Poul Klausen

C# 2

Programs With a Graphical User Interface

Software Development

bookboon
LEARNING

POUL KLAUSEN

C# 2: PROGRAMS WITH A GRAPHICAL USER INTERFACE

SOFTWARE DEVELOPMENT

C# 2: Programs with a graphical user interface: Software Development
1st edition

ISBN 978-87-403-3463-0

CONTENTS

FOREWORD

This book is the second in a series of books on software development. The programming language is C#, and the purpose of this book is to introduce the development of programs with a graphic user interface. The book requires a knowledge about programming and the language C# as explained in the book C# 1. In order to be able to write programs that used in practice, it is necessary today that they have a graphical user interface where the user communicates with the program using components in a window. The goal of this book is to introduce programming of Windows programs, and it requires quite a bit in comparison to what said in the previous book. In fact, I do not have presented important concepts about both object-oriented programming and C#, and I will therefore apply concepts first dealt with in the subsequent books. However, when I introduce GUI development already in this place, it due to the desire, quickly to enable the reader to write programs used to solve everyday tasks.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in C#. You can learn that by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it, and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance treated in the books. All books in the series are built around the same skeleton and will consist of text and examples, exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance presented in the text, and furthermore it is relatively accurately described what to do. Problems in turn, are more loosely described and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code for the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with a larger sample program, which focus primarily is on the process and an explanation of how the program is written. On the other hand appears the code only to a limited extent, if at all, and the reader should instead study the finished program code, perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

All sample programs are developed and tested on a Windows machine, and the development tool is Visual Studio. Therefore, you must have Visual Studio installed and Microsoft provides a free version that is fully adequate. Visual Studio is an integrated development environment that provides all the tools needed to develop programs. I do not want to deal with the use of Visual Studio, which is quite straightforward, and is a program that you quickly become comfortable with.

Finally a little about what the books are not. It is not "a how to write" or for that matter a reference manuals to C#, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

This book is an introduction to writing GUI applications in C# using WPF. Far from everything is treated, but conversely sufficient that you after reading the book can write small applications with a graphical user interface.

A program with a graphical user interface is a program that runs in a window. Since it is largely based on the principles of object-oriented programming, should the topic be examined only after the next two books in this series, and the following will use a lot of classes and interfaces which are first mentioned there. When I still have chosen to address GUI programs now, it is because I want to be able to write applications that are more interesting and better match the programs that you meet in everyday life, than what is possible purely as commands or console applications.

To write a program with a graphical user interface you must work on basis of a wide range of finished classes, for example classes that creates and opens a window, classes representing a button, classes for an input field, etc. These are very many classes, and they are assembled in an API called *WPF*. In the following I will call programs with a graphical user interface for *GUI* programs, where GUI stands for *Graphical User Interface*, and I will start to make it clear that one can not learn to write GUI applications by learning all the many classes and their methods and properties. Instead, you must learn some basic principles for the development of these kinds of programs, and once you have learned it, it is all not so difficult. You quickly get an idea of what it takes, and the question is then what the things are called and what you actually have to write. Here, however, there is only one way and that is to turn up the documentation, which fortunately is online available and there is also a variety of other sources on the Internet that provides advice on how to solve specific problems.

A GUI application will typically consist of several or many classes, but starting, you can think of each window as a class. In addition, the data, that a program must handle, usually also are defined using classes, and there will also be classes that implements the logic that must process the application's data. It is primarily in this context that I am missing something of the theory of classes and interfaces, as first are explained in the third book on object-oriented programming. The following will therefore to some extent deal with object-oriented concepts, but so that I postpone all the details for later.

The *Windows Presentation Foundation* abbreviated WPF is an API for developing graphical user interface applications and is an alternative to the former Windows Forms API. WPF is inspired of the development of web application and ASP.NET:

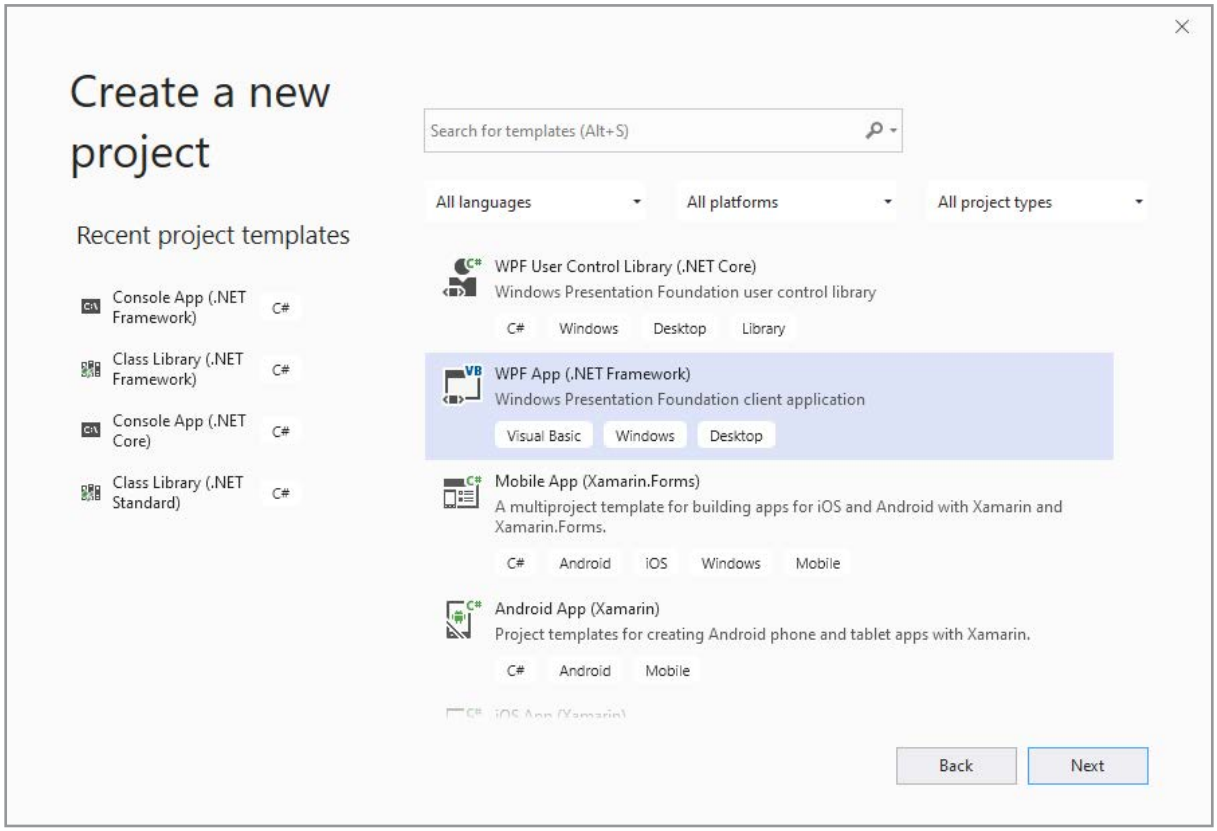
- The visual presentation in the form of HTML with web controls and then the code are clearly separated, so that the visual part and the code can be found in each there file.
- There are two ways to work with the visual presentation. Either by dragging components to a window with the mouse, or by working with the source where you can enter HTML yourself.

The two things are included in WPF, but where HTML is replaced by XML or an XML variant that Microsoft calls the *Extensible Application Markup Language* (XAML). Now you might think that it means you have to learn a new XML language, but it is not necessary as Visual Studio provides the necessary support and in fact, it is quite easy to work with a program by directly manipulating XML code.

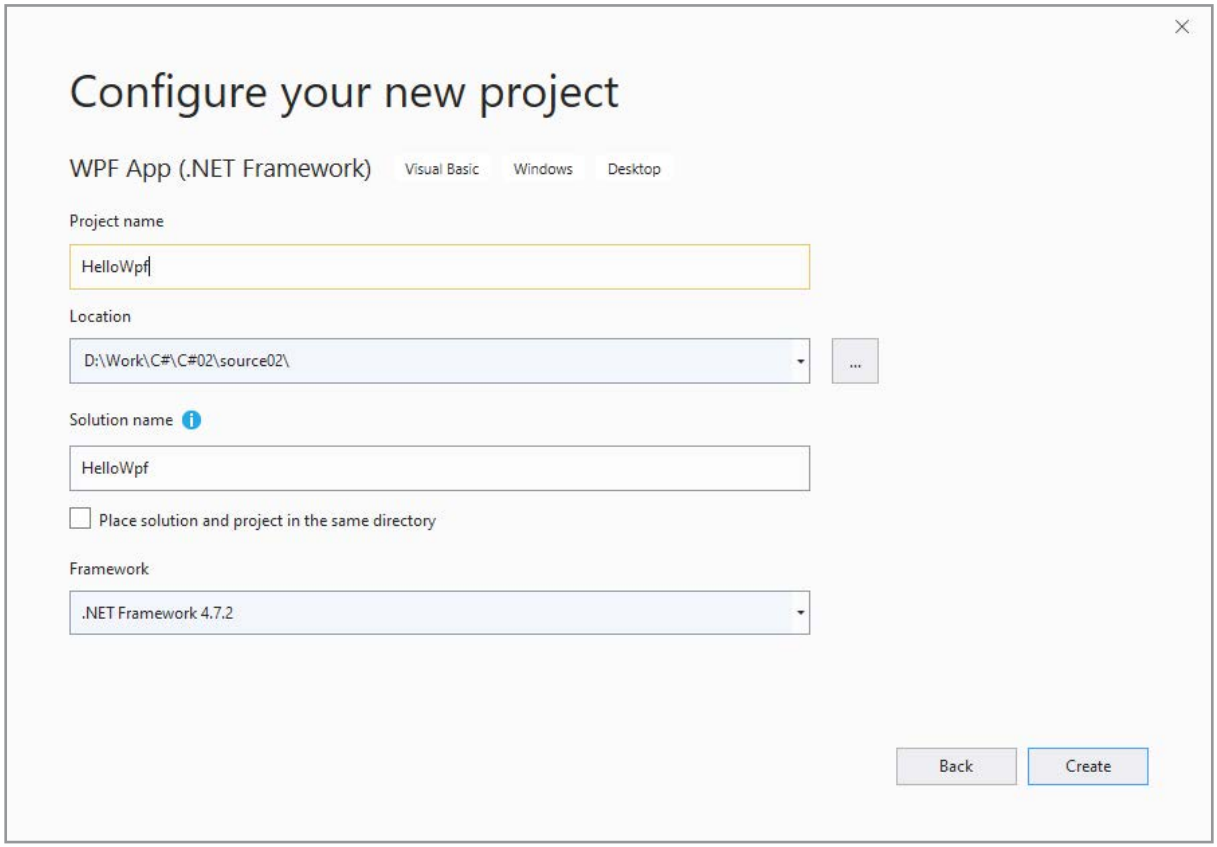
1.1 HELLOWPF

To show what it is all about I will to write a program, which I have called *HelloWpf*. The program should open a window with two fields and a button. The user can then enter a name and a job position, and when the user click the button the program must open a message box, which shows what the user has entered.

I start to create a new project in Visual Studio but this time it must be a *WPF App* project:

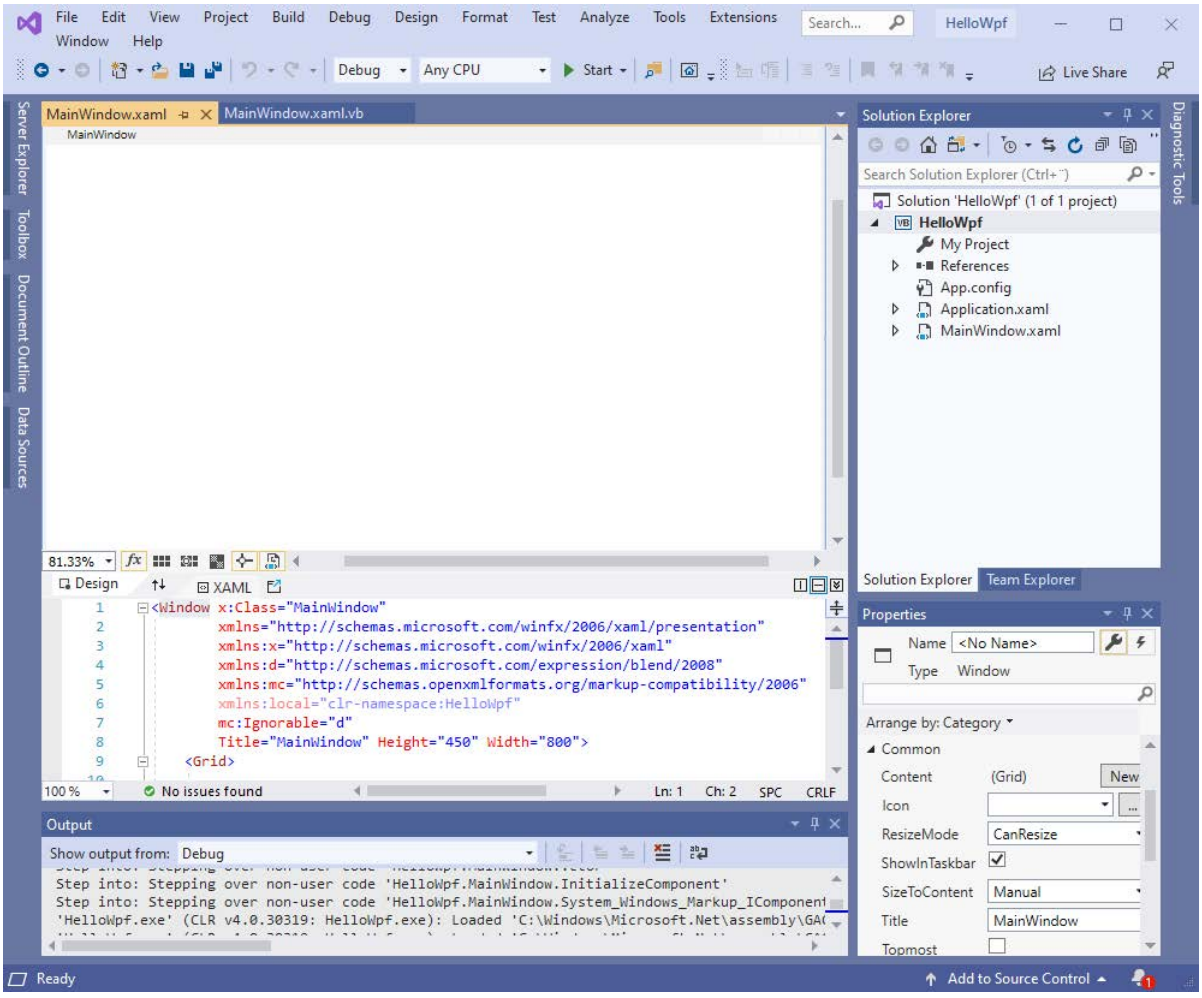


When I click the mouse on *Next* I get the next window where to enter the project name and select where to create the project (see below). You should note that it is the same window as you have seen before.



When you click the mouse on *Create* Visual Studio creates a project and opens the project window (see below) and the content of this window is different than before. To the left (and upper) Visual Studio shows how the program's window will look like when you run the program (for the moment an empty window), and below the window is the XML that defines the window, and although you can hardly see the meaning yet, you can at least see that it looks like XML. To the right is the Solution Explorer pane, which show the project's files. There is more files than shown in the previous book, but else it is the same tool. Below there is a *Properties* tool that can be used to modify properties for components in the user interface.

What Visual Studio has generated is actually a full-fledged program, and if you try to run the program, you get the following window:



At the top of the above window there is two tabs, and if you clicks on the tab *MainWindow.xaml.cs* you get a window which shows some code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

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

There is a lot of *using* statements, but else you can see that the code defines a class called *MainWindow* that inherits a class *Window*. The class has a constructor which calls a method *InitializeComponent()* which is a method that is created of the compiler based on the XML code on the fly and the method that creates the window. The code doesn't look like much, but at least you can see that it's a regular C# class.

If you in *Solution Explorer* click on the arrow in front of *App.xaml* you find a file *App.axml.cs* and if you open the file the content is:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;

namespace HelloWpf
{
  /// <summary>
  /// Interaction logic for App.xaml
  /// </summary>
  public partial class App : Application
  {
  }
}
```

It defines the class *App*, which inherits a class *Application* and is the class that starts the program.

As the next step I will design the window, and there are several possibilities. At the left in the window you find more hidden tabs, and if you open the tab *Toolbox* you get a list with components, which can be placed in a window. You can place a component in a window by double clicking the component in the toolbox, and in this case I have

1. placed two *Label* components in the window (double click *Label* two times)
2. placed two *TextBox* components in the window (double click *TextBox* two times)
3. placed a *Button* component in the window (double click *Button*)

All components are placed at the left upper corner of the window on top of each other, but if you look at the XML you can see, that Visual Studio has inserted five lines.

I have to design the window and I have dragged the components with the mouse to the positions where I want them, and I have changed the size of the two *TextBox* components. I have also changed the values for the components, which I have done directly by modifying the XML. For the two *Label* components and the *Button* I have changed the *Content* attribute, while for the two *TextBox* components I have changed the *Text* attribute:

Note that I also has changed the size of the window as well as the text in the title line. If you run the program it opens the above window, and you should note that I designed the window to write no C# code at all.

The window has five components and I want to give these components some readable names. If you select a component with the mouse, you can update the components properties in the *Properties* window by entering a value for the *Name* property. I have entered the following names:

1. The two *Label* components: *lblName*, *lblJob*
2. The two *TextBox* components: *txtName*, *txtJob*
3. The *Button* component: *cmdOk*

You should note that the XML is updated. The names, I have entered are used for variable names, when the program is compiled.

As the last thing, I must assign an *event handler* to the button that is a method, which is performed when the user clicks on the button. If I in the window double click the button, Visual Studio shifts to the tab with the code for the class *MainWindow*, and you will see that Visual Studio has added a method:

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

    private void cmdOk_Click(object sender, RoutedEventArgs e)
    {

    }
  }
}
```

For now, you should just accept the syntax as it is, but you can see that it is a method with two parameters. This method is performed, when the user runs the program and clicks the button, but I have to write what should happen and the code to be executed, when the method is performed. The final method is:

```
private void cmdOk_Click(object sender, RoutedEventArgs e)
{
  string name = txtName.Text.Trim();
  string job = txtJob.Text.Trim();
  if (name.Length > 0 && job.Length > 0)
  {
    string text = string.Format(
      "Hej {0}\nYour job is: {1}\nThank you for your contact", name,
      job);
    MessageBox.Show(text, "Information");
    txtName.Clear();
    txtJob.Clear();
    txtName.Focus();
  }
  else MessageBox.Show("You must enter both the name and job title",
  "Error");
}
```

The method starts to store what is entered in the two *TextBox* fields in two local variables. Note how I use the name of these fields and also note how to refer to text by the components *Text* property. Then the method tests if there are entered something in both fields, and if so, I create a string for the result. The string is shown in a *MessageBox* which is a small popup window. After the user has closed the message box the two fields are cleared and the *txtName* field is assigned focus. If the user not has entered text in both fields another message box opens and shows an error message.

If you run the program and for example enter:

and then click OK you get the message box:

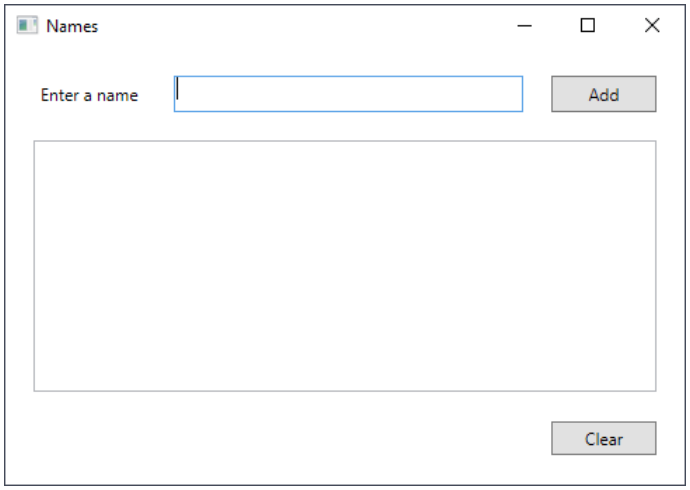
The result is a program with a graphical user interface, and although it is a simple program with limited functionality, it still has all the features that a Windows program has. You should especially note that the program is written without entering much code, and most of it is generated by Visual Studio by clicking and dragging the mouse. All the programmer has to do is to write the C# code for the button's event handler. Of course, you may not know that the event handler must be written that way, but when you see the code, it is easy enough to understand.

However, you should be aware that it is a C# program and such a program consists of classes with variables and methods. You don't see that much. The program has two classes, but it does not contain much. The important one is the class *MainWindow*, which is the class that represents the window, and here you should note that the class is defined *partial*, which briefly means that there must be something else. The class has another part that Visual Studio creates and that you do not immediately see, since it is Visual Studio that is responsible for maintaining this part, but it is by no means trivial and is comprehensive. When in the XML section I have chosen the names for the components, these names become the names of objects that represents the components, which is why in the event handler I can refer to the components using these names. I will later look at what it is for a code that Visual Studio generates and thus the connection between XML and C#, but Visual Studio uses the XML code to create the necessary C# code.

You are encouraged to study the XML code in details and there are some, but if you look through the code it is actually easy enough to understand.

1.2 ENTER NAMES

In this section I will write another WPF application. The program is basically identical to the above, but the program is written in a different way. Executing the program opens the following window:



Here the user can enter a name, and when the user click *Add* the name is added to the list box below, and the user can enter a new name. If the user click on the button *Clear* the content of the list box is removed. The window has as so five components like the previous example. An important difference is that if you resize the window, the upper input field (width only) and the list box will follow the size of the window.

To write the program I have created a new project called *EnterNames*, and the project is created in exactly the same way as the project *HelloWpf*. As the first step, I want to design the user interface, but instead of selecting components from the Visual Studio toolbox of subsequently placing the components with the mouse, this time I manually entered the XML code:

```
<Window x:Class="EnterNames.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:EnterNames"
        mc:Ignorable="d"
        Title="Names" Height="350" Width="500" >
  <Grid Margin="20">
    <DockPanel >
      <DockPanel DockPanel.Dock="Top">
```

```
        <Label Content="Enter a name" DockPanel.Dock="Left"
        Margin="0,0,20,0" />
        <Button Content="Add" Width="75"
          DockPanel.Dock="Right" Margin="20,0,0,0" />
        <TextBox />
      </DockPanel>
      <DockPanel DockPanel.Dock="Bottom">
        <Button Content="Clear" Width="75" Height="24" DockPanel.
        Dock="Right" />
        <Label Content="" />
      </DockPanel>
      <DockPanel Margin="0,20,0,20">
        <ListBox />
      </DockPanel>
    </DockPanel>
  </Grid>
</Window>
```

Looking at the code it can seem overwhelming and you may be unsure how to learn it, but it is not at all impossible again. First, note that it is limited how much XML I have entered, as it is simply the code enclosed by the *Grid* element. Secondly, Visual Studio gives so much help that you can easily write the code without having to remember it all. As soon as you press a key, Visual Studio comes up with suggestions for what to write.

To place components in a window WPF uses layout controls, and *Grid* is an example and is inserted by Visual Studio as default. A Layout control can contain other controls or components, which means that controls can be nested. In this case the *Grid* control contains a *DockPanel* which again contains three other *DockPanel*s. A *DockPanel* is a layout control, which divides the window in 5 areas: *Left*, *Top*, *Right*, *Bottom* and *Center*. Each of the first four areas can contain all the components you want, and they are placed in the order they are added to the panel. The center can only contain one component and this component is always the last to be added to the panel, and it will always fill the entire area of the window that is not occupied by the other four areas. If you look at the control, the attribute tells that the control should be placed *Top* in the outer *DockPanel*:

```
<DockPanel DockPanel.Dock="Top">
  <Label Content="Enter a name" DockPanel.Dock="Left" Margin="0,0,20,0" />
  <Button Content="Add" Width="75" DockPanel.Dock="Right"
  Margin="20,0,0,0" />
  <TextBox />
</DockPanel>
```

The control contains a *Label*, which should be placed *Left* in the control. The *Label* is not assigned a *Width*, and the width is then determined from the text. Next is added a *Button* component to the *DockPanel* such the *Button* is placed *Right*. As the last is added a *TextBox* and when it is the last component in the control it will use the space not used by the two other components, which means that the width of the *TextBox* is determined by the window size. Note that nothing is stated regarding the height of the components. The height is then determined by the current font and so the height of the *DockPanel*. After this *DockPanel* another *DockPanel* is added:

```
<DockPanel DockPanel.Dock="Bottom">
  <Button Content="Clear" Width="75" Height="24" DockPanel.Dock="Right" />
  <Label Content="" />
</DockPanel>
```

This *DockPanel* must be placed *Bottom* in the outer *DockPanel* and contains a *Button*, which must be placed *Right*. When the last component in a *DockPanel* always is placed *Center* and fill it all, is added an empty *Label*. The result is that *Button* is placed right in the window and follows the window's size.

The outer *DockPanel* has yet another component, which is a *DockPanel* with a *ListBox*. This *DockPanel* have no other components, and the *ListBox* will then fill it all, and when the *DockPanel* is the last component in the outer *DockPanel* it will fill that panel, and as a result the *ListBox* will use all space not used by the other components.

It seems a bit complicated whatever it is, but there are other layout controls, that can make it easier and with a little practice, it helps. Note, however, that the result is a stable design that behaves nicely as the window size changes.

