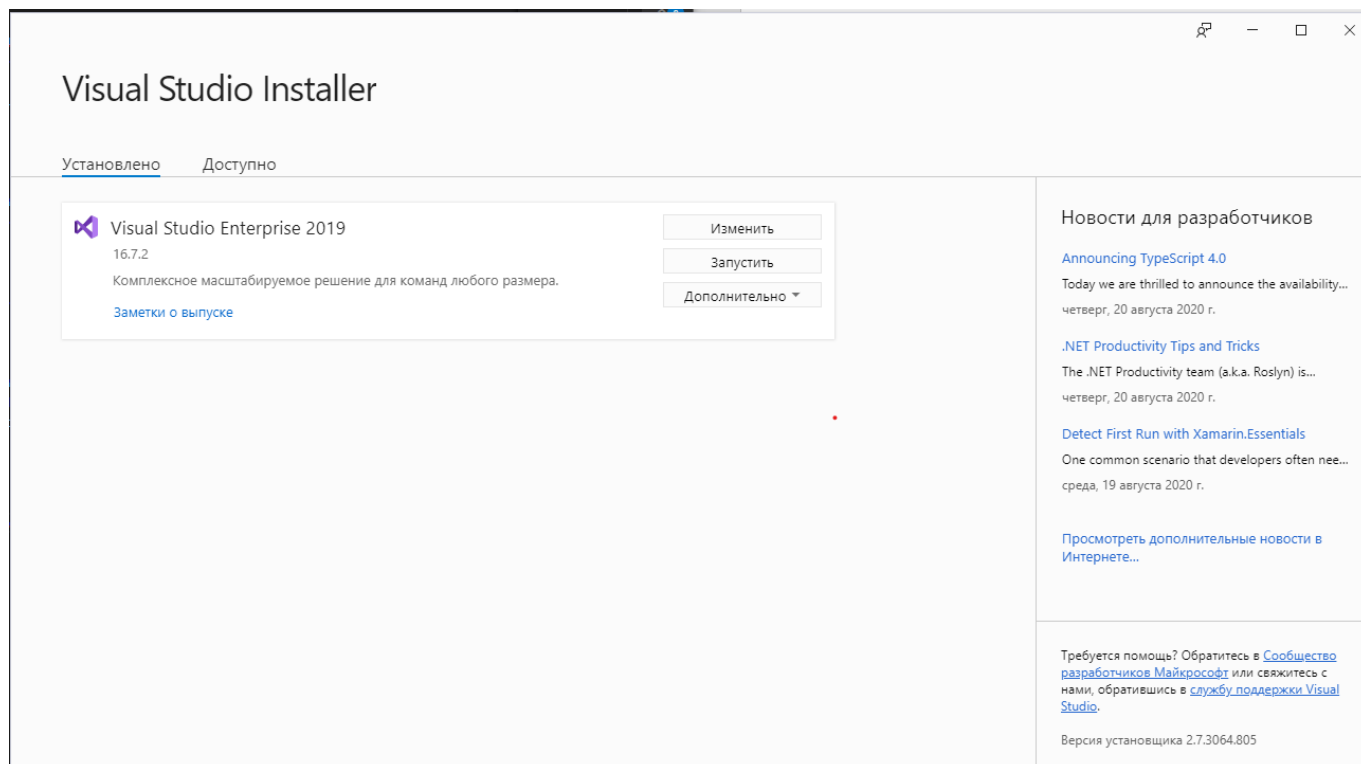
## Подготовка рабочего места

Прежде чем мы начнём погружение в сам процесс разработки ПО с применением современных технологий, давайте разберёмся с главной функцией нашего будущего приложения - с функцией рассылки почты. Назовём это ядром логики нашего будущего приложения. Надо понять - возможно ли это вообще в инфраструктуре *.NET* и языка *C#*, а если да, то на сколько это сложно и трудоёмко? По сути, этот этап можно назвать прототипированием. Эффективнее всего это делается с помощью маленького консольного приложения, в котором будет выполняться небольшая *C#* программа, которая, в нашем случае, должна будет суметь отправить одно электронное письмо на наш адрес электронной почты.

## Установка Visual Studio

Для начала, давайте определимся с инструментарием. Для работы нам потребуется среда разработки. Будем использовать *Visual Studio 2019*. Бесплатная версия *Community Edition* обеспечивает полный и достаточный для наших целей функционал. В качестве платформы мы будем использовать *.NET Core* версии 3.1 (и более старших версий). Для установки Visual Studio необходимо перейти на официальный сайт Microsoft и скачать веб-установщик: сайт [Visual Studio Community Edition](#).

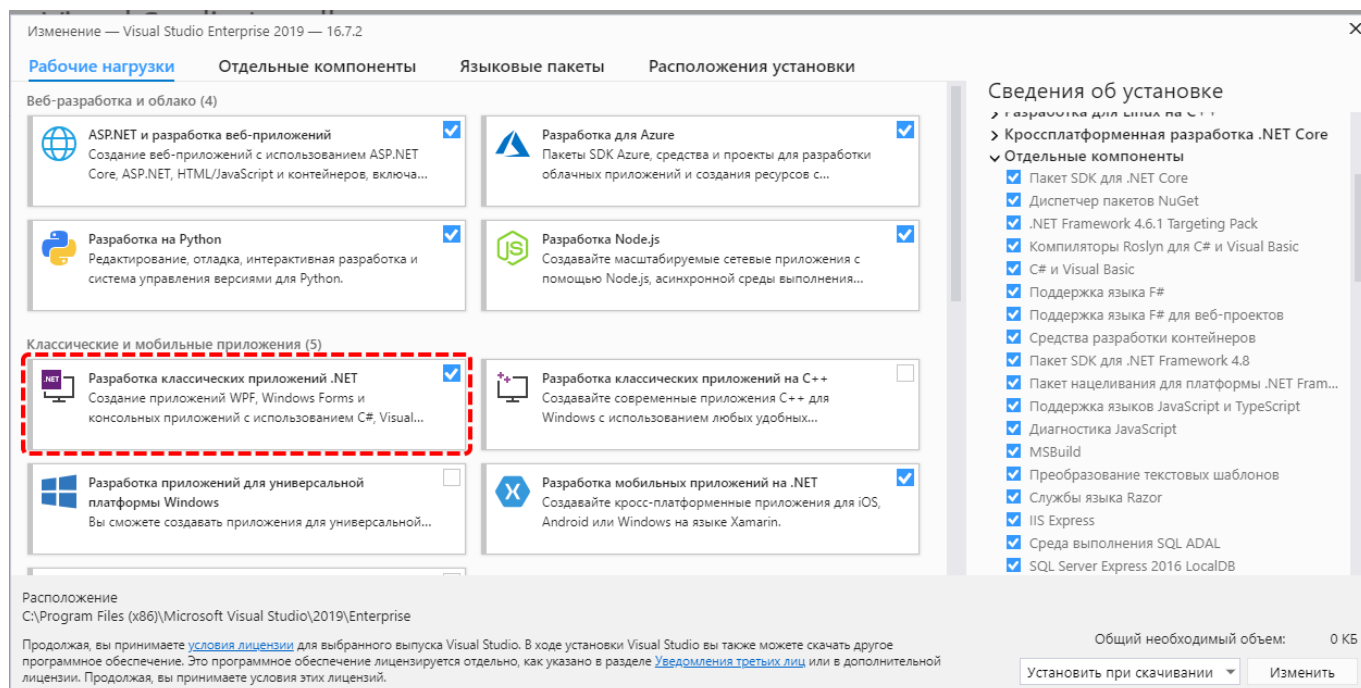
Выберем для установки *Visual Studio 2019 Community*



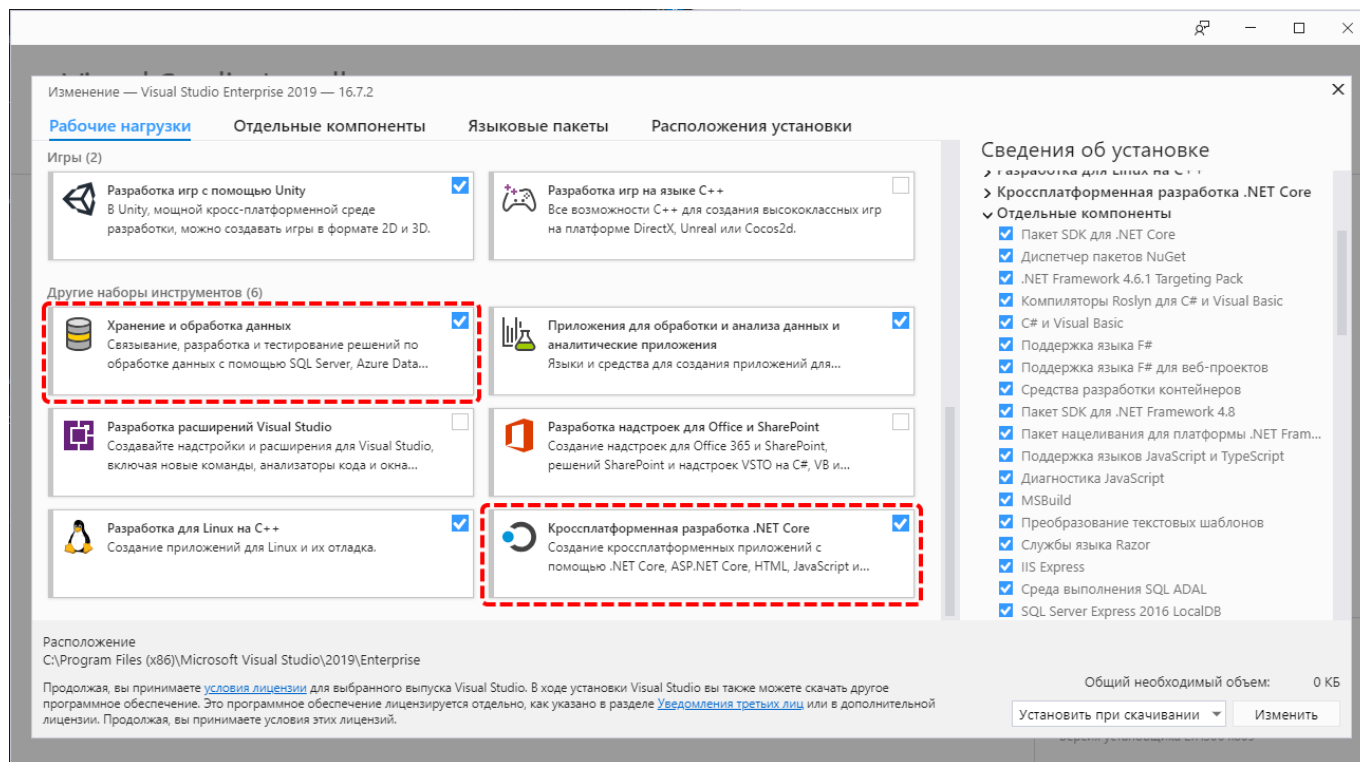
## Выбор компонентов

Среди основных устанавливаемых блоков требуется отметить:

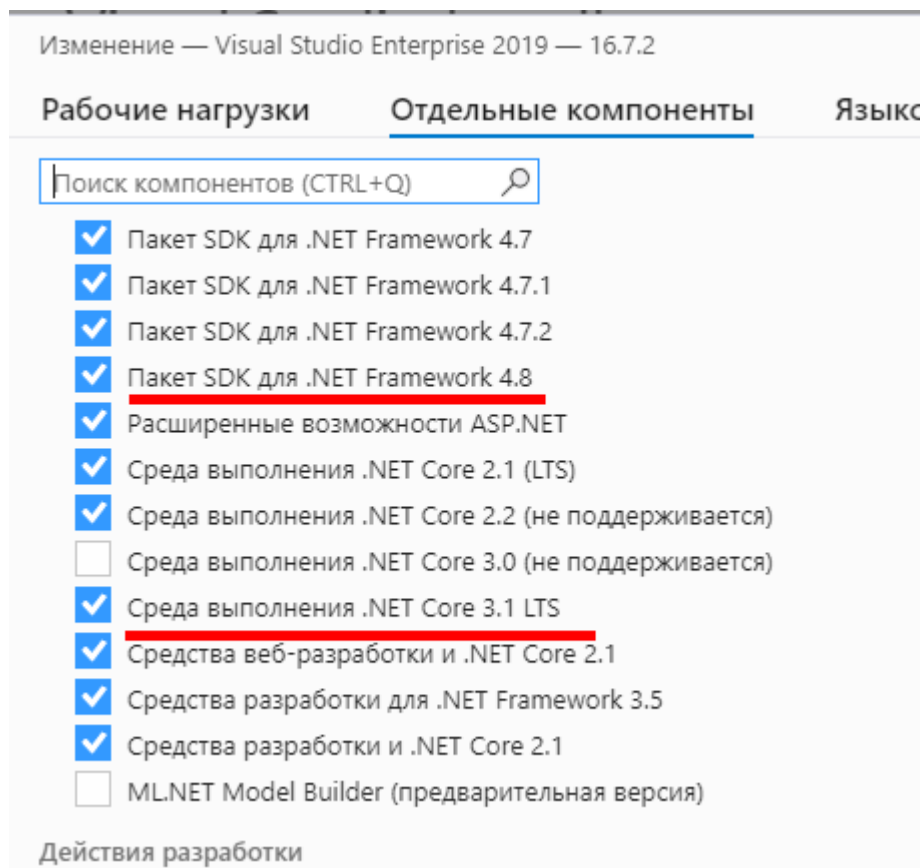
- Разработка классических приложений .NET



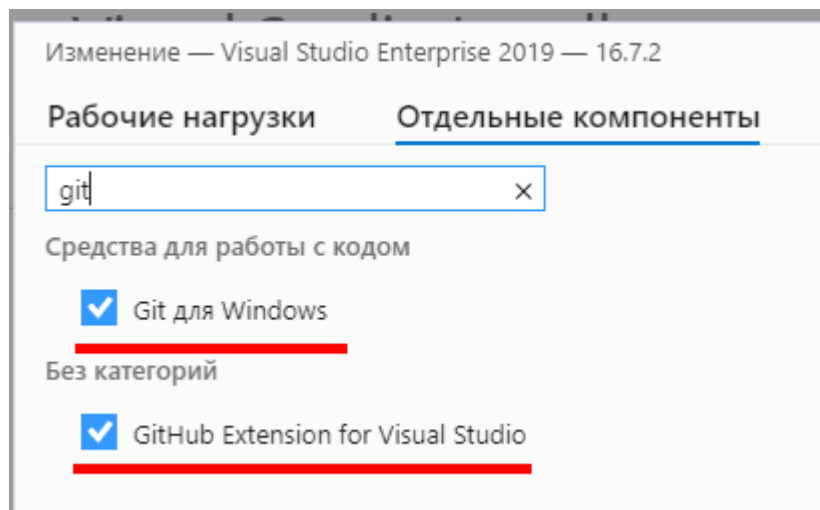
- Хранение и обработка данных
- Кроссплатформенная разработка .NET Core



В разделе отдельных компонентов следует проверить, что фреймворки, под которые требуется осуществлять разработку, выбраны. А при выборе, сделанном выше, они будут выбраны автоматически. Также следует отметить, что с выходом обновлений как фреймворков, так и Студии, данный перечень будет изменяться.



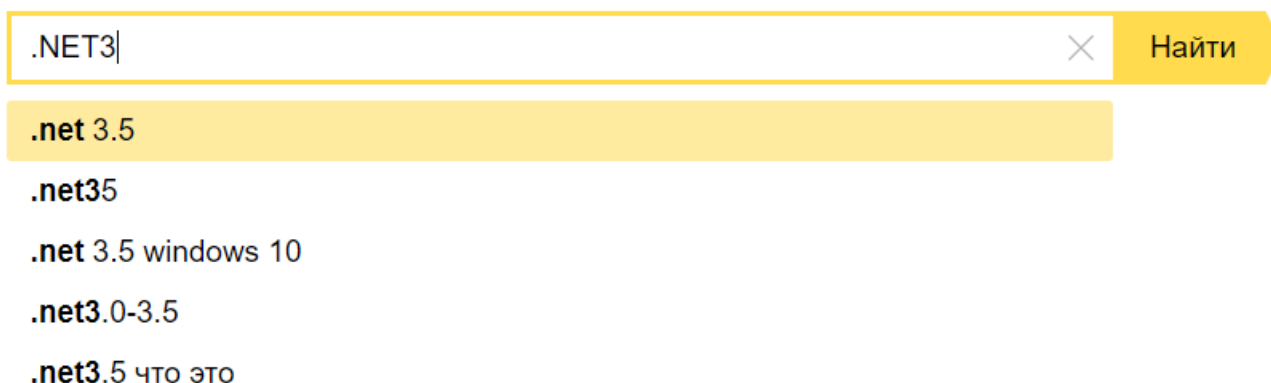
Также включаем установку компонентов, отвечающих за работу с системой контроля версий *GIT*



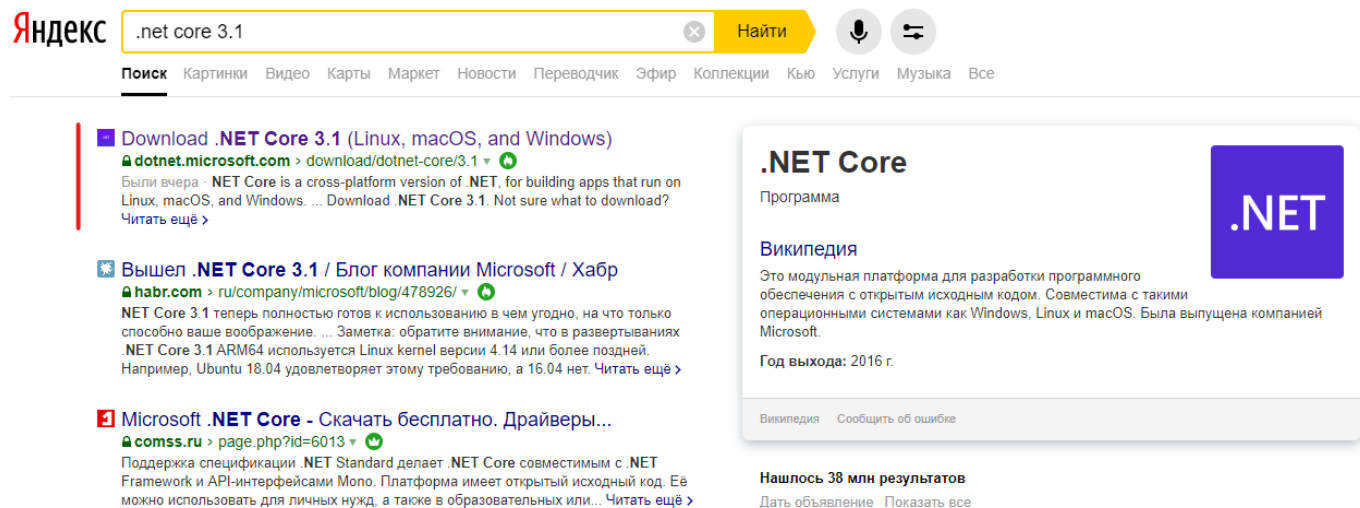
## Платформа .NET Core

Последнюю версию платформы *.NET* и инструменты разработки (*SDK*) можно установить либо через установщик Visual Studio, либо с [официального сайта .NET Core](#).


Найти официальный сайт для скачивания платформы поможет поисковик:



Первый же результат ведёт туда, куда нам нужно.



Мы можем выбрать то, что нас интересует: либо исполнительная среда платформы, либо полные инструменты разработки

 | .NET About Learn Architecture Docs Downloads Community All Microsoft

Home > Download

# Download .NET

Downloads for .NET Framework and .NET Core, including ASP.NET and ASP.NET Core

❓ Not sure where to start? See the [Hello World in 10 minutes tutorial](#) to install .NET and build your first app.

Windows

Linux

macOS

Docker

## .NET Core

### .NET Core 3.1

.NET Core is a cross-platform version of .NET for building websites, services, and console apps.

Run Apps ⓘ	<a href="#">Download .NET Core Runtime</a>
Build Apps ⓘ	<a href="#">Download .NET Core SDK</a>
Advanced ⓘ	<a href="#">All .NET Core downloads...</a>

## .NET Framework

### .NET Framework 4.8

.NET Framework is a Windows-only version of .NET for building any type of app that runs on Windows.

Run Apps ⓘ	<a href="#">Download .NET Framework Runtime</a>
Build Apps ⓘ	<a href="#">Download .NET Framework Dev Pack</a>
Advanced ⓘ	<a href="#">All .NET Framework downloads...</a>

Также мы можем выбрать версию платформы.

