# Introduction

When I started creating WPF applications, I just mixed all markup directly in the WPF code. Gradually, I discovered this is not the way to go.

* Re-use of style information is hard.
* Changing what your application looks like is a lot of work.

This article intends to help you to create a style library that separates markup from WPF functional layout and makes re-sue a bit easier. I do not claim to have a perfect solution, and like to invite you to express your ideas and improvements.

You can download a demo project from Github.

You need some knowledge on WPF (elementary) and you need a basic knowledge of Resource Dictionaries to understand this article.

# Application setup

The demo application has two projects:

* StyleDemo.DesktopUI a WPF .Net C ore 3.1 project. This is only for demo and testing purposes.
* Styles.Library a WPF User Control library, .Net Core 3.1 Make sure to use a user control library and not an ordinary class library.

This should work for .Net framework as well.

The Destop UI depends on the Styles.Library project, so do not forget to include these dependencies.

You do not need any Nuget packages. I use Visual Studio Community Edition 2019, version 16.5.

# Setting up the Styles.Library project

The basic concept is to create a number of resource dictionaries. To avoid the need to create references to each individual dictionary, first a dictionary that collects all other dictionaries is created. I use the convention to put the term “Dictionary” in the name of each resource dictionary.

In the demo I name it StylesDictionary.

To be able to test this, you need at least one dictionary to include. In the user control library. To do this, I create two empty dictionaries:

* ColorSchemaDictionary which will contain all colors used in the application.
* SizeSchemaDictionary for all sizes we like to give standard values.

The code for Styles.Dictionary looks like this:

<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">

<!-- https://www.codeproject.com/Articles/35346/Using-a-Resource-Dictionary-in-WPF -->

<ResourceDictionary.MergedDictionaries>

<!-- Basic markup -->

<ResourceDictionary Source="ColorSchemaDictionary.xaml" />

<ResourceDictionary Source="SizeSchemaDictionary.xaml" />

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

As the number of dictionaries grows, you can add more dictionaries.

Now, we can make this StylesDictionary available in in the desktop application.

To do this, adapt App.xaml, to refer to this resource dictionary:

<Application x:Class="StyleDemo.DesktopUI.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

StartupUri="MainWindow.xaml">

<Application.Resources>

<ResourceDictionary>

<ResourceDictionary.MergedDictionaries>

<ResourceDictionary Source="pack://application:,,,/Styles.Library;component/StylesDictionary.xaml"/>

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

</Application.Resources>

</Application>

Make sure to get all elements in the complex URI right. There is a complex explanation for this, but I can live without fully understanding the logic behind this.

If it is not working, Make sure your desktop project has a dependency set to the Styles.Library dll.

Now the application should run as before.

# Setting colors

The ColorSchemaDictionary is intended to collect all colors. To give a idea of the syntax, as very simple schema is created, just enough to show how it works. You need to set up colors as brushes. In this example I only use solid brushes, but it also works for other brush types. If you

<!-- Window colors -->

<SolidColorBrush x:Key="WindowBackground" Color="LightBlue" />

<SolidColorBrush x:Key="WindowBorderBrush" Color="CornflowerBlue" />

<SolidColorBrush x:Key="ControlBackground" Color="LightBlue" />

<SolidColorBrush x:Key="TextBoxBackground" Color="Oldlace" />

<SolidColorBrush x:Key="HeaderBackground" Color="DarkGray" />

<!-- Border colors -->

<SolidColorBrush x:Key="BorderDefault" Color="DarkBlue" />

<SolidColorBrush x:Key="BorderAlert" Color="OrangeRed" />

<!-- Text colors -->

<SolidColorBrush x:Key="LabelText" Color="Black" />

<SolidColorBrush x:Key="DataText" Color="Black" />

<SolidColorBrush x:Key="AlertText" Color="OrangeRed" />

<!-- Button colors -->

<SolidColorBrush x:Key="ButtonBackground" Color="DarkBlue" />

<SolidColorBrush x:Key="ButtonText" Color="Lavender" />

<SolidColorBrush x:Key="ButtonDisabled" Color="Gray" />

<SolidColorBrush x:Key="ButtonHover" Color="CornflowerBlue" />

<SolidColorBrush x:Key="ButtonPressed" Color="LightBlue" />

If you view this code in Visual Studio, it will show small color samples as well. The big advantage is, that you have all colors at one place, which makes it easy to review if you have a nice balance.

You should consider naming carefully. Intellisense will work, but if you start typing a B, it will only show everything starting with the character B. Therefore I prefer to mention the control type first in the name and then a descriptive text of what the function of the resource is. This may seem an open door, but I still regret the projects where I did this in a wrong order, e.g. setting things like DefaultWindowBackgroundColor and so on. The you have a long list of Default to search through.

You can refer to these colors directly from the desktop:

<Button Background="{StaticResource ButtonBackground}"

Foreground="{StaticResource ButtonText}">Test button</Button>

This code will create a window filling blue button, with a more or less white text. As you see in the code above, this way of using named resources is very cumbersome, so we will do it better. The button still uses the whole window size. This is not what you normally want, so the next step is to define some default dimensions.

# Setting dimensions

For this purpose the SizeSchemaDictionary can be used.