Then there is the C# code, and as the next step I have assigned names to the components, but only the components to which references is made in the code:

1. The *Add* button: *cmdAdd*
2. The enter field: *txtName*
3. The list box: *lstNames*
4. The *Clear* button: *cmdClr*

Then I must assign event handlers for the two buttons. If in the XML for a button type *Click*= and press *Enter* will Visual Studio creates an event handler for the button. If I do so for both buttons and go the code there are two handlers, and I must write the code:

```
private void cmdClr_Click(object sender, RoutedEventArgs e)
{
  if (MessageBox.Show("Are you sure you want to delete the names?",
  "Warning",
    MessageBoxButton.YesNo) == MessageBoxResult.Yes) lstNames.Items.
    Clear();
  txtName.Focus();
}

private void cmdAdd_Click(object sender, RoutedEventArgs e)
{
  string name = txtName.Text.Trim();
  if (name.Length > 0) lstNames.Items.Add(name);
  txtName.Clear();
  txtName.Focus();
}
```

There is not much to explain but note how the user get a warning before deleting the content of the list box.

Now the program is complete and can be tested. Again, the program is written without writing much C# (although it is not a goal in itself), but this time I have directly written the XML code for the user interface. It is no better than clicking and dragging with the mouse, but with practice it is actually as easy and often the result is a more stable user interface.

1.3 A GRID CONTROL

As shown above Visual Studio as default add a *Grid* control to the XML for a window, and it is also the most important. The layout controls as *DockPanel* and other explained later are also important, but a *Grid* is the most flexible and then an example that shows how it works. The program *GridLayout* opens the following window:

The window shows seven *Button* components. The buttons have no function and the purpose is only to show how the buttons are placed in the window using a *Grid* control. The XML is as follows:

```
<Window x:Class="GridLayout.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:GridLayout"
        mc:Ignorable="d"
        Title="A Grid control" Height="350" Width="500">
    <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="50"/>
      <RowDefinition Height="*"/>
      <RowDefinition Height="auto"/>
      <RowDefinition Height="40"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="150"/>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Button Grid.Row="0" Grid.Column="0">A</Button>
    <Button Grid.Row="0" Grid.Column="1">B</Button>
    <Button Grid.Row="0" Grid.Column="2">C</Button>
    <Button Grid.Row="1" Grid.Column="0" Grid.RowSpan="2"
      Grid.ColumnSpan="2">D</Button>
```

```
    <Button Grid.Row="1" Grid.Column="2">E</Button>
    <Button Grid.Row="2" Grid.Column="2">F</Button>
    <Button Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="3">G</Button>
  </Grid>
</Window>
```

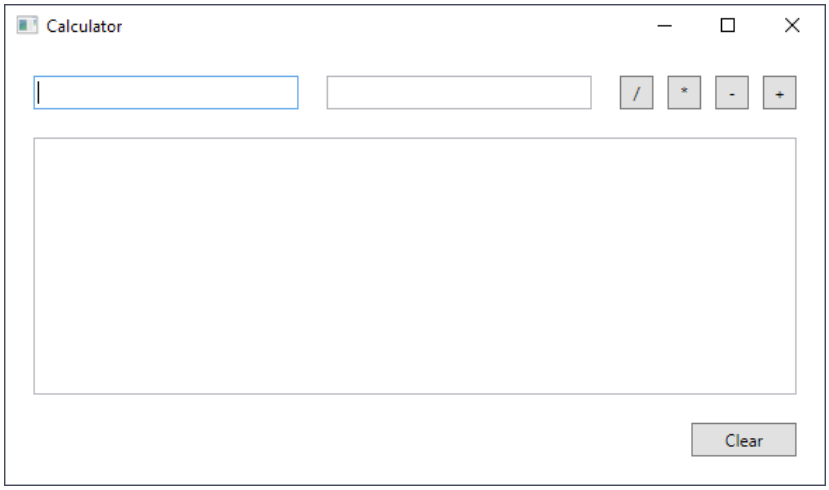
A *Grid* control divides the window into a number of rows and a number of columns. Default are one row and one column. As shown, you can define which rows the control should have, and in this case there are four. For each row you define the height, and here you can specify a size, a * or *auto*. The last means that the height is determined by the highest of the components in the row. A *, which is default means that the remaining height is evenly distributed between the remaining components. In the same way you can define columns, and in this case the *Grid* has three columns. The result is that the *Grid* divides the window in 12 cells. In a cell you can add components and indicates there size and position as in the program *HelloWpf*.

When you place a component in the window you indicate with the row index and the column index (default is 0) in which cell to add the component. You can also indicate a row span and a column span, which tells how many cells a component should use. If you look at the component *D* it has both a row span and a column span that is two, and the component will then occupy four cells.

Also note how the buttons are defined where the text is defined as a nested value and not as a *Content* attribute. It has no advantage in this case, but in other cases it is important that the content of a component can be nested.

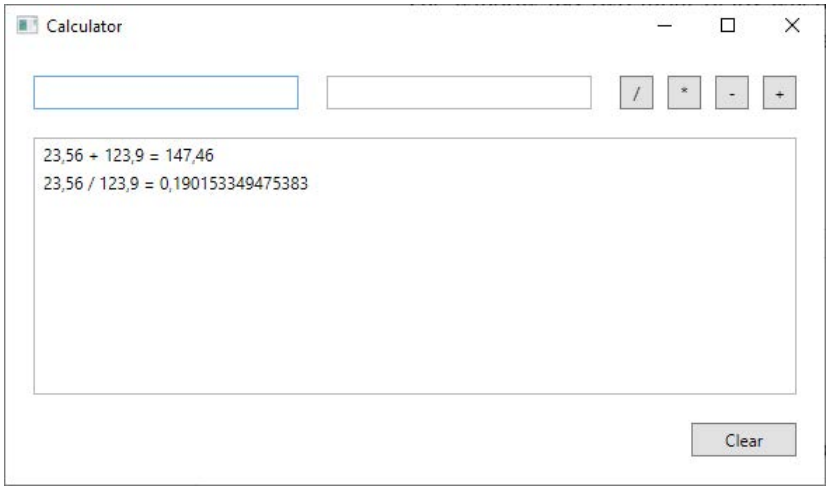
EXERCISE 1: A CALCULATOR

Create a new Visual Studio project, that you can call *SmallCalc*. You should write a program that opens a window, as shown below:

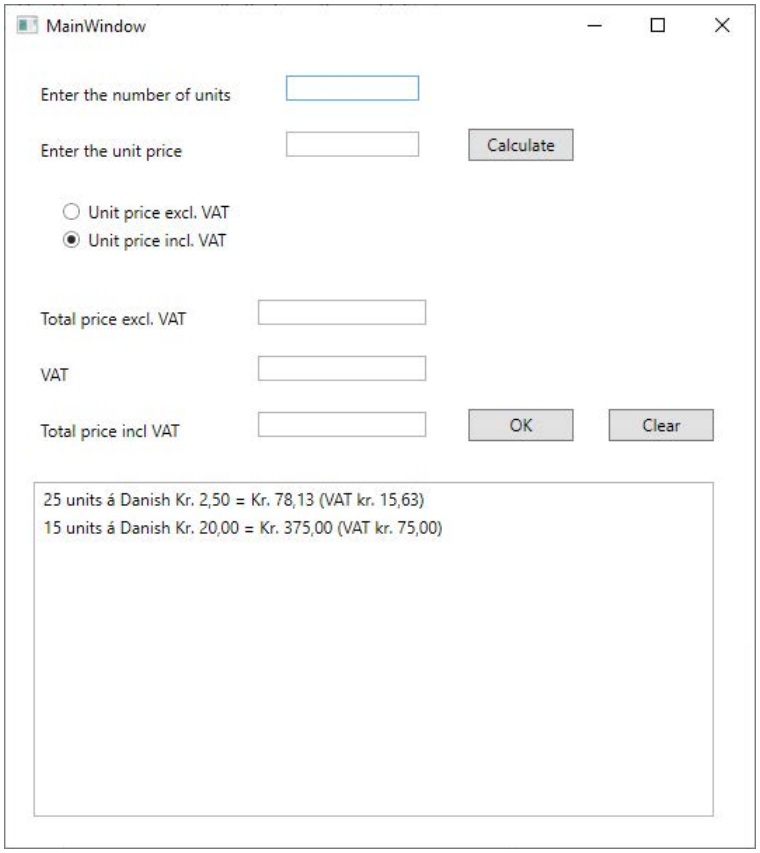


The window has two input fields where you must enter numbers. Furthermore, there are four buttons, one for each of the four arithmetic operations. When you click a button, the program should insert a line in the list box, which shows the results of that calculation. If there is an error, the program should show a message box. The *Clear* button should work in the same way as in the previous program.

When the window size is changed, all buttons must follow the window's right edge, while the two input boxes should use the remaining space equally. Here follow an example on a running program:



EXERCISE 2: THE ITEMSPROGRAM



You must write a program, which opens the above window. Here you can enter a number of units and a unit price. You can also click a radio button to indicate where the unit price is with or without VAT. When you click *Calculate*, the program calculates the total price without VAT, the VAT and the total price with VAT. If you then click *OK* the program inserts a line in the lower list box. In the above window are entered two items and the result inserted in the list box.

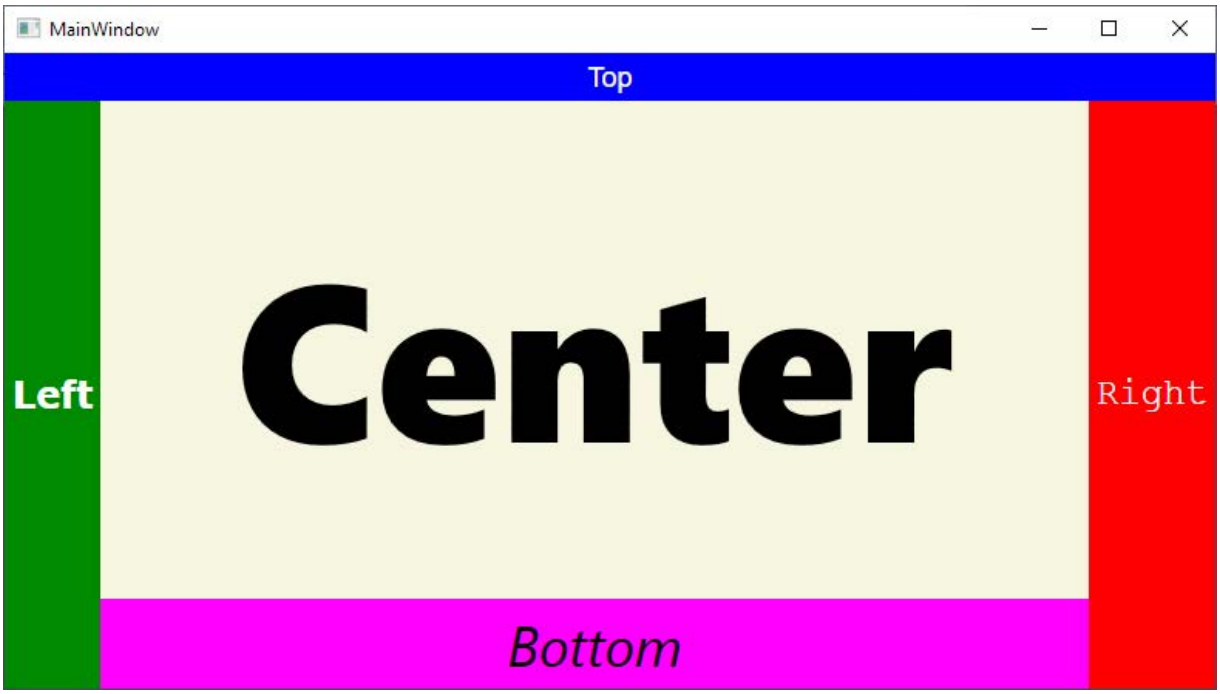
In this case the layout control should be a *Grid* with 1 cell (as Visual Studio creates the window), and you should place the components in the same way as in the example *HelloWpf*, that means as components with a fixed size and on fixed positions.

If the user enter illegal values, the program should show a message box with an error message.

When you have written the program and tested the program, you must write another version of the program, which performs exactly the same. The new version of the program must have the same user interface, but this time the *Grid* control must be divided into rows and columns. The result should be that *Clear* button follows the right edge of the window and the list box follows the right and the bottom of the window. The C# code should be exactly the same and can be copied from the previous version of the program.

2 FONTS AND COLORS

Most of the components that can be inserted into a window / panel displays text and you can define the font, which they should use. Similarly, you can define the color of the text, and finally you can define the background color. The program *FontProgram* opens the following window:



It is a very simple program that in fact do nothing, but shows 5 *Label* components arranged using a *DockPanel*:

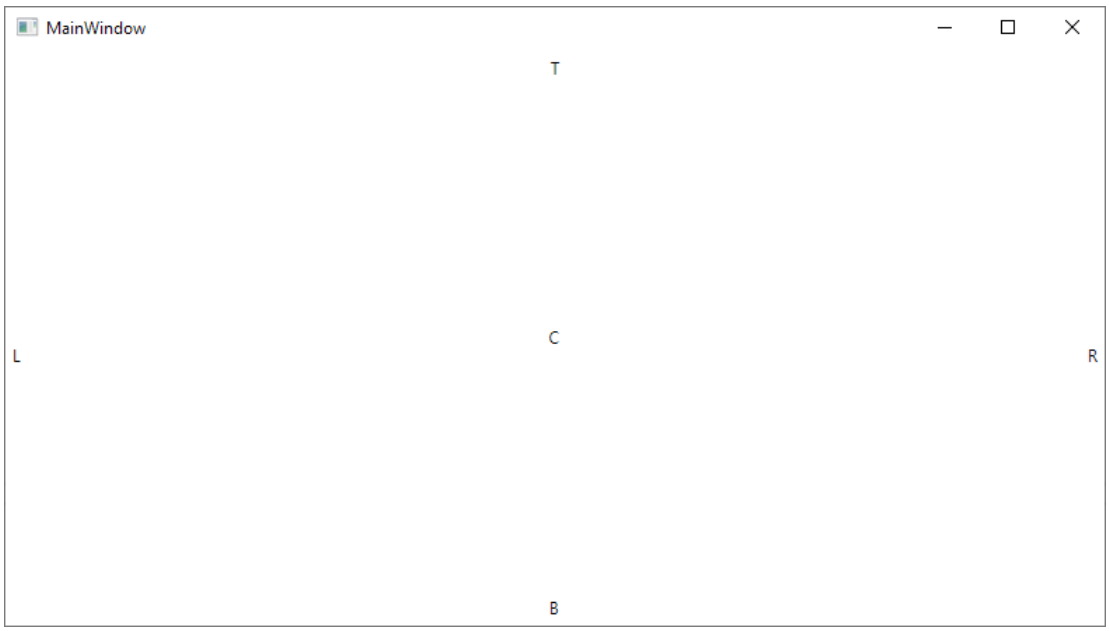
```
<DockPanel>
  <Label Content="Top" DockPanel.Dock="Top" Background="Blue"
  Foreground="White"
    FontFamily="Arial" FontSize="18" HorizontalContentAlignment="Center" />
  <Label Content="Left" DockPanel.Dock="Left" Background="Green"
  Foreground="White"
    FontFamily="Verdana" FontSize="24" FontWeight="Bold"
    VerticalContentAlignment="Center" />
  <Label Content="Right" DockPanel.Dock="Right" Background="Red"
  Foreground="White"
    FontFamily="Courier New" FontSize="24" VerticalContentAlignment="Ce
    nter" />
  <Label Content="Bottom" DockPanel.Dock="Bottom" Background="Magenta"
    FontSize="36" FontStyle="Italic" HorizontalContentAlignment="Cent
    er" />
  <Label Content="Center" Background="Beige" FontSize="144"
  FontWeight="ExtraBold"
    VerticalContentAlignment="Center" HorizontalContentAlignment="Cent
    er" />
</DockPanel>
```

When you see the code it is quite simple and there is not much to note besides observing the syntax and noting that this is how you specify colors and fonts. For example, if you specify a color, Visual Studio displays an overview of the default colors. Similarly, Visual Studio displays possible fonts values except for *FontFamily*, where you need to enter a text. If you enter the name of a font family that the system does not know or specify no, a default font is used.

A WPF application is a C# program, and it means as mentioned that Visual Studio converts all XML to C#. It also means that all what you do in XML, you can also do in C#. The program *ColorProgram* opens exactly the same window as above which shows a window with 5 *Label* components, but this time the XML is:

```
<DockPanel>
  <Label Name="lblTop" Content="T" DockPanel.Dock="Top"
    HorizontalContentAlignment="Center" />
  <Label Name="lblLeft" Content="L" DockPanel.Dock="Left"
    VerticalContentAlignment="Center" />
  <Label Name="lblRight" Content="R" DockPanel.Dock="Right"
    VerticalContentAlignment="Center" />
  <Label Name="lblBottom" Content="B" DockPanel.Dock="Bottom"
    HorizontalContentAlignment="Center" />
  <Label Name="lblCenter" Content="C" VerticalContentAlignment="Center"
    HorizontalContentAlignment="Center" />
</DockPanel>
```

If you run the program without doing anything else the result will be:



as all attributes concerning colors and fonts are removed. However, these values can be defined in the code:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    InitializeWindow();
  }

  private void InitializeWindow()
  {
    lblTop.Content = "Top";
    lblTop.Background = Brushes.Blue;
    lblTop.Foreground = Brushes.White;
    lblTop.FontFamily = new FontFamily("Arial");
    lblTop.FontSize = 18;

    lblLeft.Content = "Left";
    lblLeft.Background = Brushes.Green;
    lblLeft.Foreground = Brushes.White;
    lblLeft.FontFamily = new FontFamily("Verdana");
    lblLeft.FontSize = 24;
    lblLeft.FontWeight = FontWeights.Bold;

    lblRight.Content = "Right";
    lblRight.Background = Brushes.Red;
    lblRight.Foreground = Brushes.White;
    lblRight.FontFamily = new FontFamily("Courier New");
    lblRight.FontSize = 24;

    lblBottom.Content = "Bottom";
    lblBottom.Background = Brushes.Magenta;
    lblBottom.FontStyle = FontStyles.Italic;
    lblBottom.FontSize = 36;

    lblCenter.Content = "Center";
    lblCenter.Background = new SolidColorBrush(Color.FromRgb(245, 245,
                           220));
    lblCenter.FontWeight = FontWeights.ExtraBold;
    lblCenter.FontSize = 144;
  }
}
```

The method *InitializeWindow()* is called from the constructor, and the method set some properties on the components. *lblTop* is the name of a component, a *Label* component, and it is a variable which reference a *Label* object. Such an object has many properties with *get* and *set* methods, and you can assign values to this properties as you like. Of course you must know what the properties are called as well the type of the values. If you as an example look at the property *Content* it is simple, as it is a *string*. If you look at *Background* the type is *Brush* which is an abstract class that defines how to paint an area on the screen. .NET creates a family of standard brushes, and these brushes are defined by an *enum Brushes*. Enums are explained in the next book, but you can think of an enum as a family of constants. If you for example works in XML and you set a *Background* attribute Visual Studio shows the possibilities, which are the values of the enum *Brushes*. Of course, all this is not something, you need to think about in your daily life, but it does tell you that what you do in XML and in C# is really the same.

All the other properties and components are initialized in the same way, and when you looks through the code it is easy to recognize the initialization of attributes in XML. There is only one exception, which is the property *Background* for the object *lblCenter*. The value must be a *Brush*, but instead of choosing a standard brush I have created a *Brush* object myself:

```
new SolidColorBrush(Color.FromRgb(245, 245, 220))
```

The type is *SolidColorBrush* that is a class, and the constructor needs a *Color* object as parameter. The class *Color* has a static method *FromRgb()* which returns such an object, where the parameters are the intensity of the colors red, green and blue (see below). The result is the *Beige* color, and the purpose is alone to show the syntax and you in this way can create an arbitrary color.

In this case, the above solution has no advantage over initializing components in XML, and the purpose is also alone, to show that it is possible to do the same in the code, but in other contexts it may be important to know how. For example, you might be interested in changing the color of a component dynamically in an event handler, and then it must be done in the code as shown in the above example.

Colors are defined using three intensities of red, green and blue, and wee talk about this color encoding as *RGB colors*. Each intensity has a value between 0 and 255, and the number of possible colors is then

$$256^3 = 16777216$$

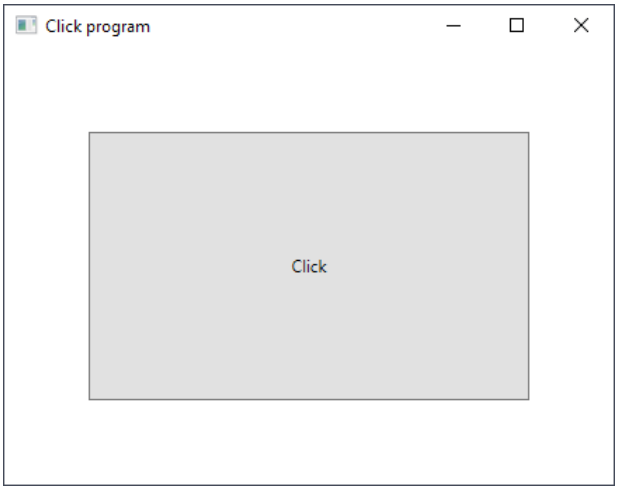
The following table shows some examples of color encodings

R	G	B	Color
0	0	0	Black
255	255	255	White
255	0	0	Red
0	255	0	Green
0	0	255	Blue
255	255	0	Yellow
128	128	128	Gray

Here you need specifically note the last encoding, where the same value for all intensities provides a gray-scale. In C# you can define a color as an object to the type *Color* with color intensities as parameters to a static method.

EXERCISE 3: CHANGE COLOR

Write a program as you can call *ClickProgram* that opens the following window with a single button:

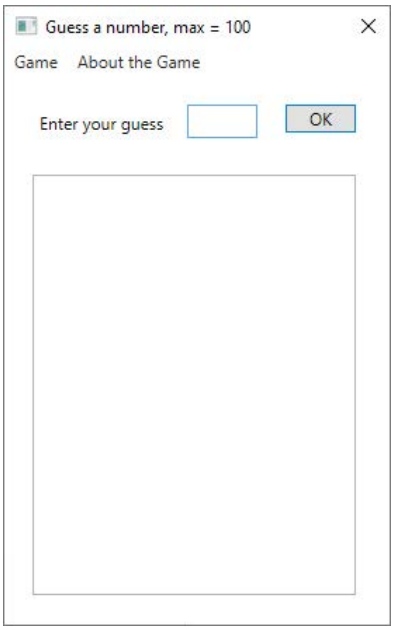


When you click the button, the program must change the button's background and foreground to an arbitrary color, and change the size of the font to an arbitrary value between 12 and 128.

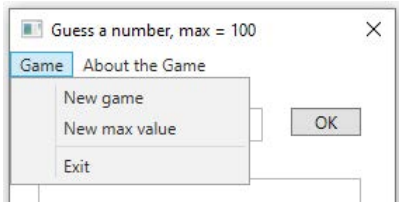
3 DIALOG BOXES

In this chapter I will show examples of a GUI program that uses a dialog box, that is a program which opens another window.

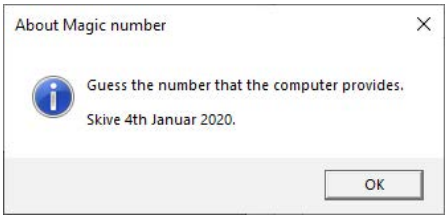
I want to start with a program where the computer determines a number that the user has to guess. It is decided that the computer must determine a random number between 1 and an upper limit (for example 100) and then the user must guess the number by entering a number. Each time the user enters a number, the computer responds whether the number is guessed or whether the user's guess is too large or too small. When the number is guessed, the computer tells you how many attempts the user has had. An example could be:



The program has a menu line. The first menu has three menu items:



where the first starts a new game, the second opens a dialog box where the user can enter a new max value, and the last menu item terminates the program. The last menu has only one menu item, which opens a message box with information about the program:



The XML for *MainWindow* is:

```
<Window x:Class="GuesNumber.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:GuesNumber"
        mc:Ignorable="d"
        Title="Gues a number" Height="450" Width="290"
        ResizeMode="NoResize"
          Activated="Window_Activated">
  <Grid>
    <Menu HorizontalAlignment="Left" VerticalAlignment="Top"
    Background="White"
      Margin="0,0,0,0">
      <MenuItem Header="Game">
        <MenuItem Header="New game" Click="NewGame_Click" />
        <MenuItem Header="New max value" Click="MaxValue_Click" />
        <Separator />
        <MenuItem Header="Exit" Click="Exit_Click" />
      </MenuItem>
      <MenuItem Header="About the Game" Click="About_Click" />
    </Menu>
    <Label Content="Enter your guess" HorizontalAlignment="Left"
      VerticalAlignment="Top" Margin="20,40,0,0" />
    <TextBox Name="txtNumber" HorizontalAlignment="Left"
    VerticalAlignment="Top"
      Margin="130,40,0,0" Width="50" Height="24"
      VerticalContentAlignment="Center" />
    <Button Name="cmdOk" Content="OK" HorizontalAlignment="Left"
      VerticalAlignment="Top" Margin="200,40,0,0" Width="50"
      IsDefault="True"
      Click="cmdOk_Click" />
    <ListBox Name="lstResult" HorizontalAlignment="Left"
    VerticalAlignment="Top"
      Margin="20,90,0,0" Width="230" Height="300"/>
  </Grid>
</Window>
```

First you should note the *Title* element which defines that it not should be possible to change the size of the window. The element also sets an attribute *Activated* to *Window_Activated*. It is the name of an event handler which is performed immediately after the window is shown on the screen, and then a method in the code where you can do some initialization that has to take place after the window appears on the screen.

You should also note how to define a menu, which is quite simple. A menu defines *MenuItem* elements, and each item defines a text to show in the menu and an event handler assigned for this menu item. Else the layout defines five components and there is not much new, but you should note the button, which has an attribute *IsDefault* with the value *True*. This means that if you enter a number and hit the *Enter* key, it is the same as click the button.

Then there is the code and this time there is five event handlers:

```
namespace GuesNumber
{
  public partial class MainWindow : Window
  {
    private static Random rand = new Random();
    private int max = 100;
    private int number = 0;
    private int count = 0;

    public MainWindow()
    {
      InitializeComponent();
    }

    private void SetTitle()
    {
      Title = string.Format("Guess a number, max = {0}", max);
    }

    public void NewMaxValue(object sender, MaxEventArgs e)
    {
      max = e.Max;
      SetTitle();
      NewGame();
    }

    private void MaxValue_Click(object sender, RoutedEventArgs e)
    {
      MaxWindow dlg = new MaxWindow();
      dlg.MaxChanged += NewMaxValue;
      dlg.ShowDialog();
    }
```

```
private void NewGame_Click(object sender, RoutedEventArgs e)
{
  NewGame();
}

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

private void cmdOk_Click(object sender, RoutedEventArgs e)
{
  try
  {
    int user = int.Parse(txtNumber.Text.Trim());
    if (user > 0 && user <= max)
    {
      if (user < number)
      {
        lstResult.Items.Add(
          string.Format("The Number {0} is too small", user));
        ++count;
      }
      else if (user > number)
      {
        lstResult.Items.Add(
          string.Format("The Number {0} is too great", user));
        ++count;
      }
      else
      {
        lstResult.Items.Add(
          string.Format("The number is {0} after {1} attempts",
          user, count));
        cmdOk.IsEnabled = false;
      }
      Clear();
      return;
    }
  }
  catch
```

```
      {
      }
      MessageBox.Show(string.Format(
        "Illegal number. You must enter a number between 1 and {0}", max),
        "Warning", MessageBoxButton.OK, MessageBoxImage.Warning);
      Clear();
    }

    private void Clear()
    {
      txtNumber.Clear();
      txtNumber.Focus();
    }

    private void NewGame()
    {
      count = 0;
      number = rand.Next(1, max + 1);
      lstResult.Items.Clear();
      cmdOk.IsEnabled = true;
      Clear();
    }

    private void Window_Activated(object sender, EventArgs e)
    {
      NewGame();
      SetTitle();
    }

    private void About_Click(object sender, RoutedEventArgs e)
    {
      MessageBox.Show(
        "Guess the number that the computer provides.\n\nSkive 4th
        Januar 2020.",
        "About Magic number", MessageBoxButton.OK, MessageBoxImage.
        Information);
    }
  }
}
```

First note that class has a static variable and three instance variables. The static variable is a random generator and then an object of the type *Random*. The other three variables are used for maximum value for number that the computer select, the users guess and the number of guess.

Then there is the event handler *Window_Activated()* defined in the *Title* element in the XML. The handler calls a method *NewGame()* and a method *SetTitle()*. The first method sets *count* to 0, so the number of guesses is set to 0, and then the method selects a new random value for *number*. This is the value that the user must guess. Next the list box is cleared, and the button is enabled so you a ready to play again. As the last *NewGame()* calls a method *Clear()* which clears the input field and set focus to the field. The event handler also calls *SetTitle()* which sets the title of the window.

You should then note the event handler *cmdOk_Click()* which is the event handler for the *OK* button. The handler is filling some but is actually quite simple. It determines the value of the user's entered number. Is it an illegal number either because, the user has entered a value that is not an integer or a value less than 1 or greater than the maximum value, the handler shows a message box with an error message. Otherwise, the number is compared to the computer's choice of number, and a line is inserted in the list box that tells whether the user guessed the number or whether the user's guess was too small or too large.

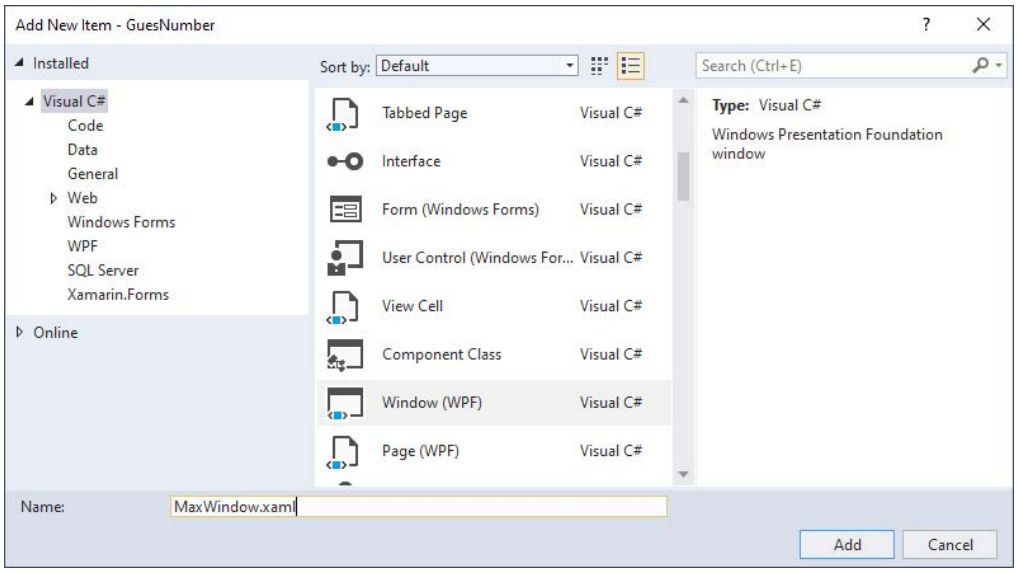
Also note the event handlers

- *NewGame_Click()*
- *Exit_Click()*
- *About_Click()*

which are all trivial but are assigned as event handlers for three of the menu items. In particular, you should note the middle one that closes the window and thus ends the program.

The last event handler is different, as it must open another window, a *dialog box*.

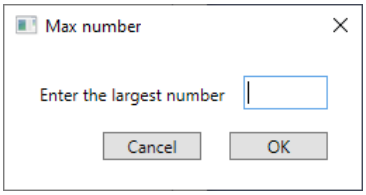
A dialog box is a window and as so, it has its own class. In Visual Studio, you should in the menu select *Project* and here *Add Window (WPF)* and you get the following window:



Here you select *Window (WPF)* and enter the name of the window as I have called *MaxWindow*. When you click *Add* Visual Studio add a new window to your project, and it is exactly the same as the main window and consists of a XML part and a C# part. You can then create the window in exactly the way as you have done in the other examples in this book, and the XML is the following:

```
<Window x:Class="GuesNumber.MaxWindow"
        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:GuesNumber"
        mc:Ignorable="d"
        Title="Max number" Height="140" Width="270" ResizeMode="NoResize"
          Activated="Window_Activated">
  <Grid Margin="20">
    <Label Content="Enter the largest number" />
    <TextBox Name="txtMax" HorizontalAlignment="Left"
    VerticalAlignment="Top"
      Margin="150,0,0,0" Width="60" Height="24"
      VerticalContentAlignment="Center" />
    <Button Name="cmdOk" Content="OK" HorizontalAlignment="Left"
      VerticalAlignment="Top" Margin="140,40,0,0" Width="70"
      Click="cmdOk_Click"/>
    <Button Name="cmdCancel" Content="Cancel" HorizontalAlignment="Left"
      VerticalAlignment="Top" Margin="50,40,0,0" Width="70"
      Click="cmdCancel_Click"/>
  </Grid>
</Window>
```

The XML is simple, but you should note the window as *MainWindow* is defined not resizable and defines a *Window_Activated* event handler. Else the window consists only of a *Label*, a *TextBox* and two *Button* components. If the window is opened the result is:



where the user must enter a value. The code is:

```
namespace GuesNumber
{
  public class MaxEventArgs : EventArgs
  {
    private int max;

    public MaxEventArgs(int max)
    {
      this.max = max;
    }

    public int Max
    {
      get { return max; }
    }
  }

  public delegate void MaxHandler(object sender, MaxEventArgs e);

  public partial class MaxWindow : Window
  {
    public event MaxHandler MaxChanged;

    public MaxWindow()
    {
      InitializeComponent();
    }

    private void cmdOk_Click(object sender, RoutedEventArgs e)
   {
      try
      {
        int max = int.Parse(txtMax.Text.Trim());
        if (max > 0)
        {
          if (MaxChanged != null) MaxChanged(this, new MaxEventArgs(max));
          Close();
          return;
        }
      }
      catch
      {
```

```
      }
      MessageBox.Show("You must enter a number greater then 1", "Error",
        MessageBoxButton.OK, MessageBoxImage.Error);
      txtMax.Clear();
      txtMax.Focus();
    }

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

    private void Window_Activated(object sender, EventArgs e)
    {
     txtMax.Focus();
    }
  }
}
```

The code defines two classes and in fact the code uses a very central concept which I have not yet explained, but shortly the following happens. After the user has entered a number and the user clicks *OK*, a message must be sent to *MainWindow*, which tells what the new max value is. The only way this can happen is that the event handler for the *OK* button calls a method on the *MainWindow* object. This problem can be solved in several ways, but here I will use a delegate, which is also the standard way to solve the problem.

The problem is that this window (the dialog box) do not now the *MainWindow*, but this window can raise an event when an action occurs, and the action is when the user click the *OK* button. When a method raises an event it sends an object to others (called observers or listeners) who want to be informed about the event. Such an event object is an object defined by a class that inherits the class *EventArgs*. In this case the class is called *MaxEventArgs* and it has only one property which represents the new max value. The class *MaxWindow* needs a data structure used to register other objects (listeners) who wants to be informed, when the event occurs. Such a data structure is called a delegate and the code defines a delegate used for *MaxEventArgs* events:

```
public delegate void MaxHandler(object sender, MaxEventArgs e);
```

MaxHandler is a type (a class) and the program can then creates an object *MaxChanged* of that type:

```
public event MaxHandler MaxChanged;
```

You should note that the variable is defined *public* as it should be referenced from other objects. All this seems complicated, but in this book you just have to take note of the way a dialog box uses to send a message to other windows. Later I will explain delegates and events in detail, but once you have seen it in use a few times, it is quite straightforward.

The window *MaxWindow* uses the delegate in the event handler for the *OK* button. It first determines the value of the number, the user has entered and if it is not a legal value the event handler shows a message box with an error message. If the value is legal the event handler must raise a *MaxEventArgs* event and close the window. To raise the event it first test if there are registered listeners (if there are some objects who has registered as a listener for this event) and if so raise the event by performing a method *MaxChanged()*:

```
if (MaxChanged != null) MaxChanged(this, new MaxEventArgs(max));
```

This was the window *MaxWindow* and apart from all that about events and delegates, the code for the window is written in the same way as other windows. The next is to use the window where the event handler for the menu item in *MainWindow* must open the new dialog box. In *MainWindow* is defined a method:

```
public void NewMaxValue(object sender, MaxEventArgs e)
{
  max = e.Max;
  SetTitle();
  NewGame();
}
```

You should note that the method has the same signature as the delegate in the class *MaxWindow*. It is the method, which the event handler for the *OK* button in the class *MaxWindow* calls and the method that receive the message from *MaxWindow*. The method do three things: Set the new *max* value, set the text in the title line and starts a new game. Last, there is the event handler for the menu item:

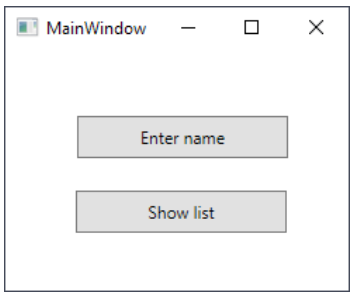
```
private void MaxValue_Click(object sender, RoutedEventArgs e)
{
  MaxWindow dlg = new MaxWindow();
  dlg.MaxChanged += NewMaxValue;
  dlg.ShowDialog();
}
```

First, the handler creates an object representing a *MaxWindow*. Next, this window (*MainWindow*) register itself as listener for *MaxEventArgs* events. Accept the syntax, but it means that this class must have a method with the same signature as defined by the delegate. As the last, the event handler opens the dialog box.

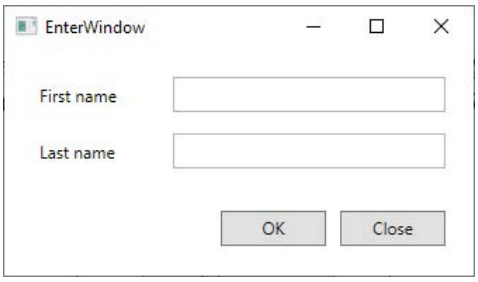
3.1 ENTER NAMES

In this example I will show a program, which has two dialog boxes. The one is used for entering names, while the other is used to show the entered names in a list box. The dialog box used in the previous example is opened as a modal dialog box. It means that when the dialog box is open you can not work in another window. Another window and its components can not get focus. A dialog box can also be opened as a modeless dialog box, which is a dialog box you can open and then change focus to another dialog box or window. This example shows a program with 3 windows:

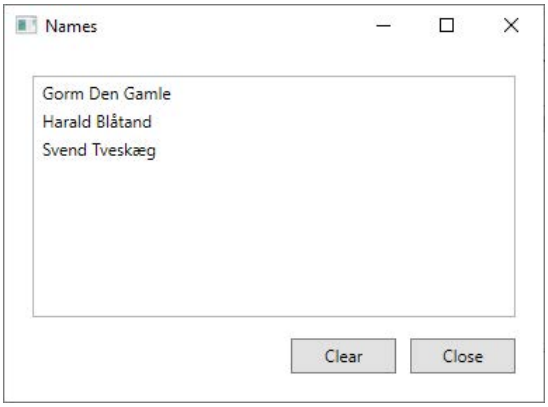
where *EnterWindow* is a modal dialog box used to enter names, while *ListWindow* is a modeless dialog box used to show the entered names. If you run the program it opens the following simple window:



If you click on the top button, the program opens a modal dialog box to enter a name:



When you click on *OK* the fields are cleared and you can enter the next name. When you close the dialog and click on the lower button in *MainWindow* you opens the window:



which shows the entered names. This dialog box is a modeless dialog box, which means that you can open *EnterWindow* again without closing *ListWindow*.

It is simple to write a program with these three windows as they all are very simple windows, but the communication between the windows is not quite simple. When you enter a name it must be stored somewhere, and if the list box *ListWindow* is open it must be updated. Another problem to solve is that it must not be possible to open the modeless dialog box if it is already open. It results all in some code.

I will not show the XML code for the layout since there is nothing new.

To represent a name I have added the following class to the project:

```
public class Name
{
  private string firstname;
  private string lastname;

  public Name(string firstname, string lastname)
  {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  public string Firstname
  {
    get { return firstname; }
  }

  public string Lastname
  {
    get { return lastname; }
  }

  public override string ToString()
  {
    return firstname + " " + lastname;
  }
}
```

It is a simple model class with two properties. The code for the *MainWindow* is and consists primarily of event handlers for the two buttons:

```
namespace ADialog
{
  public partial class MainWindow : Window
  {
    private List<Name> names = new List<Name>();
    private ListWindow listDlg = null;

    public MainWindow()
    {
      InitializeComponent();
    }

    private void cmdEnter_Click(object sender, RoutedEventArgs e)
    {
      EnterWindow dlg = new EnterWindow();
      dlg.NameEntered += NewName;
      if (listDlg != null) dlg.NameEntered += listDlg.NewName;
      dlg.ShowDialog();
    }

    private void cmdShow_Click(object sender, RoutedEventArgs e)
    {
      if (listDlg == null)
      {
        listDlg = new ListWindow(names);
        listDlg.ListCleared += ListCleared;
        listDlg.ListClosed += ListClosed;
        listDlg.Show();
      }
    }

    public void NewName(object sender, NameEventArgs e)
    {
      names.Add(e.Name);
    }

    public void ListCleared(object sender, EventArgs e)
    {
      names.Clear();
      listDlg.NewName(this, null);
    }

    public void ListClosed(object sender, EventArgs e)
    {
      listDlg = null;
    }
  }
}
```

The class has a *List<Name>* to store the entered names. It is obvious to use a list, but it is not quite as obvious where the list should be. It cannot be in *EnterWindow* as this window is closed when names are entered and the list would then disappear (be deleted). Likewise, it cannot be in *ListWindow* as this window may not be open (the window object does not exists). The list must therefore be in *MainWindow* as this window exists as long as the program is running. The class also has a variable *listDlg* which type is *ListWindow* and the variable is used to reference a *ListWindow* if a dialog box is open.

The event handler for the *Enter name* button must open the dialog box to enter a name and for this, it creates a new *EnterWindow* object. This object (window) fires an event of the type *NameEventArgs* when the user has entered a name, and *MainWindow* is listener for this event and register the event handler *NewName()*, which adds a *Name* object to the list *names.* If *listDlg* is different from *null* it means that the dialog box *ListWindow* is open and if so this dialog box must also be informed, if the user enter a new name, and this dialog box must also be registered as listener for *NameEventArgs*. Note that the class *ListWindow* must have a method *NewGame()* with the correct signature.

The event handler for the other button in *MainWindow* starts to test if the variable *listDlg* is *null*. If not the dialog box is already open and nothing should happens. Else the handler creates a *ListWindow* object and assign the object reference to *listDlg*. The dialog box *ListWindow* can also fires events and events that *MainWindow* must be aware of. When the user clicks on the *Clear* button *MainWindow* must be notified as it has to clear the list *names*. Also, when the user close the dialog box *ListWindow*, *MainWindow* must be notified to set the variable *listDlg* to *null*. Else the dialog box could not be opened again. For this *MainWindow* has subscribed for two events from *ListWindow*, and *MainWindow* must create event handlers called from *ListWindow*. Note that the handler for the *Clear* button must notify the *ListWindow* that the names are removed. You should note the dialog box is shown (opened) using the method *Show()* instead of *ShowDialog()*. This means that the dialog box is opened as a modeless dialog box, and this is the only difference to open a dialog box as a modeless or modal dialog box.

As the next, I will show the code for dialog box *EnterWindow*:

```
namespace ADialog
{
  public class NameEventArgs : EventArgs
  {
    private Name name;

    public NameEventArgs(Name name)
    {
      this.name = name;
    }

    public Name Name
    {
      get { return name; }
    }
  }

  public delegate void NameListener(object sender, NameEventArgs e);

  public partial class EnterWindow : Window
  {
    public event NameListener NameEntered;

    public EnterWindow()
    {
      InitializeComponent();
    }

    private void cmdOk_Click(object sender, RoutedEventArgs e)
    {
      string firstname = txtFname.Text.Trim();
      string lastname = txtLname.Text.Trim();
      if (firstname.Length > 0 && lastname.Length > 0)
      {
        if (NameEntered != null)
          NameEntered(this, new NameEventArgs(new Name(firstname,
          lastname)));
        txtFname.Clear();
        txtLname.Clear();
        txtFname.Focus();
      }
```

```
      else MessageBox.Show("You must enter both first name and last name",
    }

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

The class has two event handlers (there are two buttons), and the event handler for the *Close* button is trivial. The event handler for the *OK* button starts to determine the entered text, and the user has entered text for both first name and last name the handler creates a *Name* object and fires this object as an event to listeners (and this time there are two). To fire the event is defined a class *NameEventArgs* which is an enclosure of a *Name* object and is used as an event argument. The event type is defined as a delegate called *NameListener* and the class defines an event of this type. You should note that the class fires an event in exactly the same way and with the same syntax as you have seen in the above example *GuesNumber*.

Then there is the last dialog box *ListWindow* and the code is:

```
namespace ADialog
{
  public delegate void CloseListener(object sender, EventArgs e);
  public delegate void ClearListener(object sender, EventArgs e);

  public partial class ListWindow : Window
  {
    public event ClearListener ListCleared;
    public event CloseListener ListClosed;

    private List<Name> names;

    public ListWindow(List<Name> names)
    {
      InitializeComponent();
      this.names = names;
      lstNames.ItemsSource = names;
    }
```

```
    public void NewName(object sender, NameEventArgs e)
    {
      lstNames.ItemsSource = null;
      lstNames.ItemsSource = names;
    }

    private void cmdClear_Click(object sender, RoutedEventArgs e)
    {
      if (ListCleared != null) ListCleared(this, new EventArgs());
    }

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

    private void Window_Closed(object sender, EventArgs e)
    {
      if (ListClosed != null) ListClosed(this, new EventArgs());
    }
  }
}
```

The dialog box can fire two events and for that are defined two delegates. When these events does not send data to the listeners the event arguments has the type *EventArgs,* which do not an enclosure data and as so it is not necessary to define your own types for event arguments. You should note that the constructor has the *names* as a parameter and the dialog box then knows the name list. Note also that the list box in the constructor is filled by assigning the list to the property *ItemsSource*.

If the dialog box is open and a new name is entered the list box most be updated to show the new name. It happens in the method *NewName()* which is event handler for this event. You should note how to do that. It works, but it is not the way to do it, but the right way will have to wait until later until I in another book will deal with WPF in details.

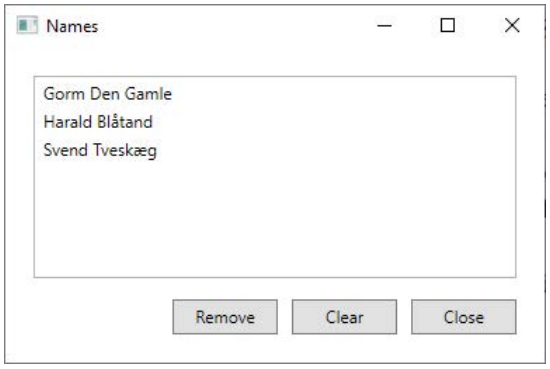
The two event handlers for the buttons are simple, but note the handler for the *Clear* button fires an event. The last event handler *Window_Closed()* is fired by the system, when the window is closed, and it is this handler which notify *MainWindow* that this dialog box is closed. To fire this event when you click on the cross in the title bar you must define the handler in XML:

```
<Window x:Class="ADialog.ListWindow"
        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:ADialog"
        mc:Ignorable="d"
        Title="Names" Height="450" Width="400" Closed="Window_Closed">
  <Grid Margin="20">
```

EXERCISE 4: ADIALOG ENHANCED

In this exercise you have to make some improvements to the program above. Start by creating a copy of the project *ADialog* and open the copy in Visual Studio.

The window *ListWindow* must have an extra button:



If you select a name in the list box and clicks on the button *Remove* the program should show a message box with a waring that the name will be removed, and if the user accept the name should be removed. If you clicks on the button, but no name is selected you should get a message box with an error message.

Next, it must be such that if you double click on a name in the list box, then the dialog box *EnterWindows* opens initialized with the name that is clicked on, and you should then be able to edit the name. It is immediately a little more complicated, but you can go as follows.

Modify the class *Name*, so it has *set* properties for both variables.

Add another constructor to class *EnterWindow*:

```
public EnterWindow(Name name)
```

and store the value in an instance variable. The constructor must initialize the two *TextBox* fields with the values of the parameter. In the event handler for the *OK* button you can then distinguish whether there is an update or enter a new name.

In the class *ListWindow* you must add an event handler for double click with the mouse. You do that in XML:

```
<ListBox x:Name="lstNames" MouseDoubleClick="lstNames_MouseDoubleClick" />
```

and the event handler can be written as:

```
private void lstNames_MouseDoubleClick(object sender,
MouseButtonEventArgs e)
{
  ListBox source = (ListBox)e.Source;
  int n = source.SelectedIndex;
  if (n >= 0)
  {
    EnterWindow dlg = new EnterWindow(names[n]);
    dlg.NameEntered += NewName;
    dlg.ShowDialog();
  }
}
```

4 COMPONENTS AND LAYOUT

A program with a graphical user interface has a user interface build from windows which contains controls or components and all this is made available for the programmer through an API that in C# is called WPF. As a programmer you must know what components exist and how to place these components in a window and the previous examples in this book tell a lot about what is needed, but there is more and that is what this chapter is about. I will not show all the details but mainly through examples and exercises point to the most important.

WPF has the following components:

Border	DockPanel	Label	RichTextBox	TextBox
Button	DocumentViewer	ListBox	ScrollBar	ToolBar
Calendar	Ellipse	ListView	ScrollViewer	ToolBarPanel
Canvas	Expander	MediaElement	Separator	ToolBarTray
CheckBox	Frame	Menu	Slider	TreeView
ComboBox	Grid	PasswordBox	StackPanel	ViewBox
ContentControl	GridSplitter	ProgressBar	StatusBar	WebBrowser
DataGrid	GroupBox	RadioButton	TabControl	WindowsFormsHost
DatePicker	Image	Rectangle	TextBlock	WrapPanel

Many of the components are simple, while others are more complex and others even extremely complex. WPF basically has three kinds of components (or controls, and I will generally use both names):

- *Individual controls*, for example *Label*, *Button* and *TextBox* that are controls with a single well defined using. It is controls that the user can do something with.
- *Item controls*, for example *ListBox*, *Menu* and *TreeView* that are controls which contains a number of other elements and typical let the user select one or more of these elements. The items in an item control can in principle be whatever and then another object, but often it will be other controls.
- *Layout controls*, for example *Grid*, *StackPanel* and *DockPanel* which are controls that contains other controls and determines how these controls should be shown on the screen. A layout control may specifically contain another layout control and is then used to define the layout of a window as a hierarchy of components inside each other.

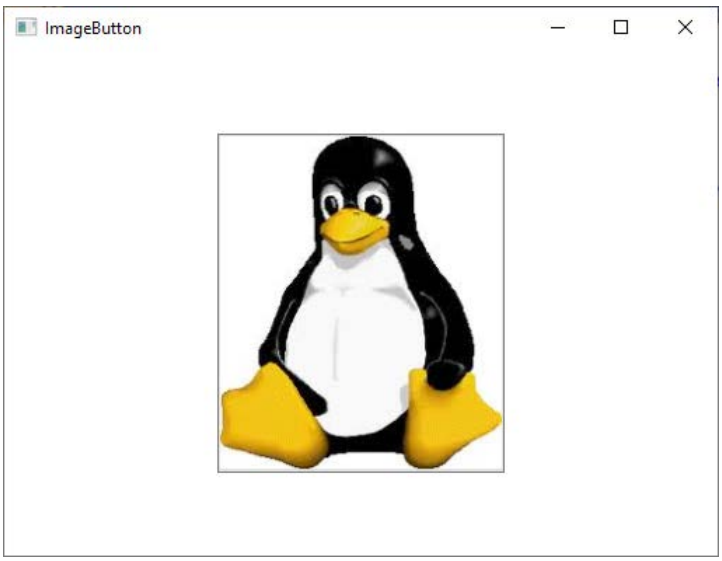
Many controls are *content controls* which are controls that inherits from the class *ContentControl* and can contain a single nested element which can be just about anything. As an example you can define a button as:

```
<Button Name="cmdButton" Width="150" Height="23" >It is a big button</
Button>
```

as a *Button* is a content control and it contains an element that is a *string*. You can defines the same button as

```
<Button Name="cmdButton" Width="150" Height="23" Content="It is a big
button" />
```

and the result is the same and in this case defining the content as a nested element does not offer any advantages, but in other cases it is necessary. The program *ImageButton* opens the following window:



The window has one component, which is a *Button*, but the *Button* does not show a text but an image. If you click on the button (the image) the program shows a message box and you can see that it is a "real" button.

The program needs an image and it is an image called *penguin.jpg*. The image is copied to the project library, but to use the image it is easiest to add the image as a resource. You can do this by right-clicking on the image in Solution Explorer and selecting *Include in project*. You can then define the user interface as:

```
<Grid>
  <Button Name="cmdImage" Height="242" Width="205" Click="cmdImage_
  Click" >
    <Image Source="penguin.jpg" />
  </Button>
</Grid>
```

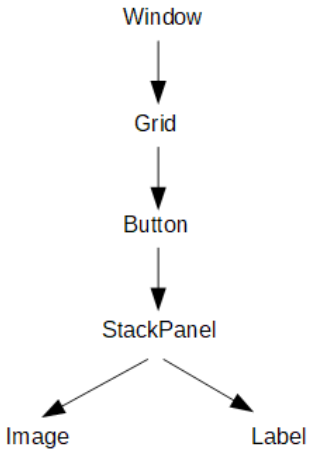
Here you should note how the content of the button is an *Image* element.

A content control can only contain one element, but since this element can be anything it can be a layout control in particular, and the user interface could be written as

```
<Grid>
  <Button Name="cmdImage" Height="265" Width="205" Click="cmdImage_Click" >
    <StackPanel>
      <Image Source="penguin.jpg" />
      <Label HorizontalAlignment="Center">It is a penguin</Label>
    </StackPanel>
  </Button>
</Grid>
```

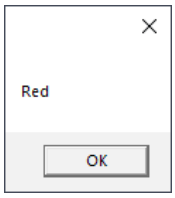
You should note that it still is a button that you can click and it does not matter if you click on the image or the label.

The content of a window is always a hierarchy of components called the visual tree, and in this case, the visual tree is a *Window* with a *Grid* containing a *Button* whose content is a *StackPanel* with an *Image* and a *Label*:

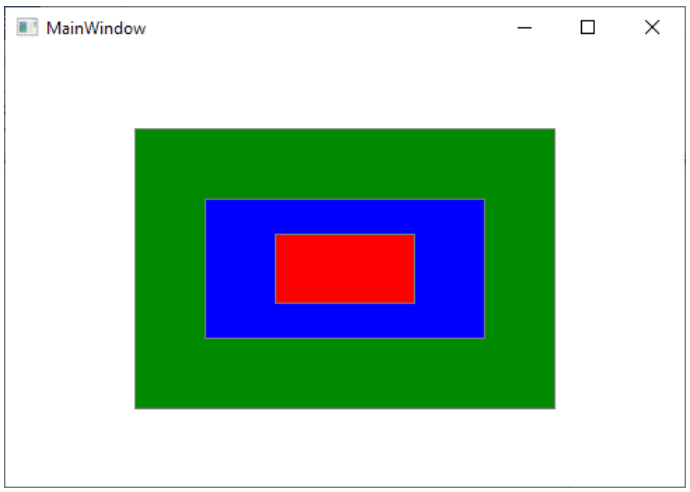


EXERCISE 5: THREE BUTTONS

Write a program you can call *ThreeButtons* which opens the window below. The window has three buttons inside each other. If you click on one of the buttons, the program must show a message box, for example:

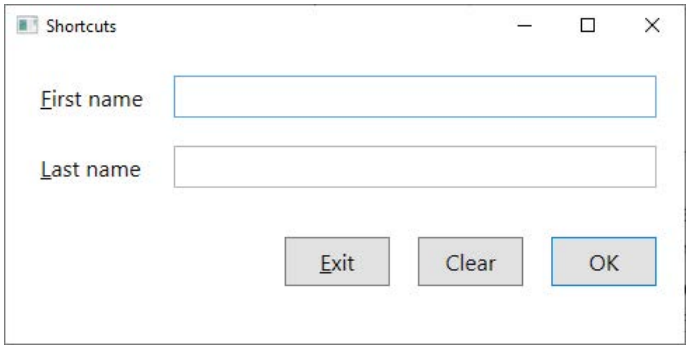


May be you have a problem. When you click on the blue button you get the message box both for the blue and the green button, and if you click on the red button you get the message box for all three buttons. Can you solve this problem?



4.1 SHORTCUTS

It is possible to assign shortcut keys to a control such you can activate the control using the keyboard. As an example has the program *Shortcuts* some shortcut keys:



If you look at the *Label* components the first character is underlined and the same goes for the button *Exit*. It means that the components can be activated with a combination of the ALT key and the underlined character as for example ALT+E for the button. Now you cannot activate a *Label*, but the labels are bounded to the two *TextBox* fields and if you for example enter ALT+F the first *TextBox* get focus.

If you click on the Button *Exit* or enter ALT+E the program show a message box and you is asked to close the program. Also the two other buttons has assigned a shortcut but defined in another way. A *Button* has an *IsCancel* property, and if this property is *true* the button is activated if the user enter ESC. In the same way a *Button* has a property *IsDefault*, and if this property is *true* the button is activated if the user enter *Enter*. In this case the cancel action is used to clear the two *TextBox* fields while the default action is used to show the entered name in a message box.

The XML is as follows:

```
<Grid Margin="20">
  <Grid.RowDefinitions>
    <RowDefinition Height="50" />
    <RowDefinition Height="50" />
    <RowDefinition Height="50" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Label Content="_First name" Target="{Binding ElementName=txtFirst}"
    FontSize="16" />
  <TextBox x:Name="txtFirst" Grid.Row="0" Grid.Column="1"
  Margin="0,0,0,20"
    VerticalContentAlignment="Center" />
  <Label Grid.Row="1" Grid.Column="0" Content="_Last name"
    Target="{Binding ElementName=txtLast}" FontSize="16" />
  <TextBox x:Name="txtLast" Grid.Row="1" Grid.Column="1" Margin="0,0,0,20"
    VerticalContentAlignment="Center" />
  <StackPanel Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2"
    Orientation="Horizontal" HorizontalAlignment="Right">
    <Button x:Name="cmdExit" Content="_Exit" Margin="0,15,0,0" Width="75"
      FontSize="16" Click="cmdExit_Click"/>
    <Button x:Name="cmdClear" Content="Clear" Margin="20,15,0,0"
    Width="75"
      FontSize="16" IsCancel="True" Click="cmdClear_Click"/>
    <Button x:Name="cmdOk" Content="OK" Margin="20,15,0,0" Width="75"
      FontSize="16" IsDefault="True" Click="cmdOk_Click"/>
  </StackPanel>
</Grid>
```

The code is relatively self-explanatory, but you should how to specify a character for a shortcut, which you do with an underscore in front of the character. Note that also characters in the middle of a text can be shortcuts. Also note how to bind a *Label* to a *TextBox*, but you should just accept the syntax as binding is a comprehensive topic, which I will discuss in detail in a later book. You should also note how to set the two properties *IsCancel* and *IsDefault* for last two buttons.

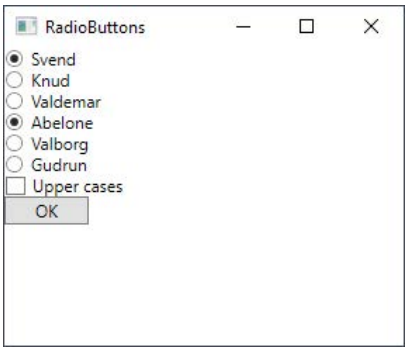
I will not show the C# code here as it only consists of simple event handlers, one for each of the three buttons.

4.2 CHECKBOX AND RADIOBUTTON

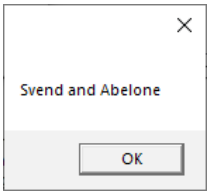
A *CheckBox* is really just a button that is drawn differently from a *Button*. It is used to indicate whether a setting is on or off. If you change the state of a *CheckBox* by clicking the mouse, it will raise an event just like a *Button*.

A *RadioButton* is also a button, but it is a control that is usually used with other *RadioButtons* (a group) and where only one of them can be pressed and then selected. By default, the radio buttons in a single container (window) form a group, where only one button can be pressed, but it is possible to divide radio buttons in the same container into several groups. Typically, a *RadioButton* is used to specify a choice between several possible choices.

The program *RadioButtons* opens a window with 6 *RadioButton* components, 1 *CheckBox* component and a *Button*, where the *RadioButton*s are grouped in two groups:



If you click on the *OK* button the result is a message box which shows the state of the components:



The XML for the user interface is:

```
<StackPanel>
  <RadioButton Name="cmd1" GroupName="male" IsChecked="True">Svend</
  RadioButton>
  <RadioButton Name="cmd2" GroupName="male">Knud</RadioButton>
  <RadioButton Name="cmd3" GroupName="male">Valdemar</RadioButton>
  <RadioButton Name="cmd4" GroupName="female" IsChecked="True">
    Abelone</RadioButton>
  <RadioButton Name="cmd5" GroupName="female">Valborg</RadioButton>
  <RadioButton Name="cmd6" GroupName="female">Gudrun</RadioButton>
  <CheckBox Name="cmdCase">Upper cases</CheckBox>
  <Button Name="cmdOk" Width="60" HorizontalAlignment="Left" Click="cmdOk_
  Click">
    OK</Button>
</StackPanel>
```

The code is simple but you should note how to group the *RadioButton*s. The code for the event handler is as shown below, and here you should especially notice how to test where a *RadioButton* or *CheckBox* is selected:

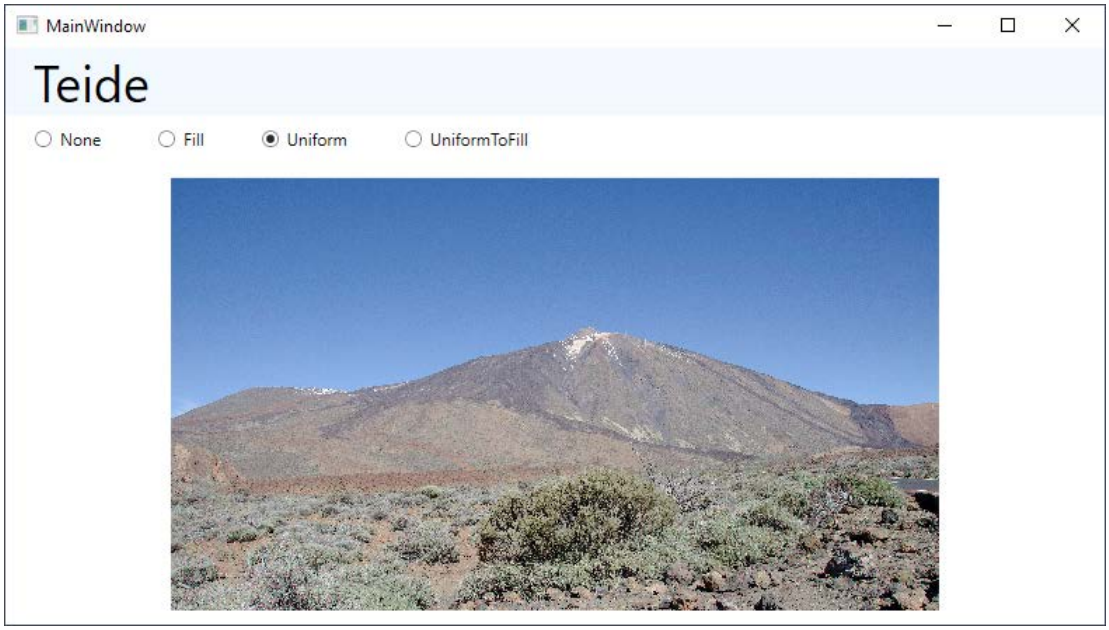
```
private void cmdOk_Click(object sender, RoutedEventArgs e)
{
  string male = cmd1.Content.ToString();
  if (cmd2.IsChecked == true) male = cmd2.Content.ToString();
  else if (cmd3.IsChecked == true) male = cmd3.Content.ToString();
  string female = cmd4.Content.ToString();
  if (cmd5.IsChecked == true) female = cmd5.Content.ToString();
  else if (cmd6.IsChecked == true) female = cmd6.Content.ToString();
  MessageBox.Show(cmdCase.IsChecked == true ?
      male.ToUpper() + "  and " + female.ToUpper() :
      male + " and " + female);
}
```

As layout control I have this time used a *StackPanel*, a control I have also used in previous examples. A *StackPanel* is a panel that arrange components in a row horizontal or vertical. As default, a *StackPanel* arrange the components vertical, but you can specify by a property *Orientation* that the panel should arrange to components horizontally. The components are arranged in the order in which they are added to the panel. Regardless of the number of components, they are laid out in one row (horizontal or vertical), and if there are more components than there is space, only those components that may be inside the window are displayed. Here the components are laid out in a row and the size is determined by the current size of the individual components. Note that by default the components are centered in the center of the panel, but this can be determined with an alignment property.

A *StackPanel* rarely has an interest as a panel used alone in a dialog box (a window), but it is often used to place components in another control, for example if a *Label* must have a *Content* that consists of other components.

4.3 IMAGES

The example *Images* is a program, which show an image in a window:



The syntax is simple when WPF has a component to show an image:

```
<Image Source="Teide.jpg"></Image>
```

and you must primarily specify the image file name, which can also be an URL. However, there is one important property called *Stretch* that specifies how the image should fill the window:

- *None*: The image is shown in its normal (physical) size. The result is that only part of the image may be displayed.
- *Fill*: The image will fill the entire window and the resolution both horizontally and vertically changes so that the image accurately fills the window. The result is that the image may deformed.

- *Uniform*: The size of the image changes so that the entire image is displayed and it is compressed either vertically or horizontally. The result is that the entire window is not necessarily filled, but you see all the image. This setting is the default.
- *UniformToFill*: The size of the image changes so that the entire window is filled, but so the image is not deformed. The result is that the image may cropped so that the part (horizontal or vertical) that cannot be in the window is not displayed.

You are encouraged to try the program so you can see the effect of the four settings. The XML for the window is:

```
<DockPanel>
  <TextBlock Text="Teide" FontSize="36" Background="AliceBlue"
  DockPanel.Dock="Top"
    Padding="20,0,0,0" />
  <StackPanel Orientation="Horizontal" DockPanel.Dock="Top" >
    <RadioButton x:Name="cmdNone" Content="None" Margin="20, 10, 20, 10"
      Click="cmdNone_Click" />
    <RadioButton x:Name="cmdFill" Content="Fill" Margin="20, 10, 20, 10"
      Click="cmdFill_Click" />
    <RadioButton x:Name="cmdUnif" Content="Uniform" Margin="20, 10, 20, 10"
      IsChecked="True" Click="cmdUnif_Click" />
    <RadioButton x:Name="cmdUnto" Content="UniformToFill" Margin="20, 10,
    20, 10"
      Click="cmdUnto_Click" />
  </StackPanel>
  <Image x:Name="imgTeide" Source="Teide.jpg" Margin="10" />
</DockPanel>
```

This time is used a *DockPanel* as layout control which arrange two components top and the last component, that is the image will fill the rest of the window. Note that the first component is a *TextBlock* and not a *Label*. There is no particular justification for it other than showing an example of this component, but a *TextBlock* is a simpler component than a *Label* and then more effective, and if the purpose is merely to display a text, you should in principle, use a *TextBlock* rather than a *Label*. To each *RadioButton* is assigned an event handler and these handlers set the property *Stretch* to see the effect of the four options for showing an image.

EXERCISE 6: A SIMPLE TEXT EDITOR

You must write a program, which you can call *TextEditor*. The program should have a very simple user interface with a menu with five menu items (see below) and a *TextBox* component, which fills the rest of the window. When the program is running the user can enter and edit text lines, and it should be possible to store the entered text in a text file. It should also be possible to open an existing text file and edit the file. The program should thus work a bit like a simple version of the program *Notepad*.

As default a *TextBox* component is used to enter a single line of text, but the component can also be used to enter several lines, and you tell that with the attribute *AcceptsReturn*. You can enter an arbitrary text, and I have defined the component as:

```
<TextBox Name="txtEdit" ScrollViewer.VerticalScrollBarVisibility="Auto"
  ScrollViewer.HorizontalScrollBarVisibility="Auto" AcceptsReturn="True"
  TextChanged="txtEdit_TextChanged" Padding="5"/>
```

where it defines that the component should have scroll bars both horizontally and vertically. You should note that I have assigned an event handler and the purpose is to keep track of when the text has changed.

It is thus quite simple to write the program so you can enter and edit any text. The challenge is to save the text to a file, and open and load a text file, as you have to get here the familiar dialog boxes to browse the file system. They are not part of WPF, but are provided by the operating system. It is your job to find out (for example by searching the web) how to do it, but as an example, I have shown the event handler to open a file:

```
private void cmdOpen_Click(object sender, RoutedEventArgs e)
{
  OpenFileDialog dlg = new OpenFileDialog();
  dlg.Filter = "Tekst Filer |*.txt";
  if (dlg.ShowDialog() == true)
  {
    filename = dlg.FileName;
    txtEdit.Text = File.ReadAllText(filename);
    modified = false;
    Title = filename;
  }
}
```

You should note to use *OpenFileDialog()* you as to add a *using* statement:

```
using Microsoft.Win32;
```

Another problem you must solve is if you close the program or you select the menu item *New* and the text is changed you must have a warning message and get the possibility to save the text.

EXERCISE 7: A SLIDER

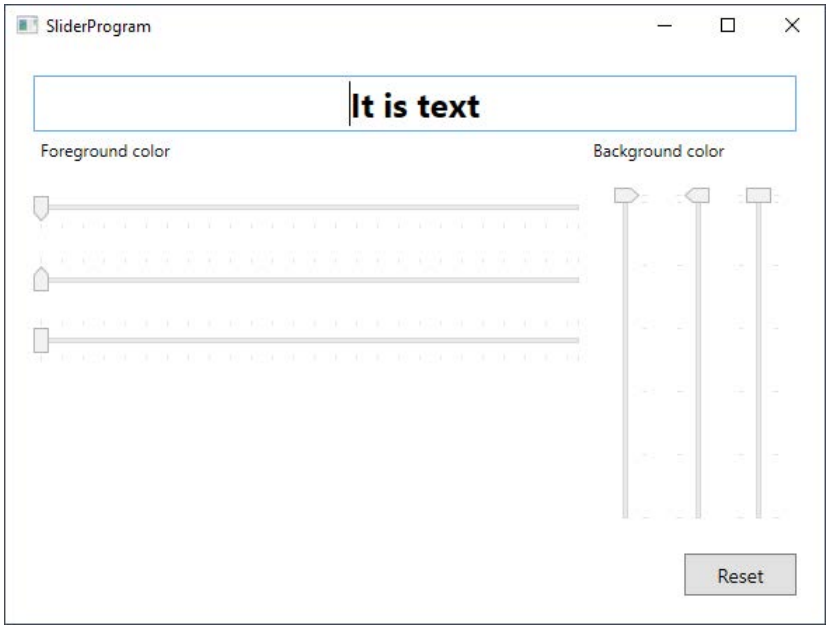
A Slider is basically the same as a scroll bar, it is simply drawn in a different way. A Slider represents a value within a range and the "slider" is drawn in a position corresponding to that value. There are a number of properties that can be set and affects how the component is displayed. Some important examples are mentioned

- *Minimum*, which indicates the minimum value of the interval
- *Maximum*, which indicates the maximum value of the interval
- *Value*, which is the value of the component.

- *TickFrequence*, which defines the distance between (the number of) selections
- *TickPlacement*, which specifies where markings should be displayed
- *Orientation*, which specifies whether a *Slider* should be oriented horizontally or vertically

You must write a program which you can call *SliderProgram*. The program must opens the following window, which has 10 components:

1 *TextBox* component
2 *Label* components
6 *Slider* components which all may have a value between 0 and 255
1 *Button* component

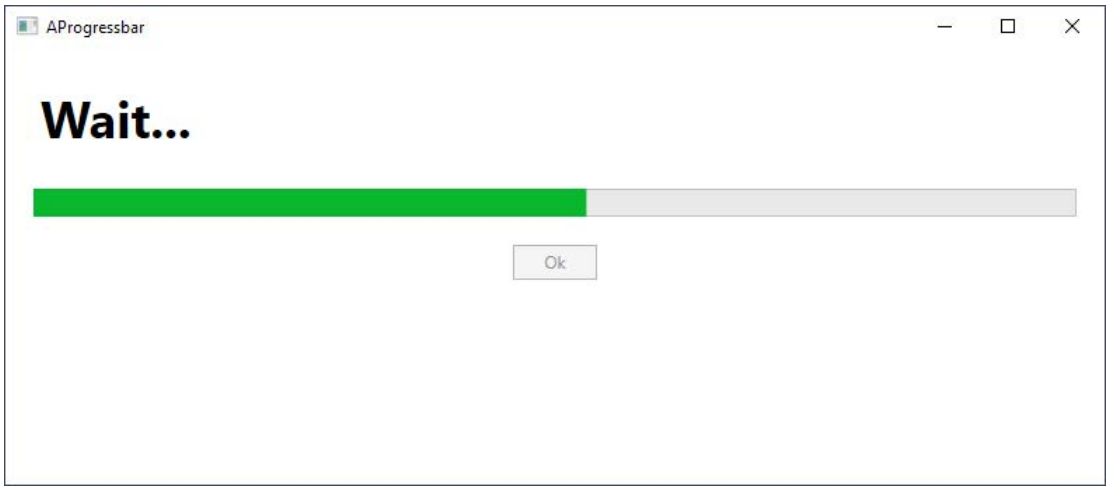


The sliders indicates a color intensity for red, green and blue and the three horizontal slides are used to adjust the foreground color, while the vertical sliders are used to adjust the background color. The *Reset* button should reset the colors and the sliders (white background and black foreground).

4.4 A PROGRESS BAR

In this section, I will show the use of a progress bar, which is a component used to show that the program is performing an operation that takes time and that one must wait. Use of the component actually requires concepts (threads) that have not yet been processed, but I have included it here as an example of a WPF component, and this is again an example where you must take note of the syntax.

A progress bar defines an interval and it has a value within that interval (which in the following example is the range from 0 to 100 defined as properties in XML) and the value is displayed as a colored bar relative to the interval (however, you can also set it to show the value itself). By changing the value of the progress bar in a loop, one can illustrate the progress of a work. If the program runs and you click on the button you start the progress bar that simulates that a work is going on:



The XML is simple and the only thing to note is how to define a *ProgressBar*:

```
<StackPanel Margin="20">
  <Label Name="lblText" Height="80" FontSize="36" FontWeight="Bold"
    Visibility="Hidden">Wait...</Label>
  <ProgressBar Name="progressBar" Height="20" Minimum="0" Maximum="100" />
  <Button Name="cmdOk" Margin="0,20,0,0" Height="25" Width="60"
    Click="cmdOk_Click" >Ok</Button>
</StackPanel>
```

The C# code is immediately more difficult to understand:

```
public partial class MainWindow : Window
{
  public delegate void UpdateProgressBar(DependencyProperty dp, Object
  value);
  private static Random rand = new Random();

  public MainWindow()
  {
    InitializeComponent();
  }

  private void cmdOk_Click(object sender, RoutedEventArgs e)
  {
    int time = 50;
    cmdOk.IsEnabled = false;
    lblText.Visibility = System.Windows.Visibility.Visible;
    double current = 0;
    double value = UpdateValue(time);
    UpdateProgressBar update = new UpdateProgressBar(progressBar.
    SetValue);
    while (true)
    {
      Dispatcher.Invoke(update, DispatcherPriority.Background,
        new object[] { ProgressBar.ValueProperty, current });
      WorkToDo(time);
      current += value;
      if (progressBar.Value >= progressBar.Maximum) break;
    }
    cmdOk.IsEnabled = true;
    progressBar.Value = 0;
    lblText.Visibility = System.Windows.Visibility.Hidden;
  }

  private double UpdateValue(int time)
  {
    int ticks = rand.Next(2000, 5000) / time;
    double min = progressBar.Minimum;
    double max = progressBar.Maximum;
    return (max - min) / ticks;
  }

  private void WorkToDo(int time)
  {
    System.Threading.Thread.Sleep(time);
  }
}
```

You can choose to accept the code as it is, and the details will be explained later, but shortly the following happens.

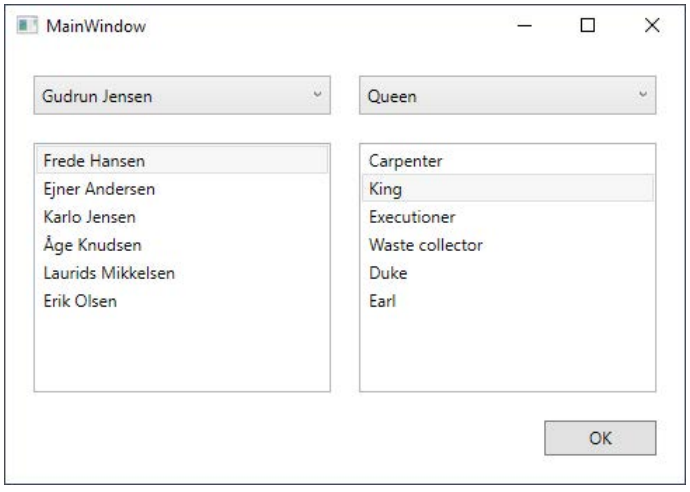
First, a delegate is defined, which can refer to a method for a progress bar, which must be called to update it. The type of the first parameter is a so-called dependency property and is a concept that is explained later. The last parameter is the value to be assigned to this dependency property.

Most of the code is in the event handler for the button. Clicking the button disables it so you cannot click it again. The event handler uses the above delegate to refer to the method in the progress bar that needs be updating, a method called *SetValue()*. Otherwise, a loop is started that iterates over the range of the progress bar, which is here from 0 to 100. For each iteration, the value of the variable *value* is counted by 1 and the method referenced by the delegate *update* is called with *value* as a parameter. It is called with a so-called *Dispatcher*, which is also explained later. This is necessary to get the progress bar updated properly. For each iteration, the program is suspended for a short period to simulate that some work is being done.

4.5 LIST AND COMBO BOXES

I have previously shown how to use a *ListBox* and a component that looks like a *ListBox* is a *ComboBox*. A *ComboBox* is similar to a *ListBox* in that it contains a number of elements of any type, but basically the elements have the type *ComboBoxItem*, which is a content control. Compared to a *ListBox*, a *ComboBox* displays only one item, which is what is selected, but the user can select another from a drop down list.

The program *ListProgram* opens the following window which has two *ComboBox* controls and two *ListBox* controls and a *Button*:



The example should primarily show that it is possible to initialize a *ListBox* and a *ComboBox* both in XML and in C#. If you run the program and clicks on the button the program opens a message box that shows what is selected in the four controls. The XML is:

```
<Grid Margin="20">
  <Grid.RowDefinitions>
    <RowDefinition Height="28"/>
    <RowDefinition/>
    <RowDefinition Height="45"/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <ComboBox x:Name="lstName1" VerticalContentAlignment="Center"
  Margin="0,0,10,0" >
    <ComboBoxItem>Valborg Kristensen</ComboBoxItem>
    <ComboBoxItem IsSelected="True">Gudrun Jensen</ComboBoxItem>
    <ComboBoxItem>Abelone Andersen</ComboBoxItem>
    <ComboBoxItem>Olga Hansen</ComboBoxItem>
    <ComboBoxItem>Helga Knudesen</ComboBoxItem>
  </ComboBox>
  <ComboBox x:Name="lstJob1" Grid.Row="0" Grid.Column="1"
    VerticalContentAlignment="Center" Margin="10,0,0,0" >
  </ComboBox>
  <ListBox Name="lstName2" Grid.Row="1" Grid.Column="0"
  Margin="0,20,10,0">
    <ListBoxItem IsSelected="True">Frede Hansen</ListBoxItem>
    <ListBoxItem>Ejner Andersen</ListBoxItem>
    <ListBoxItem>Karlo Jensen</ListBoxItem>
    <ListBoxItem>Åge Knudsen</ListBoxItem>
    <ListBoxItem>Laurids Mikkelsen</ListBoxItem>
    <ListBoxItem>Erik Olsen</ListBoxItem>
  </ListBox>
  <ListBox Name="lstJob2" Grid.Row="1" Grid.Column="1" Margin="10,20,0,0" />
  <Button Name="cmdOk" Grid.Row="2" Grid.Column="1" Content="OK"
  Width="80"
    Margin="0,20,0,0" HorizontalAlignment="Right" Click="cmdOk_Click"/>
</Grid>
```

and here you should mainly notice how to create a *ComboBox* and how to add items to the combo box. Also note how to add elements to a *ListBox* in XML. The C# code is:

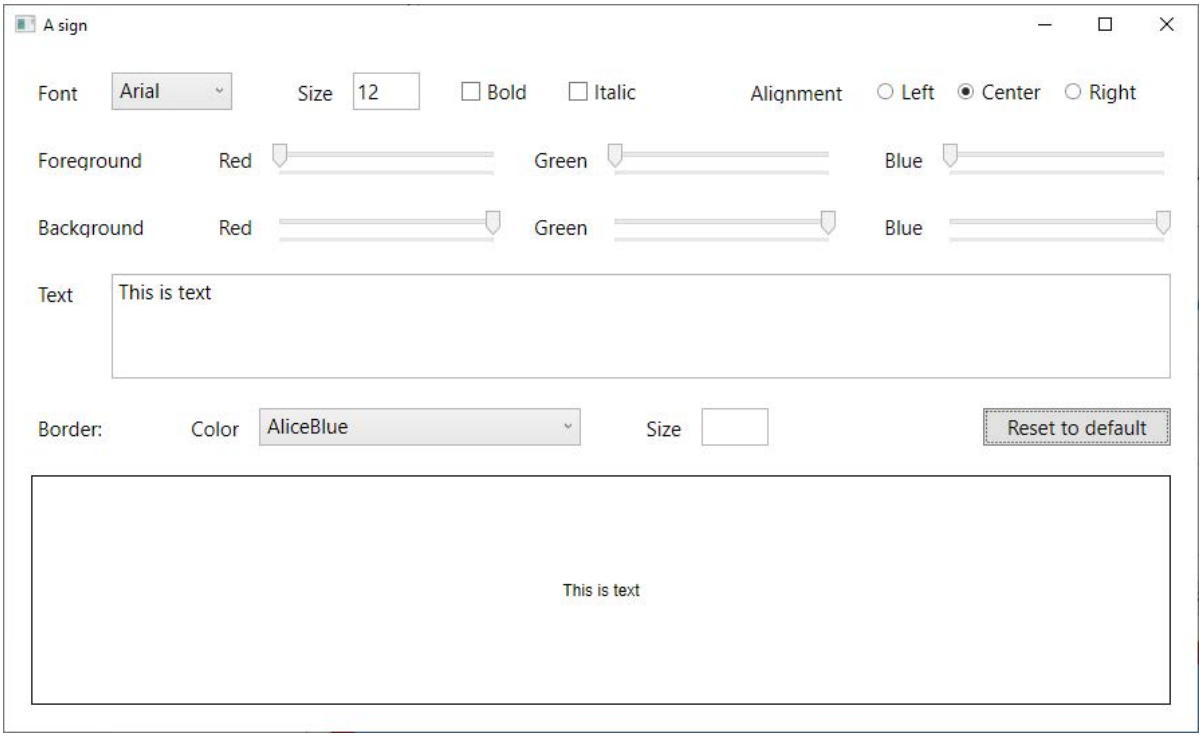
```
public partial class MainWindow : Window
{
  private string[] job1 =
    { "Witch", "Queen", "Seamstress", "Soothsayer", "Priestess",
    "Housewife" };
  private string[] job2 =
    { "Carpenter", "King", "Executioner", "Waste collector", "Duke",
    "Earl" };

  public MainWindow()
  {
    InitializeComponent();
    InitList();
  }

  private void InitList()
  {
    lstJob1.ItemsSource = job1;
    lstJob2.ItemsSource = job2;
    lstJob1.SelectedIndex = 1;
    lstJob2.SelectedIndex = 1;
  }

  private void cmdOk_Click(object sender, RoutedEventArgs e)
  {
    if (lstName2.SelectedIndex >= 0 && lstJob2.SelectedIndex >= 0)
      MessageBox.Show(((ComboBoxItem)lstName1.SelectedItem).Content + "
      " +
        lstJob1.SelectedItem + "\n" + ((ListBoxItem)lstName2.SelectedItem).
        Content
        + " " + lstJob2.SelectedItem);
  }
}
```

A *ListBox* can contain any type which is an object type (if it makes sense). In XML, objects of type *ListBoxItem* are inserted, which is a content control. The contents of the right *ListBox* inserted in the code are strings and it is necessary to take into account the difference in the message box. The same goes for a *ComboBox* where the element inserted in XML has the type *ComboBoxItem*.

PROBLEM: A SIGN

You must write a program that can be used to design a sign based on different settings. If you run the program it must open a window which is similar to the one below. The lower field is for the sign, and the other components are used for settings:

- In the upper row are settings for font and adjustment of the text.
- In the next row are three *Slider* components used to change the text color (foreground color).
- The third row contains equivalent *Slider* components to change the background color.
- The fourth row has a *TextBox* to enter the text for sign.
- In the last row you can select a color and a size for a border.



It is straightforward enough to write such a program, since you basically have to design the user interface in XML and then write the necessary event handlers, but in practice it is not necessarily so easy. Firstly, it is not very simple to design a stable user interface as there are many components, and secondly there will always be a number of issues that you do not know how to solve and where it is necessary to read the documentation or find others examples online. Finally, it can be a problem with event handlers and including which events the program should support and when these events occurs.

Below I will outline some guidelines that you should follow, but first I will clarify some of the requirements.

1. In the first line is a *ComboBox* which shows the available font as the font that are installed on your machine.
2. In the same way in the last settings line is a *ComboBox* with colors, and it must be the same colors as you can select in XML in Visual Studio.
3. The *Reset* button should reset the program settings to the same settings as shown above.
4. As a last thing, a decision is needed to when the sign should be updated after the user has made a setting. Here it is decided that it should happens after each setting are changed.

To write the program, I would suggest that you proceed as follows:

1. Create a new project in Visual Studio which you can call *ASign*.
2. Write the XML for the user interface, that is to add all the needed components (here is 31) and arrange the components so you have a nice and stable user interface even when the window size changes. It is an iterative process where you add components to layout controls, add and adjust attributes, runs and test the layout and repeat until everything looks and works the way you think it should. As layout control I have used a *Grid* control with nested *Grid* controls.
3. Assign good readable names for all components that you need to reference in the C# code. Maybe it's all components except Label components.
4. Initialize the two combo boxes. It must be done in C# code and can fist be done after the components are initialized and it should only happen one time. It's not that easy, and in fact you can't know how (unless you've seen it before), so I'll show you what to do. Start by add an event handler to the *Title* element in XML:

```
Title="A sign" Height="550" Width="1000" Loaded="Window_Loaded"
```

This event handler will be executed when the *Window* object is created in memory and then when all components are instantiated, and the event handler will only be executed this one time. To initialize the *ComboBox* with font families you can add the following statements to the event handler:

```
lstFont.ItemsSource = Fonts.SystemFontFamilies;
lstFont.SelectedIndex = 0;
```

lstFonts is the name of the combo box, and it is initialized with all system fonts. The last statement sets the first font to the selected font.

To initialize the colors is more complex as it is colors defined in C# as an *enum*. *enum* is a type explained in the next book, but you can think of an *enum* as a collection of constants. The problem is how to put these constants in a combo box and such the combo box shows readable names. Start to add the following variable (collection) as an instance variable in your class *MainWindow*:

```
private Dictionary<string, Color> colorMap = new Dictionary<string, Color>();
```

It is an example of what is called a *Dictionary* and is a collection like a *List*, and you can think of the type as a list of key / value pairs. Also dictionaries will be explained later, but here each element is identified by a *string* (the key) to which is assigned a *Color* object (the value).

In the event handler *Window_Loaded* the dictionary is initialized in a *foreach* loop:

```
foreach (var color in
  typeof(Colors).GetProperties(BindingFlags.Static|BindingFlags.Public))
  colorMap.Add(color.Name, (Color)color.GetValue(null, null));
lstColor.ItemsSource = colorMap.Keys;
lstColor.SelectedIndex = 0;
```

Accept the syntax, and you cannot know that it is the way to do it, but the technique is called *reflection* and will be dealt with later. The last two statements initialize the combo box (with the keys) and sets the first to the selected color.

5. Write a method used to set all components to there default value as shown in the above window. Call the method as the last statement in the event handler *Window_Loaded*.

Add an event handler for the *Reset* button which call your reset method. Test the program where you make some settings and test where the *Reset* button resets the settings to default.

6. Assign event handlers to the components (in all 16 components), but you show note that more components can be assigned the same event handler. You needs to take action of the following events:

1. *Click* events for check boxes and radio buttons
2. *SelectionChanged* events for the two combo boxes
3. *ValueChanged* events for the 6 slider components
4. *TextChanged* events for the three *TextBox* components for font size, border size and text for the sign

When you have created the event handlers you must write a method to update the sign with the current settings. Call this method from the above event handlers.

After that, the program is basically finished and an example of a run of the program could be:

7. However, the program has a problem. If you run the program and enter something illegal for font size or border size the program stops with an exception. You can solve the problem with some control statements in the method that updates the sign, but a better solution is to ensure that the user can only enter numbers in the two fields. Add the following method to the class *MainWindow*:

```
private bool KeyOk(Key key)
{
  switch (key)
  {
    case Key.D0:
    case Key.D1:
    case Key.D2:
    case Key.D3:
    case Key.D4:
    case Key.D5:
    case Key.D6:
    case Key.D7:
    case Key.D8:
    case Key.D9:
    case Key.Back:
    case Key.Left:
    case Key.Right: return true;
    default: return false;
  }
}
```

Key is an enum with constants representing the keys on the keyboard. The method returns *true* if the parameter is for a digit, backspace, left arrow or right arrow and else the method returns *false*.

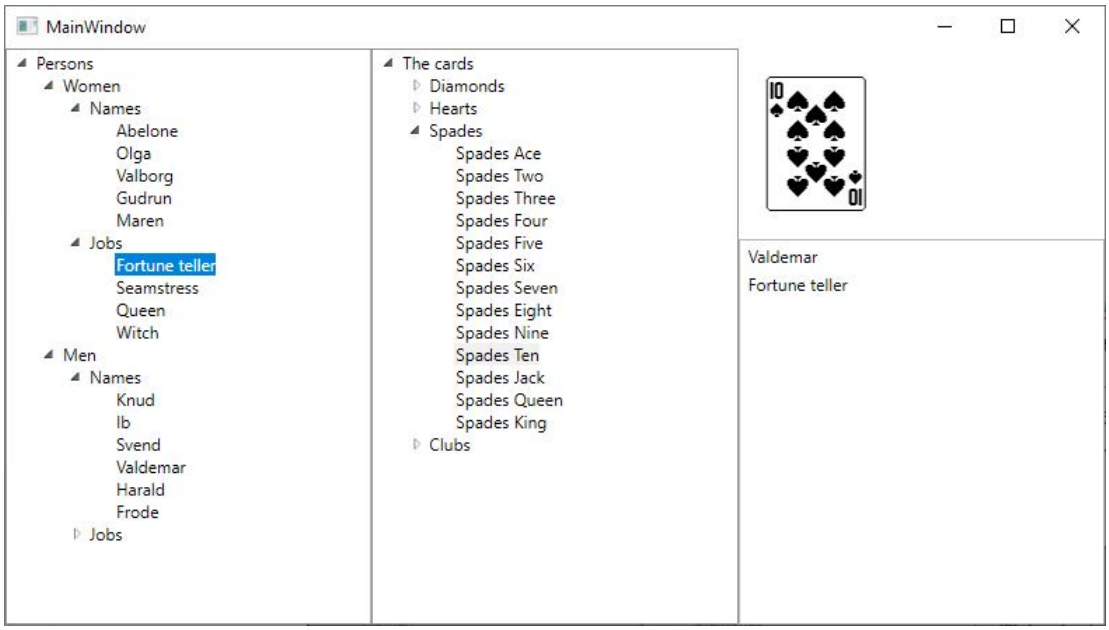
For the two *TextBox* components you should then assign an event handler for *KeyDown* events:

```
private void txt_KeyDown(object sender, KeyEventArgs e)
{
  if (!KeyOk(e.Key)) e.Handled = true;
}
```

If you now run the program the two fields only accepts digits.

4.6 A TREEVIEW

A *TreeView* is an item control that, like a *ListBox*, and contains a number of elements, but it organizes the elements into a hierarchy and displays them in a tree structure. A *TreeView* control is known from Windows Explorer and many other programs, which displays the file system as a hierarchy of directories and files. In the same way as a *ListBox*, a *TreeView* can contain arbitrary elements, but typically these are *TreeViewItem* elements, since it is an object that, in addition to the element, must also contain a collection of references to child elements. The following program opens a window with 2 *TreeView* controls, an *Image* control and a *ListBox*:



The first *TreeView* organizes some texts into a hierarchy and the entire tree structure is built into XML. If the user selects an item in the tree by clicking on it, it is inserted into the list box. The second *TreeView* organizes the playing cards in a hierarchy, and if you click on one of the cards, it appears in the *Image* component. Unlike the first *TreeView*, this tree is built into the code. You should note that the two *TreeView* controls have nothing to do with each other, and the sole purpose is to show how to define a *TreeView* in XML and code respectively. The XML is shown below, and there is nothing to explain other than noting the syntax for defining and initializing a *TreeView*:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <TreeView Grid.Column="0" Name="txtTree"
    SelectedItemChanged="txtTree_SelectedItemChanged">
    <TreeViewItem Header="Persons">
      <TreeViewItem Header="Women">
        <TreeViewItem Header="Names">
          <TreeViewItem Header="Abelone"/>
          <TreeViewItem Header="Olga"/>
          <TreeViewItem Header="Valborg"/>
          <TreeViewItem Header="Gudrun"/>
          <TreeViewItem Header="Maren"/>
        </TreeViewItem>
        <TreeViewItem Header="Jobs">
          <TreeViewItem Header="Fortune teller"/>
          <TreeViewItem Header="Seamstress"/>
          <TreeViewItem Header="Queen"/>
          <TreeViewItem Header="Witch"/>
        </TreeViewItem>
      </TreeViewItem>
      <TreeViewItem Header="Men">
        <TreeViewItem Header="Names">
          <TreeViewItem Header="Knud"/>
          <TreeViewItem Header="Ib"/>
          <TreeViewItem Header="Svend"/>
          <TreeViewItem Header="Valdemar"/>
          <TreeViewItem Header="Harald"/>
          <TreeViewItem Header="Frode"/>
        </TreeViewItem>
        <TreeViewItem Header="Jobs">
          <TreeViewItem Header="Tater"/>
          <TreeViewItem Header="Executioner"/>
          <TreeViewItem Header="King"/>
          <TreeViewItem Header="Waste collector"/>
          <TreeViewItem Header="Earl"/>
        </TreeViewItem>
      </TreeViewItem>
```

```
    </TreeViewItem>
  </TreeView>
  <TreeView Grid.Column="1" Name="imgTree"
    SelectedItemChanged="imgTree_SelectedItemChanged" />
  <Grid Grid.Column="2">
    <Grid.RowDefinitions>
      <RowDefinition Height="136"/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Image Grid.Row="0" Height="96" Name="imgCard" Stretch="Fill"
    Width="71"
      Margin="20,20,0,0" VerticalAlignment="Top"
      HorizontalAlignment="Left" />
    <ListBox Name="lstPersons" Grid.Row="1"/>
  </Grid>
</Grid>
```

Also note how to define an event handler for an event that occurs when the user selects an item in the tree.

The program needs the images of the playing cards and I want the images be packaged as resources to the program which means they are part of the program. To do that I have in Visual Studio in Solution Explorer added a folder to the project and called the folder *Cards*. Then I have added the card images to this folder as *Add* | *Existing Item*. When you build the program, the cards are added to the program's exe file. To use the cards in the program I have added a class which represents a card:

```
public class Card
{
  private readonly static string[,] cards = {
    { "bag", "The cards"},
    { "brd", "Diamonds"},
    ...
    { "klk", "Clubs King"} };
  private BitmapImage bp;
  private string name;

  public Card(int n)
  {
    name = cards[n, 1];
    bp =
      new BitmapImage(new Uri("/Cards/" + cards[n, 0] + ".GIF", UriKind.
      Relative));
  }

  public static int Count
  {
    get { return cards.GetLength(0); }
  }

  public BitmapImage Image
  {
    get { return bp; }
  }

  public override string ToString()
  {
    return name;
  }
}
```

Note that the class is a bit "hard-coded" in terms of file names, but for the purpose of creating *Card* objects, it does what it needs. The class has a constructor with an *int* as parameter. It is interpreted as index to the array containing the file names, and the constructor load the card's image as a *BitMapImage* object. The class has a property that returns this image. Also note the *ToString()* method which returns the text shown in the *TreeView* control.

Back there is the code of the window itself:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    BuildTree();
  }

  private void BuildTree()
  {
    TreeViewItem root = CreateNode(0);
    root.Items.Add(CreateColor(1, 14));
    root.Items.Add(CreateColor(15, 28));
    root.Items.Add(CreateColor(29, 42));
    root.Items.Add(CreateColor(43, 56));
    imgTree.Items.Add(root);
  }

  private TreeViewItem CreateColor(int a, int b)
  {
    TreeViewItem root = CreateNode(a);
    for (++a; a <= b; ++a) root.Items.Add(CreateNode(a));
    return root;
  }

  private TreeViewItem CreateNode(int n)
  {
    TreeViewItem node = new TreeViewItem();
    node.Header = new Card(n);
    return node;
  }

  private void imgTree_SelectedItemChanged(object sender,
    RoutedPropertyChangedEventArgs<object> e)
  {
    imgCard.Source = ((Card)((TreeViewItem)e.NewValue).Header).Image;
  }

  private void txtTree_SelectedItemChanged(object sender,
    RoutedPropertyChangedEventArgs<object> e)
  {
    TreeViewItem item = (TreeViewItem)e.NewValue;
    if (!item.HasItems) lstPersons.Items.Add(item.Header);
  }
}
```

Here you should especially notice how the tree is built in the method *BuildTree()*. First, a root is created, and then a sub-tree is added to the root for each color. Finally, the root is inserted into the *TreeView* control. You should also note the event handlers, which corresponds to selecting an element in one of the two *TreeView* controls. Each handler tests with

```
!item.HasItems
```

if the node has child nodes, and if not it is a leaf node and then a node which cannot be expanded.

4.7 TOOLBAR AND STATUSBAR

Many windows has a toolbar at the top of the window or a statusbar at the bottom of the window. A *ToolBar* is a control that is typically placed at the top of a window as an alternative to a menu. A *ToolBar* contains controls that the user is familiar with, and these controls can be anything but will often be buttons, combo boxes and so on. A *StatusBar* is another simple items control, which is typically placed at the bottom of a window and displays various information to the user. The example *ToolsProgram* opens the following window:

At the top is a toolbar (actually two) where the user in a combo box can select the text to show in the *Label* in the center of the window. The user can also in a combo box select the size of the font to show the text. If the user click on *Draw* which is a *Button* the *Label* is updated. The user can also in combo boxes select the color for background and foreground, but these combo boxes are in another toolbar that contains the two combo boxes and two *Label* components.

In the bottom is a status bar with two *Label* components. The first show a watch ticking every second, while the other *Label* shows the coordinates for the mouse if you move the mouse over the label in the center of the window. The sole purpose of the example is to show how to use a toolbar and a status bar, but some code is actually attached to the example. I starts with the XML:

```
<DockPanel>
  <ToolBarTray DockPanel.Dock="Top">
    <ToolBar ToolBar.OverflowMode="AsNeeded">
      <ComboBox Name="lstText">
        <ComboBoxItem Content="Gorm den Gamle" IsSelected="True" />
        <ComboBoxItem Content="Harald Blåtand" />
        <ComboBoxItem Content="Svend Tveskæg" />
        <ComboBoxItem Content="Knud den Store" />
      </ComboBox>
      <Label Content="Font:" />
      <ComboBox Name="lstSize">
        <ComboBoxItem Content="12" />
        <ComboBoxItem Content="18" IsSelected="True" />
        <ComboBoxItem Content="24" />
        <ComboBoxItem Content="48" />
        <ComboBoxItem Content="72" />
      </ComboBox>
      <Button Name="cmdOk" Content="Draw" Click="cmdOk_Click" />
    </ToolBar>
    <ToolBar ToolBar.OverflowMode="Always"  >
      <Label Content="Background color:"/>
      <ComboBox Name="lstBg"/>
      <Label Content="Text color:"/>
      <ComboBox Name="lstFg"/>
    </ToolBar>
  </ToolBarTray>
  <StatusBar DockPanel.Dock="Bottom" Height="30" >
    <Label Name="lblTime" Width="100" />
    <Separator/>
    <Label Name="lblPos" Width="100" />
  </StatusBar>
  <Label Name="lblText" FontWeight="Bold" HorizontalContentAlignment="
  Center"
    VerticalContentAlignment="Center" MouseMove="area_MouseMove"/>
</DockPanel>
```

This time the layout control is a *DockPanel* and to place a toolbar and a status bar is a typical use of a *DockPanel*.

Most of the XML code for the toolbar does not require special explanation, but note that the two *ToolBar* controls are placed in a *ToolBarTray* control. It allows the user to move the two *ToolBar* components, or for example you can swap the toolbars or place them below each other. The *ToolBarTry* also show an arrow for an overflow area if the toolbar cannot shows all the components. The content of the combo boxes for text and font size are defined in XML, but the content of the two other combo boxes are defined in C#. Note that except for the button *Draw* there is no event handling for the components in the toolbar.

As you can see, it is easy to define a status bar, and it can generally contain all the controls you want. Note in particular that it contains a separator.

The last component is the *Label* in the center of the window and the only thing to note is the event handler for moving the mouse.

Then there is the C# code:

```
namespace ToolsProgram
{
  public delegate void Action();

  public partial class MainWindow : Window
  {
    private Timer timer = null;

    public MainWindow()
    {
      InitializeComponent();
      Init();
      Draw();
      TimerCallback action = new TimerCallback(OnTick);
      timer = new Timer(action, null, 1000, 1000);
    }

    private void Init()
    {
      Type brushesType = typeof(System.Windows.Media.Brushes);
      var properties =
        brushesType.GetProperties(BindingFlags.Static | BindingFlags.
        Public);
      int n = 0;
      foreach (var prop in properties)
```

```
    {
      ColorItem item =
        new ColorItem(prop.Name, (SolidColorBrush)prop.GetValue(null,
        null));
      lstBg.Items.Add(item);
      lstFg.Items.Add(item);
      if (item.Name == "White") lstBg.SelectedIndex = n;
      else if (item.Name == "Black") lstFg.SelectedIndex = n;
      ++n;
    }
  }

  private void Draw()
  {
    lblText.Content = ((ComboBoxItem)lstText.SelectedItem).Content.
    ToString();
    lblText.FontSize =
      int.Parse(((ComboBoxItem)lstSize.SelectedItem).Content.
      ToString());
    lblText.Background = ((ColorItem)lstBg.SelectedItem).Color;
    lblText.Foreground = ((ColorItem)lstFg.SelectedItem).Color;
  }

  private void cmdOk_Click(object sender, RoutedEventArgs e)
  {
    Draw();
  }

  private void OnTick(object state)
  {
    Action action = new Action(UpdateLabel);
    Dispatcher.BeginInvoke(action, null);
  }

  private void UpdateLabel()
  {
    lblTime.Content = DateTime.Now.ToLongTimeString();
  }

  private void area_MouseMove(object sender, MouseEventArgs e)
  {
   lblPos.Content = string.Format("({0},{1})",
```

```
        e.GetPosition(lblText).X, e.GetPosition(lblText).Y);
    }

    class ColorItem
    {
      private Brush color;
      private string name;

      public ColorItem(string name, Brush color)
      {
        this.color = color;
        this.name = name;
      }

      public Brush Color
      {
        get { return color; }
      }

      public string Name
      {
        get { return name; }
      }

      public override string ToString()
      {
        return name;
      }
    }
  }
}
```

The method *Init()* is used to initialize the two combo boxes for colors which should be initialized with the same colors as in the class *Brushes*. It may seem complicated, but in principle it is a bit like the solution of the same problem in *Problem 1*, and again I think you just should accept the code. The objects in the two combo boxes has the type *ColorItem* which is an inner class and the class encapsulates a name and a *Brush* object. The name is used for the text shown in the combo box, and if the user selects a name in the combo box it has assign the *Brush* object to be used for ether a foreground or background color. You should note that the method *Init()* also ensure when the program starts that white is selected for background and black for foreground.

The method *Draw()* is called both from *InitializeComponent()* and from the event handler for the *Draw* button and the method uses the settings from the toolbar to update the *Label* component.

You should the event handler for moving the mouse and to read the coordinates for the mouse's position. The handler is performed when you move the mouse and the mouse is over the *Label* component.

To update the watch the class defines a timer:

```
private Timer timer = null;
```

and is an object which fires an event with a specific time interval. The program can define an event handler for this event and then perform an action every time the event occurs. The event handler for a timer is called a call back function or timer function because the event is fired in another thread (is explained later), and this timer function is defined by a delegate which above is called *Action*.

The timer is created in the constructor to tick every second. This means that every second is called the method *OnTick()*, which then calls a method to update the status bar. It presents a small problem. As the timer runs in its own thread, this thread needs to update an element in another thread (the user interface) via the timer function, and it requires it to be done using a *Dispatcher* (just accept the code as it is).

4.8 MORE LAYOUT CONTROLS

I all examples until this place I have used three layout controls

1. *Grid*
2. *StackPanel*
3. *DockPanel*

and these are the most frequently used, and it is even relatively rare to use others, but there are others and they certainly have their uses as well. I will end this chapter with mention three others

1. *WrapPanel*
2. *UniformGrid*
3. *Canvas*

and the controls will be present through a few exercises.

Since these controls can be nested, you have complete control over how components are displayed in a window. The most complex (and also most used) of them is a Grid, while the other five are relatively simple. How a component is accurately displayed in a layout control is partly determined by the component's own properties, and here you should pay particular attention to the following, where you should note that these properties do not work in all containers:

1. *FlowDirection*, which defines how for example and other content should be arranged.
2. *Height*, which defines the components height. If it is defined as *auto* the height is determined by the layout control.
3. *HorizontalAlignment*, which defines the components horizontal alignment in the current container.
4. *HorizontalContentAlignment*, which defines the horizontal alignment of the components content.
5. *Margin*, which defines the components distance to the containers edges, but it works a bit different depending on how it is used. Sometimes *Margin* refers to the container's edges (for example a *Grid*), and sometimes it refers to the neighbor components in the same container (for example a *StackPanel*), but in practice it rarely leads to major misunderstandings. You specify a margin with four values in the order left, top, right and bottom.
6. *MaxHeight*, which defines the components maximum height.
7. *MaxWidth*, which defines the components maximum width.
8. *MinHeight*, which defines the components minimum height.
9. *MinWidth*, which defines the components minimum width.
10. *Padding*, which defines the minimum distance between a control's (layout control) child components.
11. *VerticalAlignment*, which defines the components vertical alignment in the current container.
12. *VerticalContentAlignment*, which defines the vertical alignment of the components content.
13. *Width*, which defines the components width.

In many contexts, WPF uses *Attached Properties*. Because a WPF control contains the necessary information for how to place it in a container (height, width, adjustment, and so on), it is sometimes necessary that a control can refer to the properties of the container containing the control. For example, for a *Button* that sits in a *Grid*, it may need to tell in which row and column it should be placed, so the button needs to be able to refer to properties in the container. It takes a bit of a detour.

When defining a Button in XML, an object of the type *Button* is created in the program. In an XML tag, you often specify attributes as *Width* and *Height* and others, which are similar to properties defined by the class *Button*. It is what you call common properties, and if you set such a property, you change a value in a particular instance of for example a *Button*. Attached properties such as *Grid.Row*, seems something different. When a component is placed in a container, it can access properties of the container using attached properties. The name of an attached property is always of the form

TypeName.PropertyName

where *TypeName* is the name of the type (a class) that defines a property named *PropertyName*, for example *Grid.Row*. The XML parser can therefore distinguish between a regular property and an attached property. A type *TypeName* that defines an attached property *PropertyName* has a static method called *SetPropertyName()* and has two parameters, the first being the object that initializes that property while the last one is the value. For example, if in the definition of a *Button* you writes

```
Grid.Row=”2”
```

it is parsed to the following method call:

```
Grid.SetRow(cmdOk, 2);
```

where *cmdOk* is the name of the control, that defines an attached property. The static method will then call a method on the object *cmdOk*:

```
cmdOK.SetValue(Grid.Row, 2);
```

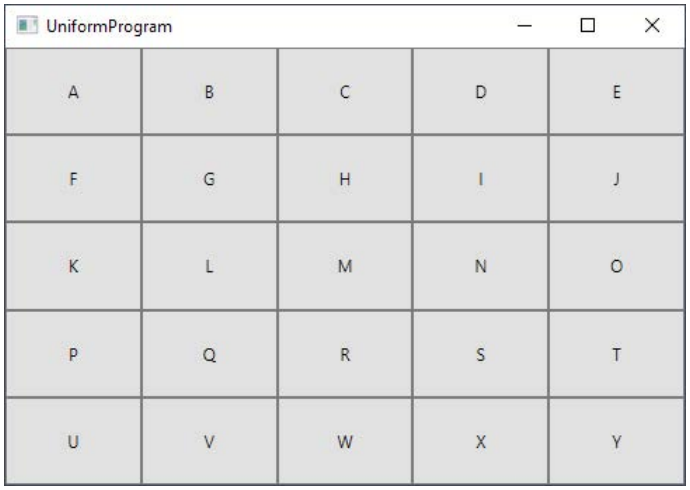
that store the actual property for the container in a collection in the *Button* object together with the value. The runtime system therefore knows which property in the component's container to initialize and with what value.

Apparently this is a long way, but the reason is that since a component can occur in many different containers, it would be in practice (impossible) to equip the component with properties for all the properties that one should be able to assign corresponding to all possible containers.

EXERCISE 8: A UNIFORMGRID

A Grid divides a window into rows and columns, but not necessarily of the same height or width. A *UniformGrid* is a simpler grid, which also divides the window into a number of rows and columns, but this time all rows have the same height and every column the same width. The result is that the window is divided into cells of the same size.

You must write a program which opens a window with 25 buttons:



when the buttons should be arranged in *UniformGrid* layout control. The XML should be as follows:

```
<UniformGrid Name="panel" Rows="5" Columns="5">
</UniformGrid>
```

This means that you must define all *Button* controls in C# and add them to the layout control. You can do that in the constructor as follows:

```
for (char ch = 'A'; ch <= 'Z'; ++ch)
{
  Button cmd = new Button();
  cmd.Content = ch;
  panel.Children.Add(cmd);
}
```

That is you dynamic add components to a window in code. Run the program and test what happens when you change the size of the window.

When you click on a button it should open a message box which shows the button's text and the cell where it sits in the grid, and an example could be:

To solve this you must add an event handler to each button. Start to define an event handler:

```
void cmd_Click(object sender, RoutedEventArgs e)
{
}
```

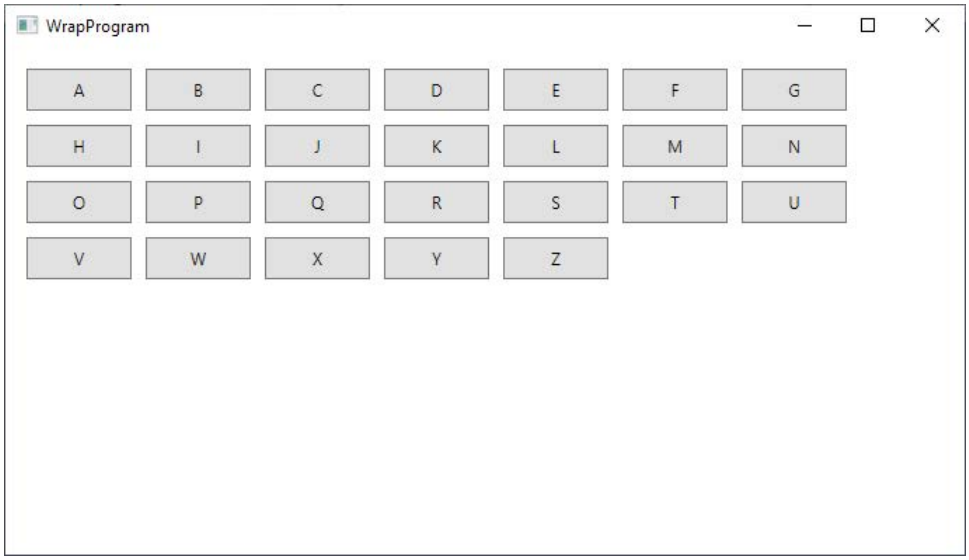
and you can then add the handler to the buttons with the following statement when you creates them:

```
cmd.AddHandler(Button.ClickEvent, new RoutedEventHandler(cmd_Click));
```

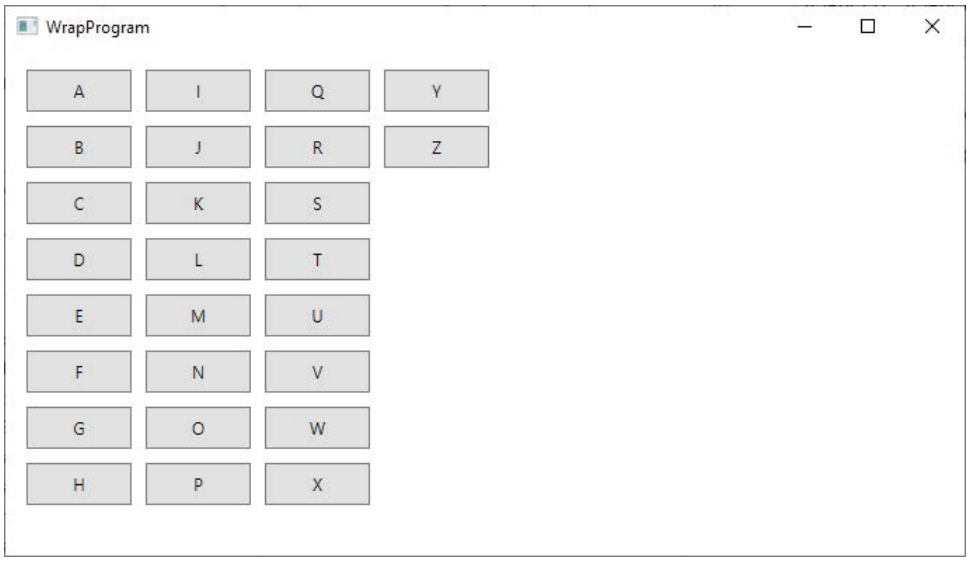
After that each button has an empty event handler and you must write the code for the message box. Here you should think of a *Button* component's *Tag* property.

EXERCISE 9: A WRAPPANEL

A *WrapPanel* is a panel similar to a *StackPanel* and places components in a row horizontally or vertically, but unlike a *StackPanel*, components are laid out on multiple rows if there is no space in the window. You must write a program which opens the following window:

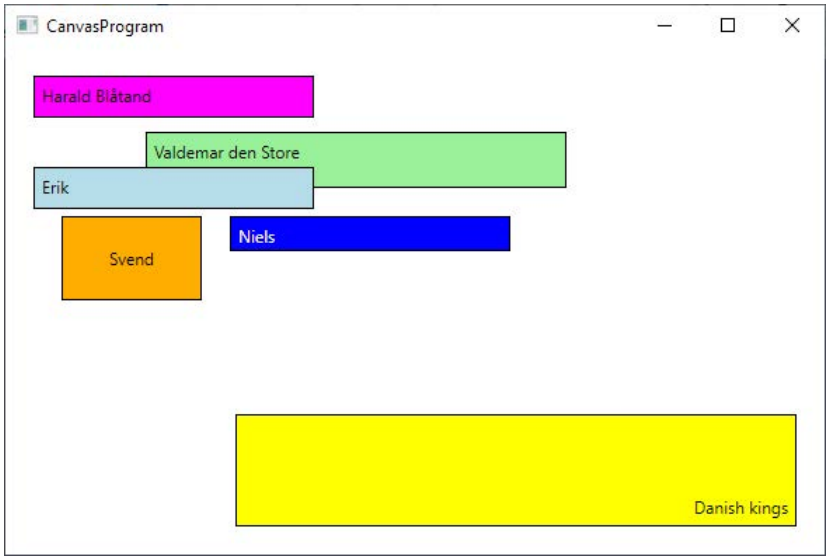


which contains 26 buttons arranged with a *WrapPanel*. Note the space between the buttons. The buttons must be added to the window dynamic in C#, and when you click on a button (the same for all buttons) the *WrapPanel* must shift orientation:



EXERCISE 10: A CANVAS

A Canvas is a panel that lays out components at fixed positions and at a fixed size. For each component, you specify coordinates for the component's location in the container (which by default is relative to the upper left corner), and also specify the component's height and width. You must write a program which opens the following window with 6 *Label* components.



The yellow component must follow the lower right corner of the window.

EXERCISE 11: A SCROLLVIEWER

A *ScrollViewer* is not really a layout control, but it is used to scroll a panel. A *Grid* is a panel where components are placed in rows and columns. If you do not define rows and columns, a *Grid* has one row and one column and thus one cell. This cell can contain all the components you want and if nothing else, the components are centered in the center of the cell according to their current size and thus on top of each other. However, as shown above, with attributes you can define which edges a component should be aligned with and with a *Margin* attribute you can specify the distance to those edges. The result is that you can lay out components in a *Grid* without defining rows and columns.

You must write a program with a window that contains 5 buttons in a *Grid* with one cell, but such that size and location are determined using the following attributes:

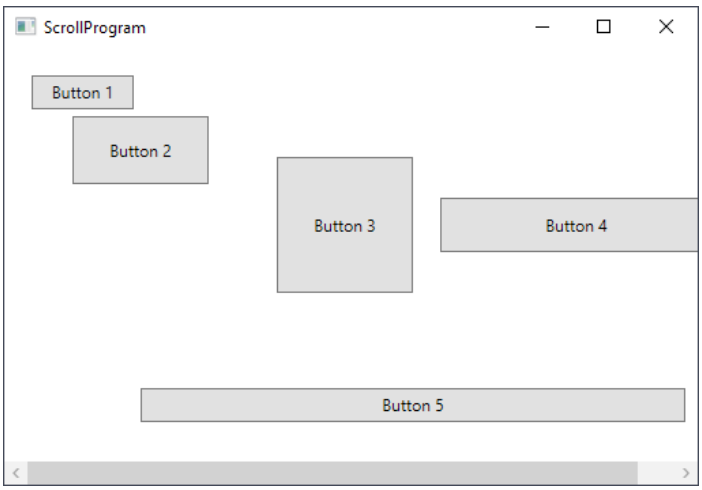
1. *Width*
2. *Heigth*
3. *HorizontalAlignment*
4. *VerticalAlignment*
5. *Margin*

The size of the *Grid* is determined by the components and if the window's size is to small not all buttons are shown. To solve this problem you must place the *Grid* in a *ScrollViewer*:

```
<ScrollViewer HorizontalScrollBarVisibility="Auto"
  VerticalScrollBarVisibility="Auto">
  <Grid>
  </Grid>
</ScrollViewer>
```

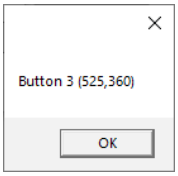
The window will then display a scroll bar if the entire panel cannot be displayed (the window is smaller than the panel size). In general, a *ScrollViewer* is a panel that allows you to scroll another component and especially another panel.

Write the program that opens the following window:



Test the program to see if the scroll bars works as you wants.

If you click on a button (the same for all buttons) the program must show a message box:

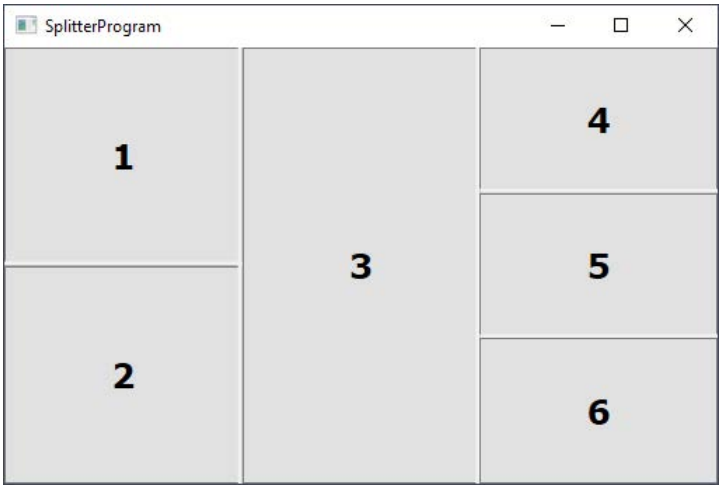


which shows the name (content) of the button and the current size of the window.

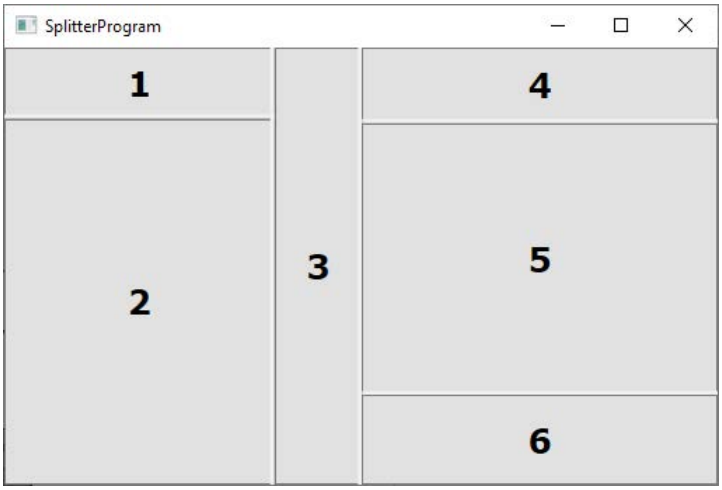
EXERCISE 12: A GRIDSPLITTER

A *GridSplitter* is a control that is used with a *Grid* and is used to dynamically change the height and width of rows and columns at runtime. The user can use the mouse to draw dividing lines between rows and columns.

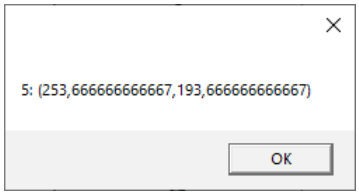
You must write a program which place 6 buttons in a *Grid* with 3 columns. The first column must contain a *Grid* with 2 rows and the last column a *Grid* with 3 buttons, and the result should be as shown below:



Between the components are *GridSplitter* components, 5 in all. Below is the window after moving the splitters:

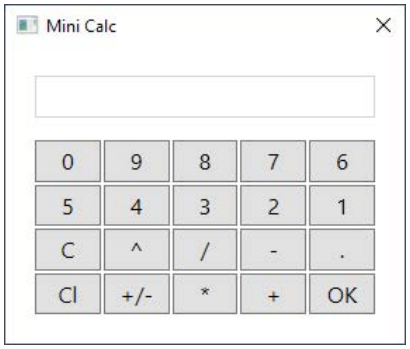


If you click on a button the program must open a message box showing the buttons name and the current size of the button:



PROBLEM 3: A SMALL CALCULATOR

Write a program that you can call *MiniCalc*. The program must open a window as shown below. The window has an input field and 20 buttons. The program should simulate a simple calculator and should only be operated by using the mouse. The input field is read only (use the method *setEditable()*), and the text is right-justified. The 10 top buttons should be self-explanatory. The buttons in the third row are from left:



1. remove the last character in the display
2. exponentation
3. division
4. subtraction
5. decimal point

The buttons in the bottom row are from left:

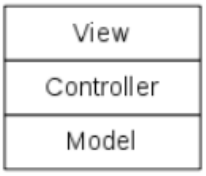
1. clear the display
2. shift sign on the content in the display
3. multiplication
4. addition
5. enter, that calculate the value of the expression in the display and update the display with the value

It should not be possible to enter directly in the display, and you can only enter an expression using the buttons. The program must validate that there only are added legal characters in the display (character which leads to a legitimate expression). If you try to perform an illegal calculation by clicking OK, you should get an error message.

It is obviously a very simplified calculator, and you can only work with very simple expressions, especially because you cannot enter parentheses.

5 GUI APPLICATIONS

Until now, all programs in this book have been simple and typically consist of one or more windows, and a window is created as an XML document that defines the user interface and an underlying C# file with event handlers. In principle, any GUI program is written that way, but it is clear that as the programs get bigger and begin to resemble the programs you encounter in everyday life, there is also more to keep track of. For this, a special pattern has been developed for designing a GUI program called MVC for *Model View Controller*. The pattern is in principle very simple and means to develop a GUI program with a three-layer architecture



where the model consists of the classes that define the program's data and state, while the controller layer has classes that mainly perform control of entries and so on, and optionally also has essential calculation functions (business logic). Finally, the view layer has all the classes for the user interface and thus for windows and dialog boxes.

The goal of the pattern is to separate the code so that the code regarding the program's data are placed in classes in the model and code used for data control and logical operations are placed in the control layer, whereas the view layer alone must contain the code which has to do with the visual representation and user interaction. Conforms to the pattern you get a code which might be a bit bigger, but in return is far easier to read and understand.

In the following books, when there is slightly larger programs, I will begin using the pattern, and so far it is only a question of the division of the program's classes in logical layers, and although it does not sound like much, the pattern has proved very appropriate as architecture for a GUI program. Therefore, I would in a small way begin using the pattern only as a way to a reasonable division of the code. There is much more to say about MVC, and including how each layer should communicate with the others, and there may also be several layers, but the details I'll defer to a later book.

In this chapter I will look at three examples. These are still small programs but nevertheless programs with greater complexity than what is the case with the examples above. In addition, these are programs with some justification and not just programs that show the use of certain components and syntax. The examples are formulated as tasks that you have to solve, but in the same way as *Problem 1* where the solution of the task is outlined as guidelines, you are encouraged to follow.

PROBLEM 3: LOAN CALCULATOR

You must write a program that is a loan calculation program and thus a program where the user can enter the amount of a loan, the interest rate and number of periods. The program should then calculate the payment when the loan is an annuity. It should be mentioned that there are many such programs on the Internet that you can compare the result with. An annuity is a loan that is amortized with a fixed payment each period. A payment consist of interest and repayment and in the beginning is a big part of the payment interest and a smaller part is repayment. This situation is changing continuously, so that towards the end of the loan period, the largest part of the payment is repayment. If

- G = the loan
- y = the payment
- n = number of periods
- r = interest rate

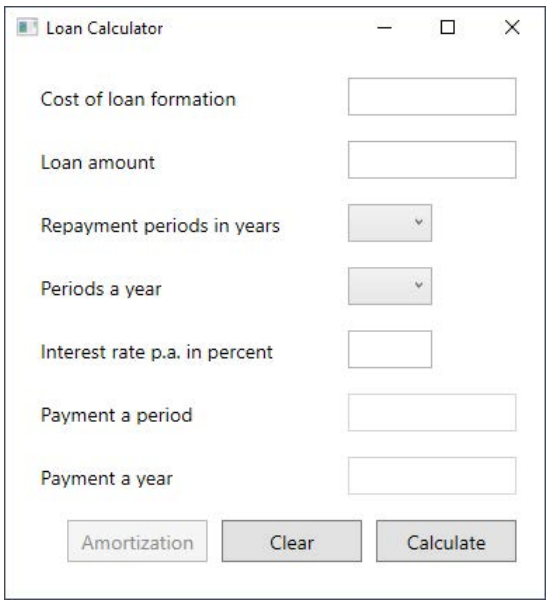
the relationship between the loan and the payment is given by the following formula:

$$G = y\frac{1-(1+r)^{-n}}{r} \Leftrightarrow y = \frac{rG}{1-(1+r)^{-n}}$$

This formula assumes that the interest rate is constant throughout the loan period and the first payment must take place 1 period after you got the loan, and you should assume that these assumptions apply. The following formula determines the outstanding debt immediately after the kth payment is paid:

$$Rest = G(1+r)^k - y\frac{(1+r)^k - 1}{r}$$

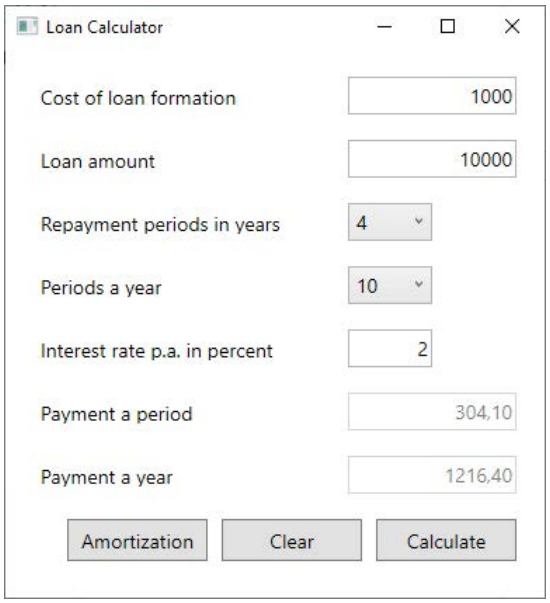
The program must open the following window



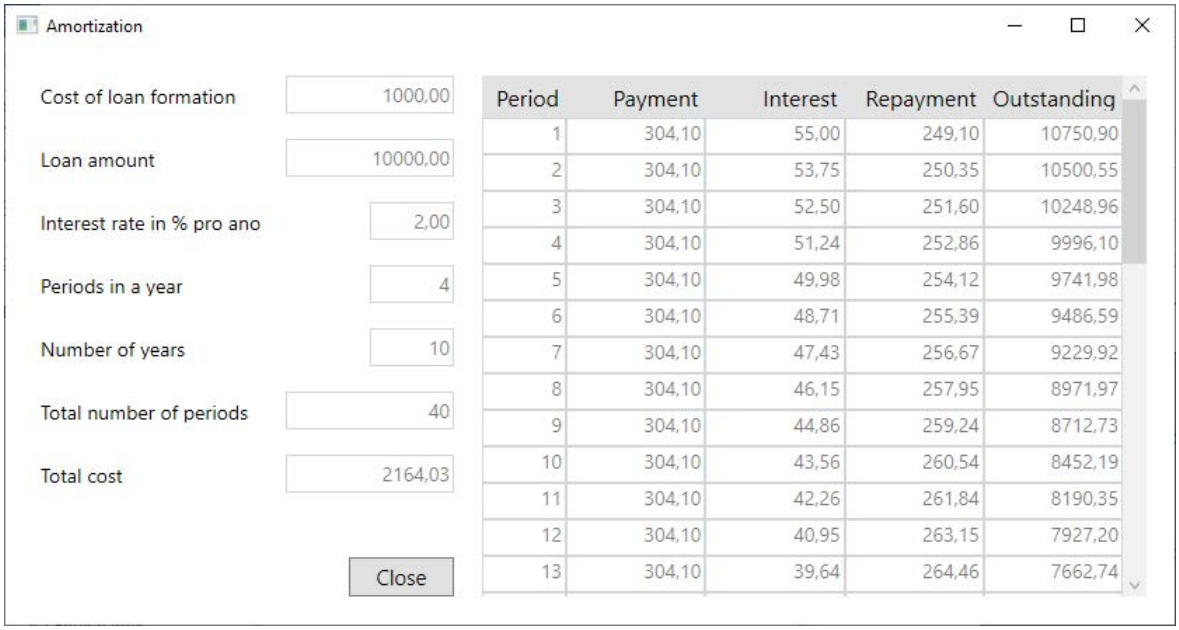
where the user must enter:

- cost of loan formation
- the size of the loan
- the interest rate in percent pro ano
- the repayment period in years
- number of periods a year

Note the button *Amortization* is disabled. If the user enter / select legal values for the 5 fields and click on *Calculate* the program must calculate the payment a period and the total payment a year. An example could be:



Now the button *Amortization* (if there is no errors) is enable, and if you clicks the button the program shows an amortization, and thus an overview of the loan that for each period shows the payment, the interest, the repayment and the debt outstanding after this period, for example:



To write the program you should follow the guidelines.

1. Create a new project which you can call *LoanCalculator*.
2. Writes the XML for *MainWindow* and test the user interface until it looks like the window *Loan Calculator* (see the window above).
3. Add readable names to the *TextBox*, *ComboBox* and *Button* components and add (empty) event handlers for the three buttons. Then the XML for the *MainWindow* should be done.
4. Add a class to the project which you can call *Loan*. The class should represents a loan as:

```
public class Loan
{
  private double formation;   // cost of loan formation
  private double amount;     // loan amount
  private double rate;       // interest rate at period as a decimal
                                number
  private int period;        // periods a year
  private int year;          // loan periods in years
  private int periods;       // total periods
```

The class must have a constructor to initialize the fields, and the class must have read-only properties for all fields. Also the class must implements the calculation methods, that is similar to the formulas mentioned above.

You can think of the class as a simple model class for the program.

5. Add another class called *MainCtrl*. For now the class should only have one instance variable and two methods:

```
class MainCtrl
{
  private Loan loan = null;

  public Loan Loan
  {
    get { return loan; }
  }

  public void Clear()
  {
    loan = null;
  }

  public string Validate(TextBox txtCost, TextBox txtAmount, TextBox
  txtInterest,
    ComboBox lstYears, ComboBox lstPeriods)
  {
    ...
  }
}
```

The method *Validate()* should be used to validate the users input, and the return value is an error message if the data are not legal:

1. the value for loan formation must be empty but if not it should be a non negative integer
2. the amount must be a positive value
3. number of periods a year must be between 1 and 12, both included
4. number of years must be between 1 and 60, both included
5. the interest rate must be a value between 0 and 100

If the value for all fields are legal, the controller must create a *Loan* object.

You can think of the class as a controller for *MainWindow*.

6. You must then write the C# code for *MainWindow*. The class must have an instance variable of the type *MainCtrl*, it needs an object representing the controller.

Implements the event handler for the *Calculate* button. It should call the controller to validate the user's data, and if there is an error the handler must show a message box with an error message. Else it must update the two payment fields. Note that the controller has a property for the *loan* object. It is a property which returns the model.

Implements the *Clear* button as it must resets all fields in *MainWindow* and also sets the model in the controller object to *null*.

You can only write the last event handler after you have created the window for the amortization.

7. Add e new window called *LoanWindow* to the project. The window should look like the window *Amortization* shown above, but creating this window is not easy.

To create the left side with the *TextBox* fields and the button is quite simple and can be done in XML. To create the table to the right is more complicated as it must primarily be done in C# as the number of rows must be dynamical assigned. Start by creating the left side in XML and then the right side as a *Grid* in a *ScrollView*:

```
<ScrollViewer Grid.Row="0" Grid.Column="2" Grid.RowSpan="8"
  HorizontalScrollBarVisibility="Auto" VerticalScrollBarVisibility="Auto"
  Margin="20,0,0,0">
  <Grid x:Name="tblData">
    <Grid.RowDefinitions>
      <RowDefinition Height="30"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="60" />
      <ColumnDefinition />
      <ColumnDefinition />
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Label Grid.Row="0" Grid.Column="0" Content="Period" FontSize="16"
      HorizontalContentAlignment="Right" Background="#E0E0E0" />
    <Label Grid.Row="0" Grid.Column="1" Content="Payment" FontSize="16"
      HorizontalContentAlignment="Right" Background="#E0E0E0" />
    <Label Grid.Row="0" Grid.Column="2" Content="Interest" FontSize="16"
      HorizontalContentAlignment="Right" Background="#E0E0E0" />
    <Label Grid.Row="0" Grid.Column="3" Content="Repayment" FontSize="16"
      HorizontalContentAlignment="Right" Background="#E0E0E0" />
    <Label Grid.Row="0" Grid.Column="4" Content="Outstanding" FontSize="16"
      HorizontalContentAlignment="Right" Background="#E0E0E0" />
  </Grid>
</ScrollViewer>
```

After creating the XML (and also the XML for the rest of the window) the table has only one row, which is the table header. The rest of the rows must be created in code.

When you have created the window you must implement the last event handler in *MainWindow*, and the handler must open *LoanWindow*. Test the new window, and when it all look nice you should add a parameter to the constructor in *LoanWindow*:

```
public LoanWindow(Loan loan)
{
```

and the event handler in *MainWindow* must send the model object as actual parameter.

8. You must then write the code for *LoanWindow* and the most important is how to dynamic add rows to the *Grid* in the user interface. Add the following method which is a method that creates and initialize a *TextBox* component

```
private TextBox CreateField(string text, int row, int col)
{
  TextBox field = new TextBox();
  field.FontSize = 14;
  field.HorizontalContentAlignment = HorizontalAlignment.Right;
  field.Text = text;
  field.IsReadOnly = true;
  field.IsEnabled = false;
  Grid.SetRow(field, row);
  Grid.SetColumn(field, col);
  return field;
}
```

Compare to what you would have written in XML and it is easy enough to understand the meaning of the individual statements. With this method you can dynamic creates the table:

```
private void Initialize(Loan loan)
{
  // initialize all fields in the left side of the window

  // loop over all periods for the loan
  for (int i = 1; i <= loan.Periods; ++i)
  {
    RowDefinition rowDef = new RowDefinition();
    rowDef.Height = new GridLength(26);
    tblData.RowDefinitions.Add(rowDef);
    tblData.Children.Add(CreateField("" + i, i, 0));
    tblData.Children.Add(CreateField(string.Format("{0:F2}", loan.
    Payment), i, 1));
    // add the other fields
  }
}
```

Test the program, and after some adjustments it should all works.

You should note that creating the table as shown above and dynamically creating a *Grid* is not the best solution and if the table gets large you can observe a performance problem. There are better solutions, but they will have to wait until later after I have dealt with WPF in more detail.

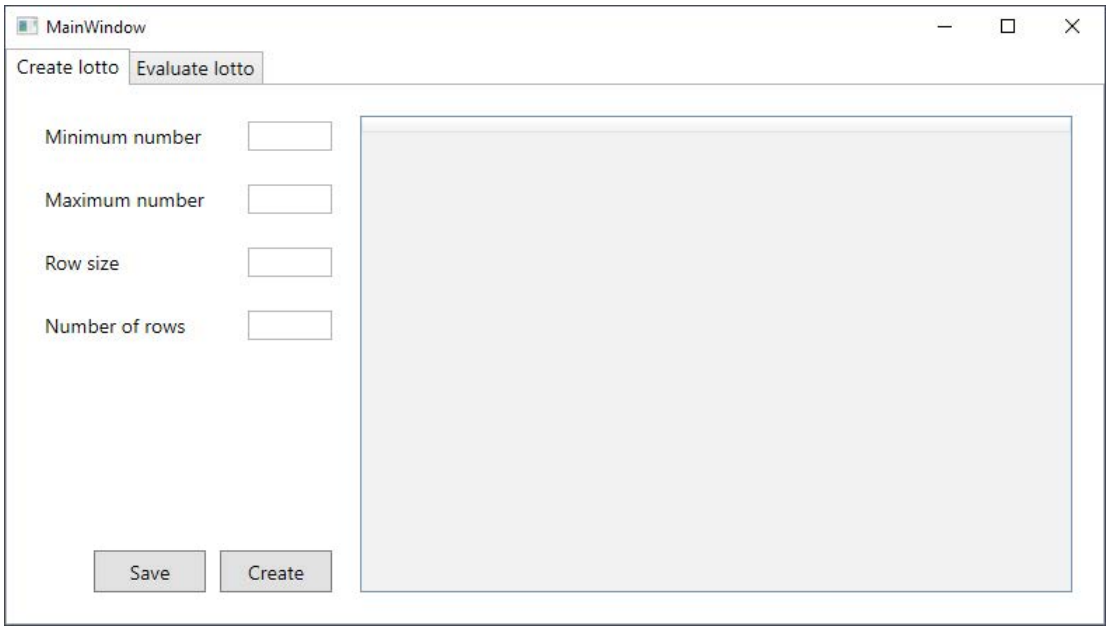
9. As a final improvement you should in the same way as in *Problem 1* change the program so the user in *MainWindow* can only enter numbers. There is however an additional challenge as the user must also enter a decimal point. Try to solve this problem when

1. only one decimal point must be accepted
2. only the decimal point for current culture must be accepted
3. if should be possible to use the numeric keyboard

PROBLEM 4: LOTTO

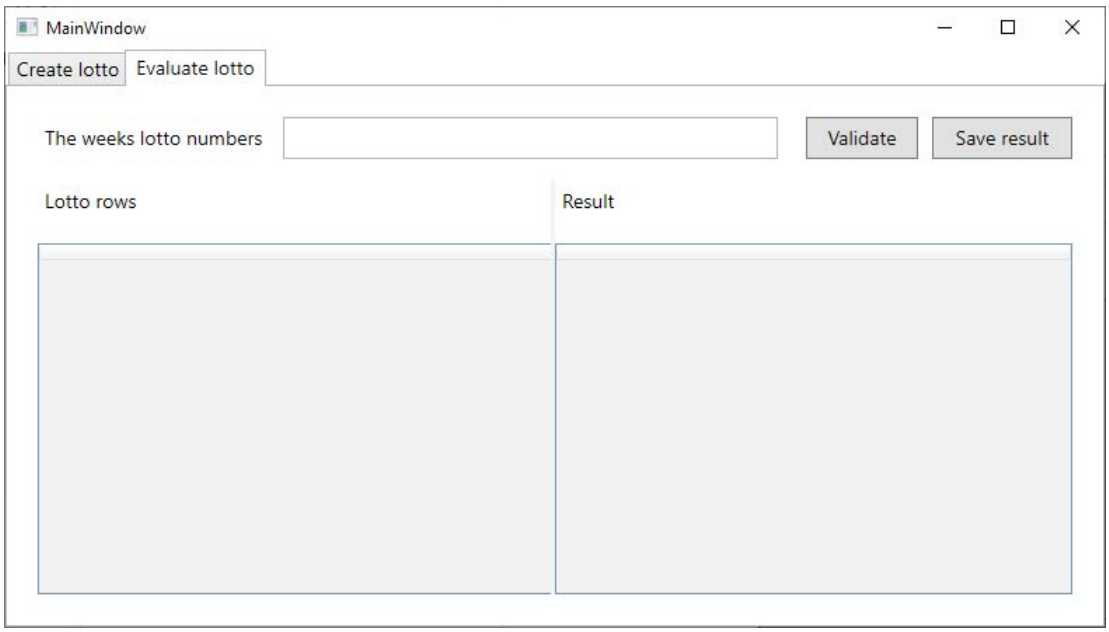
The final example in the previous book shows a command used to create lotto rows. The rows could be saved in a file, and the command could also be used to evaluate the rows against the weeks lotto. You are encouraged to read the last chapter of the previous book to make sure you remember the problem and how the program was written. In this problem you must write the same program, but with the difference that this time the program should have a graphical user interface. In fact, it is not a completely unknown issue where you have a program that needs to be changed so that it works in a different way and for example so that it gets a different user interface.

When the program opens it should opens the following window:



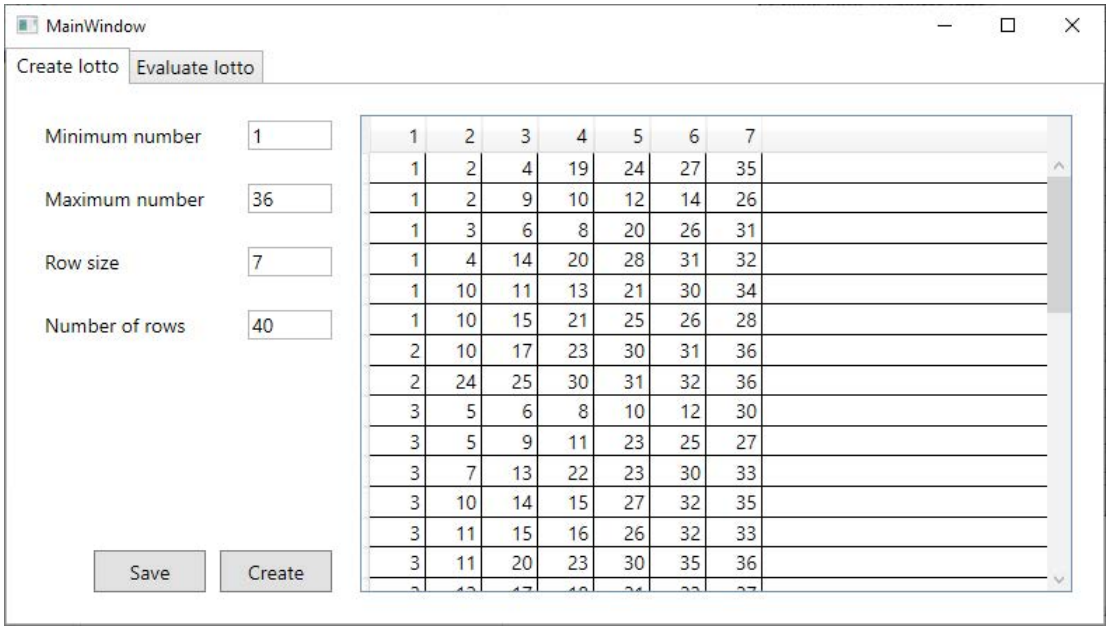
There are four input fields which must be used to enter the values for current lotto and the number of rows the program should create. The gray area to the right should be used for a table to show the created rows.

The window has two tabs, and the first tab is used to create the rows, and the second tabs is used to evaluate a saved lotto game:

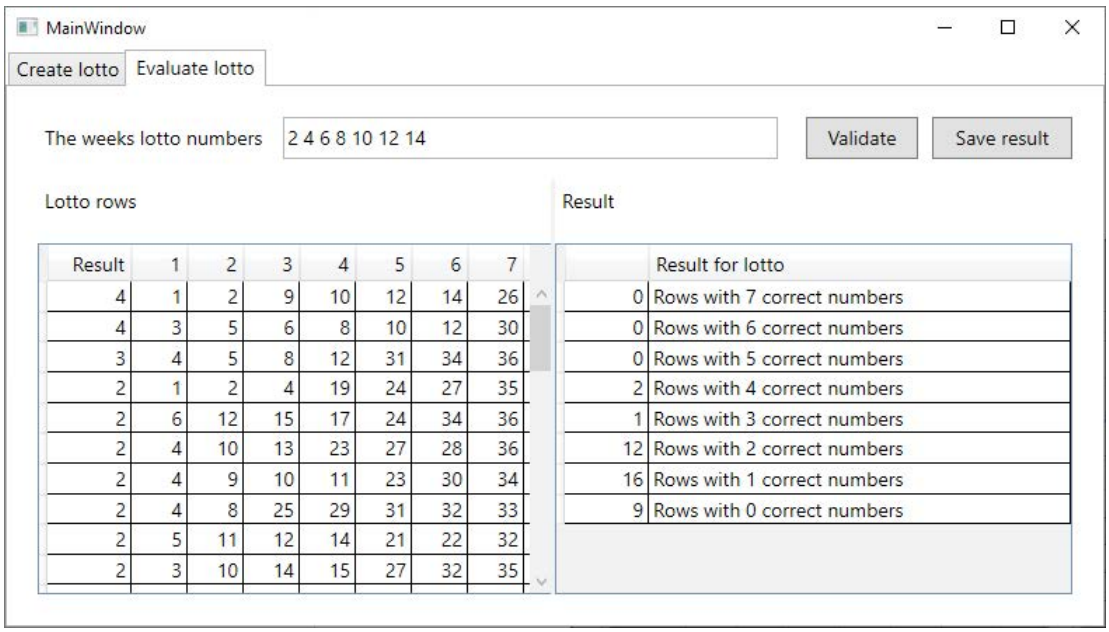


The *TextBox* is used to enter the weeks lotto separated by spaces, while the two tables are used for lotto rows and the result.

Below is shown an example where I have entered parameters for a lotto (the default values) and entered that the program must creates 40 rows:



When the rows are saved in a file I can select the second tab and enter the lotto numbers to evaluate the rows, and the result could be as below:



To solve the problem you should use as much of the code from the solution from the previous book as possible. You should note that the graphical user interface uses concepts that I have not mentioned and here tabs and tables with lottery rows. Therefore, you should follow the following guidelines.

1. Create a new WPF project which you can call *Lotto*. The project should only have one window, but, on the other hand, it is a relatively complex window.

After you have created the project you must from the previous project (the previous book) add the classes

- *LottoGame*
- *LottoNumber*
- *LottoRow*
- *Util*

These classes should all be used without changes.

2. Create the window, that is write the XML for the window. You must start to add a *TabControl* to the *Grid* layout. A *TabControl* contains *TabItem* elements with one *TabItem* for each tab, and in this case it means two. Each *TabItem* contains a control which can be a *Grid* control, and you must add a *Grid* control to each *TabItem* and you then design each tab writing XML for each *Grid* control.

Basically, it doesn't include anything new, but the first tab should contain one table, while the second should contain two tables. In the previous example (the amortization table) you solved the problem with a *Grid*, but for large tables it is not the solution and then you can use a *DataGrid*, which is a component that displays data organized in rows and columns. In principle, it is quite simple to use a *DataGrid*, but partly you lack some prerequisites and partly the tables must be created in C#, since you do not know the number of columns or the number of rows. For now you must add three *DataGrid* elements to your design which you can call:

```
<DataGrid x:Name="tblCreate" ... />     <!-- first tab -->
<DataGrid x:Name="tblLotto" ... />      <!-- second tab -->
<DataGrid x:Name="tblResult" ... />     <!-- second tab -->
```

The result of this step should be a program that show the two tabs as shown at the beginning of this problem.

3) A *DataGrid* shows a collection of objects with one object at each row. In each column is shown the value of a property for the objects, and you needs to tell which properties should be used. For that you must write some classes to represent the data for the tables. You should start with the last table, since it should have only two columns and is therefore the simplest. Add the following class to you project:

```
namespace Lotto
{
  class ResultType
  {
    private int count;
    private int index;

    public ResultType(int count, int index)
    {
      this.count = count;
      this.index = index;
    }

    public int Res
    {
      get { return count; }
    }

    public string Text
    {
      get { ... }
    }
  }
}
```

The class represents a result, for example

6 Rows with 3 correct numbers

and for that the class has two properties which are the properties that will be used for columns in the table.

The two other tables should show both show lotto rows and then be defined by the same data type. A lotto row can have between 3 and 15 numbers, and there should also be a column for the number of correct numbers and the data type for a row in the table must then have 16 properties for columns. You must then add the following class to your project:

```
namespace Lotto
{
  class TableType : IComparable<TableType>
  {
    private LottoRow row;

    public TableType(LottoRow row)
    {
      this.row = row;
    }

    public LottoRow Row
    {
      get { return row;  }
    }

    public int Res
    {
      get { return row.Result; }
    }

    public int N0
    {
      get { return 0 < LottoGame.Size ? row[0].Number : 0; }
    }

    public int N1
    {
      get { return 1 < LottoGame.Size ? row[1].Number : 0; }
    }

    ...

    public int N14
    {
      get { return 14 < LottoGame.Size ? row[14].Number : 0; }
    }

    public int CompareTo(TableType tableType)
    {
      return row.CompareTo(tableType.Row);
    }
  }
}
```

The class is in principle quite simple (note I have not shown the full class), but is a bit of a work around. Basically, the class is nothing more than an encapsulation of a lottery row for the purpose of associating properties with the individual lottery numbers. As the number of numbers in a lottery row varies, it is necessary to test whether the value of a property is defined and if not set it to 0. The class defines 17 properties at all, and unless otherwise noted, the tables will display 17 columns. Also note that the class is defined comparable.