# Download .NET Core 3.1

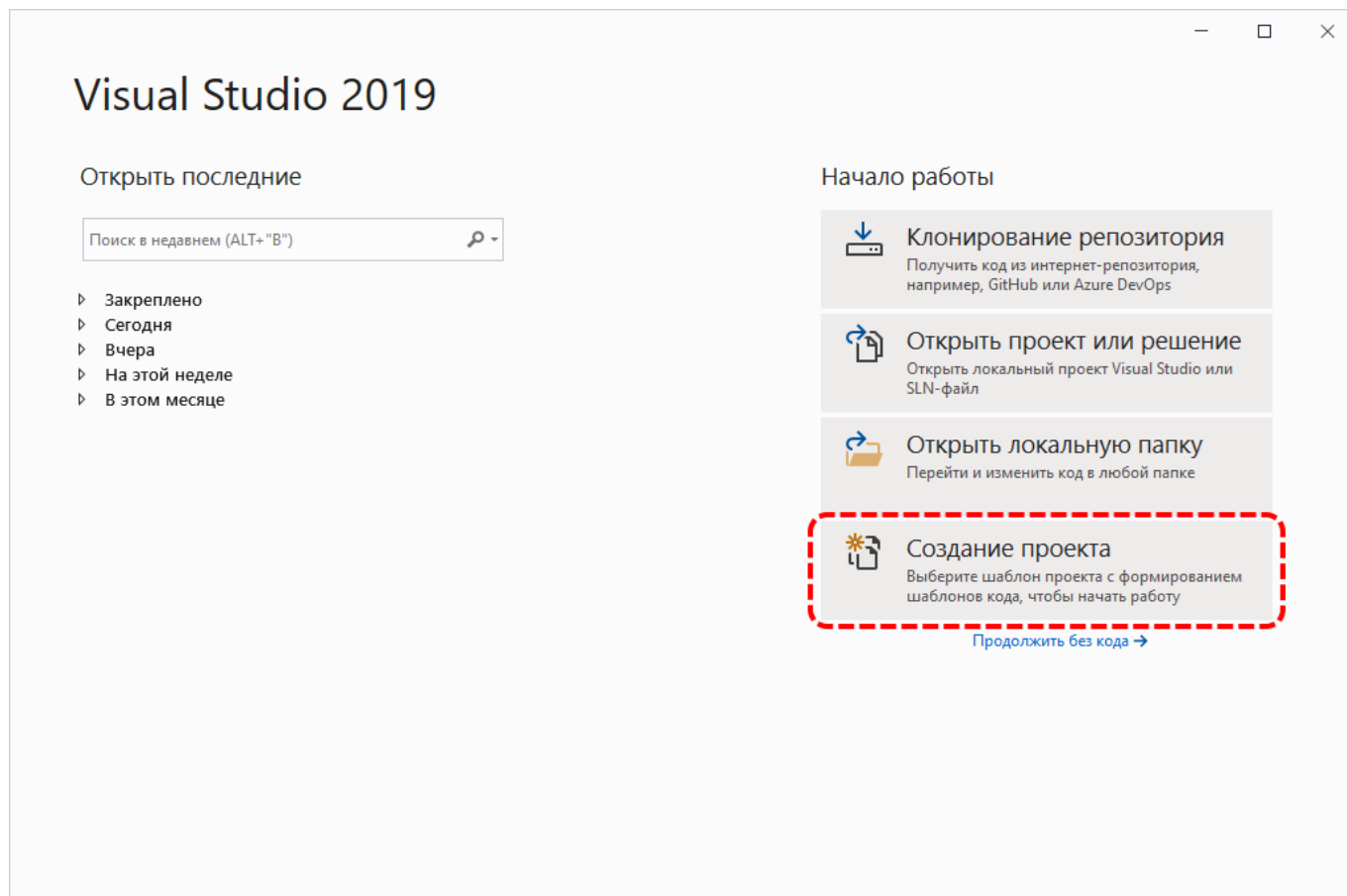
❓ Not sure what to download? [See recommended downloads for the latest version of .NET.](#)

Release information	Build apps - SDK ①	Run apps - Runtime ①												
<b>v3.1.7</b> <b>Security patch</b> ① <a href="#">Release notes</a> <b>Released</b> 2020-08-11	<p>❓ This release contains multiple SDKs. If you're using Visual Studio, look for the SDK that supports the version you're using. If you're not using Visual Studio, install the first SDK listed.</p> <p><b>SDK 3.1.401</b></p> <p><b>Visual Studio support</b>            Visual Studio 2019 (v16.7)            Visual Studio 2019 for Mac (v8.7)</p> <p><b>Included in</b>            Visual Studio 16.7.1</p> <p><b>Included runtimes</b>            .NET Core Runtime 3.1.7            ASP.NET Core Runtime 3.1.7            Desktop Runtime 3.1.7</p> <p><b>Language support</b>            C# 8.0            F# 4.7</p>	<p><b>ASP.NET Core Runtime 3.1.7</b></p> <p>The ASP.NET Core Runtime enables you to run existing web/server applications. <b>On Windows, we recommended installing the Hosting Bundle, which includes the .NET Core Runtime and IIS support.</b></p> <p><b>IIS runtime support (ASP.NET Core Module v2)</b>            13.1.20205.7</p> <table> <tr> <th>OS</th><th>Installers</th><th>Binaries</th></tr> <tr> <td>Linux</td><td><a href="#">Package manager instructions</a></td><td><a href="#">ARM32</a>   <a href="#">ARM64</a>   <a href="#">ARM64 Alpine</a>   <a href="#">x64 Alpine</a>   <a href="#">x64</a></td></tr> <tr> <td>macOS</td><td></td><td><a href="#">x64</a></td></tr> <tr> <td>Windows</td><td><a href="#">x64</a>   <a href="#">x86</a>   <a href="#">Hosting Bundle</a></td><td><a href="#">ARM32</a>   <a href="#">x64</a>   <a href="#">x86</a></td></tr> </table> <p><b>Desktop Runtime 3.1.7</b></p>	OS	Installers	Binaries	Linux	<a href="#">Package manager instructions</a>	<a href="#">ARM32</a>   <a href="#">ARM64</a>   <a href="#">ARM64 Alpine</a>   <a href="#">x64 Alpine</a>   <a href="#">x64</a>	macOS		<a href="#">x64</a>	Windows	<a href="#">x64</a>   <a href="#">x86</a>   <a href="#">Hosting Bundle</a>	<a href="#">ARM32</a>   <a href="#">x64</a>   <a href="#">x86</a>
OS	Installers	Binaries												
Linux	<a href="#">Package manager instructions</a>	<a href="#">ARM32</a>   <a href="#">ARM64</a>   <a href="#">ARM64 Alpine</a>   <a href="#">x64 Alpine</a>   <a href="#">x64</a>												
macOS		<a href="#">x64</a>												
Windows	<a href="#">x64</a>   <a href="#">x86</a>   <a href="#">Hosting Bundle</a>	<a href="#">ARM32</a>   <a href="#">x64</a>   <a href="#">x86</a>												

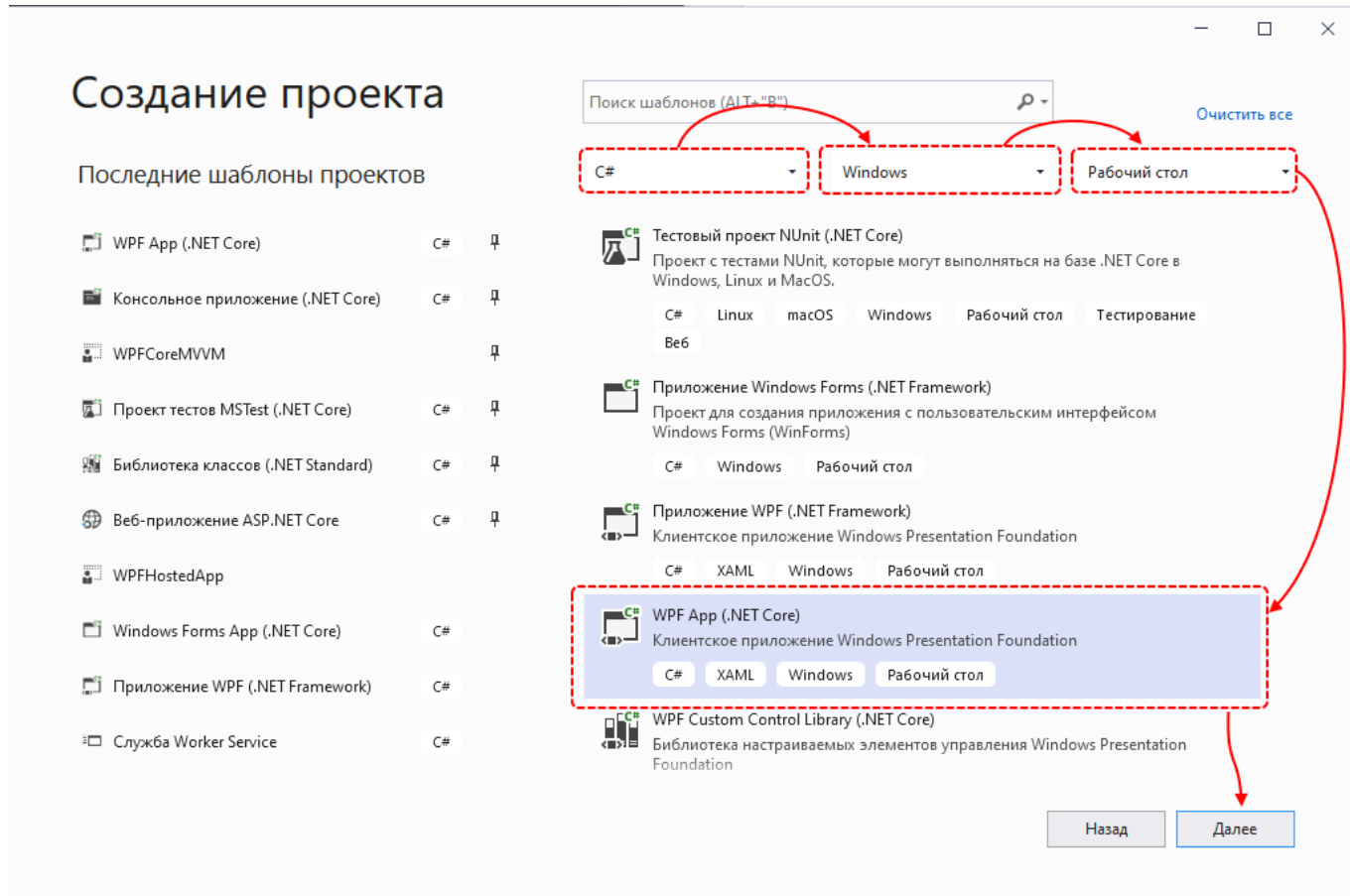
Периодически следует заглядывать сюда за последними новостями и обновлениями платформы.

## Начало - Новый проект

Давайте начнём с того, что с помощью *Visual Studio* создадим новое решение, нацеленное изначально на язык *C#*, ОС *Windows* и *Рабочий стол*. Среди вариантов шаблонов выберем приложение *WPF* под платформу *.NET Core*. В последствии у нас будет возможность в любой момент изменить структуру проекта (решения) так как нам будет необходимо.



Выбираем проект приложения *WPF* под платформу *.NET Core*.



Указываем где будет расположен исходный код проекта. Здесь ни в коем случае не надо устанавливать галочку. Иначе проект будет расположен в папке с решением (не будет создана подпапка) и в



последствии при попытке добавления ещё одного проекта в решение мы получим кашу из файлов.

Настроить новый проект

WPF App (.NET Core) C# XAML Windows Рабочий стол

Имя проекта

MailSender

Расположение

C:\Users\shmac\Documents\IGB\C#3\src

Имя решения ⓘ

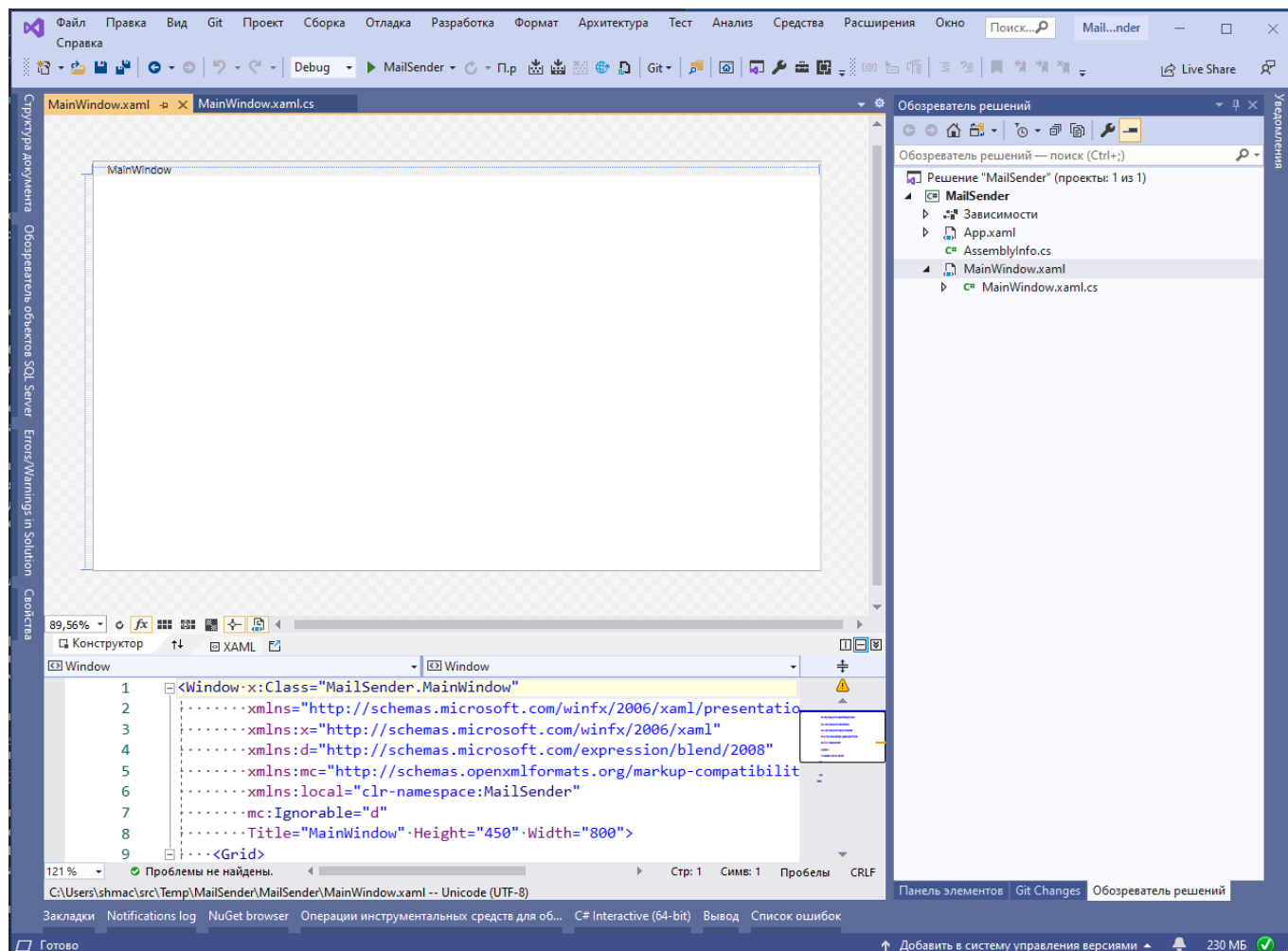
MailSender

☐ Поместить решение и проект в одном каталоге

Не ставить!!!

Назад Создать

Visual Studio создаёт шаблон проета приложения для рабочего стола Windows, использующего технологию WPF.



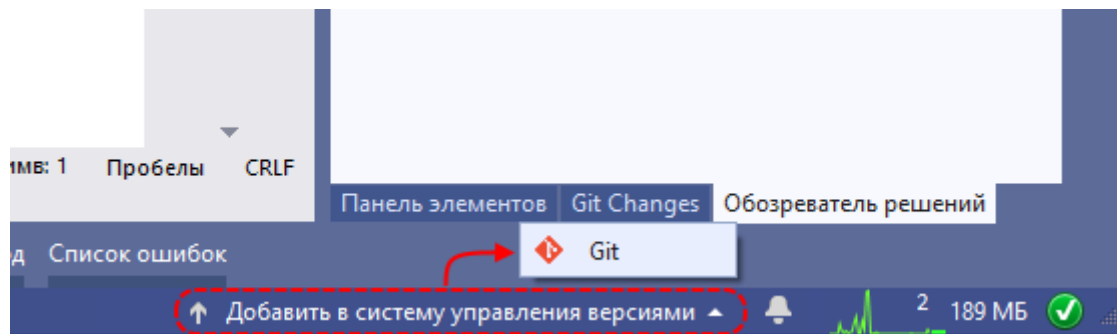
## Первые шаги для нового проекта

В открывшемся окне среды разработки справа (на скриншоте) можно увидеть главное навигационное окно - Обзорщик решений. С его помощью мы имеем возможность осуществлять навигацию между элементами (файлами) проекта, и не только... В основе идеи разработки ПО в среде *Visual Studio* лежит понятие "Решения". Решение - это комплекс (несколько) проектов, которые разрабатываются в рамках одного цикла и предназначены для решения какой-то сложной большой комплексной задачи. Более мелкими единицами, из которых состоит решение являются "Проекты". Каждый проект упрощённо можно ассоциировать с одним приложением (одним исполнительным файлом, либо файлом библиотеки, одним веб-сайтом, одним приложением под iOS, или Android и т.п.). Проекты могут быть индивидуальными и независимыми внутри решения, а могут быть связанными между собой ссылками. В нашем случае создан один единственный проект приложения, который можно (и стоит первым делом) скомпилировать и запустить путём нажатия кнопки запуск на панели инструментов VS, либо системной клавишей на клавиатуре *F5*. После этого VS выполнит компиляцию проекта и его запуск. Мы должны увидеть окно приложения. Это означает что инструменты разработки и платформа *.NET* в рабочем состоянии, и мы можем продолжать дальше.

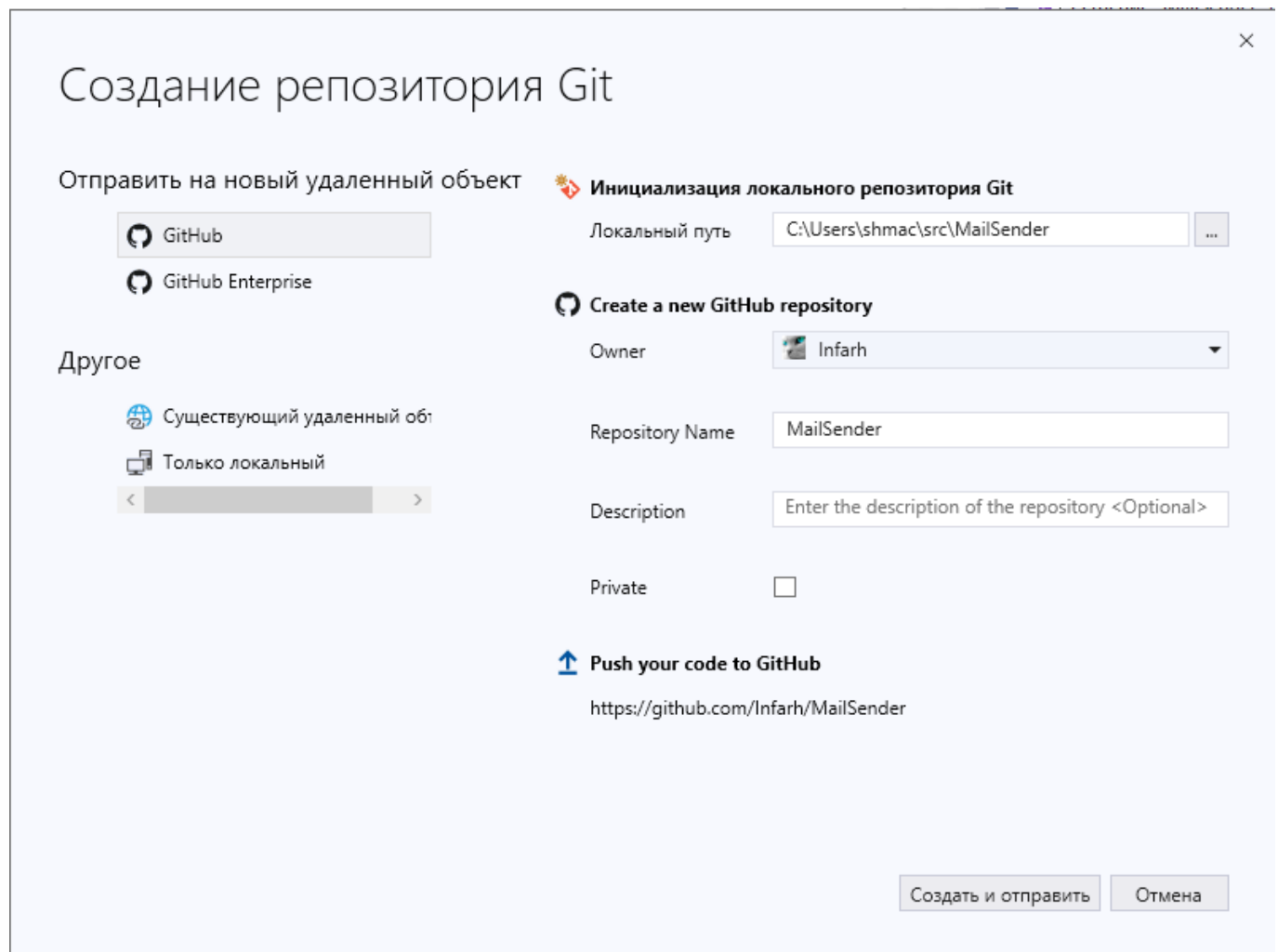
## Добавление в систему контроля версий

Первым делом добавим созданный проект в систему контроля версий. Это позволит нам отслеживать все вносимые в проект изменения и не беспокоиться о том, что сделанные нами правки в угаре творческого азарта не разрушат проект до такого состояния, что мы потеряем возможность их откатить.

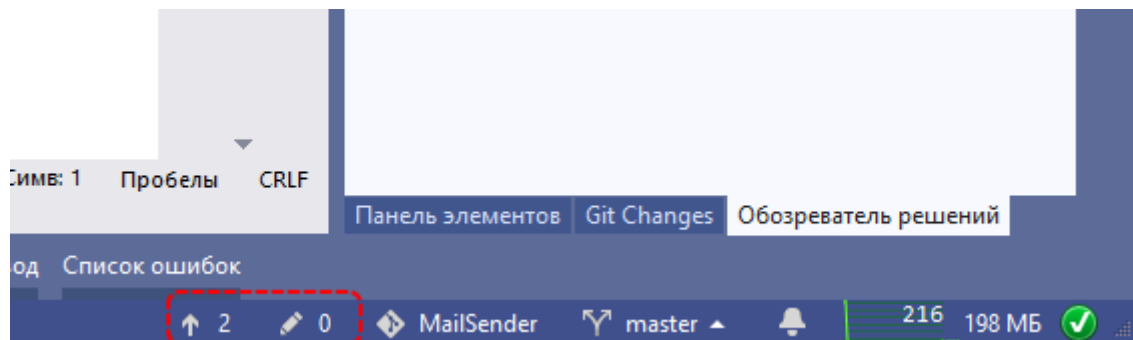
Для этого в правом нижнем углу в статусной строке окна VS есть кнопка "Добавить в систему управления версиями".



Локальный путь создания репозитория менять не следует. Слева указано, что будет использован удалённый сервер *GitHub*. Учётная запись, в которой мы залогинились, также доступна для выбора. Можно задать название репозитория на сервере, и задать описание. Кроме этого можно поставить галочку сделав репозиторий приватным.



После добавления в систему контроля версий мы в статусной строке можем увидеть состояние: две фиксации выполнены локально и ожидают отправки на удалённый сервер; файлов с незафиксированными изменениями нет; имя репозитория *MailSender*; имя ветки, в которой мы работаем - *master*.

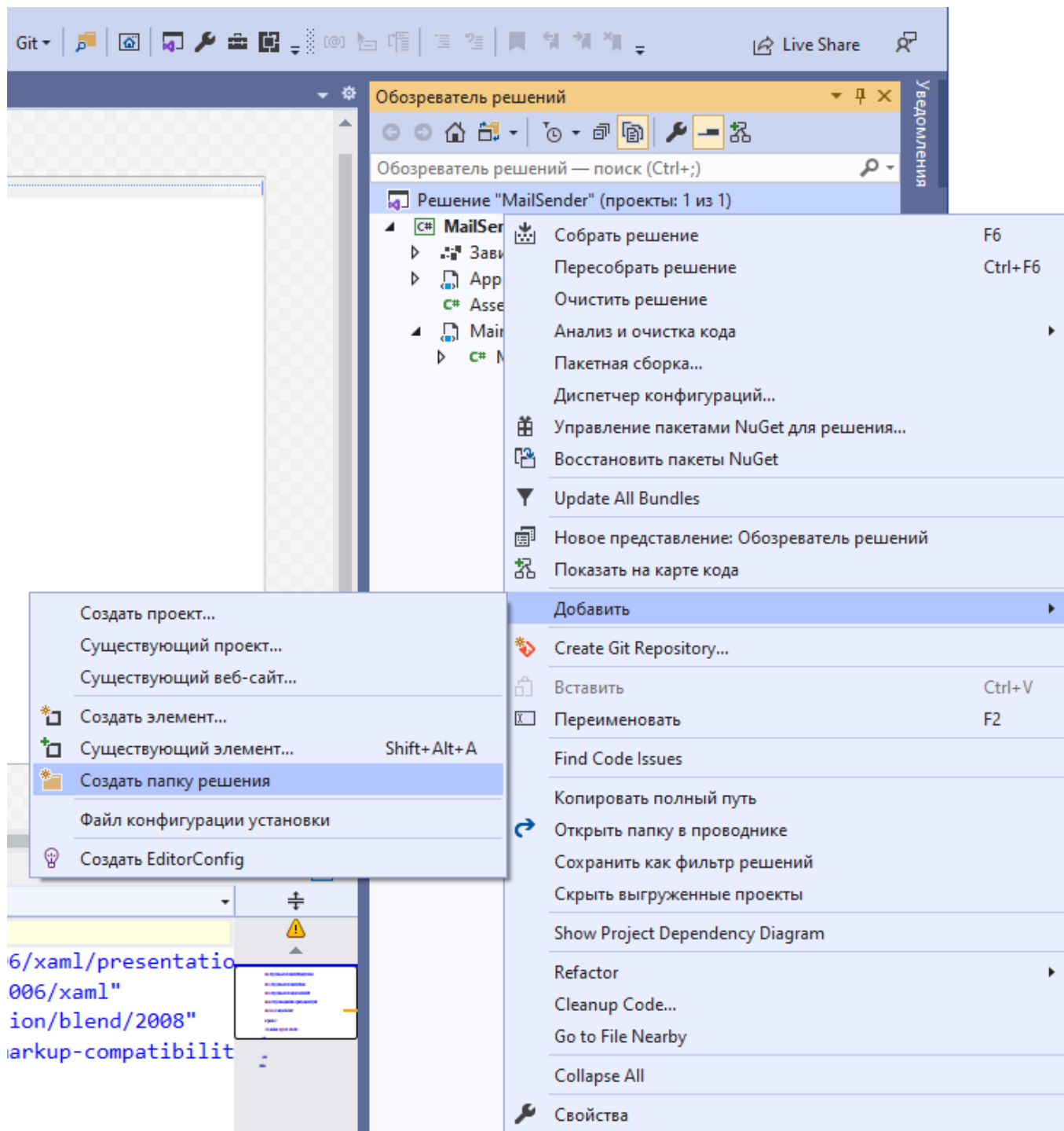


Для повышения эффективности процесса разработки в современном мире крайне рекомендуется (а местами требуется) использовать сервер системы контроля версий. Мы будем использовать для этих целей [github.com](https://github.com). Для этого требуется зарегистрироваться на этом ресурсе и в окне VS войти в систему. Это позволит нативным (естественным) образом быстро и эффективно передавать вделанные изменения на удалённый сервер и, при необходимости, скачивать их с сервера. Особенно это эффективно если мы будем вести разработки на нескольких рабочих местах (компьютерах), или в составе группы разработчиков.

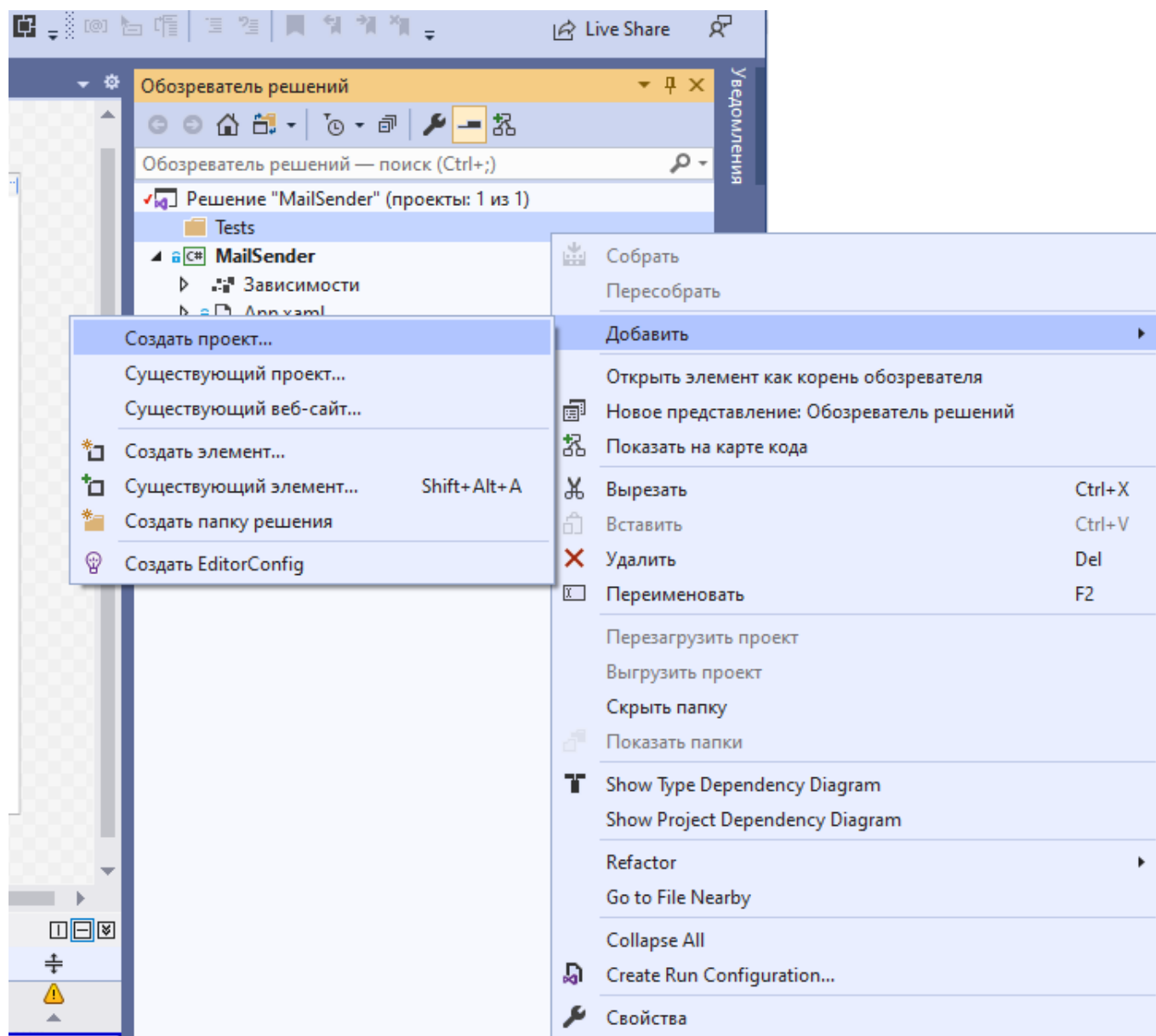
Теперь мы имеем возможность экспериментировать и не бояться что мы потеряем что-либо.

### Тестовый консольный проект

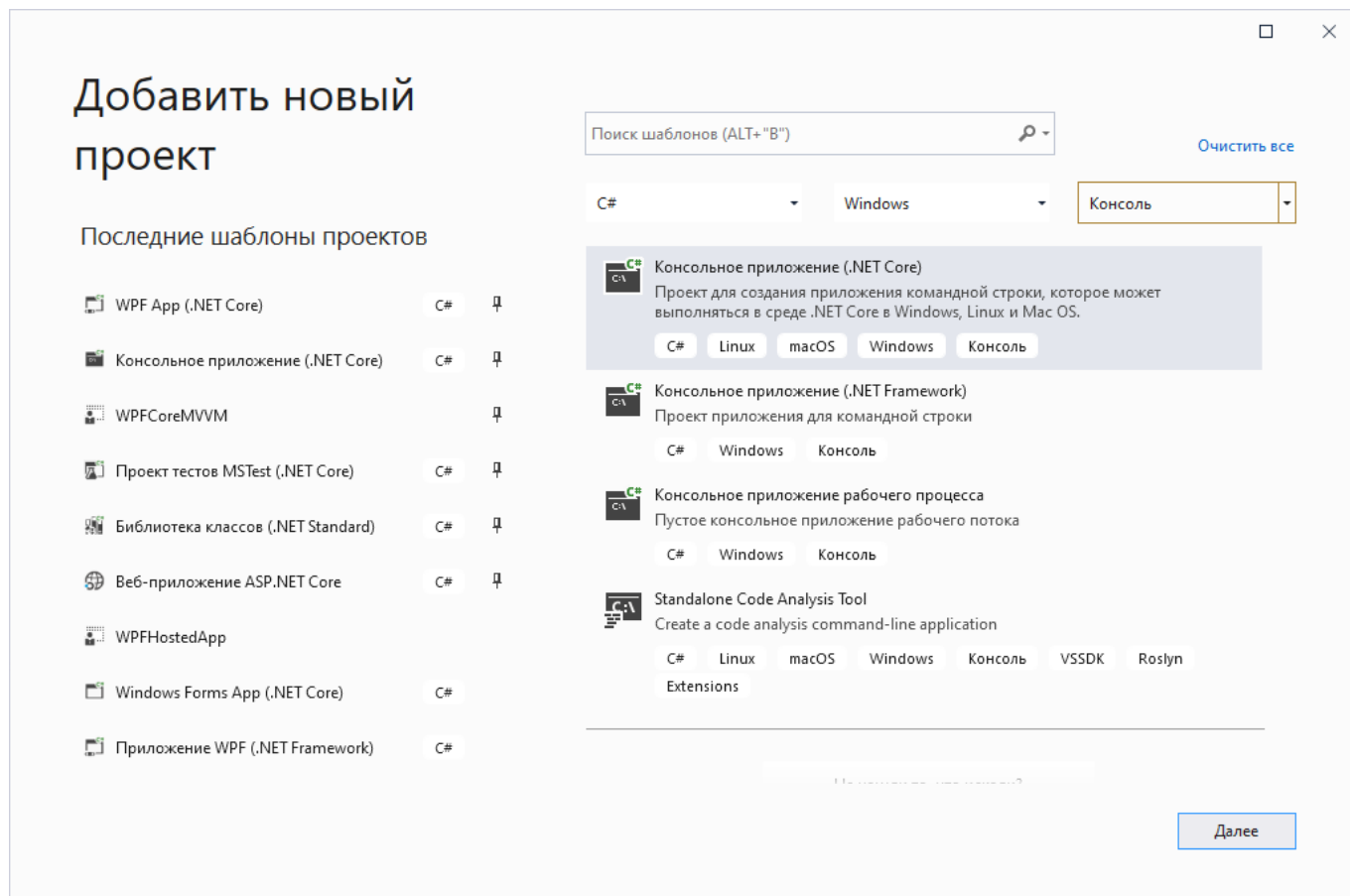
Для отработки главной функции нашего приложения создадим дополнительный проект - испытательный полигон в виде Консольного приложения *.NET Core*. Создадим виртуальную папку в решении с названием "Tests" и добавим в неё новый проект Консольного приложения *.NET Core*.



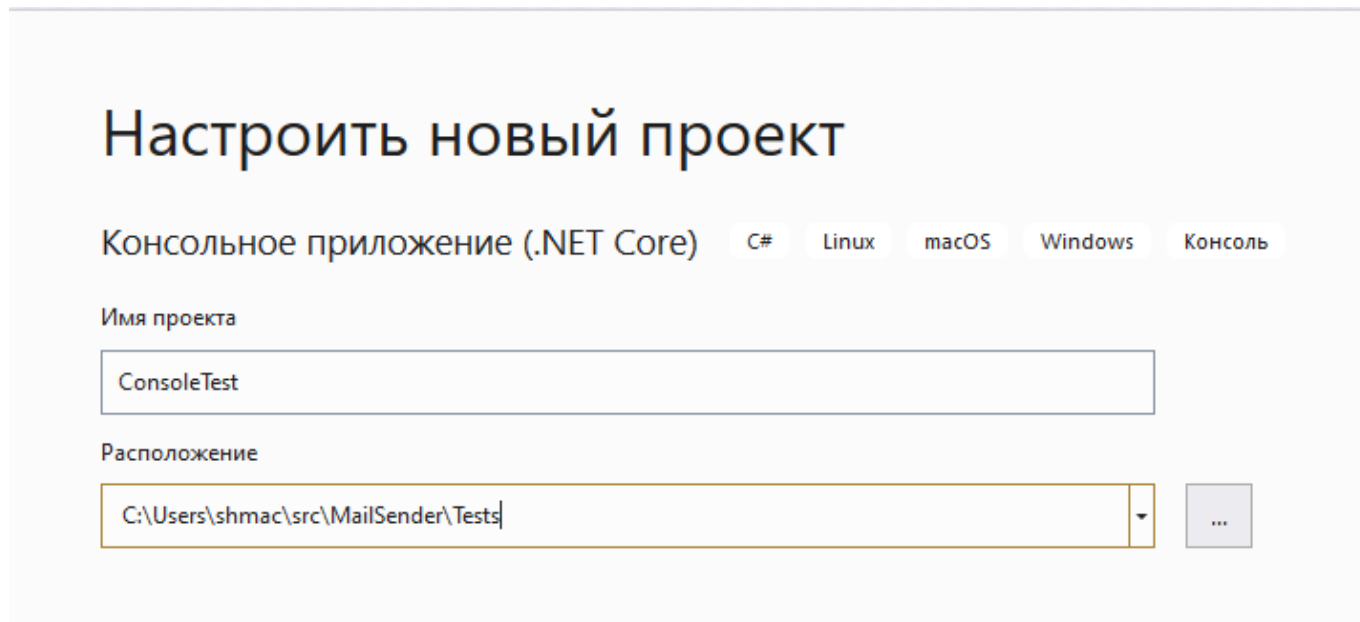
Укажем физическое место размещения проекта тестовой консоли и название проекта. Физическое размещение следует отличать от размещения проекта в решении (виртуальное размещение).



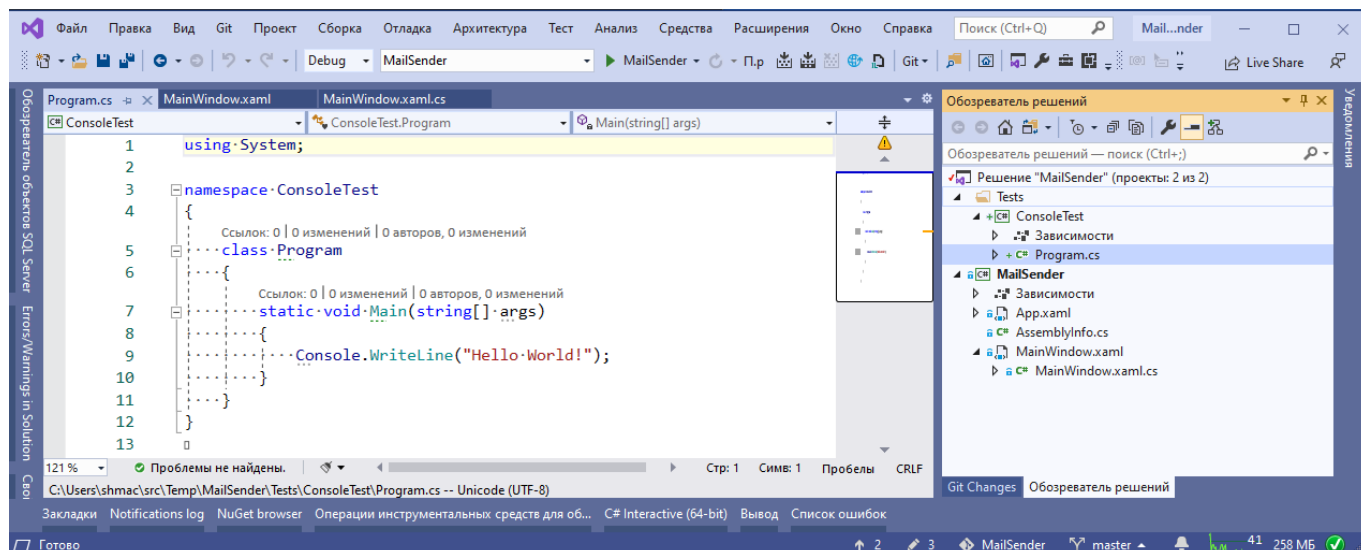
Выбираем консольное приложение *.NET Core*



Указываем название проекта и размещение - указываем, что проект будет "жить" во вложенной папке **Tests**. Папки физически на диске не существует, и она будет создана.

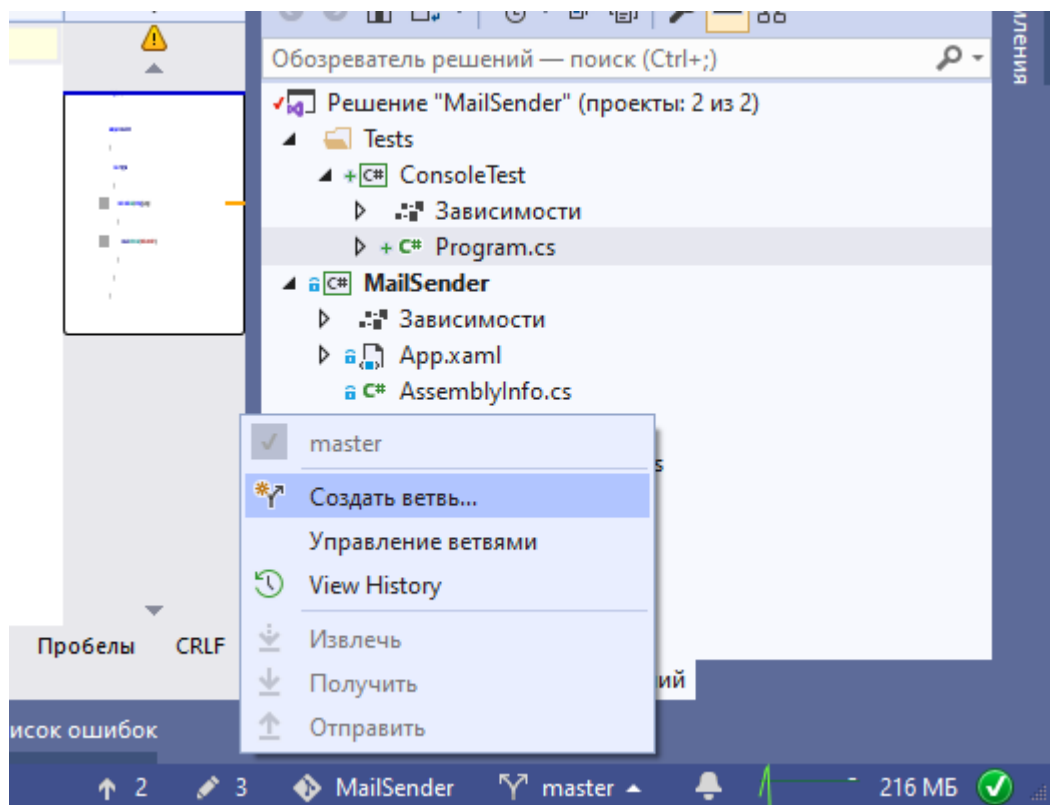


В обозревателе решений мы можем увидеть (и проконтролировать) созданный тестовый консольный проект.



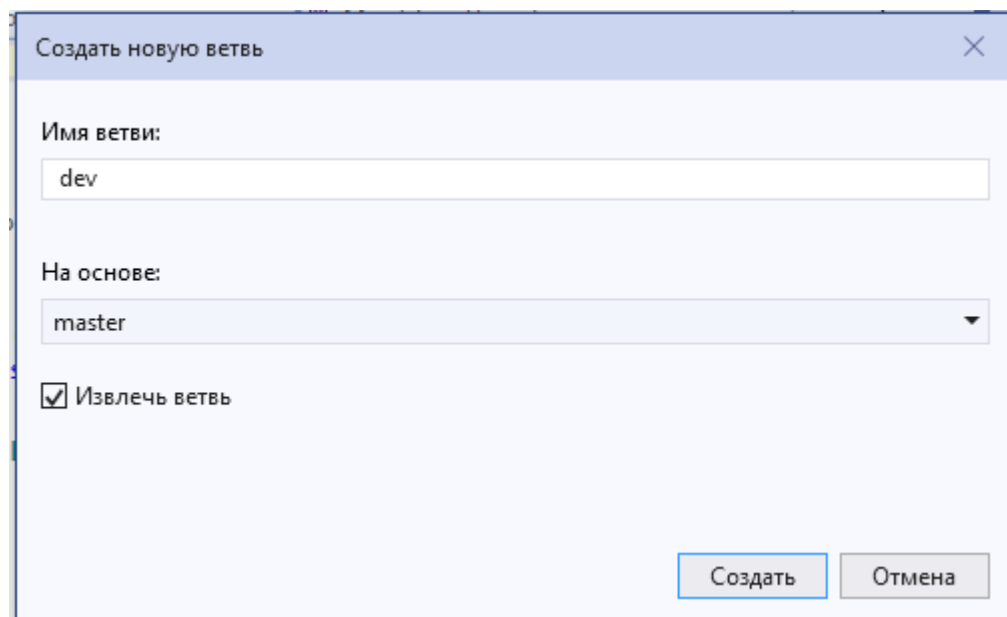
## Прототип бизнес-функции

Тестовый проект готов. В нём теперь можно экспериментировать... Давайте закрепим эти изменения в нашем репозитории. Для этого немного доработаем его структуру: создадим ветку, в которой будем вести разработку. Репозиторий GIT представляет собой граф. Каждый узел графа - это состояние репозитория - одна из версий которые система и контролирует. В любой момент мы имеем право переходить от одной вершины графа к другой. Обычно процесс разработки представляет собой линейную последовательность версий. Для упрощения этого процесса в системе *GIT* применяется понятие "ветви". Ветвь — это, по сути, ссылка на последнюю вершину из некоторой последовательности в этом графе. Обычно в системе контроля версий существует главная ветвь (часто она называется master) в которую отправляются финальные изменения релизов. Также существует ветвь в рамках которой ведётся активная разработка (часто её называют development, или dev). Создадим ветку dev. Для этого в правом нижнем углу в панели статуса окна VS найдём имя текущей ветви "master" и в меню выберем пункт "Создать ветвь...".

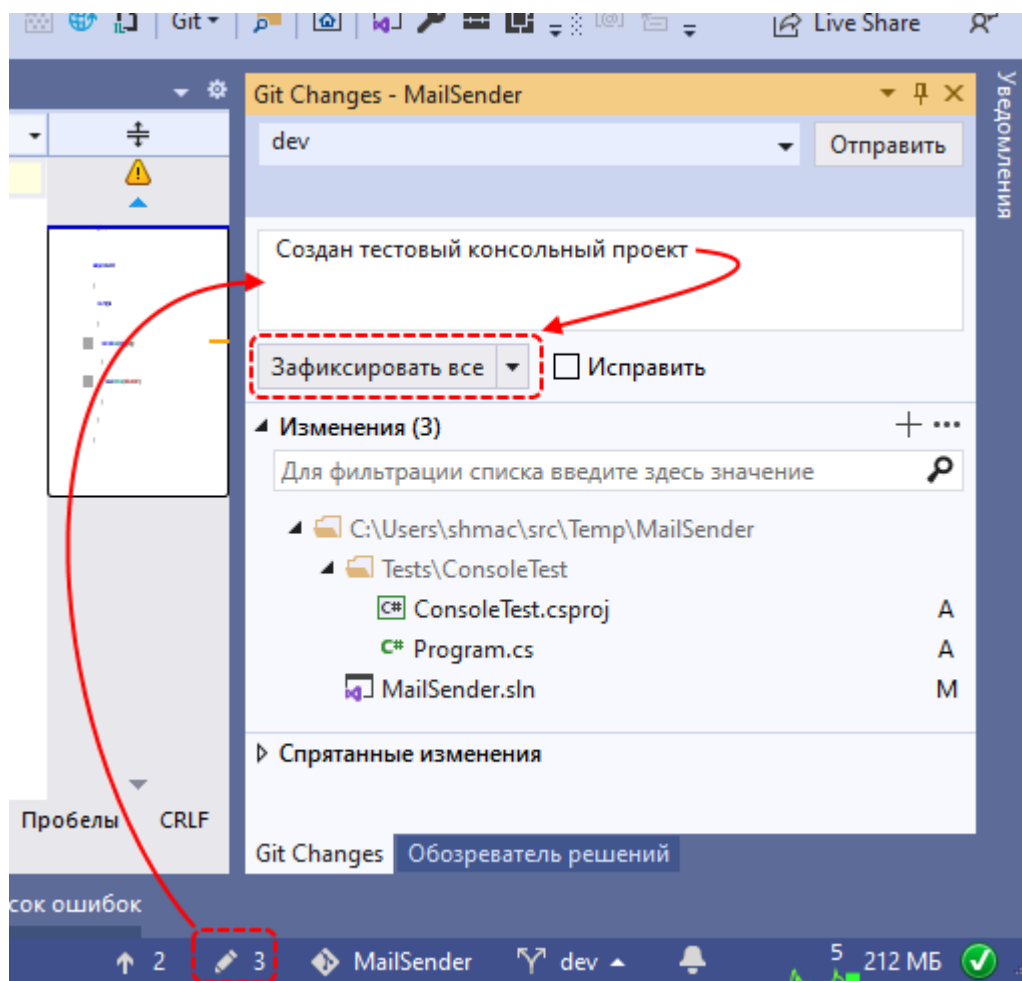




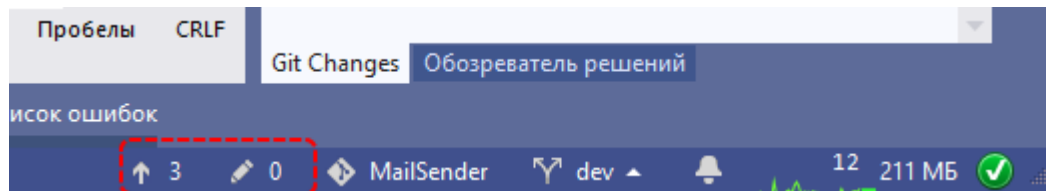
После создания новой ветви её смену можно заметить в статусной строке.



Теперь зафиксируем внесённые изменения - созданный проект-полигон.

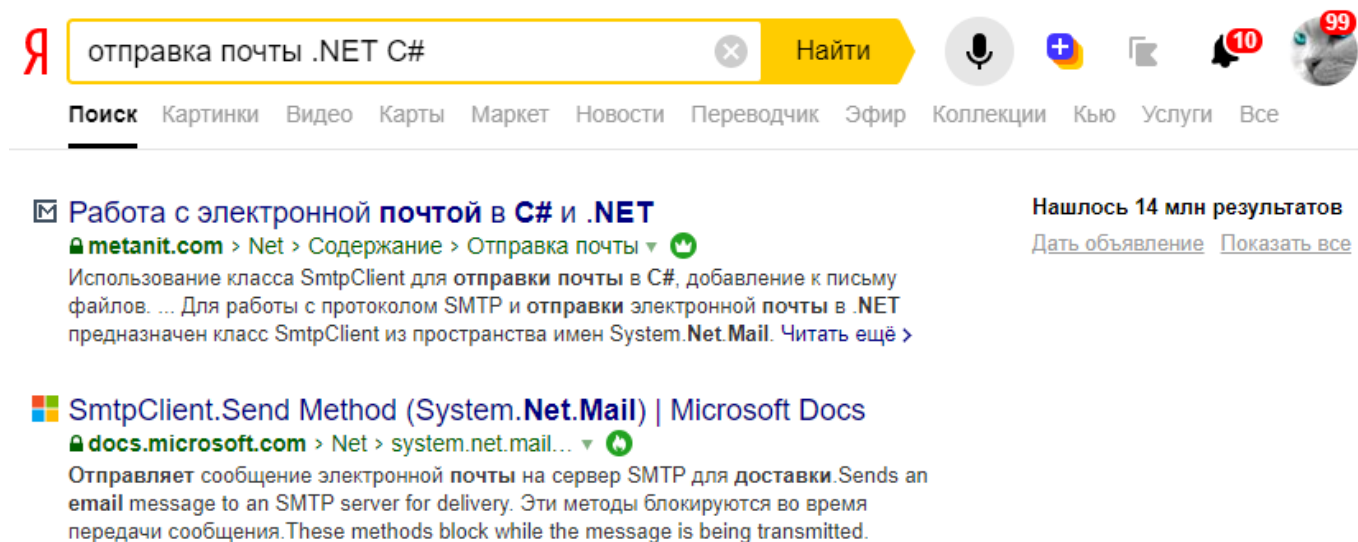


После выполнения фиксации изменений можно заметить что число файлов с изменениями стало равно 0, а число исходящих изменений увеличилось на 1.



Давайте попробуем заставить созданный консольный проект отправлять почту.

Первое с чего можно начать - спросить Яндекс "отправка почты .NET C#".



Первый же результат ведёт нас на очень хороший источник [metanit.com](https://metanit.com) на котором мы можем найти фрагмент кода, осуществляющий решение задачи отправки почты по средствам протокола SMTP.

```
using System;

// Пространства имён, содержащие нужные нам инструменты
using System.Net;
using System.Net.Mail;

namespace ConsoleTest
{
    class Program
    {
        static void Main(string[] args)
        {
            // Определим от кого и кому будем отправлять почту
            // Тут надо подставить рабочие адреса электронной почты
            var sender = new MailAddress("sender_user@yandex.com", "Sender");
            var recipient = new MailAddress("recipient_user@yandex.ru");

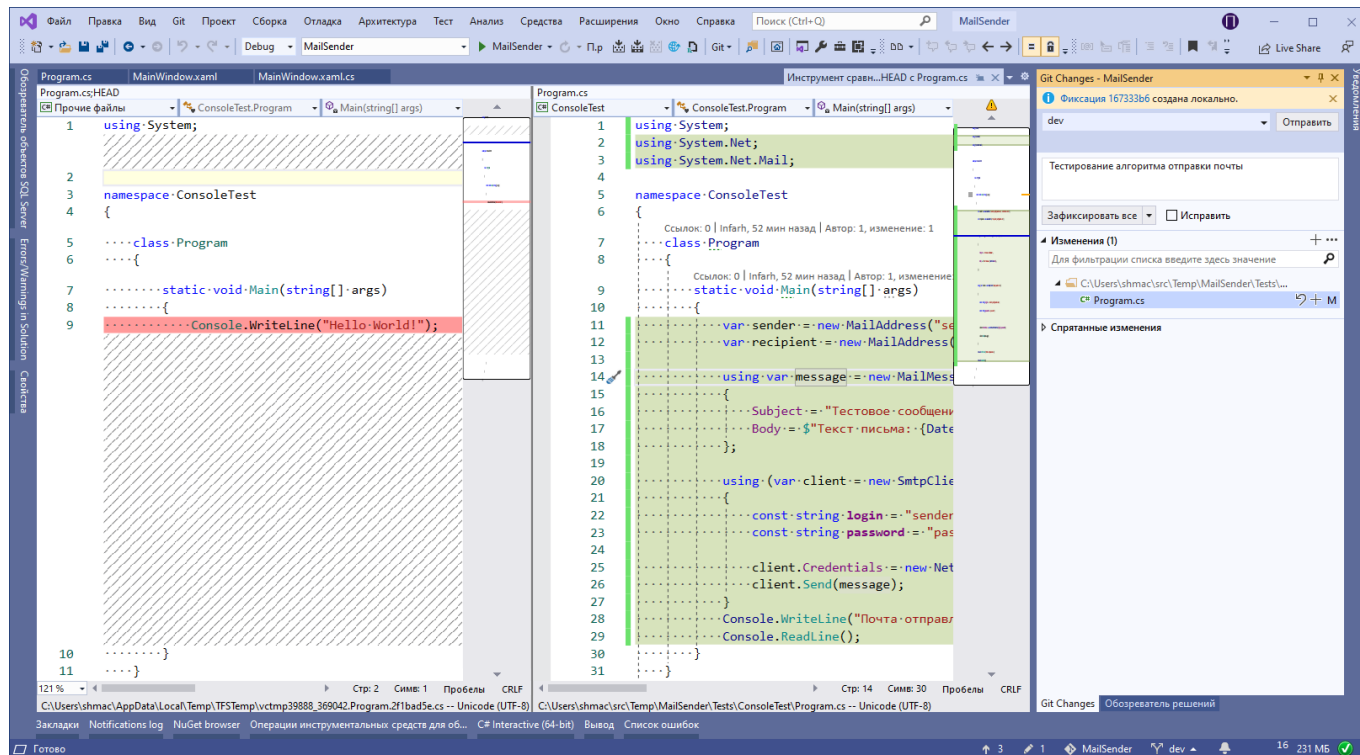
            // Создадим объект сообщения указав отправителя и адресата
            using var message = new MailMessage(sender, recipient)
            {
                // а также заполним свойства заголовка и тела сообщения
                Subject = "Тестовое сообщение",
                Body = $"Текст письма: {DateTime.Now}",
            };
        }
    }
}
```

```
// Создадим клиента для общения с почтовым сервером по протоколу SMTP
// В конструкторе надо указать адрес сервера и порт (по умолчанию 25)
using (var client = new SmtpClient("smtp.yandex.ru", 587))
{
    const string login = "sender_user@yandex.com";
    const string password = "password";

    // Обычно сервер не разрешает анонимам отправлять почту
    // Нам необходимо добавить информацию с нашими учётными данными
    client.Credentials = new NetworkCredential(login, password);
    client.EnableSsl = true;
    // После чего можем попросить клиента открыть канал с сервером
    // и отправить наше сообщение
    client.Send(message);
}
// Если всё прошло нормально и сервер принял наше сообщение
// то мы увидим на консоли соответствующую надпись.
// При указании неверного адреса сервера/порта,
// получим ошибку таймаута
// Если мы ввели неверные учётные данные, или забыли их указать
// то получим ошибку протокола подключения к серверу
Console.WriteLine("Почта отправлена");
Console.ReadLine();
}
}
```

Когда нам удалось решить главную задачу в рамках ядра логики нашего будущего приложения, мы можем попробовать создать прототип, который будет обладать уже полноценным интерфейсом.

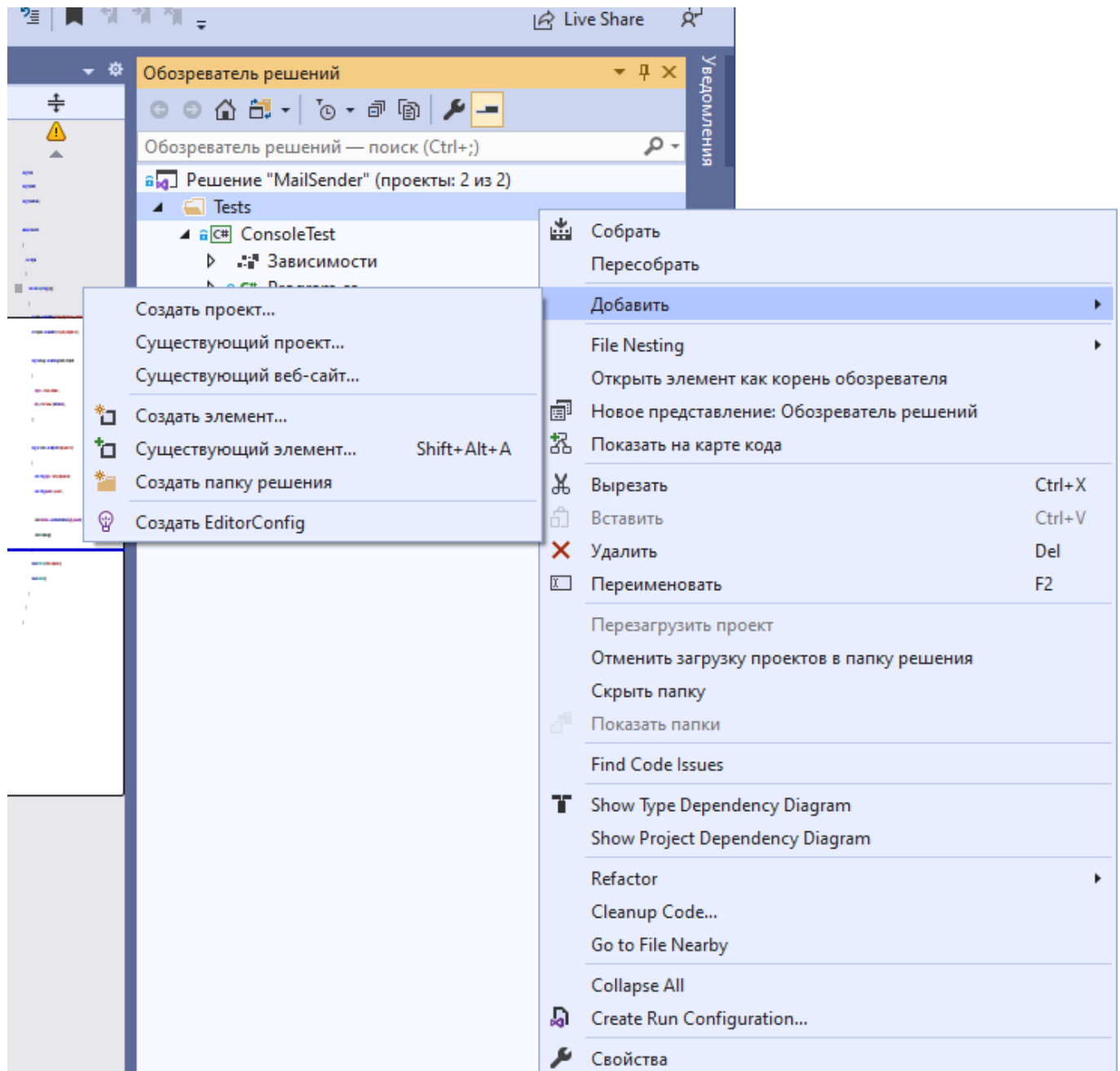
Давайте зафиксируем изменения в системе контроля версий и создадим ещё один тестовый проект, но на этот раз уже с шаблоном WPF-приложения *.NET Core 3.1* и зафиксируем изменения в репозитории.



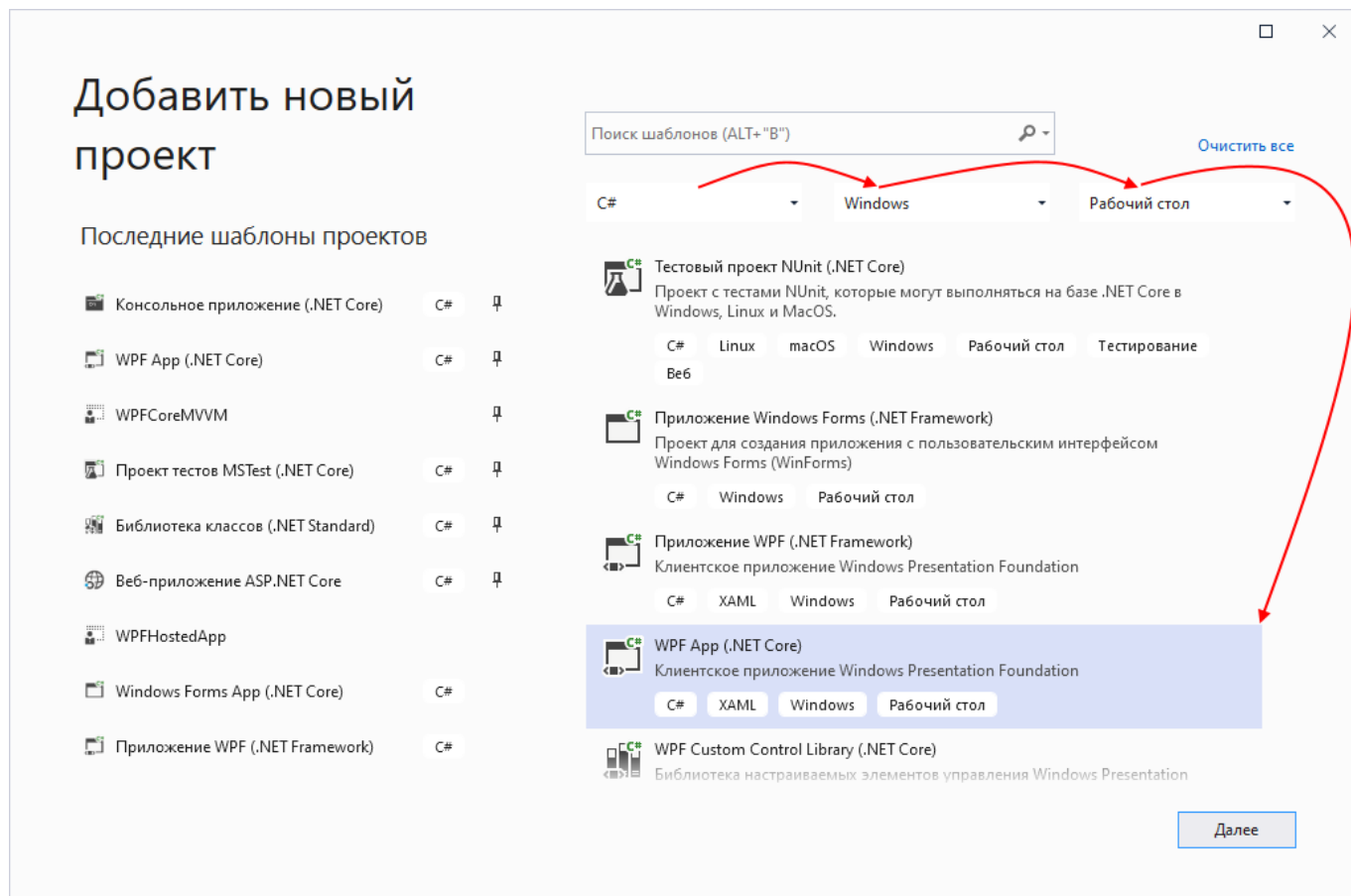
На этом этапе мы накопили уже ряд изменений в репозитории, зафиксированных локально. Их стоит отправить на удалённый сервер (синхронизироваться).

## Структура проекта WPF-приложения

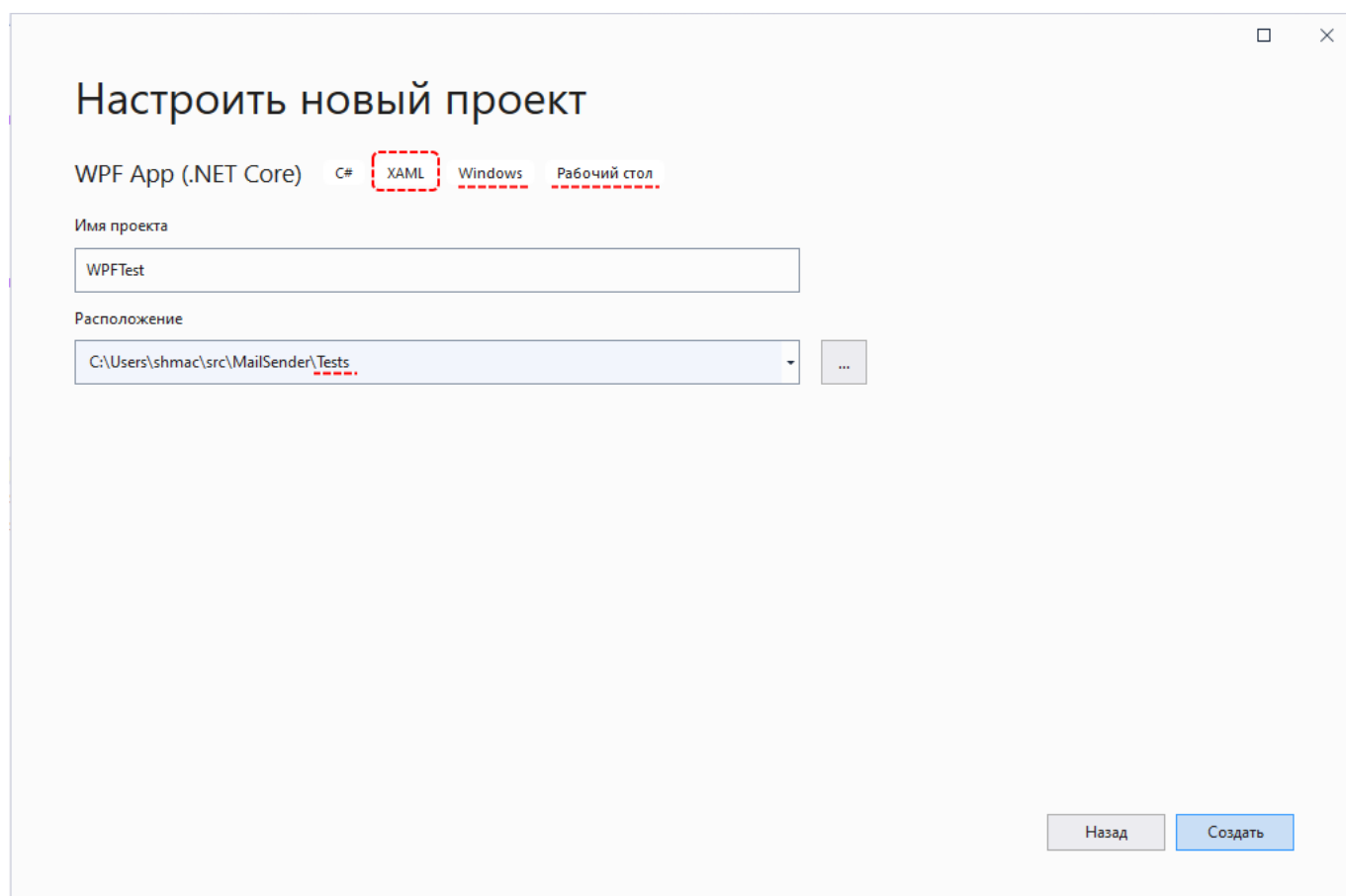
Давайте рассмотрим структуру созданного WPF-проекта и предназначение его основных элементов.



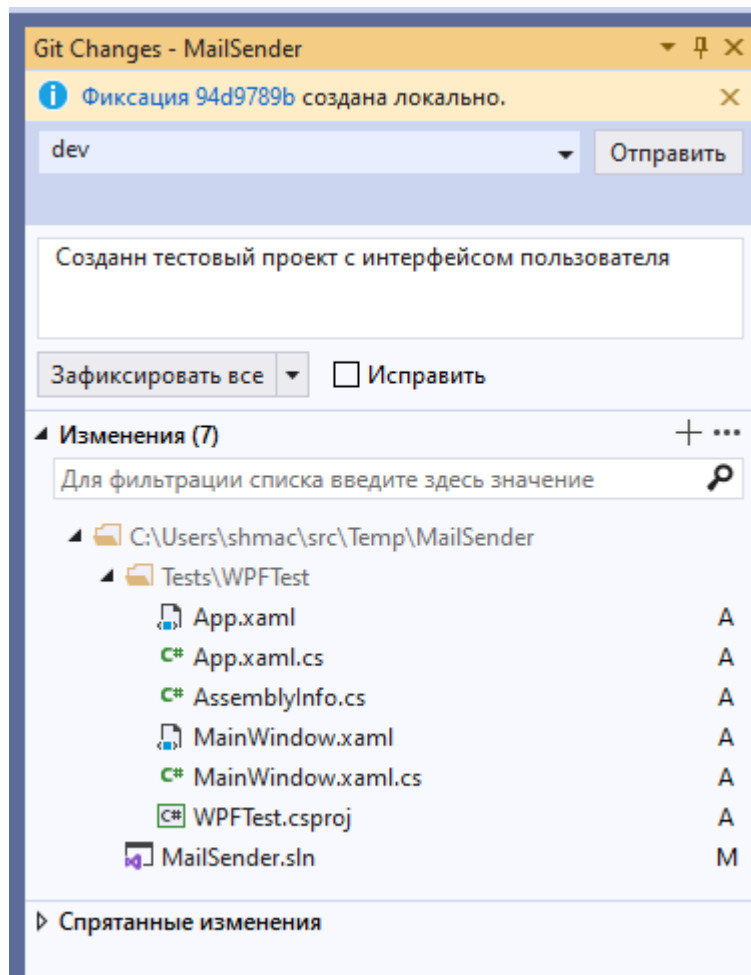
Выбираем проект WPF для платформы .NET Core



Указываем "место жительства" нашего тестового проекта по соседству с консольным проектом

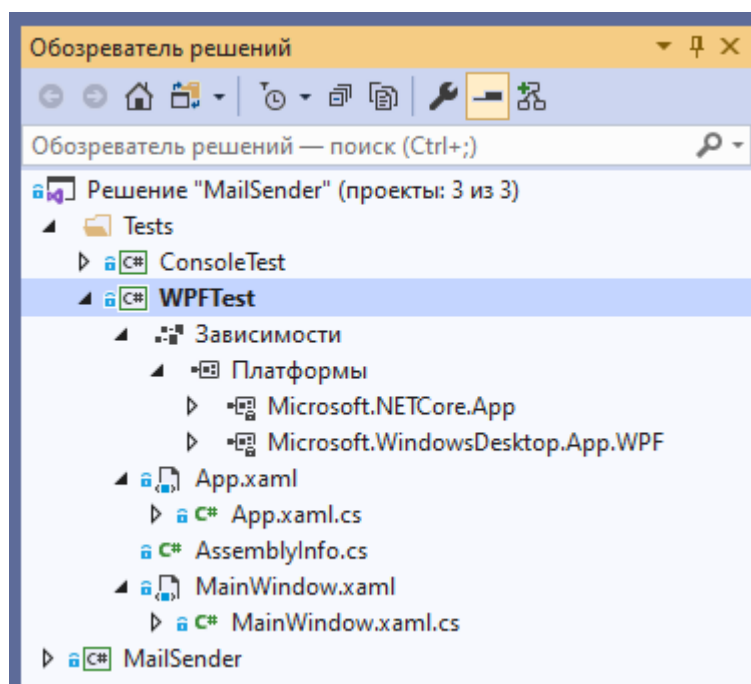


И сразу зафиксируем изменения в системе контроля версий.



## Файлы проекта WPF

В проекте находится два основных файла: `App.xaml` и `MainWindow.xaml`. Они представляют собой разметку нашего приложения. К каждому из них создан парный файл "Code-behind" - файл прилагаемого кода, или файл обработчиков событий: `App.xaml.cs` и `Mainwindow.xaml.cs`. Кроме того, здесь у нас присутствует файл `AssemblyInfo.cs` и раздел зависимостей нашего приложения.



- Зависимости приложения определяют ссылки на те внешние ресурсы, которые будут использованы нами. Обычно это либо дополнительные библиотеки с кодом, которые мы будем использовать, либо целые пакеты библиотек и ресурсов, которые также могут нам понадобиться. По умолчанию к нашему проекту сразу подключаются пакеты *Microsoft.NETCore.App* и *Microsoft.WindowsDesktop.App.WPF*, обеспечивающий нас инструментами для создания настольного WPF-приложения.

## AssemblyInfo.cs

- Файл **AssemblyInfo.cs** - это необязательный файл, определяющий атрибуты сборки (информация о версии, авторе, компании и т.п.). Его можно смело удалить, а всю информацию перенести в файл проекта **WPFTest.csproj**

## App.xaml

- Файл **App.xaml** определяет разметку (структуру) всего нашего приложения. Его расширение говорит о том, что внутри находится содержимое, определяемое как разметка в формате XAML (*eXtensible Application Markup Language*) - язык-наследник XML.

```
<Application x:Class="WPFTest.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WPFTest"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Разметка представляет собой древовидную структуру вложенных друг в друга элементов. Каждый элемент можно рассматривать как скобки с именем. Внутри одной "скобки" может находиться несколько других. Но каждой открывающей **<Application>** должна быть сопоставлена соответствующая закрывающая **</Application>**. Внутри элемента может не быть данных вообще: **<TextBlock></TextBlock>**. В этом случае запись можно упростить: **<TextBlock/>**. В открывающем элементе могут присутствовать атрибуты **<TextBlock Foreground="Red">Hello World</TextBlock>**. Идея XAML-разметки заключается в том, что каждый элемент в памяти программы представляет собой объект класса, имя которого указано как имя элемента. То есть если мы пишем **<Grid><TextBlock Text="Hello World!"/></Grid>**, то при анализе этой разметки будет создан объект Grid и ему в дочерние объекты будет добавлен созданный объект класса TextBlock. Причём у последнего будет установлено его свойство Text = "Hello World!".

Помимо обычных атрибутов у элемента могут быть указаны специальные атрибуты (см. атрибуты XML) **xmlns**, определяющие пространства имён внутри разметки. Атрибут **xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"** говорит о том, что все элементы разметки будут иметь пространство имён по умолчанию "http://schemas.microsoft.com/winfx/2006/xaml/presentation" - это пространство имён XAML. Обычно эти интернет-адреса ведут в никуда. Это просто строка текста, используемая анализатором. Но изредка по



этому адресу находится некоторое описание, или адрес компании, предоставляющей используемый код. В данном случае по умолчанию для всех элементов разметки применяется пространство имён разметки XAML. Также с помощью атрибутов `xmlns:...="..."` устанавливаются псевдонимы для нужных пространств имён. К примеру, здесь применяется псевдоним "x":

`xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"`, устанавливающий связь с пространством имён инструментов разметки. С его помощью указывается что атрибут разметки `Application` будет ассоциирован с классом `App`, описанным в нашем приложении:

`x:Class="WPFTest.App"`. Также здесь добавляется псевдоним `local` для пространства имён, используемого в нашем коде платформы: `xmlns:local="clr-namespace:WPFTest"`. Для этого применяется ключевое слово `clr-namespace`, которое ассоциирует псевдоним `local` с нашим пространством имён `WPFTest`. Это сделает возможным использование в разметке классов, описанных нами в этом пространстве имён. Свойство `StartupUri` указывает что будет запущено как главное окно нашего приложения. Тут можно указать путь к файлу разметки. Также в элементах разметки помимо установки значений свойств и определения псевдонимов пространств имён можно "подписываться" на события, которые могут возникать в выбранном элементе. При этом обработчиком события указывается метод, который должен быть описан в файле прилагаемого кода.

### App.xaml.cs

- Файл `App.xaml.cs` - файл прилагаемого кода для разметки приложения.

```
using System.Windows;

namespace WPFTest
{
    public partial class App : Application
    {
    }
}
```

Если присмотреться, то можно заметить, что класс `App`, объявленный в это файле, наследуется от класса `Application`. На самом деле это объявление в данном случае избыточно и для компилятора неинформативно по той причине, что в разметке мы этот факт уже указали:

```
<Application x:Class="WPFTest.App"
    ...>
...
</Application>
```

Кроме того, здесь видно, что класс объявлен как `partial`. Для компилятора это означает, что класс объявлен в этом и ещё в нескольких файлах. Это сделано для того, чтобы компилятор при сборке приложения проанализировал разметку `xaml` и на её основе в папке `/obj` создал файл `app.xaml.g.cs` с кодом на C#, который воспроизводит логику `xaml`-файла. И в нём будет определена вторая часть класса `App`, содержащая логику установки значений свойств, подписки на события и т.п.

## MainWindow.xaml

- Файл **MainWindow.xaml** представляет собой описание разметки главного окна. Идея его разметки полностью соответствует идее разметки файла **App.xaml** за исключением того, что корневым элементом служит **<Window> ... </Window>** - то есть класс, формируемый на основе данного файла разметки будет наследником класса **Window**.

```
<Window x:Class="WPFTest.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WPFTest"
        mc:Ignorable="d"
        Title="MainWindow"
        Width="800" Height="450">
    <Grid>

    </Grid>
</Window>
```

Здесь подключены пространства имён:

- xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"** - инструменты разметки
- xmlns:d="http://schemas.microsoft.com/expression/blend/2008"** - специальные инструменты разметки времени разработки
- xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"** - средство, позволяющее исключить неудобные пространства имён при компиляции **mc:Ignorable="d"** - пространство имён **d** при компиляции будет проигнорировано.

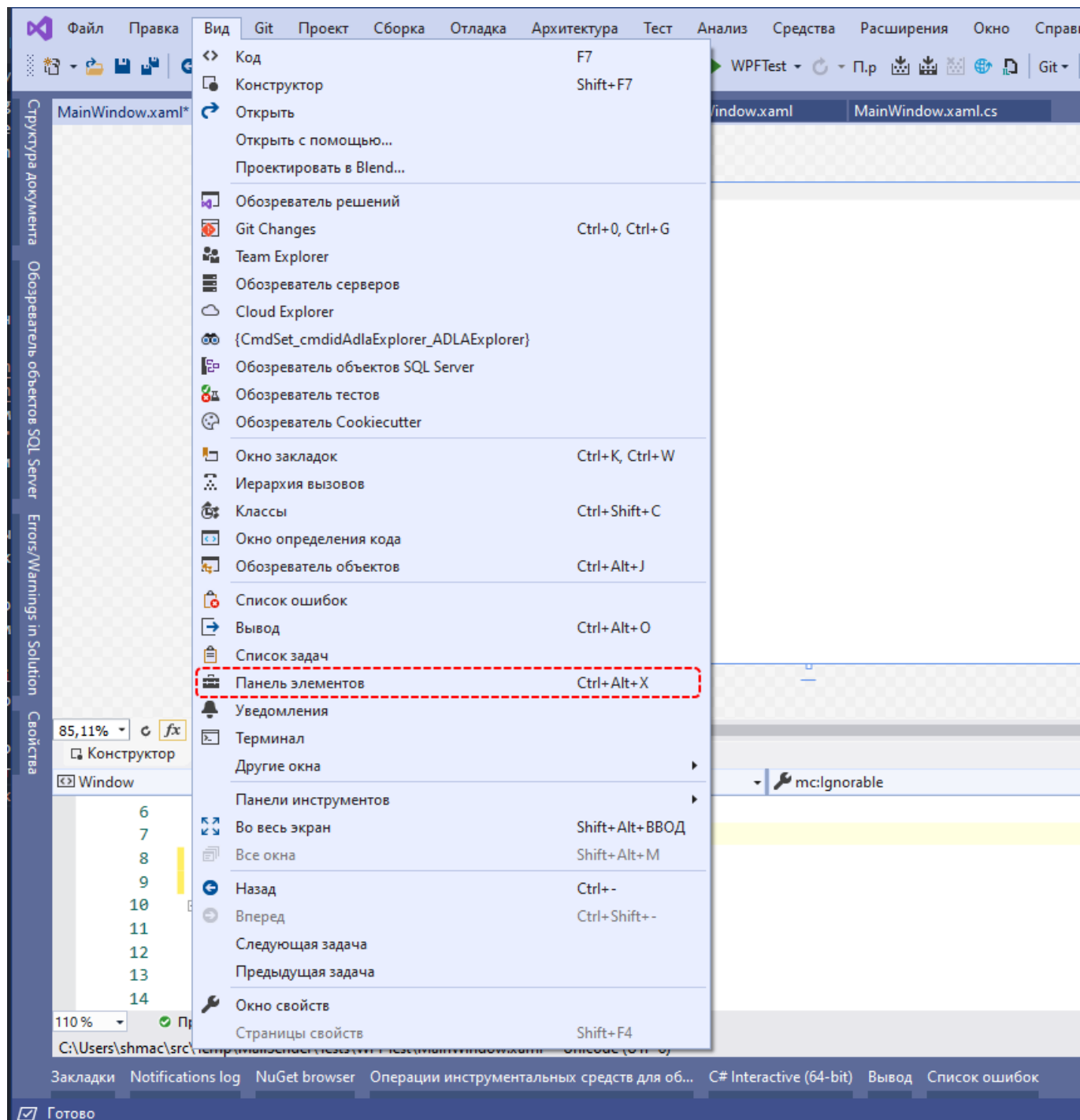
Также в открывающемся теге указывается какой текст будет в заголовке окна и какие будут его размеры (800x450).

Внутри окна размещается объект **<Grid></Grid>** - панель, в которую можно "уложить" нужные нам элементы разметки пользовательского интерфейса.

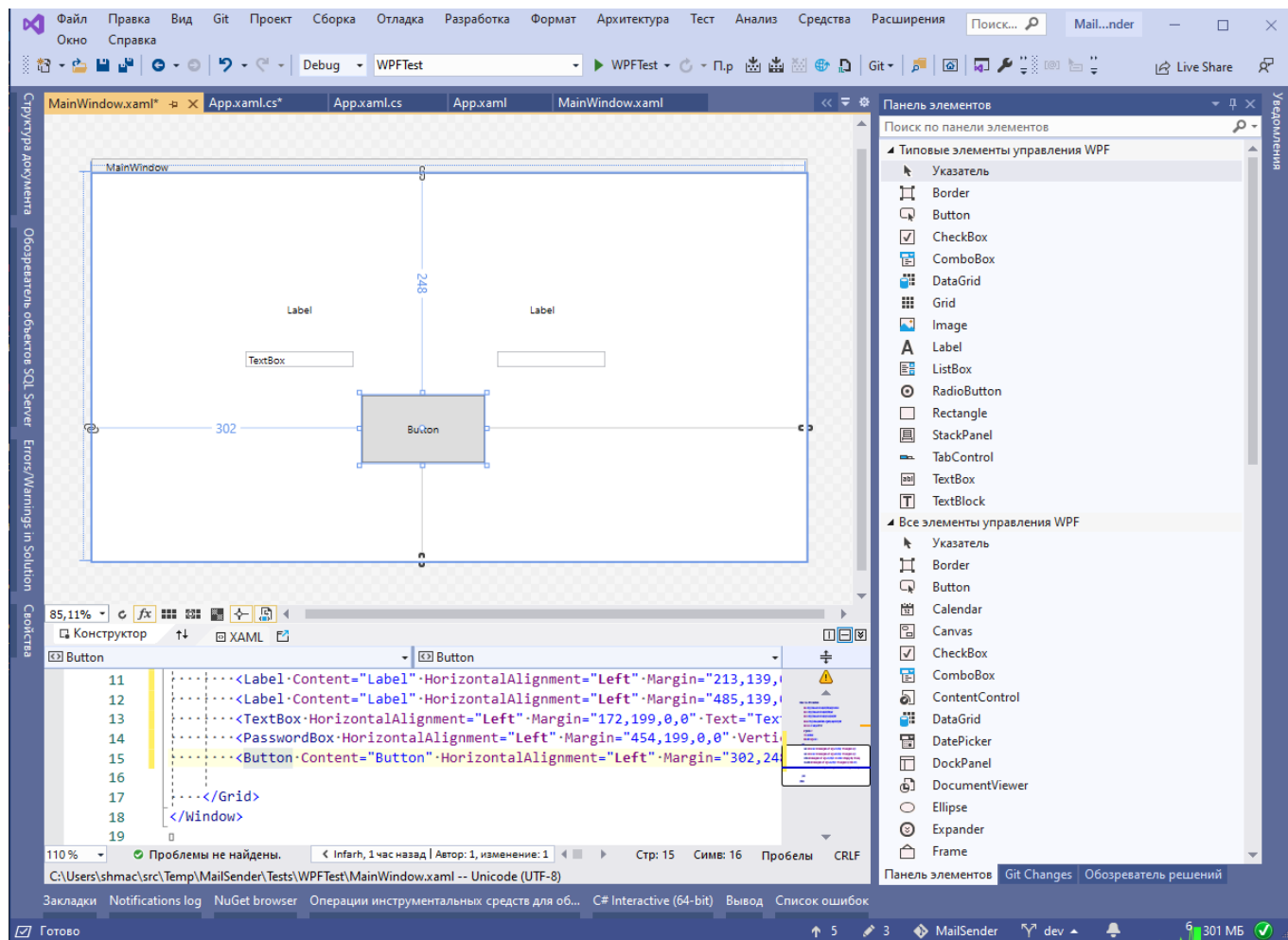
## MainWindow.xaml.cs

- Файл **MainWindow.xaml.cs** предоставляет код на C# второй части описания класса главного окна программы в котором может быть определена логика его работы и обработчики событий.

Визуальный дизайнер интерфейса

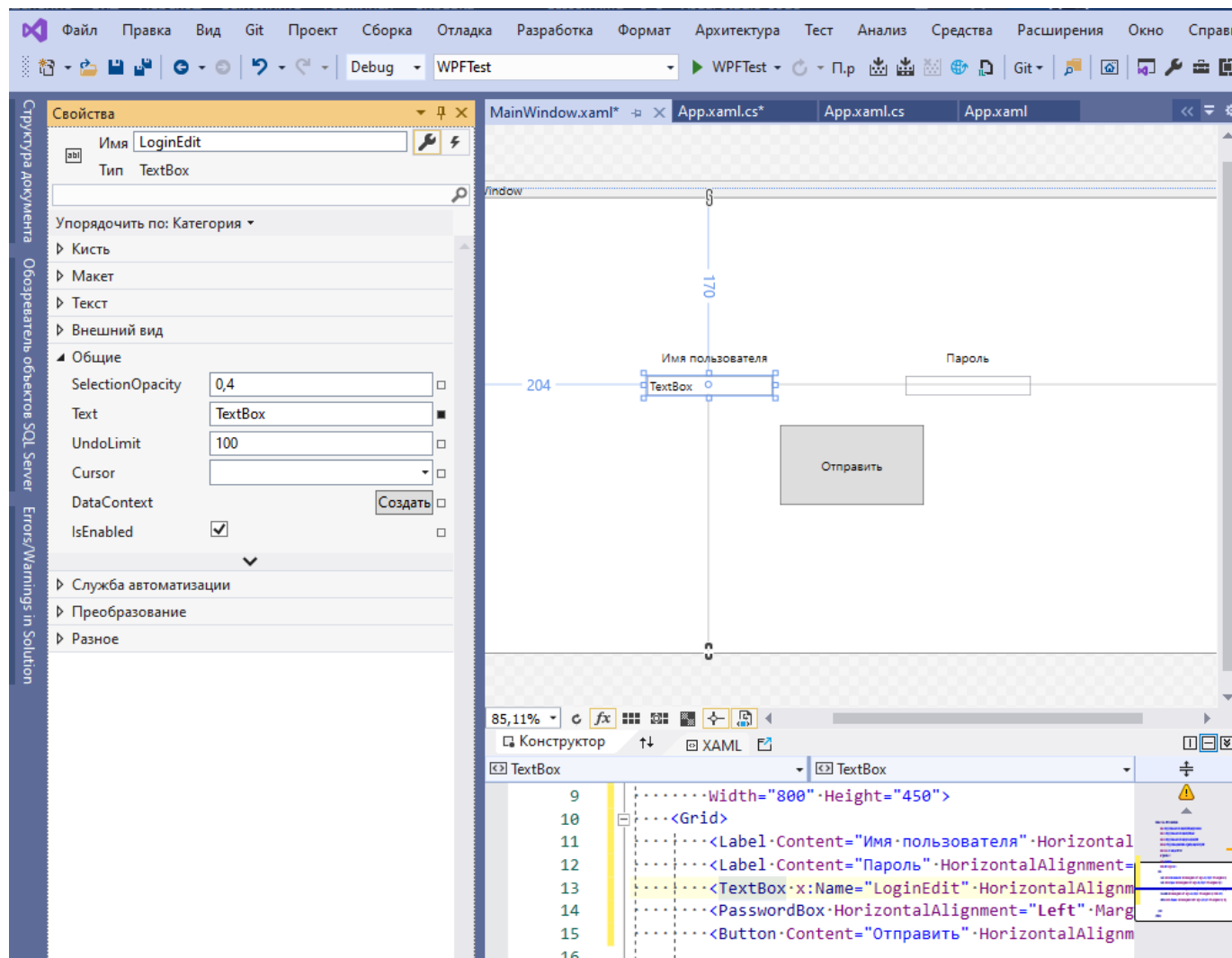


Давайте откроем "Панель элементов" Visual Studio и мышкой перетянем на поверхность окна пару текстовых меток (**Label**), поле для ввода текста (**TextBox**), поле для ввода пароля (**PasswordBox**) и кнопку (**Button**). А дальше, используя панель "Свойства" и поочередно выбирая каждый из добавленных элементов, настроим их параметры так, чтобы они отвечали задаче ввода имени пользователя и пароля и отправки сообщения. Для "метки" (**Label**) свойство, которое задаёт её содержимое, будет **Content**. Для кнопки - тоже самое. Свойство **Text** определяет содержимое для поля ввода текста (**TextBox**), а **Password** - пароль для (**PasswordBox**).



В панели "Свойства" в самом верху есть место для ввода имени элемента управления. Так как в последствии в коде нам потребуются данные, которые хранятся в поле ввода текста и пароля, то этим двум элементам управления необходимо задать имя.

### Свойства визуальных элементов



Если присмотреться к происходящему, то можно заметить, что любые изменения в визуальной части конструктора окна, или в панели "Свойства" приводят к немедленным изменениям в области разметки окна. На самом деле область разметки окна является главной по отношению к его визуальной части, и визуальная часть лишь повторяет (визуализирует по мере возможности) то, что описано в разметке. Таким образом, накопив небольшой багаж опыта работы с разметкой Вы заметите, что писать разметку интерфейса в коде гораздо быстрее и эффективнее, чем пользоваться визуальным дизайнером и панелями элементов и свойств. Визуальная часть удобна на первых порах пока отсутствует опыт и знания инструментария. Кроме того, визуальный дизайнер очень удобен для быстрой навигации по интерфейсу. Для этого удобно пользоваться колёсиком мышки и клавишей Ctrl позволяющей изменять масштаб. А щелчок мышкой по интересующему визуальному элементу тут же перебросит курсор ввода в нужную часть разметки.

### События визуальных элементов

Теперь у нас есть визуальная часть интерфейса. Нужно "объяснить" теперь нашему приложению что делать, когда пользователь щёлкает мышкой по кнопке. В момент щелчка кнопка генерирует событие **Click**. И нам необходимо добавить обработчик этого события. Есть три способа это сделать:

1. Можно в дизайнера щёлкнуть дважды мышкой по кнопке, и это заставит Студию создать обработчик события для "главного события" этого визуального элемента. Тоже самое можно сделать и с другими визуальными элементами. Для кнопки главным событием является событие **Click**;

2. Можно открыть панель свойств и в верхнем её правом углу найти кнопку с "молнией". Это переключит панель на редактирование обработчиков событий выбранного элемента управления. Здесь можно ознакомиться (вспомнить, посмотреть) какие события умеет генерировать тот, или иной элемент управления. К каждому событию здесь можно дописать имя обработчика и нажать **Enter**, либо просто дважды кликнуть мышкой по пустому полю. Это также заставит Студию сгенерировать обработчик, но уже для выбранного события.
3. В разметке в открывающем теге выбранного элемента (в нашем случае кнопки) можно указать интересующее событие (в нашем случае **Click**) и установить его обработчик. В случае отсутствия метода с указанным именем Студия предложит его автоматически сгенерировать.

В созданный средой разработки обработчик события перенесём код из нашего консольного приложения.

```
using System;
using System.Net;
using System.Net.Mail;
using System.Windows;

namespace WPFTest
{
    public partial class MainWindow : Window // Излишнее указание на базовый класс
    {
        public MainWindow() // Конструктор окна
        {
            // Метод, который по разметке и воссоздаёт внешний вид
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            var mail_sender = new MailAddress(
                "mail_sender_user@yandex.com",
                "Sender");
            var mail_recipient = new MailAddress("recipient_user@yandex.ru");

            using var message = new MailMessage(mail_sender, mail_recipient)
            {
                Subject = "Тестовое сообщение",
                Body = $"Текст письма: {DateTime.Now}",
            };

            using (var client = new SmtpClient("smtp.yandex.ru", 587))
            {
                // Забираем из интерфейса строку с именем пользователя
                var login = LoginEdit.Text;
                // Забираем из интерфейса строку с паролем
                // и полчаем её в открытом виде
                //var password = PasswordEdit.Password;
                // Пароль можно забрать в защищённом виде. И это будет хорошо.
                var password = PasswordEdit.SecurePassword;
```

```

        client.Credentials = new NetworkCredential(login, password);
        client.EnableSsl = true;
        client.Send(message);
    }

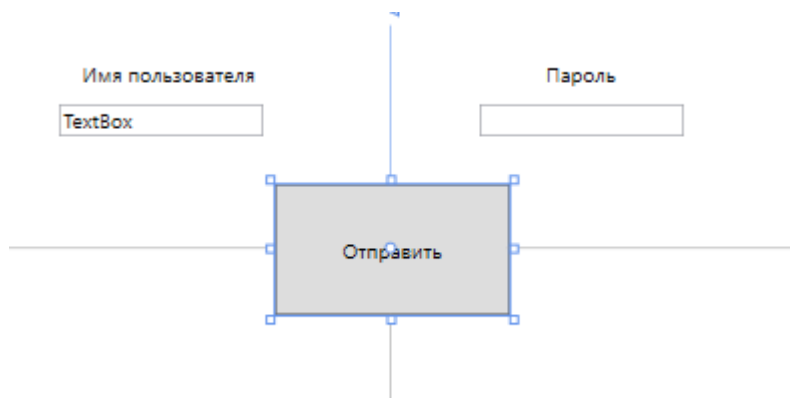
    // Если всё прошло нормально, то выводим сообщение в диалоговом окне
    MessageBox.Show(
        "Почта отправлена", // Текст окна
        "Отправка почты"); // Заголовок окна
    }
}

```

В результате мы получили прототип приложения, обеспечивающий отправку почты на заданный нами в коде адрес электронной почты через почтовый сервер. А имя пользователя и пароль вводится из оконного интерфейса.

## Различие WinForms и WPF

Концептуально, сделанное нами, не сильно отличается от процесса разработки ПО с применением технологии *WinForms*. Здесь также присутствует визуальный дизайнер окна, панели элементов и свойств, файл с кодом обработчиков событий. И даже поведение визуального дизайнера интерфейса схоже: любые действия в дизайнере отражаются им в генерируемый код. Только в *WinForms* этот код пишется дизайнером на C#, а в *WPF* - формируется разметка. Концептуальной разницей между технологиями является основа их движка. Если в основе *WinForms* лежит набор вызовов к ядру операционной системы Windows - библиотекам *WinAPI* (*user32.dll*, *kernel32.dll*), с помощью которых ОС рисует изображение на экране и обрабатывает реакцию пользователя (щелчки мышкой, ввод клавиатуры и т.п.). В *WPF* в основе визуального движка лежит не *WinAPI*, а *DirectX*. Все визуальные элементы платформы были с нуля переписаны на основе *DirectX* и их изображение формируется не CPU, а графическим устройством компьютера. Это существенно образом положительно влияет на производительности и качестве формируемой картинки, позволяет делать сложные визуальные эффекты (вроде размытия, поворотов, наклонов), и анимации.



При этом, визуальные трансформации могут быть определены в разметке органичным образом:

```

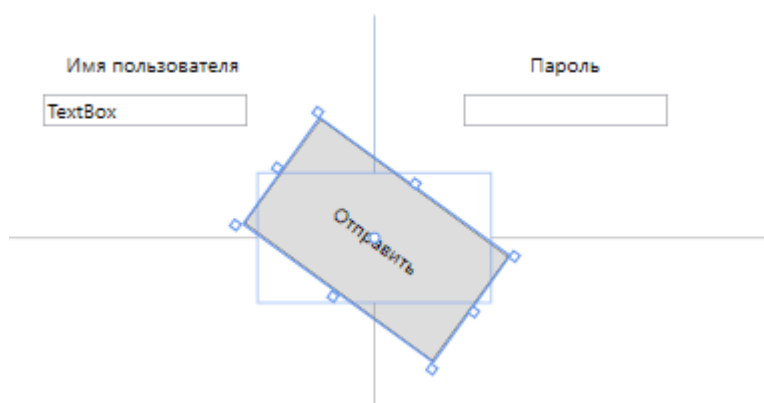
<Button Content="Отправить"
        HorizontalAlignment="Center" VerticalAlignment="Top"

```

```

Margin="0,217,0,0"
Width="137" Height="76"
RenderTransformOrigin="0.5,0.5"
Click="Button_Click">
<Button.RenderTransform>
  <TransformGroup>
    <ScaleTransform/>
    <SkewTransform/>
    <RotateTransform Angle="36.231"/>
    <TranslateTransform/>
  </TransformGroup>
</Button.RenderTransform>
</Button>

```



Но это не главное отличие *WPF* от *WinForms*. Главным отличием является язык разметки *XAML*. Его применение позволяет использовать особую магию, которая называется "привязкой к свойствам объекта". Её идея заключается в том, что два объекта, расположенные в памяти приложения и обладающие некоторым набором свойств, можно связать так, что изменении значения свойства одного объекта значение связанного свойства второго объекта будет автоматически установлено в значение, которое было установлено для свойства первого объекта. То есть, предположим, у Вас есть объект "*Студент*" со свойством "*Имя*" и есть поле ввода на поверхности окна (`<TextBox/>`). Можно выполнить привязку свойства `Text` у поля ввода к свойству `Name` у объекта "*Студент*". И в этом случае при вводе данных в поле ввода свойство имени студента будет автоматически изменено. Это же работает и в обратную сторону. Если вдруг по каким-то причинам "*Студент*" изменит своё имя, то поле ввода это обнаружит и визуализирует новое значение имени. Данная концепция привязки позволяет реализовать более простым способом особые шаблоны проектирования приложений, упростив тем самым организацию его логики и кода.

## Проектирование приложения

И так, на данном этапе у нас готов прототип - приложение, реализующее главную бизнес-функцию. То, ради чего затевается весь процесс. Прототип позволяет отработать базовую логику и показать возможность реализации всей программной системы в целом. Но он не должен служить фундаментом для последующего развития всей программной системы. Если продолжить развитие прототипа добавляя ему новые и новые функции, то рано, или поздно он "рухнет под собственным весом". Количество исправлений, время, которое надо на них потратить, время, которое надо потратить на то чтобы разобраться что тут происходит и как это работает, начнёт превышать время на добавление



новой функции. И дальнейшая поддержка и развитие данного проекта станет нецелесообразным. Будет проще всё взять и написать с нуля.

Мы будем использовать инженерный подход к разработке ПО. В его основе находится два основных процесса: "*анализ*" и "*синтез*".

На этапе анализа, с которого мы обязаны(!) начать, нам необходимо определить:

- Какова цель создания ПО?
- Набор реализуемых функций
- Где, кем и в каких условиях будет эксплуатироваться ПО?

## Главные этапы

Разработка программного обеспечения должна идти по заранее запланированному сценарию. Цикл разработки как всего программного комплекса, так и отдельных его элементов должен начинаться с анализа требований к разрабатываемому модулю, условий его функционирования. Далее может следовать этап прототипирования на котором исследуются возможные варианты решения поставленной задачи. После этого следует этап непосредственно разработки завершаемый этапом тестирования сформированной функциональности. Последним этапом цикла идёт развёртывания, на котором осуществляется внедрение созданной функциональности в то место, в которое подразумевалось внедрение. После этого цикл повторяется до полного удовлетворения заказчика.

Давайте составим для самих себя некоторое подобие технического задания.

### 1. Целью создания нашего приложения будут:

1. Обучение. Научиться использовать инструментарий *WPF*, а также средства и методы разработки
2. Заложить основу проекта, который может стать частью Вашего портфолио
3. Создать проект-библиотеку примеров кода, которую в последствии можно использовать при создании других проектов

### 2. Приложение должно обеспечивать:

1. Уметь отправлять почту указанному адресату
2. Иметь возможность выбирать получателя из заранее определённого перечня
3. Управлять перечнем получателей, отправителей и серверов для рассылки
4. Уметь автоматизировано рассылать почту по расписанию
5. Давать возможность управлять расписанием рассылки
6. Иметь возможность управлять перечнем рассылаемых сообщений
7. Вести статистику рассылки
8. Ведение журналов

### 3. Условиями работы нашего приложения определим:

1. Компьютер с процессором архитектуры x86, x64, 1ГБ RAM, ОС Windows 7 SP2 и старше
2. Хранение данных перечней адресов рассылки, писем и расписания в БД, либо в *XML/Json*-файлах

## Архитектура

В основе процесса проектирования программного обеспечения лежит понятие его архитектуры. Одним из видов архитектуры приложения, обеспечивающая расширяемость, простоту тестирования и

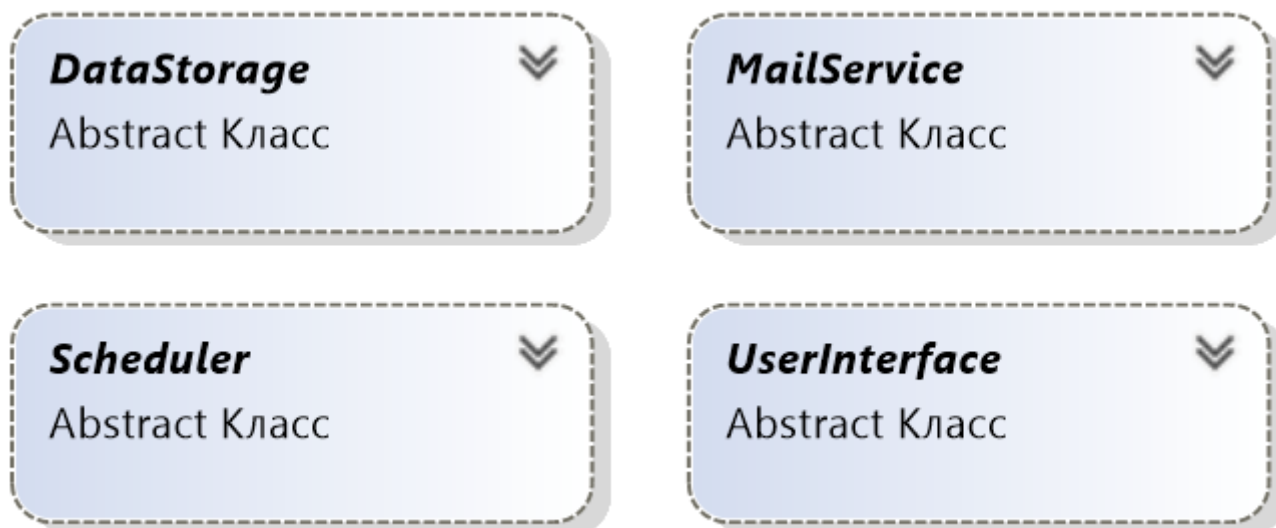
распределение обязанностей при разработке является слоистой структурой, где каждый слой максимально изолирован от остальных.

В нашем будущем приложении помимо основной бизнес-функции отправки почты нам потребуется область кода, обеспечивающая хранение и управление данными и визуальная часть - пользовательский интерфейс. Таким образом мы можем выделить слой данных, слой бизнес-логики и слой пользовательского интерфейса.



Выделим основные структурные единицы приложения и очертим его границу: всё что находится внутри периметра, является предметом нашей ответственности. Всё что находится во вне - обеспечивается инфраструктурой окружения в котором наше приложение будет работать.

## Сервисы



В структуру нашего приложения войдёт следующие модули (сервисы):

1. Сервис хранения данными и управления ими
2. Сервис отправки почты
3. Сервис планировщика рассылки
4. Интерфейс пользователя

Снаружи нашего приложения будут:

1. База данных (место, где будут жить наши данные)

2. Средства хранения конфигурации нашего приложения
3. Служба точного времени ОС, которой будет пользоваться планировщик
4. Сетевая инфраструктура, обеспечивающая соединение с почтовым сервером

На основе всего вышеописанного можно нарисовать схему приложения. Эту схему легче всего создать на основе определения языка *UML*-диаграммы классов.

## Эскиз интерфейса

После этого можно сформировать эскиз интерфейса пользователя.

Если Вы не обладаете опытом работы в векторных графических редакторах, то наиболее эффективным методом создания эскиза будет использование листа бумаги и простого карандаша.

На листе бумаги нарисуем прямоугольник, олицетворяющий форму нашего главного окна. Отчеркнём в нём заголовок окна, напишем в нём текст заголовка и нарисуем системные кнопки закрытия, свёртывания и развёртывания. Также вверху окна предусмотрим главное меню, а внизу - панель статуса.

Далее - необходимо задуматься об основных выполняемых функциях нашего приложения. Что оно должно делать, и как пользователь сможет взаимодействовать с этими функциями?

В нашем приложении есть 4 основных раздела: управление списками рассылки и серверов, планировщик заданий, редактор писем и статистика.

Представим каждую функцию в виде отдельной вкладки. И нарисуем эскиз одной из них - вкладки редактирования списков.

Пусть в верхней части этой области экрана будет набор панелей инструментов для управления списками серверов и отправителей, а основная часть оставшегося пространства будет отведена для управления списком получателей. Панели инструментов управления списками серверов и отправителей должны содержать сам список с возможностью выбора и кнопки для добавления, редактирования и удаления элементов. Список получателей пусть для начала будет просто представлен некоторой таблицей значений.

Целью создания эскиза является рассуждение по вопросу доступа пользователя к функциям приложения. Итоговый вариант интерфейса может существенным образом отличаться от того, что будет спроектировано на этапе эскизного проектирования.

Рассылщик

Файл

Списки

Расписание

Письма

Статистика

Сервера

.....

Рассылка

Отправители

.....

ID	Имя	eMail	Комментарий

Статус

## Обзор контейнеров компоновки

Интерфейс пользователя приложения обычно состоит из комплекса взаимодействующих между собой визуальных компонентов. Компоненты необходимо размещать в окне некоторым способом, допускающим их динамическое позиционирование при изменении размеров окна, или его содержимого. В *WinForms* обычно компоненты позиционировались путём указания расстояний от левого верхнего угла контейнера (окна, либо панели в котором элементы размещаются), а также указывается размеры элемента (ширина и высота). *WPF* допускает подобный режим позиционирования. Но в большинстве случаев применение концепции разметки подразумевает структурирование интерфейса путём разбиение всего пространства окна на зоны и замену абсолютного позиционирования элементов относительным (в пределах контейнера).

Каждый визуальный элемент имеет понятие внешней рамки (**Margin**), определяющий расстояние от стенок контейнера, либо соседних элементов до рамки элемента. Элементы с содержимым имеют понятие внутренней рамки (**Padding**) - расстояние от рамки до содержимого. И ряд элементов имеют саму рамку (**BorderThickness**) - толщина визуальной рамки элемента.

Каждый элемент имеет систему управления ориентацией - расположение:

- **VerticalAlignment** - по вертикали
  - **Stretch** растянутый на всю высоту контейнера,
  - **Top** прижатый к верхней части,
  - **Bottom** нижней части,

- **Center** либо центрированный
- **HorizontalAlignment** - по горизонтали
  - **Stretch** растянутый на всю ширину контейнера,
  - **Left** прижатый к левой,
  - **Right** правой части,
  - **Center** либо центрированный

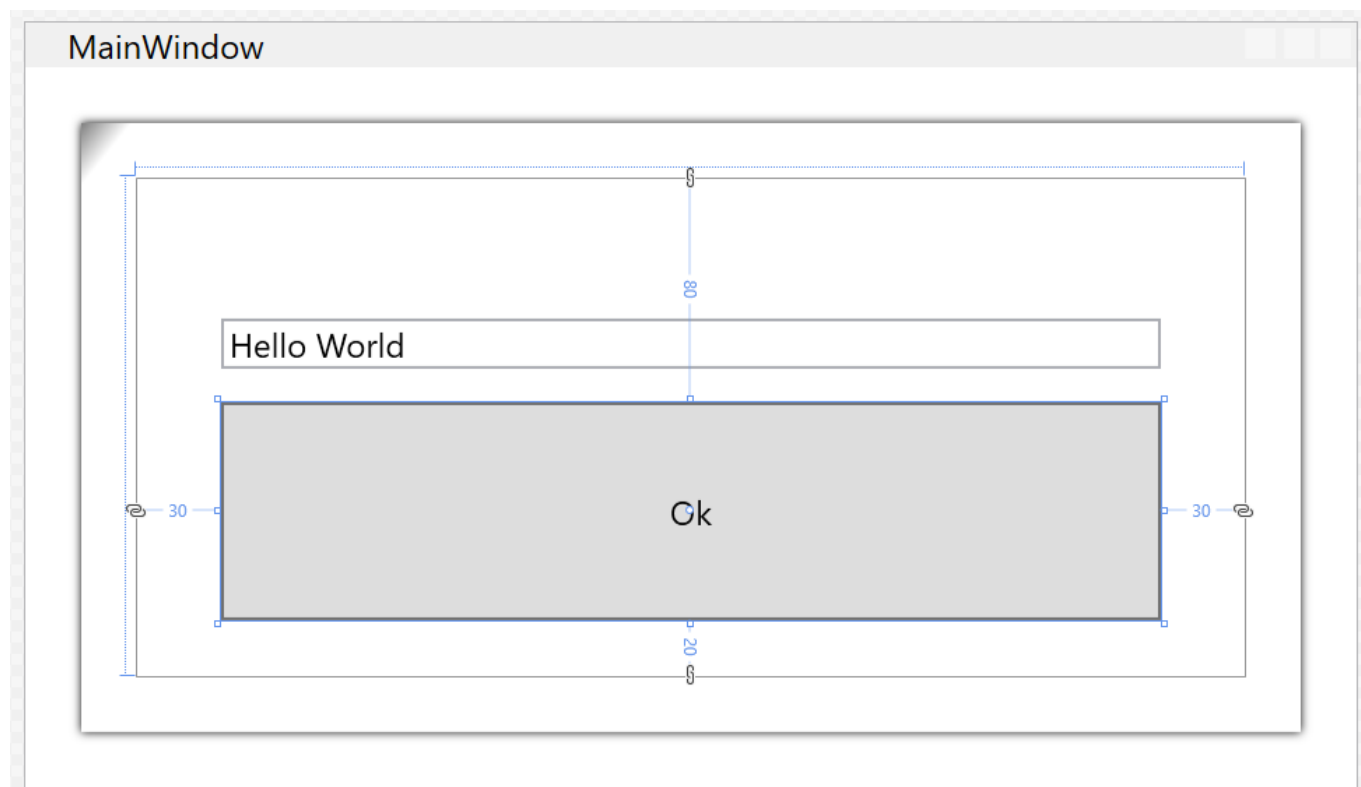
Элементы, обладающие содержимым, имеют свойство, определяющее как содержимое будет расположено внутри элемента: **VerticalContentAlignment**, **HorizontalContentAlignment**.

Для структурирования элементов в окне существует 7 видов контейнеров компоновки - панелей:

1. **Grid**
2. **StackPanel**
3. **DockPanel**
4. **UniformGrid**
5. **WrapPanel**
6. **Canvas**
7. **VirtualizedStackPanel**

## Таблица (**Grid**)

Таблица позволяет размещать элементы внутри в едином координатном пространстве. Все элементы располагаются слоями друг над другом (элемент, расположенный позади закрывается элементом, располагаемым сверху). При этом внутри элементы можно расположить указывая значения внешней рамки **Margin**, как расстояния от границы **Grid** до границ элемента, а также путём указания привязок к краям (**VerticalAlignment**, **HorizontalAlignment**) и размеров (**Width**, **Height**).

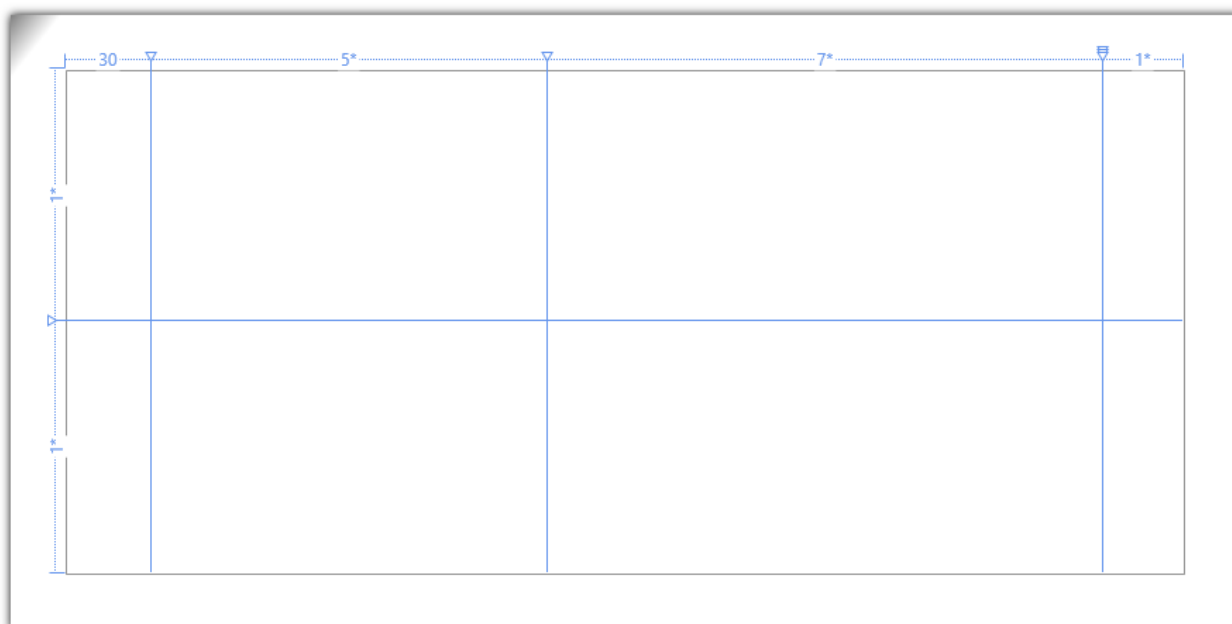