<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:system="clr-namespace:System;assembly=System.Runtime">

<Thickness

x:Key="MarginDefault"

Bottom="5"

Left="5"

Right="5"

Top="5" />

<Thickness

x:Key="MarginSmall"

Bottom="3"

Left="3"

Right="3"

Top="3" />

<Thickness

x:Key="PaddingDefault"

Bottom="2"

Left="2"

Right="2"

Top="2" />

<Thickness

x:Key="PaddingSmall"

Bottom="1"

Left="1"

Right="1"

Top="1" />

<!-- Button dimensions -->

<system:Double x:Key="ButtonDefaultWidth">100</system:Double>

<system:Double x:Key="ButtonWideWidth">120</system:Double>

<system:Double x:Key="ButtonDefaultHeight">30</system:Double>

</ResourceDictionary>

In the example you see that Margin and Padding are represented by a Thickness object. The height and with of , for instance a button must be set as a double. This is not obvious, and it may take some research to find out how to do this in different cases. Until now I only use these two ways to specify dimensions.

If we apply these settings to the butting, it goes like this:

<Button Background="{StaticResource ButtonBackground}"

Foreground="{StaticResource ButtonText}"

Width="{StaticResource ButtonDefaultWidth}"

Height="{StaticResource ButtonDefaultHeight}"

Margin="{StaticResource MarginDefault}"

Padding="{StaticResource PaddingDefault}">Test button</Button>

In this way your markup is much more re-useable, but your xaml specs will grow large and it still is a lot of typing.

One comment. I tend to apply this MarginDefault to every single control in this way. There may be other options, e.g. to apply margin at the right and bottom sides only. I noticed that your markup soon gets ugly if the way you apply margin is not very consistent. Then you may need to apply manual fixes, to get controls properly aligned, which may result in more inconsistencies, resulting in broken markup if you change anything. Because I apply the same margin to all controls, everything looks well aligned always.

# Defining simple styles

Markup tends to be large and not very readable. To make this better, you can define styles for your controls. Because this tutorial focusses on setting up the way of working, defining complex styles will not be covered.

It is possible to change the default style for each control. Being lazy, I tried to do so, but in a number of cases you will run into trouble. The reason is that controls may be used in other controls. If you start applying fancy default styles, these styles will be applied also where you really do not want this. Therefore, 99% of the styles I use do have a key and must be applied explicitly. I learned my lesson here …

This article gives some nice examples: <https://ikriv.com/dev/wpf/TextStyle/>

So, as a rule, always specify x:Key to keep control over the useage of a style.

It helps to make a division in style files. For this demo, four additional style dictionaries will be created and wired up::

* WindowDictionary for Window styles
* ButtonDictionary for buttons
* TextBlockDictionary for TextBlocks
* TextBoxDictionary for TextBoxes

# WindowDictionary

As a first step, a resource dictionary for windows is created:

<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<!-- Window style -->

<Style x:Key="WindowDefault" TargetType="Window">

<Setter Property="Background" Value="{DynamicResource WindowBackground}" />

</Style>

</ResourceDictionary>

An important thing to note: if you create the resource dictionary inside the project where it is used, you can use StaticResource as resource type. For dictionaries created in a separate library project, always use DynamicResource as shown in the example. You can see the property here refers to another resource.

Now you must make sure the resource is known the library interface, so adapt StylesDictionary:

<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">

<!-- https://www.codeproject.com/Articles/35346/Using-a-Resource-Dictionary-in-WPF -->

<ResourceDictionary.MergedDictionaries>

<!-- Basic markup -->

<ResourceDictionary Source="ColorSchemaDictionary.xaml" />

<ResourceDictionary Source="SizeSchemaDictionary.xaml" />

<ResourceDictionary Source="WindowDictionary.xaml" />

<ResourceDictionary Source="ButtonDictionary.xaml" />

<ResourceDictionary Source="TextBlockDictionary.xaml" />

<ResourceDictionary Source="TextBoxDictionary.xaml" />

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

And finally, apply the style to the window (here you can use a static resource):

<Window x:Class="StyleDemo.DesktopUI.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"

mc:Ignorable="d"

Style="{StaticResource WindowDefault}"

Title="MainWindow" Height="450" Width="800">

<Grid>

<Button Background="{StaticResource ButtonBackground}"

Foreground="{StaticResource ButtonText}"

Width="{StaticResource ButtonDefaultWidth}"

Height="{StaticResource ButtonDefaultHeight}"

Margin="{StaticResource MarginDefault}"

Padding="{StaticResource PaddingDefault}">Test button</Button>

</Grid>

</Window>

The result should be a nice blue screen with a dark blue button in the centre.

When creating this style, I would have liked to make default for WindowStartupLocation as well. This will not work, because it is not a XAML dependency property. At stackoverflow you find a workaround. I guess it is also possible to derive your own window class and fix this there.

<https://stackoverflow.com/questions/10596515/setting-windowstartuplocation-from-resourcedictionary-throws-xamlparseexception>

This allows you to make all windows to look similar and you may change the background color easily.

# Some other controls

To show some slightly more complex examples, let’s make the style for the button more general in the ButtonDictionary: