Урок 1

Введение в WPF. Архитектура приложения на C#

Общие рекомендации по созданию приложений на C#. Введение в WPF, его отличие от WinForms. Приложение «Рассыльщик» на WPF. Подключение базы данных к приложению, создание интерфейса с помощью Combobox, Grid. ADO.NET, MS SQL.

Введение

Введение в WPF

- Знакомство с XAML
- Знакомство с контролами WPF
- Изменение стиля приложения WPF

Итог

Рекомендации по созданию приложений

- Постановка цели
- План по созданию программного продукта
  - 1. Для чего нужно приложение?
  - Структура программного продукта
  - Визуальное воплощение приложения

Приложение «Рассыльщик». Интерфейс и WPF

- Прототип приложения «Рассыльщик»
- Компоновочные элементы управления в системе WPF (панели)

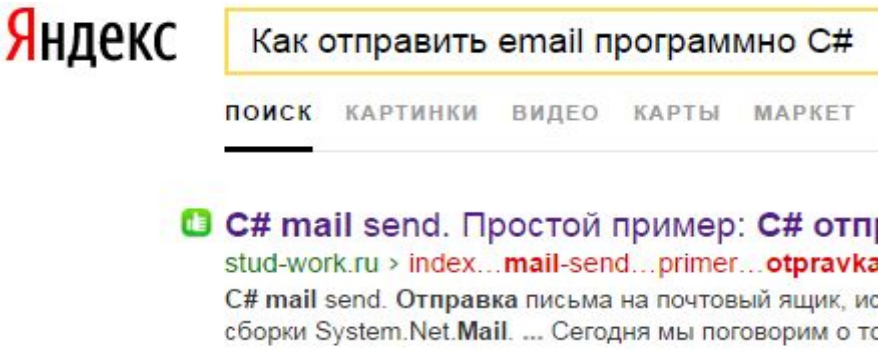
Домашнее задание

Используемая литература

Введение

Предположим, вы решили написать программу, которая рассылает электронную почту автоматически всем вашим друзьям или коллегам. На первом этапе вы еще четко не представляете ее интерфейс и какие задачи она будет выполнять. Нужно проверить, возможно ли в принципе отправить электронное письмо средствами C#.

Используем для этого yandex.ru или google.com: в строке поиска набираем «отправка email c#» или «рассылка писем программно C#»:



Зачастую работа программиста заключается в том, что он ищет в интернете куски кода, подходящего для решаемой задачи. В этом нет ничего зазорного. Главное — понимать и уметь адаптировать код, который вы нашли под свой проект.

Вернемся к нашей задаче. В интернете можно найти множество вариантов, как отправить электронное письмо программным способом. Рассмотрим один из них.

Будем использовать протокол **SMTP** (простой протокол передачи почты — simple mail transfer protocol). В .NET Framework есть специальная библиотека, при помощи которой можно отправить электронное письмо, используя протокол **SMTP**.

Создадим консольное приложение, чтобы проверить возможность отправки электронного письма средствами .Net Framework. Подключим два пространства имен:

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

Для отправки email в пространстве имен **System.Net.Mail** реализовано два класса: **SmtpClient** и **MailMessage**. В коде это может быть реализовано так:

```
// Формирование письма
MailMessage mm = new MailMessage("оправитель@yandex.ru",
"получатель@yandex.ru");
mm.Subject = "Заголовок письма";
mm.Body = "Содержимое письма";

mm.IsBodyHtml = false; // Не используем html в теле письма
```

Далее нужно организовать авторизацию на smtp-сервере. Здесь пригодится класс **NetworkCredential** из **System.Net**.

```
// Авторизуемся на smtp-сервере и отправляем письмо
SmtpClient sc = new SmtpClient("smtp.yandex.ru", 25);
sc.EnableSsl = true;
sc.DeliveryMethod = SmtpDeliveryMethod.Network;
sc.UseDefaultCredentials = false;
sc.Credentials = new NetworkCredential("отправитель@yandex.ru", "password");
sc.Send(mm);
```

Мы использовали **smtp.yandex.ru** в качестве службы отправки почты, где 25 — это порт. Мы выбрали **yandex**, потому что email отправителя находится на **yandex.ru**.

Если почта отправителя находится на **gmail.com**, то строка получения экземпляра класса **SmtpClient** будет выглядеть так: **SmtpClient sc = new SmtpClient("smtp.gmail.com", 58);**

Если на **mail.ru: SmtpClient sc = new SmtpClient("smtp.mail.ru", 25);**

Если ящик отправителя находится где-то еще, придется воспользоваться интернетом, чтобы определить, какой smtp-сервер и порт нужно использовать.

Немного усовершенствуем код: добавим блок **try…catch** к строке **sc.Send(mm**).

```
using System;
using System.Windows;

...

try
{
      sc.Send(mm);
}
catch (Exception ex)
{
      MessageBox.Show("Невозможно отправить письмо " + ex.ToString());
}
```

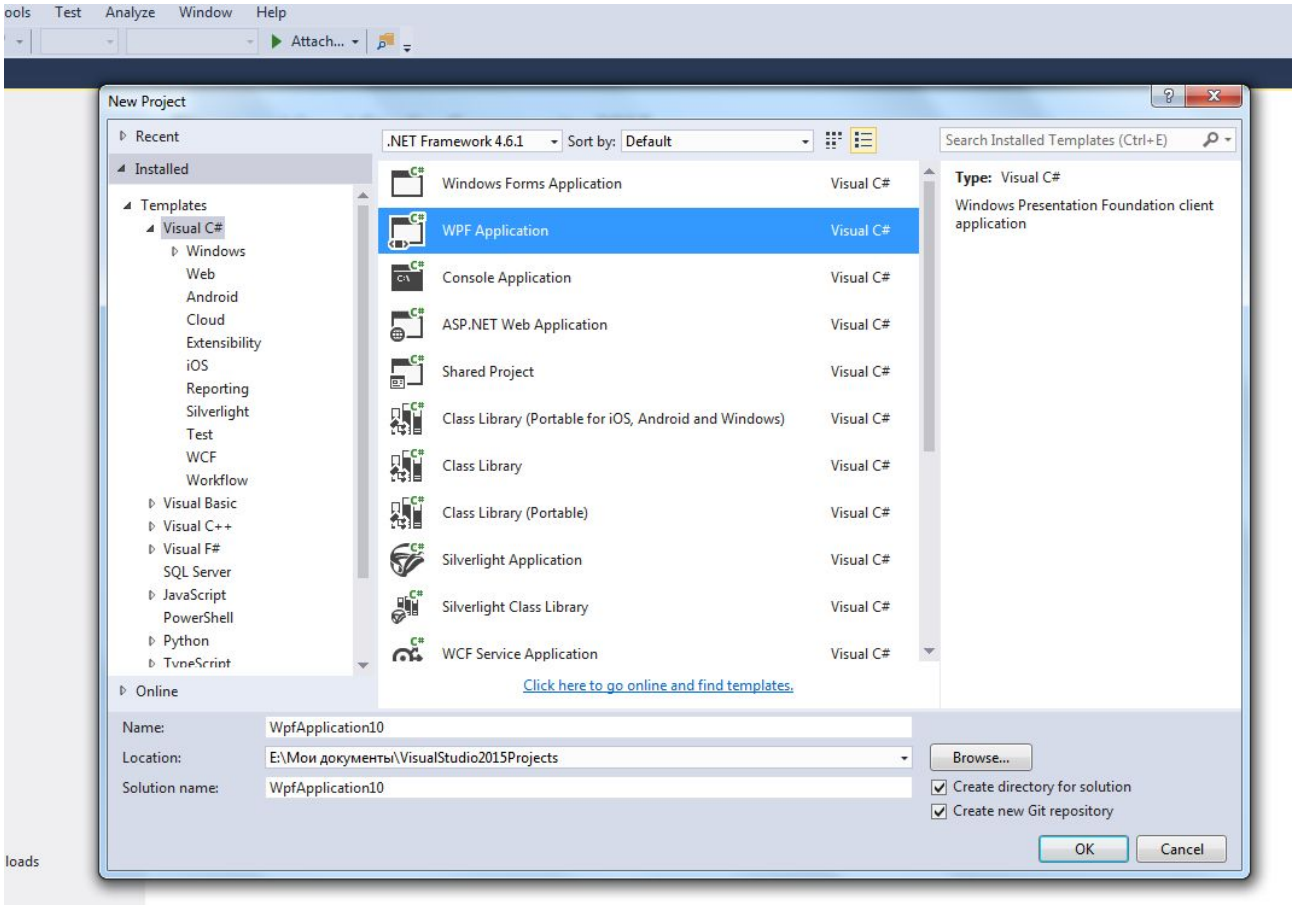
Введение в WPF

Проверим код и посмотрим, отправиться ли письмо из нашей программы. Не будем создавать консольное приложение, а перейдем сразу к WPF.

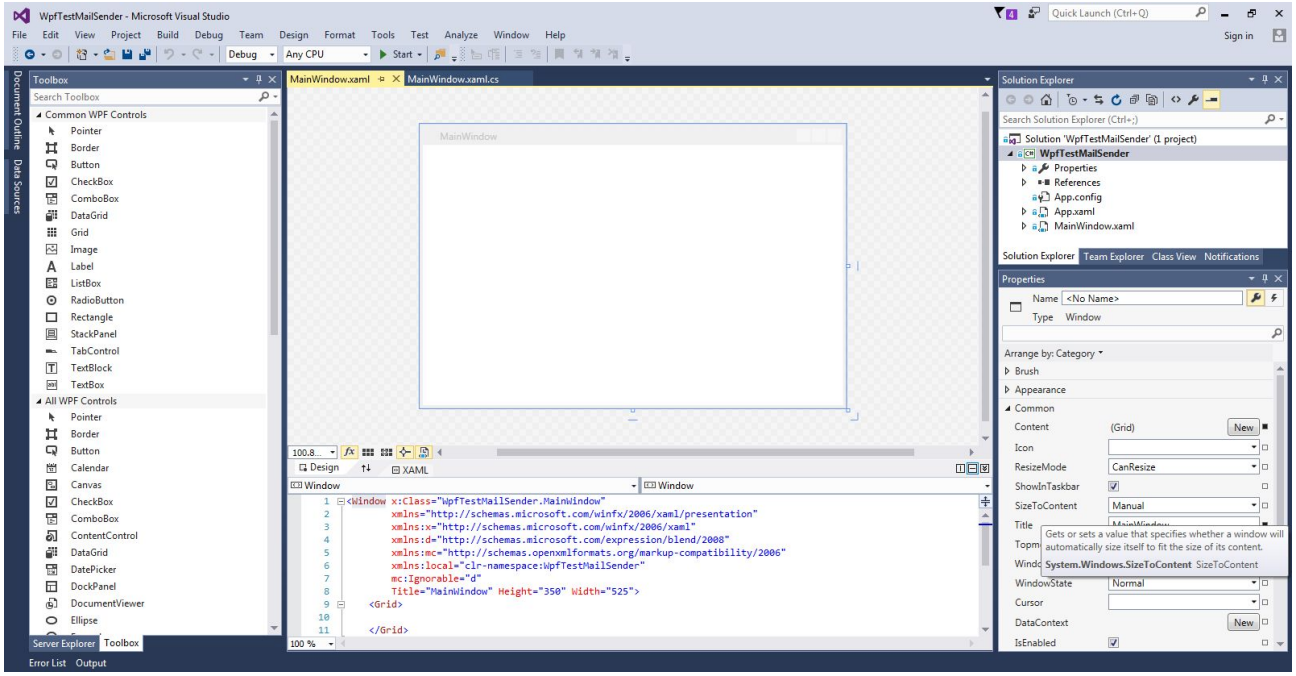
Если разрабатываем клиентское приложение под Windows, можем выбрать из двух технологий: **Windows Forms** и **WPF** (Windows Presentation Foundation).

WPF — это новый интерфейс прикладного программирования, который позволяет управлять слоем презентации пользовательского приложения.

Открываем **Visual Studio**, кликаем **New Project** и выбираем **WPF Application**:

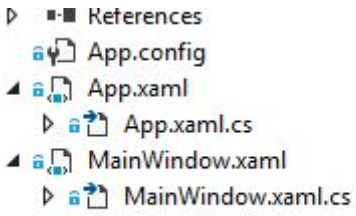


Переименовываем проект в **WpfTestMailSender**. Нажимаем **ОК** и видим структуру проекта:



Сгенерированная структура состоит из файлов **MainWindow.xaml** и **App.xaml**.

Если развернуть узлы проекта, соответствующие этим файлам, увидим исходные файлы **.cs**:



App.xaml определяет первоначально загружаемый файл. Если открыть **App.xaml**, увидим элемент **Application XAML**. В нем в атрибуте **StartupUri** по умолчанию прописан **MainWindow.xaml**, то есть первоначально загружаемый XAML-файл.

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

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

Переименуем файл **MainWindow**, например, в **WpfMailSender**, а файл **MainWindow.xaml** в **WpfMailSender.xaml**. Исходный файл **.cs** переименуется автоматически. Делайте так, чтобы название файла не совпадало с названием проекта, и основного **Namespace**. Самые любопытные могут проверить, какие ошибки при этом появятся.

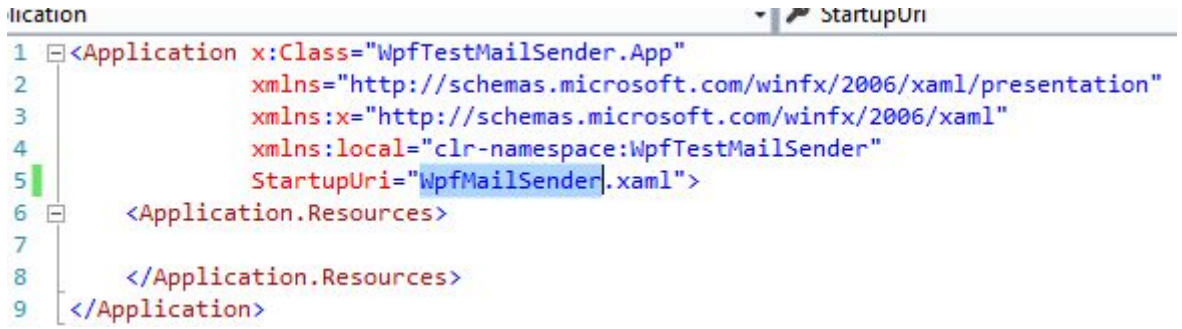
Переименуем название класса в файле **.cs**:

```
namespace WpfTestMailSender
{
    /// <summary>
    /// Interaction logic for WpfTestMailSender.xaml
    /// </summary>
    public partial class WpfMailSender : Window
    {
        public WpfMailSender()
        {
            InitializeComponent();
        }
    }
}
```

Переименуем в файле **.xaml** значение атрибута **x:Class** элемента **Window**:

```
ow                                                    x:Class
<Window x:Class="WpfTestMailSender.WpfMailSender"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pr
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blenc
```

Переименуем в файле **App.xaml** атрибут **StartupUri**:



Если посмотрим на окно визуального дизайнера основного файла проекта **WpfMailSender.xaml**, то увидим, что по сравнению с технологией **Windows Forms** почти ничего не поменялось. Так же можно мышью перетаскивать контролы из **Toolbox** на окно. Но одно новшество появилось.

На области предварительного просмотра и редактирования есть участок для изменения кода XAML.

Несмотря на внешнее сходство, технология **WinForms** отличается от **WPF**. На **WinForms** программист с помощью визуального конструктора определяет пользовательский интерфейс, при этом код генерируется в файле с расширением **.designer**. Таким образом, интерфейс определяется кодом на языке C#. На **WPF** для этой цели есть язык **XAML**.

XAML, или **Extensible Application Markup Language** — это язык разметки для определения пользовательского интерфейса. XAML разрабатывался на основе языка XML и внешне очень на него похож. Благодаря ему появилось много новых возможностей сделать для приложения более красивый дизайн, добавить 3D-графику. Но код, написанный на C#, практически не отличается от того, который предназначался бы для приложения на WinForms.

Знакомство с XAML

XAML покажется вам простым, если вы знакомы с языком XML, так как их синтаксис практически совпадает.

```
<Window x:Class="WpfTestMailSender.WpfMailSender"
        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:WpfTestMailSender"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

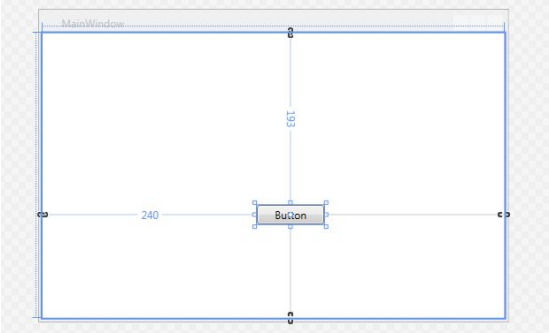
Это XAML-код, который студия создала автоматически при добавлении проекта. Единственное, что мы сделали — заменили название класса основного окна **MainWindow** на **WpfMailSender**.

XAML может иметь только один корневой узел, в данном случае это **Window**. Элемент, который в нем содержится, это **Grid**. В XAML элементы вкладываются друг в друга и в корневой элемент, определяя, как выглядит пользовательский интерфейс и что в нем содержится. Наименования атрибутов и элементов зависят от регистра. Каждому элементу XAML соответствует класс **.NET**, а атрибутам — события или свойства этого класса. Атрибут **x:Class** содержит название класса, который соответствует корневому узлу. В корневом узле есть свойства **Height**, **Width**, **Title** и префиксы пространств имен. Последние объявлены атрибутом **xmlns** и соответствуют определению пространств имен в **.cs**-файлах при помощи **using**.

Знакомство с контролами WPF

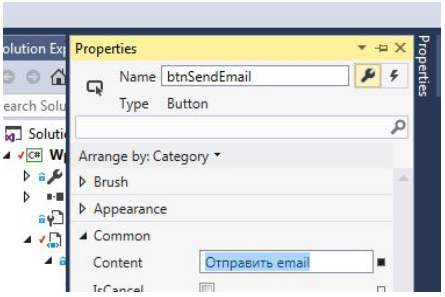
Пришло время поместить кнопку (**button**) на форму приложения, определить метод по событию нажатия этой кнопки и поместить код, который рассылает письма, в тело этого метода.

Поместим кнопку на проект:



Заходим в **Property** и видим, что свойства **Text** у кнопки нет, вместо него есть свойство **Content**. Свойство **Name** на самом верху. Заменим наименование контрола (свойство **Name**) с **Button** на **btnSendEmail**. Чтобы было легче разбираться с программой, называем контролы по такому принципу: сначала — краткое обозначение контрола (для кнопки это **btn**), затем описание его действий (каждое слово с прописной буквы, без пробелов и подчеркиваний).

Теперь заменим надпись на кнопке (свойство **Content**) с **Button** на «Отправить email».



Теперь посмотрите на XAML-код: внутри элемента **Grid** появился элемент **Button**.

```
    <Grid>
        <Button x:Name="btnSendEmail" Content="Отправить email"
HorizontalAlignment="Left" Margin="240,193,0,0" VerticalAlignment="Top"
Width="124"/>
    </Grid>
```

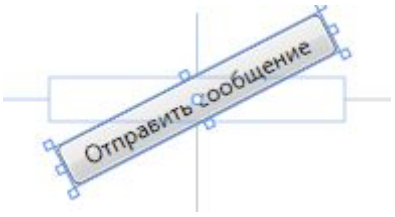
Кстати, в XAML-коде вы можете изменить название кнопки и другие свойства. Поиграйтесь с кнопкой, поменяйте ей свойства: наименование, длину, ширину, расположение на экране:

- на экране в дизайнере;
- в XAML-коде;
- в свойствах **Property**.

В основном **Name** и **x:Name** взаимозаменяемы. **Name** является свойством класса, x:Name — директивой из **x:“пространство имен”**, используемая XAML-парсером.

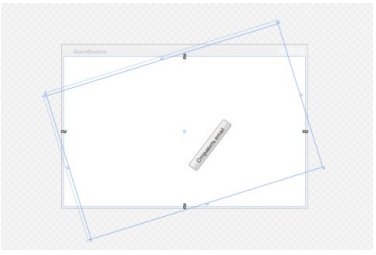
Если подвести курсор мыши к любому углу кнопки, он станет изогнутой стрелкой:

Можно повернуть кнопку:



Посмотрите, как поменяется при этом XAML-код.

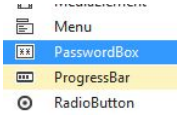
Так же можно повернуть и весь **Grid**.



Чтобы вернуть Grid и кнопку в исходное положение, удалим подузлы **Grid.RenderTransform** и **Button.RenderTransform**.

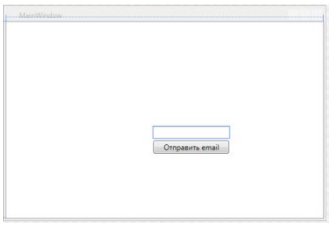
Чтобы работать только в дизайнере, можно два раза кликнуть по вкладке **Design**, а только с **XAML** — по вкладке **XAML**. Чтобы вернуть отображение и дизайнера, и кода XAML, кликните справа по этой кнопке:

Добавим на форму еще один контрол — **PasswordBox**:



Он нужен для ввода пароля к ящику при отправке писем. Не нужно писать свои пароли, да еще и рядом с логином, даже в таких тестовых приложениях — вдруг их считает недоброжелатель или вирус.

Свойство **Name** менять этому контролу не будем, пусть называется **passwordBox**.



Чтобы почувствовать разницу, создайте такой же проект на **Windows Forms**. Киньте на форму кнопку, переименуйте свойство **Name** в **btnSendEmail** и назовите эту кнопку «Отправить email». Контрола **PasswordBox** в Windows Forms нет, поэтому киньте на форму обычный **TextBox**, поменяйте ему свойство **Name** с **textBox1** на **passwordBox**. Найдите свойство **PasswordChar** и поставьте там символ ***** или любой другой, чтобы скрыть свой пароль.

Теперь добавляем метод, обрабатывающий событие клика кнопки «Отправить email» в приложениях. В них обоих можно добавить обработчик, два раза кликнув по кнопке в дизайнере, либо добавив событие **Click** в списке событий в свойствах кнопки.

В приложении **WPF** появился еще один способ добавить обработчик события: прописать в XAML-коде в свойствах кнопки **Button** атрибут **Click="btnSendEmail_Click"**.

```
    <Grid
        <Button x:Name="btnSendEmail" Content="Отправить email"
HorizontalAlignment="Left" Margin="240,193,0,0" VerticalAlignment="Top"
Width="124" Click="btnSendEmail_Click" />
        <PasswordBox x:Name="passwordBox" HorizontalAlignment="Left"
Margin="240,170,0,0" VerticalAlignment="Top" Width="124"/>
    </Grid>

</Window>
```

В этом случае придется вручную добавить код обработчика в файл **.cs**:

```
private void btnSendEmail_Click(object sender, RoutedEventArgs e)
{
}
```

Теперь в проектах нужно добавить две директивы **using**:

- **using System.Net**;
- **using System.Net.Mai**l;

В тело обработчика **btnSendEmail_Click** добавим код, который писали в самом начале урока. Приводим его ниже с некоторыми изменениями:

```
List<string> listStrMails = new List<string> { "testEmail@gmail.com",
"email@yandex.ru" }; // Список email'ов //кому мы отправляем письмо
string strPassword = passwordBox.Password; // для WinForms - string strPassword
= passwordBox.Text;
foreach (string mail in listStrMails)
{
          // Используем using, чтобы гарантированно удалить объект MailMessage
после использования
    using (MailMessage mm = new MailMessage("sender@yandex.ru", mail))
    {
        // Формируем письмо
        mm.Subject = "Привет из C#"; // Заголовок письма
        mm.Body = "Hello, world!";       // Тело письма
        mm.IsBodyHtml = false;           // Не используем html в теле письма
      // Авторизуемся на smtp-сервере и отправляем письмо
      // Оператор using гарантирует вызов метода Dispose, даже если при вызове
      // методов в объекте происходит исключение.
        using (SmtpClient sc = new SmtpClient("smtp.yandex.ru", 25))
        {
            sc.EnableSsl = true;
                    sc.Credentials = new NetworkCredential("sender@yandex.ru",
strPassword);
            try
            {
                sc.Send(mm);
            }
            catch (Exception ex)
            {
                MessageBox.Show("Невозможно отправить письмо " + ex.ToString());
            }
        }
    } //using (MailMessage mm = new MailMessage("sender@yandex.ru", mail))
}
MessageBox.Show("Работа завершена.");
```

Остается заменить список email-адресов, на которые будем отправлять письмо, и указать свой email — от кого запускается рассылка. Не забудьте выбрать правильный smtp-сервер.

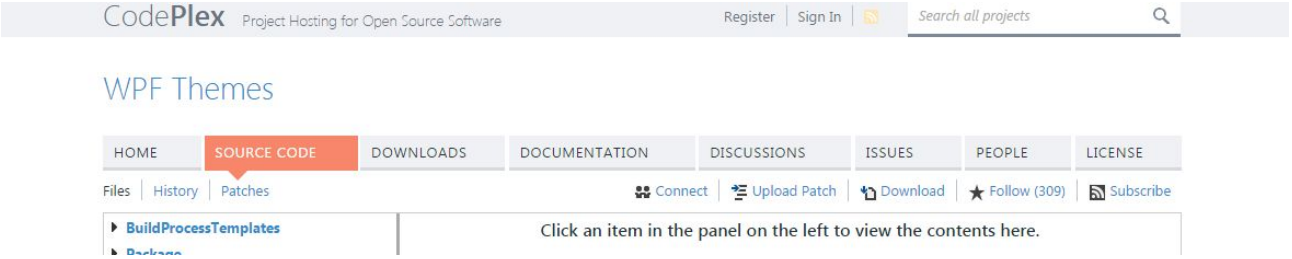
Открываем тестовые email-ы и видим письма с темой «Привет из C#», текст — «Hello, world!» Все работает, письма отправляются из обоих приложений: основанного на технологии WPF и на WinForms.

Изменение стиля приложения WPF

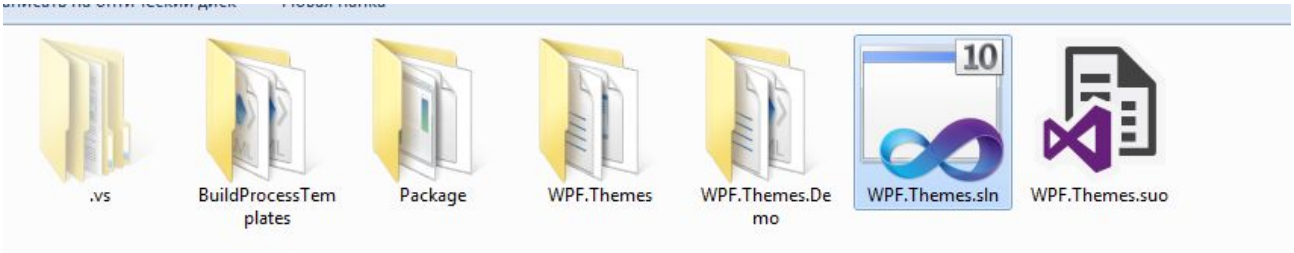
Подчеркнем преимущество технологии WPF и добавим красивый стиль для приложения.

Можно создать стиль в отдельном XAML-файле и привязать его к свойству **Resources** любого элемента или ко всему приложению — для этого добавим стиль в файле **App.xaml** элемента **Application** в свойстве **Application.Resources**.

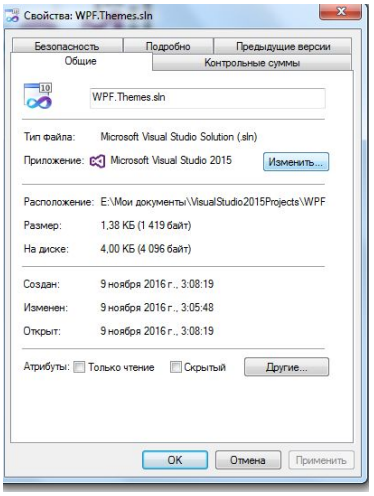
Скачаем библиотеку стилей или тем с сайта WPF Themes.



Заходим на сайт и переключаемся на вкладку **SOURCE CODE**, затем кликаем на кнопку **Download** и загружаем проект с темами. Распаковываем его.

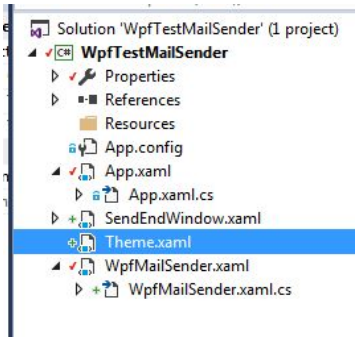


В папке **WPF.Themes** найдем список тем. Если открыть проект **WPF.Themes.sln** и запустить его, то можно увидеть, как выглядят темы, и выбрать то, что нравится. Если при клике на **WPF.Themes.sln** проект не открывается, на него надо кликнуть правой кнопкой мыши, выбрать «Свойства», нажать кнопку «Изменить» и принудительно задать, чем открывать приложение:



Изучить проект **WPF.Themes** вы можете самостоятельно.

Чтобы выбрать тему и добавить ее в приложение, открываем папку **WPF.Themes**, находим папку **ShinyBlue** и копируем файл **Theme.xaml** в корень проекта.



Добавляем в файл **App.xaml** элемента **Application** (свойство **Application.Resources**) такой код:

```
<ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Theme.xaml"/>
    </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

Если запустить приложение, можно увидеть, что оно стало синеватым.

Уберем новый код из файла **App.xaml** и добавим его к **Resources** кнопки **btnSendEmail** основного файла **WpfMailSender.xaml**.

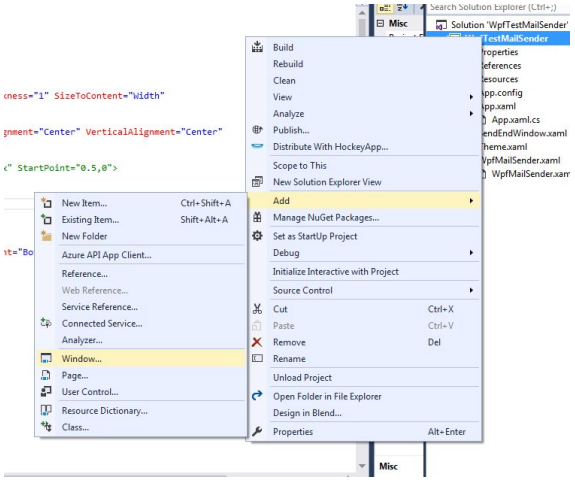
```
<Button          x:Name="btnSendEmail"          Content="Отправить          email"
HorizontalAlignment="Left"      Margin="240,193,0,0"      VerticalAlignment="Top"
Width="124" RenderTransformOrigin="0.5,0.5" Click="btnSendEmail_Click" >
    <Button.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Theme.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Button.Resources>
</Button>
```

Синеватой осталась только одна кнопка, а поле для ввода пароля не отдает синевой при наведении мыши.

Применить стиль можно к главному окну приложения полностью, добавив узел **ResourceDictionary** в подузел **Window.Resources** узла **Window**. В этом случае применяем новый стиль только на контролы основного окна приложения. Создадим окно вместо **MessageBox.Show("Работа завершена.")**.

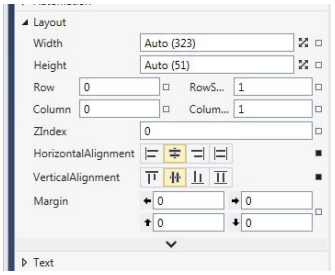
В любом случае **MessageBox** выглядят некрасиво и на них новый стиль не действует, и поэтому лучше их заменять окнами с сообщениями.

Кликнем правой кнопкой мыши в **Solution Explorer** по названию проекта и выберем «Добавить новое окно».



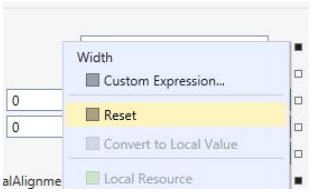
Назовем его **SendEndWindow** и поместим на него **Label**.

В **Properties** меняем название контрола (поле **Name** наверху) на **lSendEnd**, в поле **Content** прописываем «Работа завершена». В категории **Layout** сделаем так, чтобы лейбл находился в центре окна:

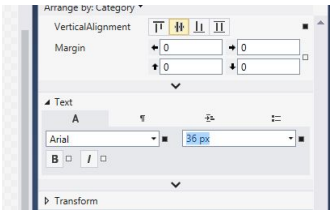


HorizontalAlignment и **VerticalAlignment** выставляем по центру (как на рисунке).

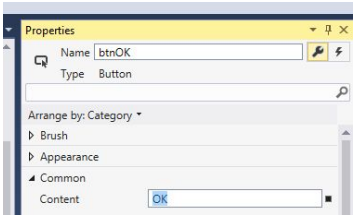
Свойства **Width** и **Height** устанавливаем в **Auto** (нажимаем на маленькие квадраты справа и в появившемся меню выбираем **Reset**).



В категории **Text** увеличим шрифт до 36, чтобы надпись была крупнее.



Теперь кидаем кнопку на новую форму. Свойство **Name** — **btnOK**, **Content** — **OK**.



В категории **Layout** и устанавливаем **Margin** — это расстояния до границ родительского контейнера, в данном случае окна **SendEndWindow**: 50 до правого края и 20 до нижнего. Длину и ширину кнопки установите мышью по своему усмотрению.

Основное окно сделаем так, чтобы его размеры нельзя было поменять. Можем случайно кликнуть мышью не по основному окну, а по элементу **Grid**, который лежит на нем. Удостоверимся, что выбрали основное окно. Кликнем в xaml-коде по узлу **Window** (прямо по слову **Window**), и увидим, как в дизайнере подсветится само окно, а в **Properties** появятся свойства основного окна. Чтобы выбрать другой элемент, нужно кликнуть по его названию (**Grid** или **Button**, например).

Заходим в **Properties** основного окна, в категорию **Appearance**, выбираем **WindowsStyle** — **None**. В категории **Common** выбираем **ResizeMode** — **NoResize**, **WindowStarupLocation** — **CenterOwner**. Заходим в категорию **Layout** и устанавливаем **Width** — 400, **Height** — 175.

Получаем красивое окно для надписи «Работа завершена».

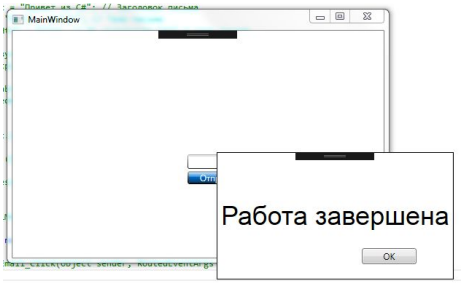
Теперь два раза кликаем по кнопке **OK** (или в свойствах выбираем событие **Click) и в обработчике добавляем строку Close()**:

```
private void btnOk_Click(object sender, RoutedEventArgs e)
{
    Close();
}
```

Заходим в файл **WpfMailSender.xaml.cs** и внизу обработчика **private void btnSendEmail_Click(object sender, RoutedEventArgs e)** вместо вызова **MessageBox.Show("Работа завершена.");** добавляем такой код:

```
SendEndWindow sew = new SendEndWindow();
sew.ShowDialog();
```

Так создаем экземпляр нового окна и открываем его. При отладке нового окна можно оставить только его вызов — закомментировать код по отправке писем.

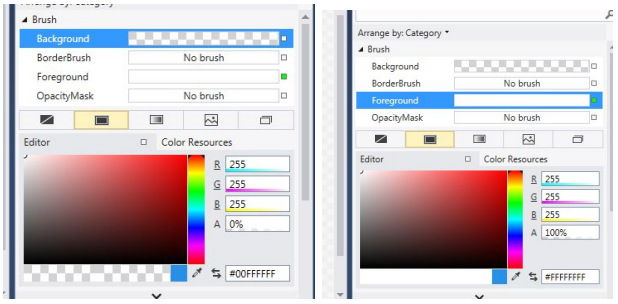


Дизайн нового окна черный и некрасивый. Чтобы исправить это, применим стиль на все приложение. Вернем его в файл **App.xaml** (элемент **Application**, свойство **Application.Resources**):

```
<ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Theme.xaml"/>
    </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

Теперь новое окно появляется, кнопка «ОК» синеватая, но пропал лейбл с надписью «Работа завершена».

Либо мы неправильно прикрутили новый стиль, либо неисправность глубже. Открываем в свойствах (**Properties**) элемента **Label** категорию **Brush**, которая отвечает за отрисовку цвета фона, шрифта, границ.

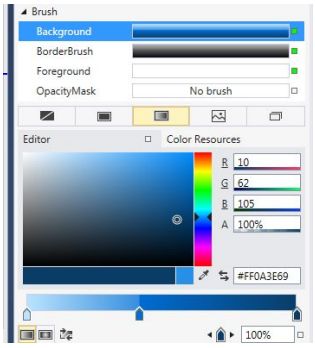


Если кликнуть на **Background**, увидим прозрачный фон (#00FFFFFF), то есть фон лейбла будет совпадать с фоном основного окна. Если кликнем на **Foreground**, увидим белый (#FFFFFFFF): цвет шрифта надписи будет белым, как и фон.

Сделаем буквы цветными. Если посмотрим на фон кнопок этой темы, увидим синий переливающийся

цвет.

Чтобы посмотреть, как устроены свойства кнопки, зайдем в категорию **Brush**.



Установим такой же цвет для букв лейбла. Можно это сделать прямо в свойствах, но правильнее скопировать xaml-код цветов фона кнопки.

Код в файле **Theme.xaml**, оформляющий кнопку:

```
<LinearGradientBrush x:Key="NormalBrush" EndPoint="0.5,1" StartPoint="0.5,0">
    <GradientStop Color="{StaticResource NormalBrushGradient1}" Offset="0" />
                <GradientStop    Color="{StaticResource    NormalBrushGradient2}"
Offset="0.41800001263618469" />
     <GradientStop Color="{StaticResource NormalBrushGradient3}" Offset="0.418"
/>
    <GradientStop Color="{StaticResource NormalBrushGradient4}" Offset="1" />
</LinearGradientBrush>
```

Эти цвета нужно добавить в описание лейбла. Удалим **x:Key=”NormalBrush”**, а все остальное должно перекочевать в лейбл.

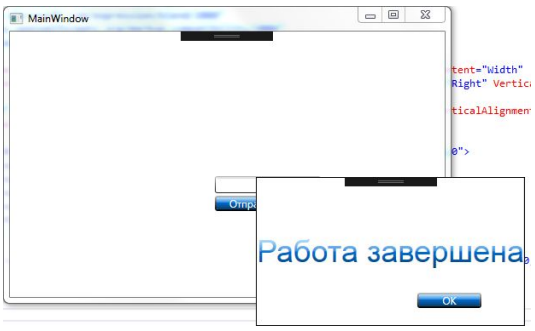
Описание лейбла выглядело так:

```
<Label     x:Name="lSendEnd"     Content="Работа     завершена"     FontSize="36"
HorizontalAlignment="Center" VerticalAlignment="Center" FontFamily="Arial"/>
```

А надо сделать так:

```
<Label     x:Name="lSendEnd"     Content="Работа     завершена"     FontSize="36"
HorizontalAlignment="Center" VerticalAlignment="Center"
                FontFamily="Arial"    >
    <Label.Foreground>
        <LinearGradientBrush EndPoint="0.5,1" MappingMode="RelativeToBoundingBox"
StartPoint="0.5,0">
            <GradientStop Color="#FFBAE4FF" Offset="0.013"/>
            <GradientStop Color="#FF398FDF" Offset="0.41800001263618469"/>
            <GradientStop Color="#FF006DD4" Offset="0.418"/>
            <GradientStop Color="#FF0A3E69" Offset="1"/>
        </LinearGradientBrush>
    </Label.Foreground>
</Label>
```

Запускаем проект:



Он выполнен в одном стиле и хорошо смотрится.

Итог

Мы проверили код по отправке писем из приложения, созданного на языке C#: это вполне возможно.

Познакомились с технологией визуального интерфейса **WPF** и сравнили ее с **Windows Forms**.

Основное отличие WPF от WinForms — создание красивых стилей, украшение проекта. Если как следует поискать в интернете, то можно и приложение, написанное на WinForms, сделать более красивым и натянуть на него корпоративный стиль. Но чтобы сделать презентабельное приложение, WPF подходит больше. Рекомендации:

- **WinForms** надо знать и уметь применять. При трудоустройстве можно столкнуться со старыми программами, которые работают под WinForms и нуждаются в поддержке. Но не нужно сильно вкладываться в изучение этой технологии — достаточно поверхностного уровня.
- **WPF** следует изучать более серьезно. Если создаете новое приложение — привлекайте именно эту технологию, даже если не нужно создавать красивый дизайн. Вы будете более востребованы на рынке труда, если хорошо освоите WPF.

Рекомендации по созданию приложений

С приложением, которое рассылает электронные письма, есть два пути дальнейшего развития:

1. Продолжать развивать приложение и «по ходу пьесы» вводить новые возможности в программу.
2. Сесть и хорошенько подумать, что хотим от программы:
    a. Какие цели мы преследуем, что хотим получить?
    b. Как должен выглядеть интерфейс приложения, дизайн?
    c. Какова структура программы?

    То есть написать техническое задание.

Первый путь приводит к спагетти-коду, когда непонятно, какой метод когда вызывается и какая переменная для чего нужна. Без четкой цели не определен и результат. Поэтому выбираем второй путь.

Постановка цели

С постановки цели и составления технического задания обычно начинается создание нового проекта

или приложения. В компаниях для этого собираются митинги, мозговые штурмы, заказчикам отправляются брифы, анкеты. Если приложение не очень большое и под силу одному программисту, то достаточно посидеть и подумать, как оно должно быть устроено.

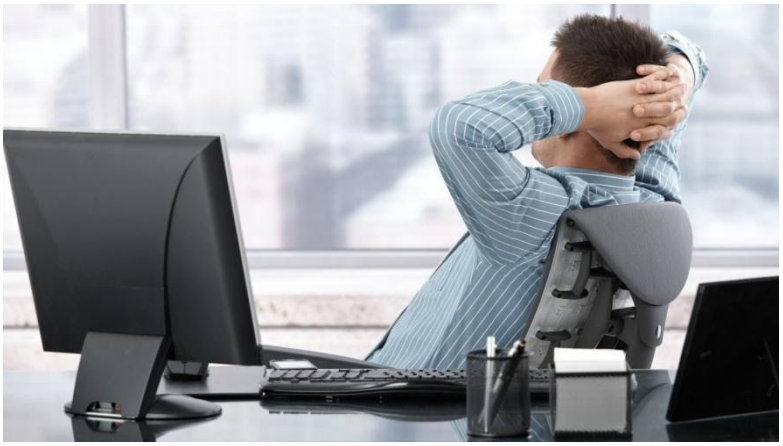
Вы спросите, почему урок не начался с пункта «Постановка цели». Чтобы быть ближе к реальной жизни: сначала стоило бы изучить задачу, действительно ли можно отправить письмо программно, и хотя бы поверхностно — новую технологию. И затем с уверенностью, что решить вопрос рассылки средствами .NET с использованием технологии WPF возможно, есть смысл продумывать цель.

План по созданию программного продукта

1. Ответить на вопросы: «Для чего нужно приложение?», «Кто его будет использовать?», «Что оно будет делать?»
2. Определить структуру программного продукта.
3. Продумать интерфейс или внешний вид приложения и нарисовать его на бумаге.
4. Создать прототип программы, который содержит весь графический интерфейс (кнопки, диалоговые окна и другое), но не обладает функциональной полнотой.
5. Выполнить кодинг: добавить функциональные блоки, методы в программу.

1. Для чего нужно приложение?

Откиньтесь на спинку кресла и подумайте, чего вы хотите от приложения, что оно должно делать.



Подумайте в комфортной обстановке. Запишите мысли.

Вот какое приложение мы будем делать:

1. Приложение «Рассыльщик», которое занимается рассылкой электронных писем по адресатам из базы данных.
2. В БД хранятся:
    a. email-адреса, имена и другая информация;
    b. email-адреса, от чьего имени делается рассылка, а также smtp-серверы и порты;
    c. тексты писем для отправки, их названия и ссылки на файлы, которые нужно к ним прикрепить;
    d. статистика, кому и что отправили.
3. В приложении можно:
    a. выбирать из базы набор адресатов; добавлять, редактировать и удалять адресатов;

загружать из файла несколько адресатов сразу;

b. выбирать адрес, от чьего имени отправляется письмо, и smtp-сервер; добавлять, редактировать и удалять отправителя;

c. составлять календарный план рассылки;

d. отправлять письмо по нажатию кнопки;

e. устанавливать напоминание, что нужно составить и отправить рассылку к определенной дате и времени;

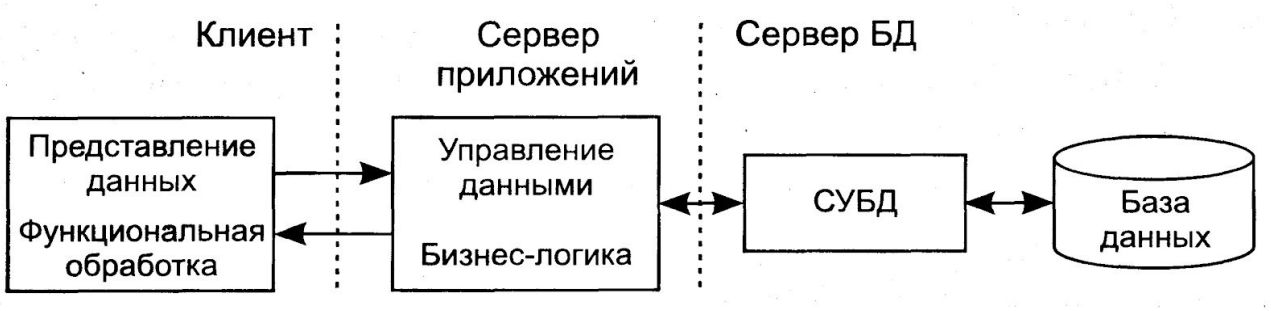
f. смотреть статистику отправленных писем;

4. Использовать приложение будем сами, чтобы рассылать письма, а также продемонстрируем потенциальному работодателю на собеседовании.

Структура программного продукта

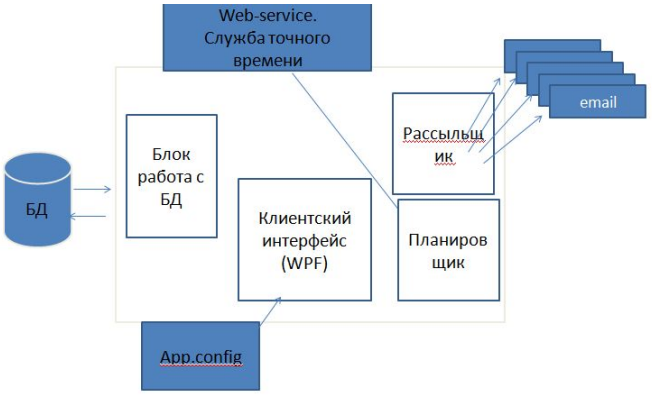
Приложение может измениться как по структуре, так и по внешнему виду. На начальном этапе все должно быть сделано качественно.

В рекомендациях по архитектуре приложений чаще всего упоминается трехзвенная модель:



Есть клиентское приложение, которое установлено на компьютере клиента — так называемый тонкий клиент. На нем находится только интерфейс. Он в свою очередь отправляет задание серверу приложения, который содержит всю бизнес-логику и занимается рассылкой писем. И отдельно расположен сервер БД, к которому обращается сервер бизнес-логики. Все сервера находятся на разных компьютерах.

Так выглядит идеальный программный комплекс, а мы сделаем его немного проще. Будет одно приложение на компьютере, к которому присоединяется база данных.



Это структура приложения, которое будем создавать. Состоит оно из нескольких блоков. По сути оно похоже на идеальный программный комплекс, созданный по принципу трехзвенной архитектуры. Те блоки, которые находятся на отдельных серверах в идеальном программном комплексе, у нас

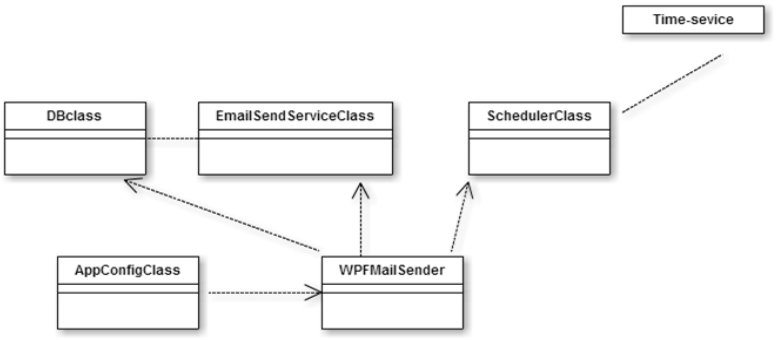
располагаются в разных блоках одной программы.

Каждый отдельный блок — это отдельный класс или сборка в зависимости от ситуации.

Старайтесь делать так, чтобы каждая логическая структура программы была отдельным классом. На нашей первоначальной схеме можно увидеть пять классов и веб-службу точного времени.

- **Блок работы с базой данных.** Класс, который отвечает за подключение к БД, за получение данных из нее, за добавление новых адресатов и другие манипуляции с данными.
- **Клиентский интерфейс.** Класс главного окна программы. Могут появиться другие классы для вспомогательных форм.
- **Рассыльщик**. Класс, который отвечает за логику программы. Именно он производит рассылку писем.
- **Планировщик**. Класс, который планирует рассылку, присылает уведомления.
- **App.config**. Это файл, в котором хранятся настройки приложения. Ему будет соответствовать отдельный класс, который будет отвечать на работы с этим файлом.
- **Web-service** — служба точного времени, нужна для планировщика и для того, чтобы напомнить, как подключаются веб-службы.

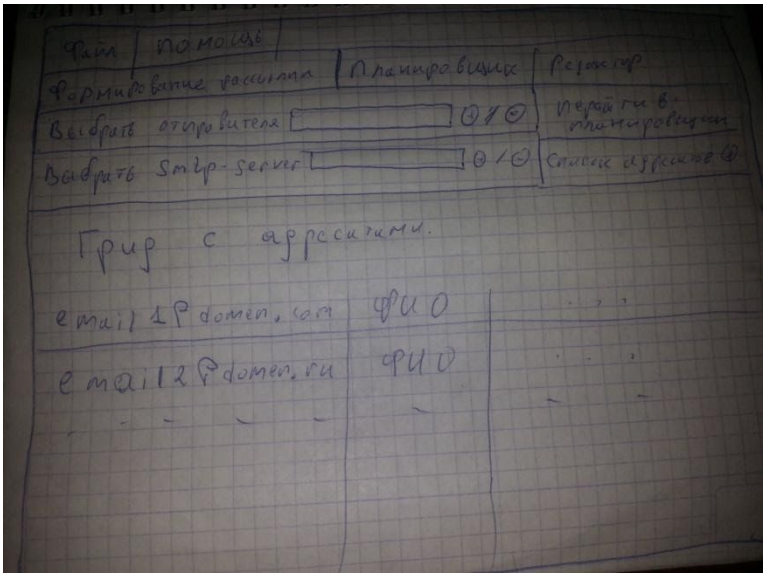
Чтобы лучше понимать, как должно быть устроено приложение, рекомендуем использовать UML-схемы. Они удобны, когда нужно понять, как должен быть устроен программный продукт: что он должен делать и в какое время, как взаимодействует с пользователем. UML стоит изучить, это повысит вашу ценность на рынке труда. Этот простой и наглядный язык действительно упрощает понимание, что будет делать программа или как уже действует приложение.



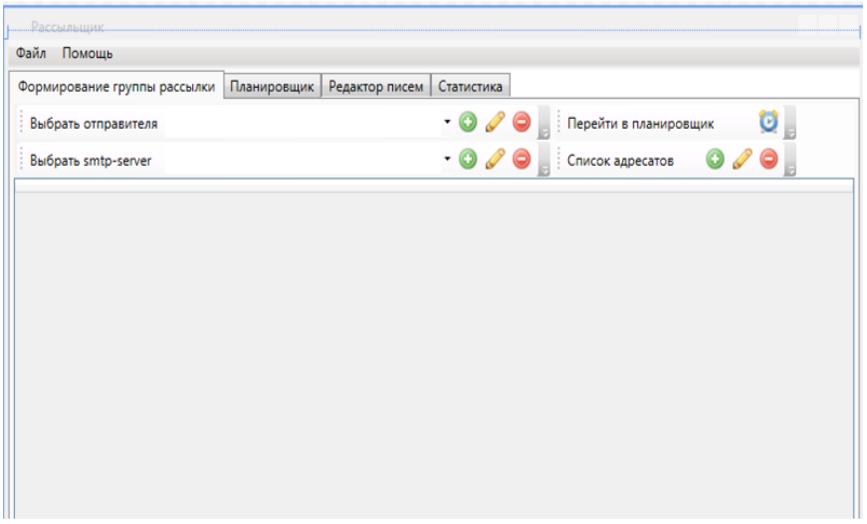
Это UML-диаграмма классов, которые предположительно будут использоваться в нашем приложении. Она похожа на схему структуры приложения, которая была представлена выше. При разработке серьезного корпоративного программного продукта всегда рисуется множество схем, диаграмм — не жалейте времени на продумывание структуры приложения. Зачастую бывает так, что на формулирование цели программы, написание ТЗ, рисование схем и диаграмм уходит больше сил и времени, чем на программирование.

Визуальное воплощение приложения

Чтобы продумать, как будет выглядеть приложение, нарисуйте в тетради формы и контролы. Пример эскиза:



Скриншот приложения:



Есть основное окно, на котором расположен **TabControl** с вкладками. На первой — возможность выбрать smtp-сервер, список адресатов и почтовый ящик, от чьего имени отправляются письма. На второй вкладке — планировщик с возможностью отправить письма сразу или установить время и дату рассылки. На третьей — просто **TextBox**, где будет сохраняться текст письма. На последней вкладке — таблица со статистикой: кому, когда и какие письма мы отправляли.

На третьем пункте плана заканчивается подготовительная работа. Мы сформулировали цель и четко представляем, что нужно делать, чтобы создать приложение. Оставшиеся четвертый и пятый пункт — непосредственно программирование.

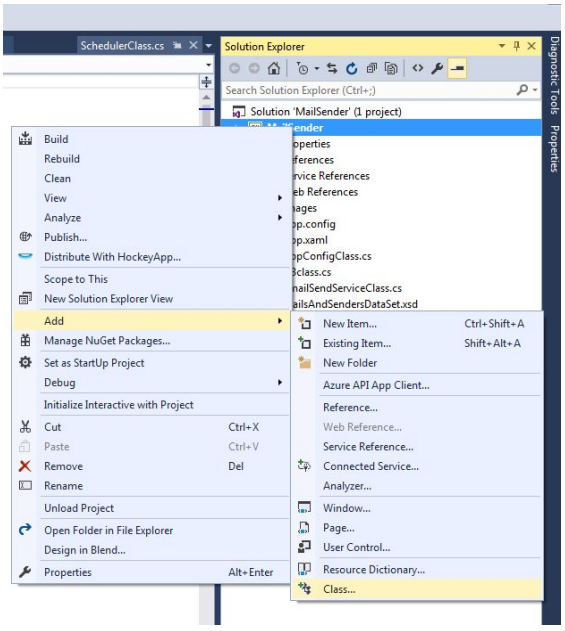
Приложение «Рассыльщик». Интерфейс и WPF

Четвертый пункт плана — создать прототип с интерфейсной частью. Пятый — создать тело приложения, выполнить кодинг. Возможно, на этом будем добавлять что-то в интерфейс или его изменять, но это не означает возврат к четвертому пункту. И создавая прототип, мы не должны педантично добавлять все контролы, формы и стиль — только оформить примерный внешний вид и внутренний каркас приложения. А вот заполнить его кодом и отшлифовать, а также отладить — это уже пятый пункт плана.

Прототип приложения «Рассыльщик»

Открываем **Visual Studio** и создаем новый WPF-проект. Назовем его **MailSender**.

В соответствии с внутренней структурой в нашем приложении должно быть пять классов. Сразу добавим их в проект. Кликаем правой кнопкой мыши по названию проекта, нажимаем «Добавить» и добавляем класс. Повторяем это четыре раза (один класс уже создан).



Вспомним правила наименования классов и переменных:

1. Классы начинаем с заглавных букв.
2. Переменные — с маленьких.
3. Без пробелов и подчеркиваний.
4. Каждое новое слово — с заглавной буквы.
5. Названия — подробные и наглядные.

В соответствии с этими правилами добавим классы:

1. **AppConfigClass** — для работы с **App.config**;
2. **DBClass** — для работы с базой данных;
3. **EmailSendServiceClass** — класс, который отвечает за рассылку писем;
4. **SchedulerClass** — планировщик;
5. Класс, который отвечает за интерфейс, добавился автоматически при создании проекта.

Компоновочные элементы управления в системе WPF (панели)

Вернемся к визуальному интерфейсу. Если вы работали в Windows Forms, то помните, что на рабочей поверхности для размещения контролов использовались абсолютные координаты: явно заданные *x* и *y*. В WinForms и WPF концепция позиционирования контролов (или элементов управления) отличается. В технологии WPF есть компоновочные элементы управления, или панели, которые нужны для позиционирования на своей поверхности других элементов управления. Панели невидимы и определяют поверхность, на которой работает программист, — так как поверхности, предназначенной для позиционирования элементов управления, в WPF нет. Компоновочные

элементы управления в иерархии XAML-файла находятся сразу под корневым узлом.

Существует несколько основных компоновочных элементов:

1. **Grid** — по умолчанию помещается на поверхность окна приложения. Определяет гибкую область сетки, состоящей из строк и столбцов. Дочерние элементы Grid могут быть расположены с помощью свойства **Margin**.

2. **Canvas** — использует абсолютную систему координат, как принято в WinForms.

3. **DockPanel** — определяет область, внутри которой можно упорядочить дочерние элементы по горизонтали или по вертикали относительно друг друга.

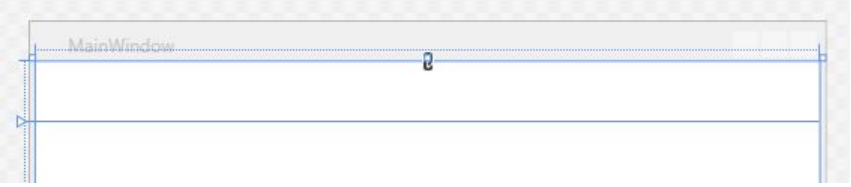
4. **StackPanel** — располагает дочерние элементы в одну строку, которую можно ориентировать по горизонтали или по вертикали.

5. **WrapPanel** — размещает дочерние элементы слева направо, перенося содержимое на следующую строку на границе поля. Дальнейшее упорядочивание происходит последовательно сверху вниз или справа налево в зависимости от значения свойства **Orientation**.

Разработчик может по своему усмотрению заменить компоновочную панель на ту, которая ему больше подходит, или разместить одну панель на поверхности другой.

Панель **Canvas** в WPF используют крайне редко, поэтому рассмотрим **Grid**. Она позволяет разделить свою поверхность на ячейки, в которых можно разместить другие контролы. **Grid** разбивается на ячейки при помощи свойств **RowDefinitions** и **ColumnDefinitions**. Это коллекции колонок и рядов, пересечения которых и определяют ячейки.

Рассмотрим это на практике. Если подвести мышь к левому краю **Grid**, то на ней появится линия, которая определяет, в каком месте размещается граница ячейки. Кликните мышью в получившейся ячейке, состоящей из одного ряда, чтобы в ней можно было бы разместить контрол меню.

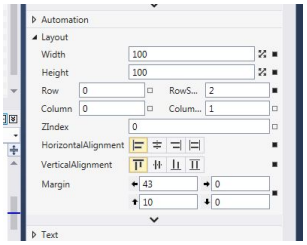


Посмотрите на XAML-код, который появился:

```
8         mc:Ignorable="d"
9         Title="MainWindow" Height="350" Width="525">
10    <Grid>
11        <Grid.RowDefinitions>
12            <RowDefinition Height="39*"/>
13            <RowDefinition Height="281*"/>
14        </Grid.RowDefinitions>
15
16    </Grid>
17</Window>
```

Если удалим свойства **Height** из **RowDefinition**, то черта, определяющая ряды, переместится в середину окна. Оставим определение ряда ближе к верху и поместим на форму «Меню» контрол. Надо выбрать его из **ToolBox** и перетащить мышью на экран, чтобы он попал на нужный ряд.

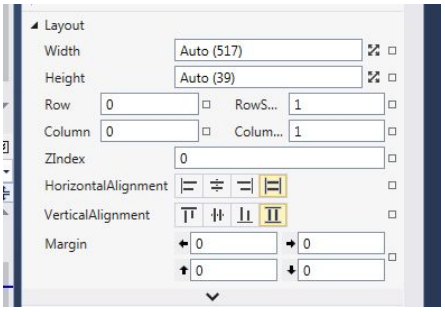
Заходим в свойства этого контрола, в категорию **Layout**.



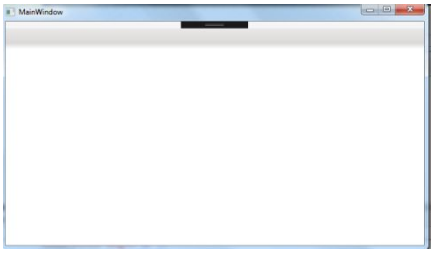
Видим, что под **Width** и **Height** появились определения, в каком ряду и колонке находится контрол (в полях **Row** и **Column** слева). Справа — поля **RowSpan** и **ColumnSpan**: в них содержится определение, на какое количество рядов и колонок растягивается контрол. На поверхности **Grid** в нашем случае находится два ряда и всего одна колонка. Так как пункт меню будет только в верхней ячейке, оставляем значения **Row** и **Column** равными нулю. **RowSpan** сейчас равен 2, а нам нужно, чтобы меню находилось только в верхнем ряду. Поэтому занимать оно будет только ячейки верхнего ряда — для этого устанавливаем свойству **RowSpan** значение 1. Поля **Width** и **Height** устанавливаем в значение **Auto**: нажимаем на маленький квадрат справа от поля и выбираем **Reset**.

Свойство **Margin** определяет, на каком расстоянии от границ ячейки будет находиться контрол. Так как мы хотим, чтобы он занимал все пространство ячейки, так же нажимаем на маленький квадрат справа и выбираем **Reset**.

В заключение выбираем свойства **HorizontalAlignment** и **VerticalAlignment**, выбираем крайние справа значения — то есть растягиваем по всей длине и ширине ячейки. Получаем такие свойства:



Теперь запускаем проект, нажимаем кнопку **Start** и видим окно с меню на самом верху:



Если поиграть с размерами окна, то можно увидеть, что меню становится шире, если окно расширяется. Не принято, чтобы меню меняло размер в зависимости от окна. Поэтому **Grid** в качестве основной панели для размещения контролов для нашего приложения не совсем подходит. Эта панель хороша для «резинового» интерфейса. Мы можем разбить его на колонки и ряды и размещать контролы в ячейках, и они вместе будут менять размер. **Grid** часто используют для

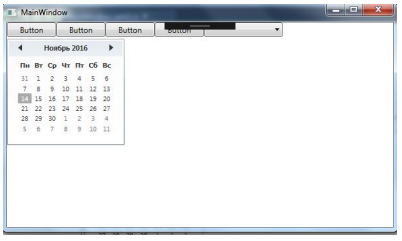
WPF-приложений в качестве основной панели, поэтому его стоит изучить.

Рассмотрим оставшиеся панели **StackPanel**, **DockPanel**, **WrapPanel**. Удалим **Grid** и **Menu** из xaml-кода. Поместим вместо **Grid** в качестве основной панели **WrapPanel**. Можно в xaml-коде добавить узел.

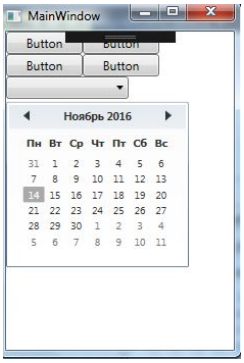
```
<Window x:Class="WpfApplication9.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfApplication9"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <WrapPanel >
    </WrapPanel>
</Window>
```

Можно просто кинуть панель из **ToolBox** на поверхность окна. В этом случае нужно удалить все лишнее (размеры и другие свойства), что соответствует **WrapPanel**, из xaml-кода.

Добавим на поверхность **WrapPanel** несколько контролов:



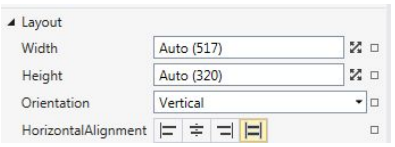
Если делаем окно шире, то и контролы растягиваются в одну линию. Если сжимаем окно по длине, контролы тоже перемещаются влево.



Это интересное свойство панели, но не совсем подходит для наших задач.

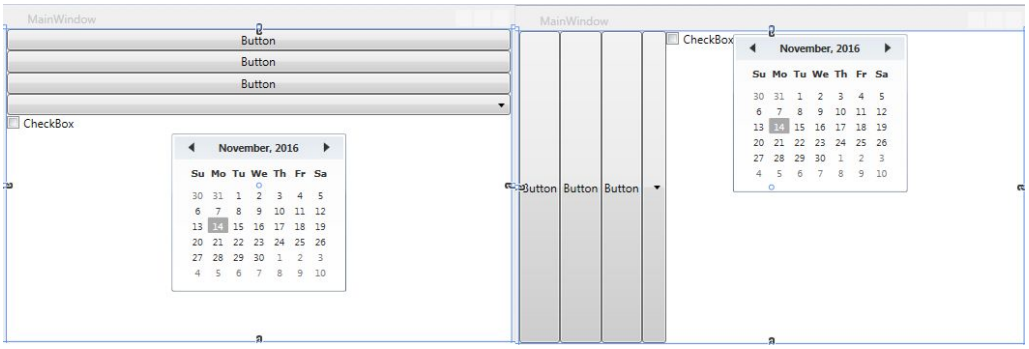
Рассмотрим **StackPanel**. Удалим **WrapPanel** и все контролы, находящиеся на ней. Разместим **StackPanel** на нашей форме.

В свойствах **StackPanel**, в категории **Layout**, есть свойство **Orientation**:



По умолчанию оно установлено как **Vertical**. Добавим несколько контролов на панель.

Посмотрите, как они аккуратно располагаются сверху вниз. Теперь поменяйте **Orientation** с **Vertical** на **Horizontal**. Теперь контролы размещаются слева направо.



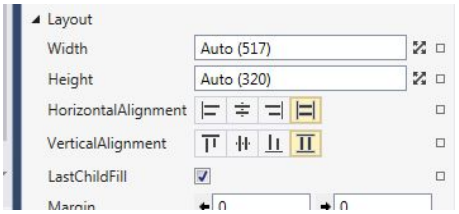
Запустите приложение и с вертикальной, и с горизонтальной ориентацией.

StackPanel удобно использовать, когда нужно аккуратно разместить несколько контролов подряд либо слева направо, либо сверху вниз.

Некоторые панели можно использовать не только как основные, но и как вспомогательные. Если в качестве основной панели использовать **Grid**, то одну из других (например, **StackPanel**) можно разместить на той части **Grid**, где хочется расположить контролы определенным образом.

Рассмотрим **DockPanel**. Удаляем **StackPanel** и все контролы на ней. Размещаем **DockPanel** в качестве основной панели. **DockPanel** определяет область, внутри которой можно упорядочить дочерние элементы относительно друг друга по горизонтали или по вертикали.

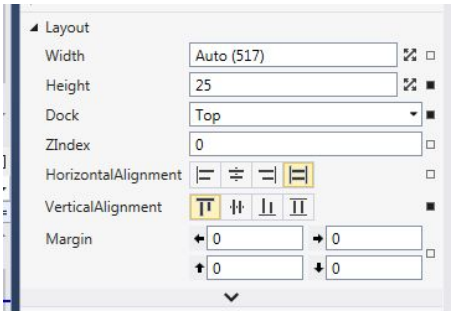
Посмотрите на свойства панели в категории **Layout**:



Появилось свойство **LastChildFill**. Если выбрать его, то последний контрол, который мы разместим на панели, заполнит все оставшееся место.

Поместим меню «Контрол» на нашей форме. Кинем его на **DockPanel** и увидим, как оно заполнило всю поверхность формы. Чтобы исправить это, идем в свойства меню, в категорию **Layout**:

1. В свойстве **Width**, если оно не выставлено как **Auto**, делаем **Reset**.
2. Свойство **Height** определяет высоту меню. В этом измерении оно не меняет размеры, поэтому не может быть **Auto**. Устанавливаем ему значение 25.
3. Появилось свойство **Dock**, выбираем **Top**.
4. **HorizontalAlignment** делаем растянутым по всей ширине.
5. **VerticalAlignment** — делаем **Top**.
6. **Margin** — если значения отличаются от 0, делаем **Reset**.

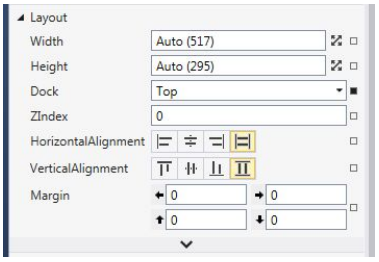


Теперь добавляем на форму **TabControl**. Надо, чтобы он заполнил все оставшееся место на нашем приложении. Идем в свойства: в **Width** и **Height** делаем **Reset**, чтобы они были **Auto**.

У свойства **Dock** нет значения **Fill**, как это было в **Windows Forms**. Вместо **Fill** у **DockPanel** есть свойство **LastChildFill**. Так что у свойства **Dock** просто выбираем **Top**.

У свойств **HorizontalAlignment** и **VerticalAlignment** выбираем **Stretch** (крайнее справа свойство). Оно означает, что контрол будет максимально растянут по длине и ширине.

В свойстве **Margin**, если там не все нули, делаем **Reset**.



Запустим приложение и посмотрим, как выглядит его окно. Именно **DockPanel** будем использовать в качестве основной панели.

Домашнее задание

1. Рассмотрим код приложения, при помощи которого мы протестировали возможность отправлять электронные письма и начали изучать WPF. В коде есть несколько моментов, которые простительны для теста, но не для серьезного приложения.

    a. Первый — жестко заданные переменные в коде. В строке **new SmtpClient("smtp.yandex.ru", 25**) таких две: "**smtp.yandex.ru**" — smtp-сервер и 25 — порт для него. В коде много и других жестко заданных переменных: адреса почтовых ящиков, тексты писем, тексты ошибок и другое.

    Задание: добавить в проект **WpfTestMailSender *public static class*** без конструктора и методов. Определить в этом классе статические переменные и задать им значения. В коде использовать эти переменные вместо жестко заданных.

    b. Второй момент — тот код, что описывает форму, и тот, который занимается непосредственно рассылкой, содержатся в одном классе.

    Задание: добавить к проекту **WpfTestMailSender *public class*** с конструктором, назвать его **EmailSendServiceClass**. Создать в этом классе методы (один или несколько), которые будут заниматься непосредственно рассылкой писем. Причем класс надо создать таким образом, чтобы его было легко перенести в другой проект.

    c. Третий сомнительный момент — в коде присутствует **MessageBox** с выводом ошибки, если невозможно отправить письмо. В принципе, это не криминал, и даже в серьезных проектах **MessageBox** присутствуют, но окно со своим стилем выглядит лучше.

Задание: по аналогии с формой, которая выводит сообщение «Работа завершена», создать еще одну для вывода текста ошибки шрифтом красного цвета. Добавить кнопку «ОК», которая будет закрывать форму.

2. Задание на укрепление знаний в технологии WPF:

    a. Добавить на главное окно тестового проекта **WpfTestMailSender**, в любое место формы, два контрола **TextBox** — одно для названия письма, второе — для его текста. Сделать так, чтобы и название, и текст брались из этих контролов.

    b. Скачать библиотеку стилей с сайта wpfthemes, как в главе «Изменение стиля приложения WPF». Выбрать любую тему и установить, как описано в этом уроке.

    c. Поиграть с контролами тестового приложения **WpfTestMailSender**: поменять им свойства, поразмещать в разных местах окна. Поменять основную панель **Grid** на другие панели, рассмотренные на этом уроке.

3. Заменить название основного окна и класса приложения **MailSender**, **MainWindow** на **WpfMailSender** — по аналогии с тем, как мы меняли название главного окна у тестового приложения **WpfTestMailSender** на уроке.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Ник Рандольф, Дэвид Гарднер, Майкл Минутилло, Крис Андерсон. Visual Studio 2010 для профессионалов.
2. MSDN.
3. Меню в WPF
4. Михаил Смирнов. Стандарты и правила оформления кода C#.