```
<Grid>
  <TextBox Text="Hello World"
    VerticalAlignment="Top"
    Margin="30,50"/>
  <Button Margin="30,80,30,20">Ok</Button>
</Grid>
```

Также таблица позволяет разбить своё пространство на любое число строк и столбцов. Для этого достаточно в дизайнера подвести мышку к левой, или верхней границе **Grid** до изменения формы указателя мышки и сделать щелчок. В дизайнера при этом должна появиться направляющая с помощью которой можно будет изменять размер строки/столбца. Но проще и эффективнее это можно сделать в разметке:

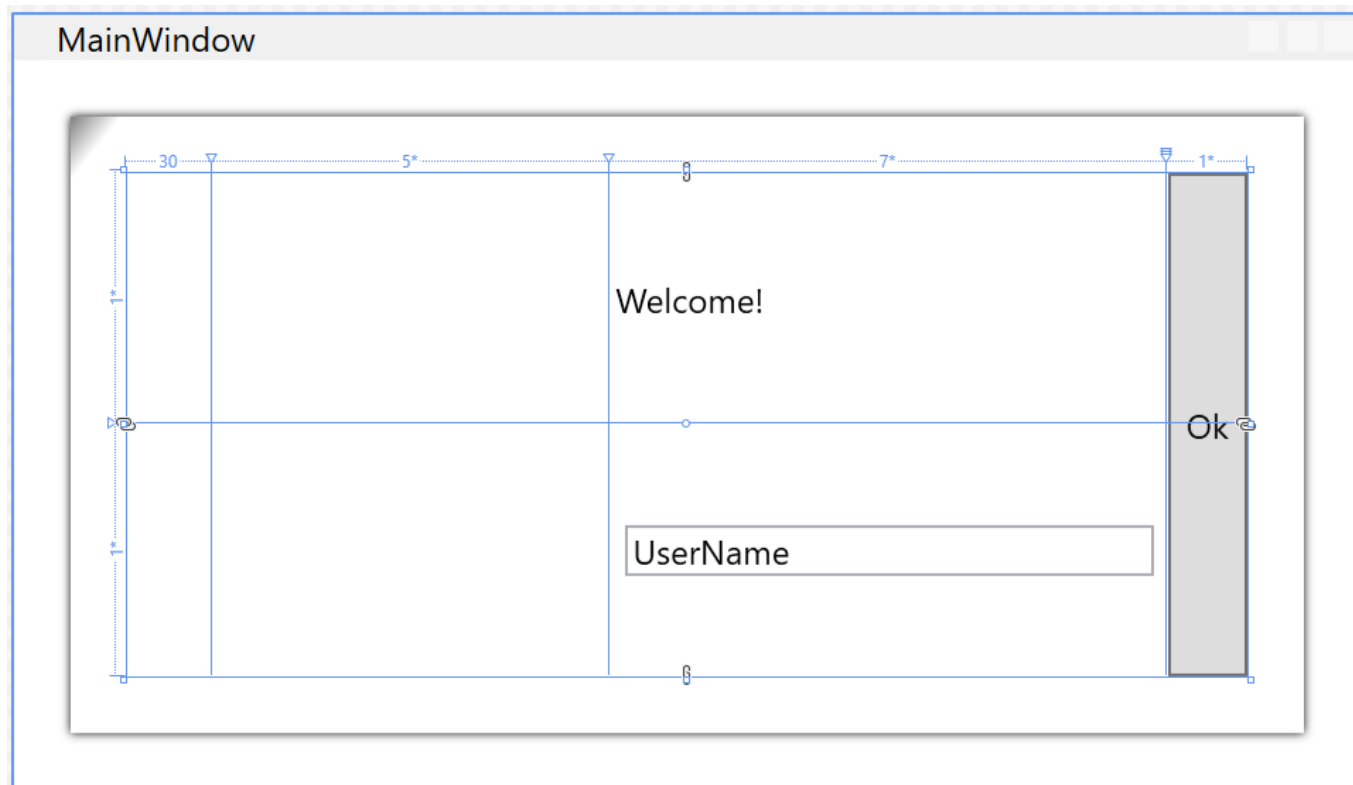
```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="30"/> <!-- Абсолютное значение ширины -->
    <ColumnDefinition Width="5*"/> <!-- Относительное значение ширины -->
    <ColumnDefinition Width="7*"/> <!-- 5 к 7 к 1 -->
    <ColumnDefinition Width="Auto"/> <!-- Колонка выберет размер по
содержимому -->
    <ColumnDefinition/> <!-- значение ширины по умолчанию = 1* -->
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition/> <!-- Делим высоту поровну на две строки -->
    <RowDefinition/> <!-- отношение 1 к 1 -->
  </Grid.RowDefinitions>
</Grid>
```

MainWindow



Для размещения элементов в таблице необходимо в размещаемом элементе указать номер строки и столбца таблицы-контейнера:

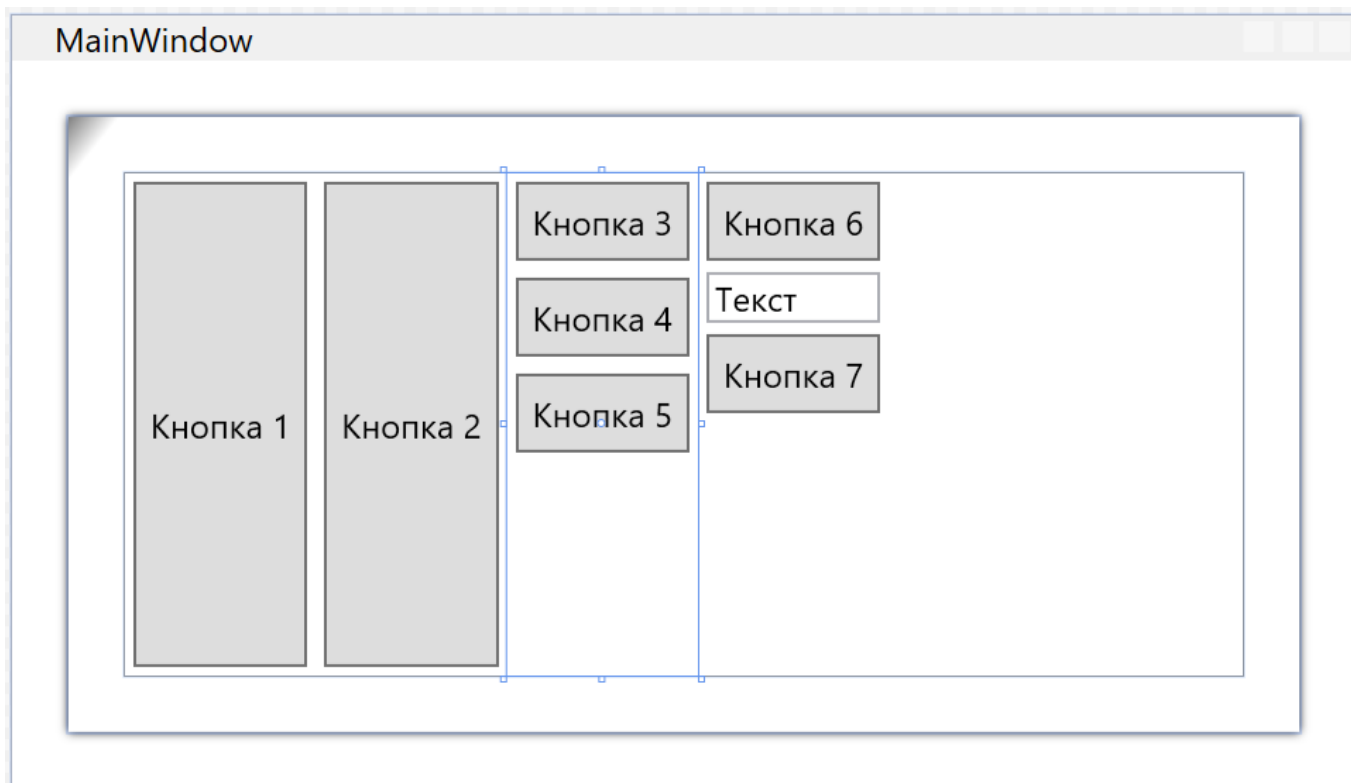
```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="30"/>
    <ColumnDefinition Width="5*"/>
    <ColumnDefinition Width="7*"/>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <TextBlock Grid.Row="0" Grid.Column="1" Grid.ColumnSpan="2"
    VerticalAlignment="Center" HorizontalAlignment="Center">
    Welcome!
  </TextBlock>
  <TextBox Grid.Row="1" Grid.Column="2"
    Text="UserName"
    VerticalAlignment="Center"
    Margin="5"/>
  <Button Grid.Row="0" Grid.Column="4" Grid.RowSpan="2" Content="Ok"/>
</Grid>
```



Стек-панель (**StackPanel**)

Стек-панель позволяет располагать элементы по горизонтали слева направо, либо по вертикали сверху вниз друг за другом.

```
<StackPanel Orientation="Horizontal">
  <Button Margin="3" Padding="5">Кнопка 1</Button>
  <Button Margin="3" Padding="5">Кнопка 2</Button>
  <StackPanel>
    <Button Margin="3" Padding="5">Кнопка 3</Button>
    <Button Margin="3" Padding="5">Кнопка 4</Button>
    <Button Margin="3" Padding="5">Кнопка 5</Button>
  </StackPanel>
  <StackPanel>
    <Button Margin="3" Padding="5">Кнопка 6</Button>
    <TextBox Margin="3,1">Текст</TextBox>
    <Button Margin="3" Padding="5">Кнопка 7</Button>
  </StackPanel>
</StackPanel>
```



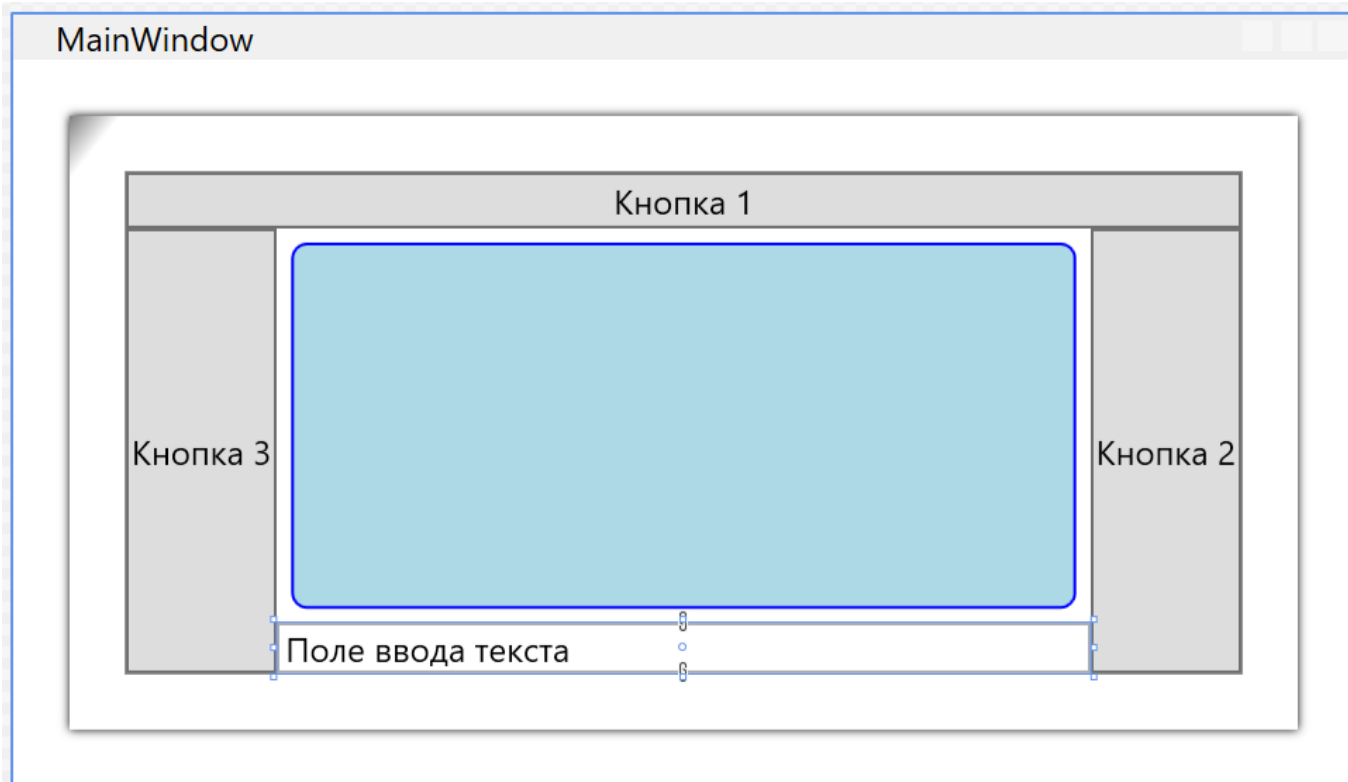
### Док-панель (**DockPanel**)

Док-панель позволяет ориентировать элементы внутри своего пространства "приклеивая" их к своим боковым граням и растягивая последний элемент на всё оставшееся пространство.

```
<DockPanel>
  <Button Content="Кнопка 1" DockPanel.Dock="Top"/>
  <Button Content="Кнопка 2" DockPanel.Dock="Right"/>
  <Button Content="Кнопка 3"/>
  <TextBox DockPanel.Dock="Bottom">Поле ввода текста</TextBox>
```



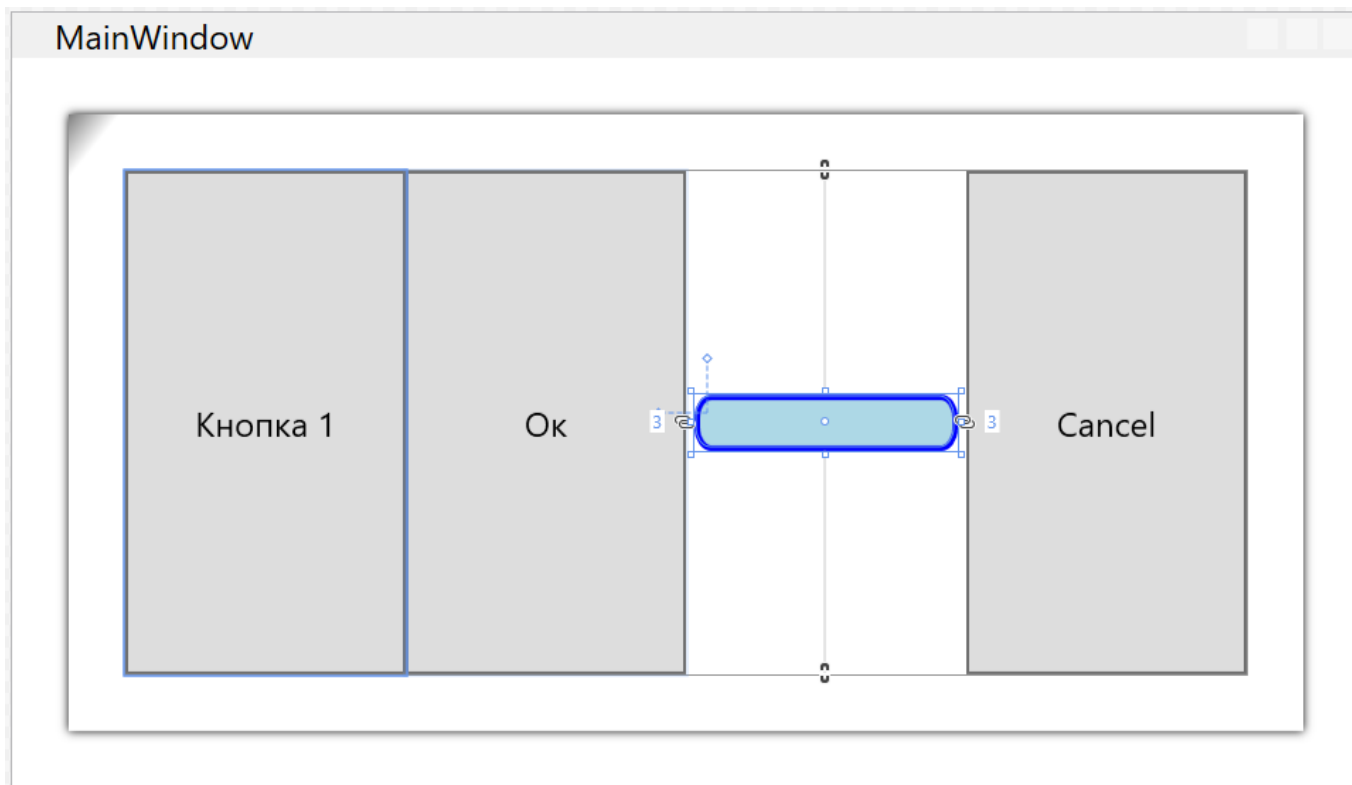
```
<Rectangle Margin="5" Fill="LightBlue" Stroke="Blue"
           RadiusX="5" RadiusY="5"/>
</DockPanel>
```



### Панель одинаковых размеров (UniformGrid)

Панель равномерных размеров автоматически разбивает своё пространство на равные ячейки (строки и столбцы) в зависимости от числа вложенных элементов и их содержимого (размеров). Число строк и столбцов панели можно указать насильно. К примеру, расположим кнопки в ряд обеспечив одинаковый размер для всех них.

```
<UniformGrid Rows="1">
  <Button>Кнопка 1</Button>
  <Button>Ок</Button>
  <Rectangle Margin="3" Fill="LightBlue" Stroke="Blue"
             StrokeThickness="2"
             RadiusX="5" RadiusY="7"
             VerticalAlignment="Center"
             Height="20"/>
  <Button>Cancel</Button>
</UniformGrid>
```



### Панель с переносом ([WrapPanel](#))

Панель с переносом элементов позволяет расположить последовательность элементов в одну строку (в один столбец) и перенести не поместившиеся элементы на следующую строку (столбец).

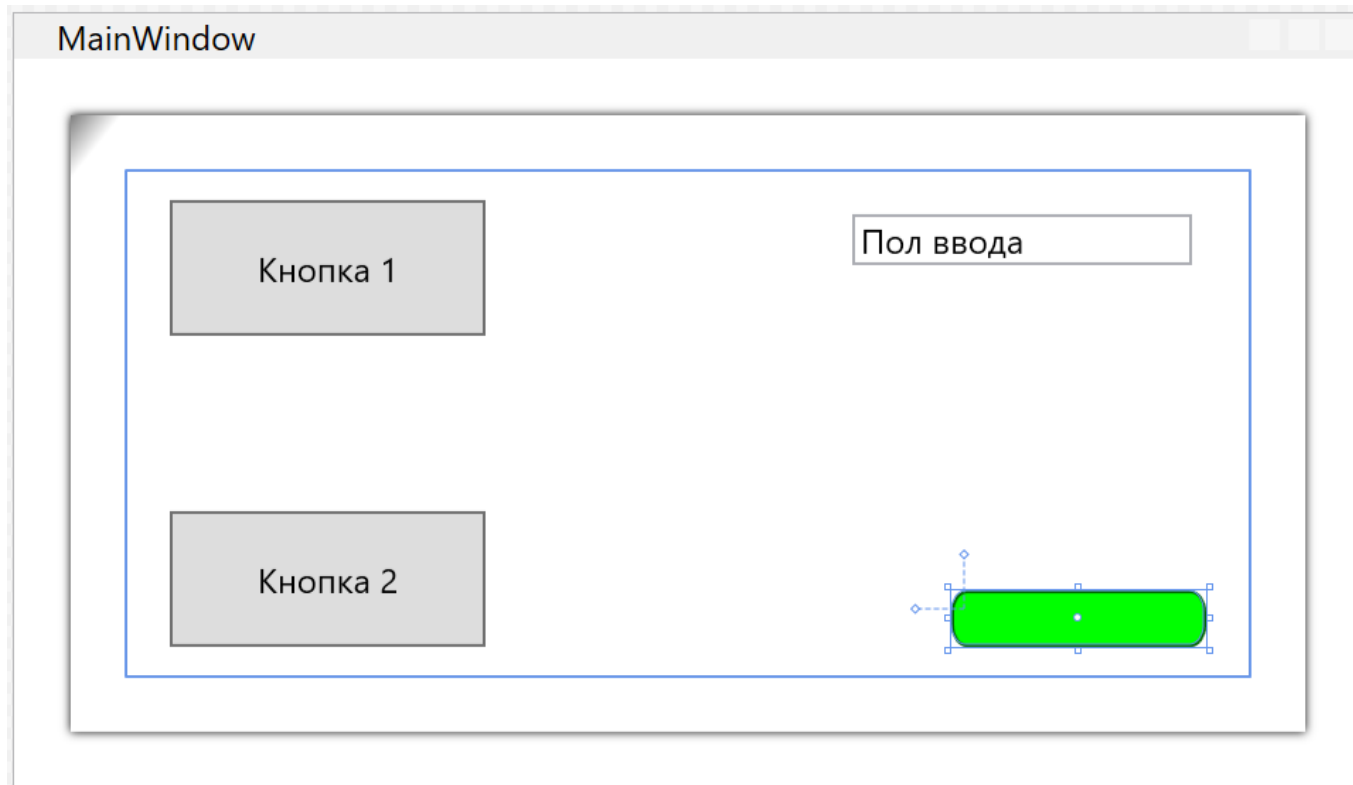
```
<!-- По умолчанию Horizontal. Можно указать Vertical -->
<WrapPanel Orientation="Horizontal">
    <Button Content="Кнопка 1" Margin="5" Padding="30,15"/>
    <Button Content="Кнопка 2" Margin="5" Padding="30,15"/>
    <TextBox VerticalAlignment="Center" MinWidth="100" Margin="10,0">
        Поле ввода 1
    </TextBox>
    <TextBox VerticalAlignment="Center" MinWidth="100" Margin="10,0">
        Поле ввода 2
    </TextBox>
    <Rectangle Margin="30,0" Fill="Red" Stroke="DarkRed" Width="70"/>
    <Rectangle Margin="30,0" Fill="Blue" Stroke="DarkBlue" Width="70"/>
    <Button Content="Кнопка 3" Margin="5" Padding="30,15"/>
</WrapPanel>
```



### Панель-холст/канва (**Canvas**)

Канва используется для координатного размещения элементов внутри относительно её краёв.

```
<Canvas>
  <Button Padding="30,15" Canvas.Top="10" Canvas.Left="15">Кнопка 1</Button>
  <Button Padding="30,15" Canvas.Bottom="10" Canvas.Left="15">Кнопка 2</Button>
  <TextBox Width="120" Canvas.Right="20" Canvas.Top="15">Пол ввода</TextBox>
  <Rectangle Fill="Lime" Stroke="DarkGreen"
    Canvas.Bottom="10" Canvas.Right="15"
    Width="90" Height="20" RadiusX="5" RadiusY="7"/>
</Canvas>
```



### Виртуализованная стек-панель ([VirtualizedStackPanel](#))

Все панели могут быть использованы не только для структурирования элементов интерфейса, но и как внутренний компонент внутри структуры списочных элементов (таких как [ListBox](#), [ComboBox](#) и т.п.) для определения того, как элементы будут расположены внутри. Виртуализованная стек-панель может быть применена в этих целях для сокращения потребления памяти при выводе огромных размеров списков данных на экран и уменьшения нагрузки на ЦПУ для обчёта геометрии их визуальной части. Панель позволяет визуализировать лишь те элементы, которые попадают в её видимую часть. Всё что выходит за пределы видимой части панели не существует и будет создано лишь в тот момент, когда визуализация потребуется.

## Домашнее задание

---

1. Требуется создать свой собственный проект приложения (новое решение)
2. Добавить проект в систему контроля версий
3. Зарегистрироваться на [GitHub.com](#) (если учётной записи там ещё нет)
4. Выгрузить созданный репозиторий в открытый репозиторий на [github.com](#)
5. Создать тестовый WPF-проект приложения с окном, обеспечивающим отправку почты
  1. Обеспечить возможность ввода адреса получателя
  2. Обеспечить ввод адреса отправителя
  3. Обеспечить ввод параметров почтового сервера (адрес и порт, использовать/нет ssl)
  4. Обеспечить возможность ввода имени пользователя и пароля для сервера
6. Разметку окна постараться структурировать с применением панелей контейнеров-компоновки.
7. ★ Реализовать стилизацию интерфейса

## Ссылки

---

- [Metanit.com](#)
- [ProfessorWeb.ru](#)
- [MSDN](#) - Создание простого приложения на C#
- [Оформление кода](#) - Стандарты и правила
- [Material design in WPF](#) - стилизация приложений
- [GIT](#) - Руководство по системе управления версиями GIT