4) You now has the two classes which defines data for the tables, and you are ready to write the code that creates the tables. Add a class called *MainCtrl* which you can think of as a controller class for the window:

```
namespace Lotto
{
  class MainCtrl
  {
    private List<TableType> rows = new List<TableType>();
    private List<ResultType> result = new List<ResultType>();

    public List<TableType> Rows
    {
      get { return rows; }
    }

    public List<ResultType> Result
    {
      get { return result; }
    }
  }
}
```

For now the class only has two collections representing data for the tables. Create an object of the class as an instance variable in the class *MainWindow*:

```
private MainCtrl ctrl = new MainCtrl();
```

To create the first table (the table in the first tab) you must change the XML as:

```
<DataGrid x:Name="tblCreate" ... AutoGenerateColumns="True" ColumnWidth="*"
  AutoGeneratingColumn="tlbCreate_AutoGeneratingColumn"/>
```

It tells the *DataGrid* to automatically create the columns from an assigned data source, that the width of the columns should be distributed equally so that all space available is used, and the last attribute reference an event handler which is executed every time a column is added to the table. You must write the event handler in C#:

```
private void tlbCreate_AutoGeneratingColumn(object sender,
  DataGridAutoGeneratingColumnEventArgs e)
{
  if (e.Column.Header.ToString().Equals("Res") ||
      e.Column.Header.ToString().Equals("Row"))
        e.Column.Visibility = Visibility.Hidden;
  else
  {
    int n = int.Parse(e.Column.Header.ToString().Substring(1));
    e.Column.Header = "" + (n + 1);
    if (n >= LottoGame.Size) e.Column.Visibility = Visibility.Hidden;
    else e.Column.Width = 40;
  }
  e.Column.HeaderStyle = RightAlignment();
  e.Column.CellStyle = RightAlignment();
}
```

You should remember that the type *TableType* defines 17 properties and for the first table I do not want to show the columns for *Res* and *Row*. The event handler then starts to test for these columns and defines not to show these columns. The rest of the columns should only be shown if they are columns representing a number for current lotto, and if not they are hidden. Else I define a column header and sets the with of the column to 40.

The last two statements in the event handler should right align the text in the columns. A *DataGrid* does not have a property for that, and you must instead use a style that is also a WPF term which is explained later, but in this case you can create a style with the following method:

```
private Style RightAlignment()
{
  Style style = new Style();
  style.Setters.Add(new Setter(HorizontalContentAlignmentProperty,
    HorizontalAlignment.Right));
  style.Setters.Add(new Setter(TextBlock.TextAlignmentProperty,
    TextAlignment.Right));
  return style;
}
```

Now you have written the code for the first table and you must assign a data source to the table. You do that in the event handler for the *Create* button:

```
private void cmdCreate_Click(object sender, RoutedEventArgs e)
{
  string message = null; //ctrl.Create(txtMin, txtMax, txtSize, txtCount);
  if (message != null)
  {
    MessageBox.Show(message, "Error", MessageBoxButton.OK,
    MessageBoxImage.Error);
    tblCreate.ItemsSource = null;
  }
  else tblCreate.ItemsSource = ctrl.Rows;
}
```

The event handler will fill the table with data from the collection *Rows* in the class *MainCtrl*. If you run the program and click on the *Create* button nothing happens since you are missing the method *Create()* in the controller class, but it is the next step.

You must then in the same way creates the two other tables, which primarily means that you have to write the event handlers that are executed when a column is automatically added to the tables.

5) In this step you must write the code to create a lottery and save the result in a file. Note that you can find the code for that in the class *LottoCoupon* in the previous solution, and although the class cannot be used directly, you can probably use part of the code.

In the class *MainCtrl* you must add a method:

```
public string Create(TextBox txtMin, TextBox txtMax, TextBox txtSize,
  TextBox txtCount)
{
}
```

The parameters are the *TextBox* components from the user interface. The method must evaluates the user input, and if one of the values are illegal it must return an error message. Else the method must create the lottery which means to fill the list *rows*. Note that you can see how in the class *LottoCoupon*.

When you can create a lottery and show the result in the table in the first tab, you must add an event handler for the *Save* button. It should call a method in the class *MainCtrl* to do the work. The method must open a usual browser dialog box to browse the file system for where to save the file. Note again that you can see in the class *LottoCoupon* how to save the lottery.

6) The last step is to validate a saved lottery. Add a method *Validate()* to the class *MainCtrl* to validate the lottery:

```
public string Validate(TextBox txtNumbers)
{
}
```

The parameter is a reference to a *TextBox* component in the second tab with the numbers for the weeks lotto separated by spaces. The method must

1. open a browser dialog box to browse the file system for the file with the lottery
2. initialize the *LottoGame* object from the file
3. create a *LottoRow* for the weeks lotto from the method's parameter
4. read the rows from the file, evaluate each row and update the list *rows*
5. update the list *result*

When you have written the method and used it from the event handler for the *Validate* button you should be able to evaluate a lottery and updates the tables in the user interface.

As the last ting you must add a method to the class *MainCtrl* used to save a validated lottery. You can find the format for the file in the class *LottoCoupon*.

PROBLEM 5: SOLITAIRE

In this problem you must want to write a program that simulates a solitaire. It is a very simple solitaire game where the rules are as follows:

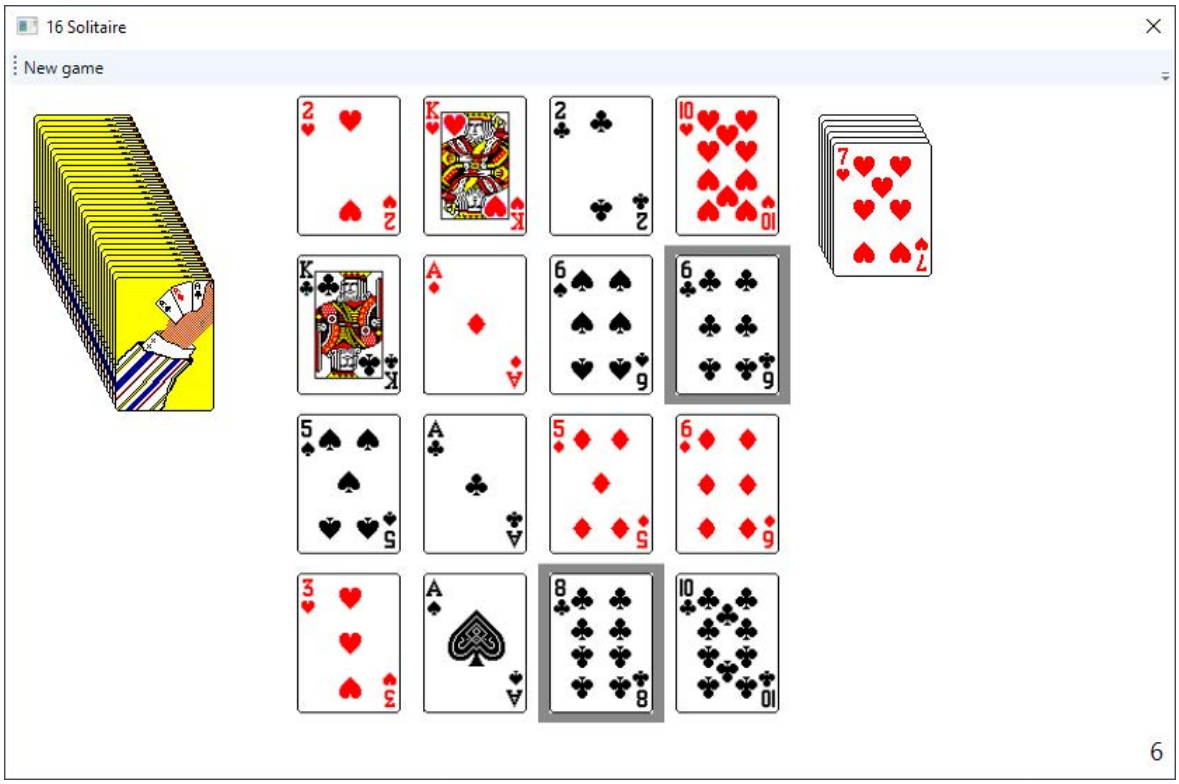
On the table are 16 cards. One must then remove the cards after two criteria:

1. cards of the same color with value less than 10, and where the sum of the card's values are 15 (thus as an example ace, five and nine of the same color, or seven and eight of the same color)
2. 10, Jack, Queen and King of the same color

When you have removed cards, the unused places must be filled with cards from the deck, if there are more cards. The solitaire is solved when all cards are removed, but when you get into a situation where you can not remove cards, it means that the solitaire can not be solved.

Below is a window which shows the program's user interface:

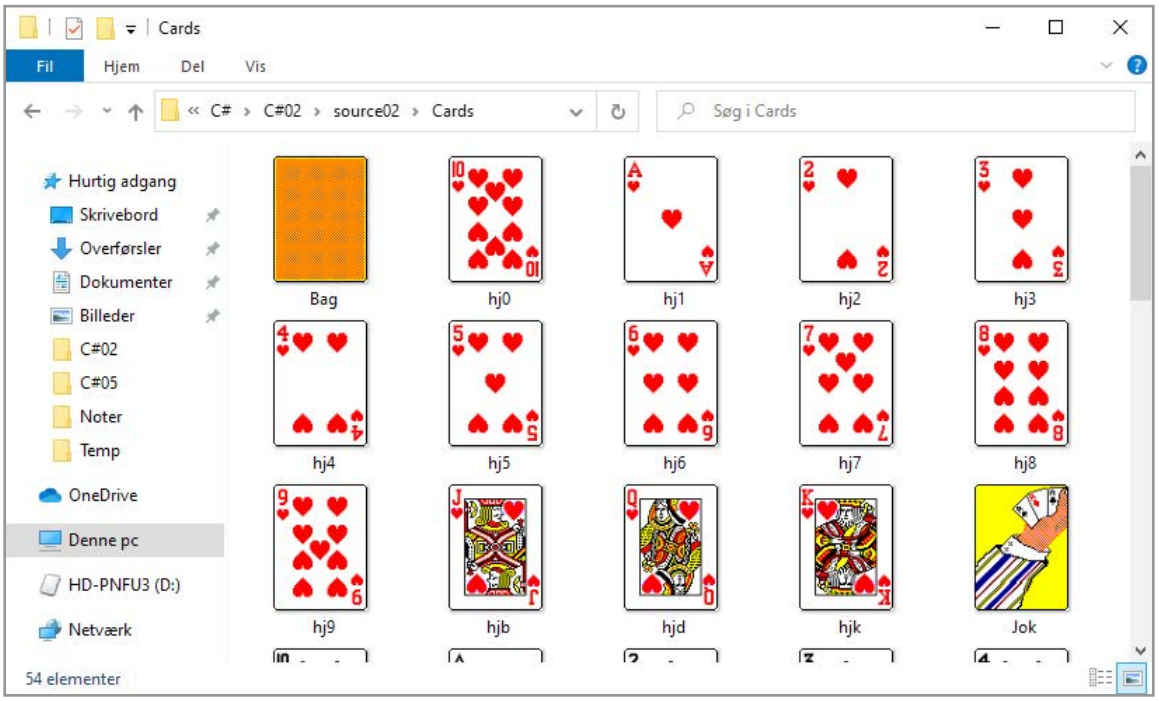
1. At the top is a toolbar with a button used to start a new game.
2. At the bottom is a status bar with a label which shows the number of cards that are moved.
3. The center contains three parts, where the left part shows the cards which has not yet been used and the right part shows the cards already moved from the table. The center represents the table, and when you click on a card it is marked with a gray border, and if you click again the border is removed. Clicking a card then select or deselect a card.



When you click cards, and the selected cards meets the rules of the solitaire, they are automatically moved to the right stack.

To solve the problem, follow these guidelines:

1. Create a new WPF project called *Solitaire*. The program must use the images for the cards. In the books source files is a folder which contains the cards, 52 cards and two background cards:



   In Solution Explorer right click and add (create) a new folder called *Cards*. Right click at your new folder and select *Add* | *Existing Items* and browse to the folder above with the cards. Select all cards, click *Add* to add the cards to the project. The result is that the cards (images) will be added as resources to the project and are then be part of the exe file.

2. Create the model. In principle, it is a very simple program in which the user interface has to display images of playing cards, which the user then should be able to click on with the mouse. The program has in principle no other functions in addition it must be a possibility to be able to select new game.

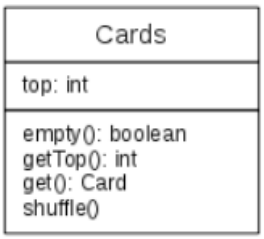
There must be kept track of which cards are on the table, which are left in the deck and which have been taken, and for this purpose the program should define a model that always represents the program's current state. The program must therefore in principle work in that way, that when the user clicks on a card, a message is sent to the model on which card is clicked, and the model should then from the cards that are marked, determine whether the cards can be removed from the table in according to the solitaires rules. If this is the case the model must remove the cards from the table, and selecting new cards from the deck.

The basic concept in the program is a playing card, which is a very simple concept. A playing card must have three characteristics:

1. a value that must be a number from 1 to 13: 1 = ace, 11 = Jack, 12 = Queen and 13 = king
2. a color which should be a character: D (diamonds), H (Hearts), S (spade) C (clubs)
3. an image of the card

Otherwise, a card is passive and can be defined as follows:

Besides a playing card it is also natural to think of a deck of 52 cards. It is not much more than an array of 52 *Card* objects and a variable that can keep track of how many cards are left in the deck:

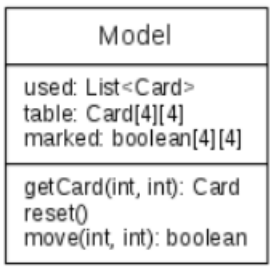


The first method returns while the deck is empty, and the next how many cards left in the deck. The third method returns and remove the card that is on top of the deck and as so simulates that a dealer share a card. The last method shuffles the cards and should be used when selecting a new game.

The model must have an object of the type *Cards*, and thus a deck of cards. In order to control the game's state is needed three data structures

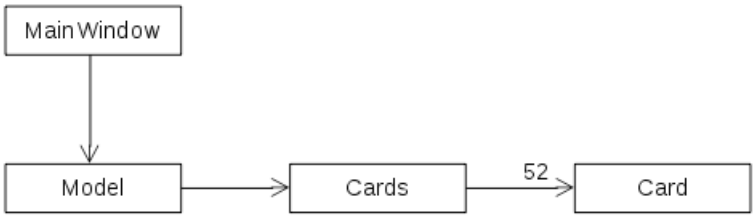
- a list of the cards that has been taken
- an array that keeps track of which cards are on the table
- an array which keeps track of which of the cards on the table is marked

and thus one can outline a model class as follows:



The class has only two essential methods, where *move()* is the most important and is used each time the user clicks on a card and then *reset()*, which is for selecting a new game.

The program's design will consist of the following classes:



Considerations such as the above regarding the program's model are in system development called for design and relate to which classes a program should consist of. You should note that the syntax does not conform to C#, nor is it a goal. The the meaning is that you in the design can work freely and independently of the many details of the programming language. Of course, a design like the above is not so much, but for the sake of the finished program and its architecture, it may be worthwhile to make these kinds of considerations before addressing the programming.

Now you must write the three model classes. In the file for that class you should add three other types. As mentioned above enumerations are not mentioned yet, but it is a type which represents a number of constants. Use of enumerations can make the code more readable, and for a playing card you can define (in the file *Cards.cs*) an enumeration to represents the card colors:

```
public enum CardColor
{
  Diamonds, Hearts, Spades, Clubs
}
```

Also define an *enum* which represents the card values:

```
public enum CardValue
{
  Ace = 1, Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten,
  Jack, Queen, King
}
```

Names on an enumerations are names for integers, and for the type *CardColor* the names means 0, 1, 2 and 3. In *CardValue* the first name is initialized as 1, and the result is that the names represents the integers 1, 2, 3, ..., 13.

With this two enumerations ready you can define a class *Card* (in the same file) which represents a playing card:

```
public class Card
{
  private CardColor color;
  private CardValue value;
  private BitmapImage image;
```

The class must have a constructor which with parameters for all three variables as well as *get* properties for the three variables.

You can then write the class *Cards* representing of deck of cards:

```
public class Cards
{
  private static Random rand = new Random();
  private Card[] cards = new Card[52];
  private int top = 0;
  private static BitmapImage back1;
  private static BitmapImage back2;

  static Cards()
  {
    back1 = new BitmapImage(new Uri("/Cards/Bag.GIF", UriKind.Relative));
    back2 = new BitmapImage(new Uri("/Cards/Jok.GIF", UriKind.Relative));
  }

  public Cards()
  {
    Load();
    Shuffle();
  }
```

The class has a static random generator used to shuffle the cards. There are two instance variables, where the first is an array for the 52 cards, while the other is used to show how many cards there is left in the deck. If the variable has the value 0 it means all cards are used. The class also has two static variables for background images. These variables must be initialized by the images for the cards and you can do that in a static constructor.

The class has a constructor that calls two methods. You must write this methods. The first should create the 52 *Card* objects. An example could be:

```
cards[0] = new Card(CardColor.Diamonds, CardValue.Ace,
  new BitmapImage(new Uri("/Cards/Ru1.GIF", UriKind.Relative)));
```

The other method is the method *Shuffle()* used to shuffle the cards. You can write this method as a loop that random swaps two cards a number of times.

You must the write the class *Model*:

```
namespace Solitaire
{
  public delegate void MoveEventHandler();

  public class Model
  {
    // define an event fired when a card is moved
    public event MoveEventHandler CardsMoved;

    // the cards
    private Cards cards = new Cards();

    // references to cards on the table
    private Card[,] center = new Card[4, 4];

    // references to cards moved the right deck
    private Card[] right = new Card[52];

    // which cards on the table that are selected
    private bool[,] marked = new bool[4, 4];

    // number of cards on the left deck
    private int leftCount = 36;

    // number of cards on the right deck
    private int rightCount = 0;

    public Model()
    {
      NewGame();
    }

    public int LeftCount
    {
      get { return leftCount; }
    }

    public int RightCount
    {
      get { return rightCount; }
    }
```

```
    public Card this[int n]
    {
      get { return right[n]; }
    }

    public Card this[int i, int j]
    {
      get { return center[i, j]; }
    }

    // Creates a new game that means shuffle the cards and set the mode
    to default
    public void NewGame()
    {
    }

    // Check if the solitaire is solved and all cards moved to the right
    deck
    public bool Ok()
    {
    }

    // Called when the user click on a card.
    // If card already is marked the method must remove the mark and
    return false.
    // Else the method must mark the card and test from the rules for
    the
    // solitaire where the marked cards must be moved to the right deck.
    // If not the method must return true. Else the method must update the model:
    //   move the marked cards to the right deck
    //   remove all marks
    //   move cards to the table as long the left deck is not empty
    //   fire a CardsMoved event
    public bool Mark(int i, int j)
    {
    }
  }
}
```

Note that the method *Mark()* is not quite simple.

3) Create the window and then the programs user interface. You can use a *DockPanel* with a toolbar at the top and a status bar at the bottom. Center you can add a *Grid* with three columns:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="200"/>
    <ColumnDefinition Width="360"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Grid Grid.Column="0" Name="leftPanel" />
  <UniformGrid Grid.Column="1" Rows="4" Columns="4" Name="centerPanel" />
  <Grid Grid.Column="2" Name="rightPanel" />
</Grid>
```

The left panel should be used for the unused cards, the right panel for the cards moved from the table and the panel at the center for the cards on the table. These three panels must be initialized in code. The left and right panel should only show images, but for the cards in the center panel it must be possible to select the cards with the mouse and it must be visible that a card is selected. One way to solve this problem is to add a user control as you should do in the next step.

4) Add a WPF User Control to the project, which you can call *CardControl*. A WPF user control is a control which can be used as all other controls to build the user interface. To write a user control you must write the XML as well as the C# code. I this example the XML should be:

```
<Grid>
  <Rectangle Name="rect"  />
  <Image Width="75" Height="100" Name="image"
    MouseLeftButtonDown="image_MouseLeftButtonDown" />
</Grid>
```

where the *Rectangle* element should be used to set a background color and else the component only consists of an *Image* control. Note that the *Image* control has an event handler for click with the mouse. You should write the C# code as follows:

```
public partial class CardControl : UserControl
{
  private static SolidColorBrush unselected = new SolidColorBrush(Colors.
  White);
  private static SolidColorBrush selected = new SolidColorBrush(Colors.
  Gray);
  private Model model;
  private int row;
  private int col;

  public CardControl(int row, int col, Model model)
  {
    InitializeComponent();
    this.model = model;
    this.row = row;
    this.col = col;
    image.Source = model[row, col].Image;
  }

  public void Refresh()
  {
    rect.Fill = unselected;
    if (model[row, col] == null) image.Source = Cards.Back1;
    else image.Source = model[row, col].Image;
  }

  private void image_MouseLeftButtonDown(object sender,
  MouseButtonEventArgs e)
  {
    if (image.Source != Cards.Back1) rect.Fill =
      model.Mark(row, col) ? selected : unselected;
  }
}
```

The class has a reference for the model and variables for the position on the table. The variables are initialized in the constructor. The method *Refresh()* is called when cards are moved from the table, and the method sets the current component to unselected and set the image to the new image or the background card. Also note the event handler which marks the image as selected or not by changing the color of the component *rect*. To change the color the class has two static *Brush* variables representing the colors.

5) You then have to write the C# code to *MainWindow*:

```
namespace Solitaire
{
  public partial class MainWindow : Window
  {
    // the model for MainWindow
    private Model model = new Model();

    // Image components for the left panel
    private Image[] left = new Image[36];

    // Image components for right panel
    private Image[] right = new Image[52];

    // CardControl components for center panel (the table)
    private CardControl[,] cards = new CardControl[4, 4];

    public MainWindow()
    {
      InitializeComponent();
      InitLeft();
      InitRight();
      InitCenter();

      // make this object listener for Cardsmoved events which are fired
      by the
      // model object when cards are moved from the table to the right
      deck
      model.CardsMoved += Cardsmoved;
    }

    private void InitLeft()
    {
      // initialize the left panel and add 36 background cards to the
      panel
      // for each card sets the following properties:
      //   Source
      //   HorizontalAlignment
      //   VerticalAlignment
      //   Width
      //   Height
      //   Margin
    }
```

```
    private void InitCenter()
    {
      initialize the center panel and add a CardContol at each location
    }

    private void InitRight()
    {
      // initialize the right panel with 52 empty Image components
      // for each component sets the following properties:
      //   HorizontalAlignment
      //   VerticalAlignment
      //   Width
      //   Height
      //   Margin
    }

    public void Cardsmoved()
    {
      // event handler for Cardsmoved events
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
      // event handler for the button in the toolbar
    }
  }
}
```

After that the solitaire should works and you should be able to try the solitaire.

7) The last thing. When you have solved the solitaire (which is actually not easy) the program must open a dialog box:

Remember that the model has e method to test if the solitaire is solved.

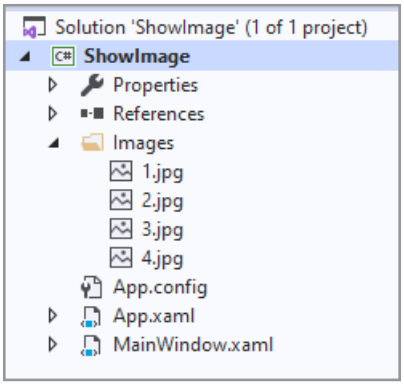
6 RESOURCES

Resources are files or other objects that an application can use without having to create them in the code, and one talks about respectively binary and logical resources, the last of which are objects intended for a particular application and are resources that can be written in XML. Binary resources are an option to compile binary files into the application's assembly, and above I have shown programs that uses playing cards, and the images of the cards was added to the program as binary resources. Finally, resources can be characterized as static or dynamic.

6.1 BINARY RESOURCES

Adding a binary resource to an application is easy, as you simply add the resource file to the project and set its *Build Action* property to *Resource*. When the program is compiled, the resource becomes embedded in the program's assembly.

As an example, I created a project called *ShowImage* and added a new folder named *Images*. To this folder I have added four images:



For each image I ensure that the image's *Build Action* is set to *Resource* (Visual Studio likely do that). Then I adds the following element to *MainWindow*:

```
<Image Name="img" Source="Images/1.jpg" Stretch="Fill"></Image>
```

Note primarily how to refer to the image as the folder and the name of the image. If you then run the program, you will get a window which shows the first image:

When the program run it should each 5 seconds random select 1 of the 4 images and show this image. It can be done using a timer which ticks every 5 second, and then random selects one of the 4 images:

```
using System;
using System.Windows;
using System.Windows.Media.Imaging;

using System.Windows.Threading;

namespace ShowImage
{
  public partial class MainWindow : Window
  {
    private static Random rand = new Random();
    private DispatcherTimer timer = new DispatcherTimer();

    public MainWindow()
    {
      InitializeComponent();
      timer.Tick += new EventHandler(timer_Tick);
      timer.Interval = new TimeSpan(0, 0, 5);
      timer.Start();
    }

    private void timer_Tick(object sender, EventArgs e)
    {
      img.Source = new BitmapImage(
        new Uri("pack://application:,,,/Images/" + rand.Next(1, 5) +
        ".jpg"));
    }
  }
}
```

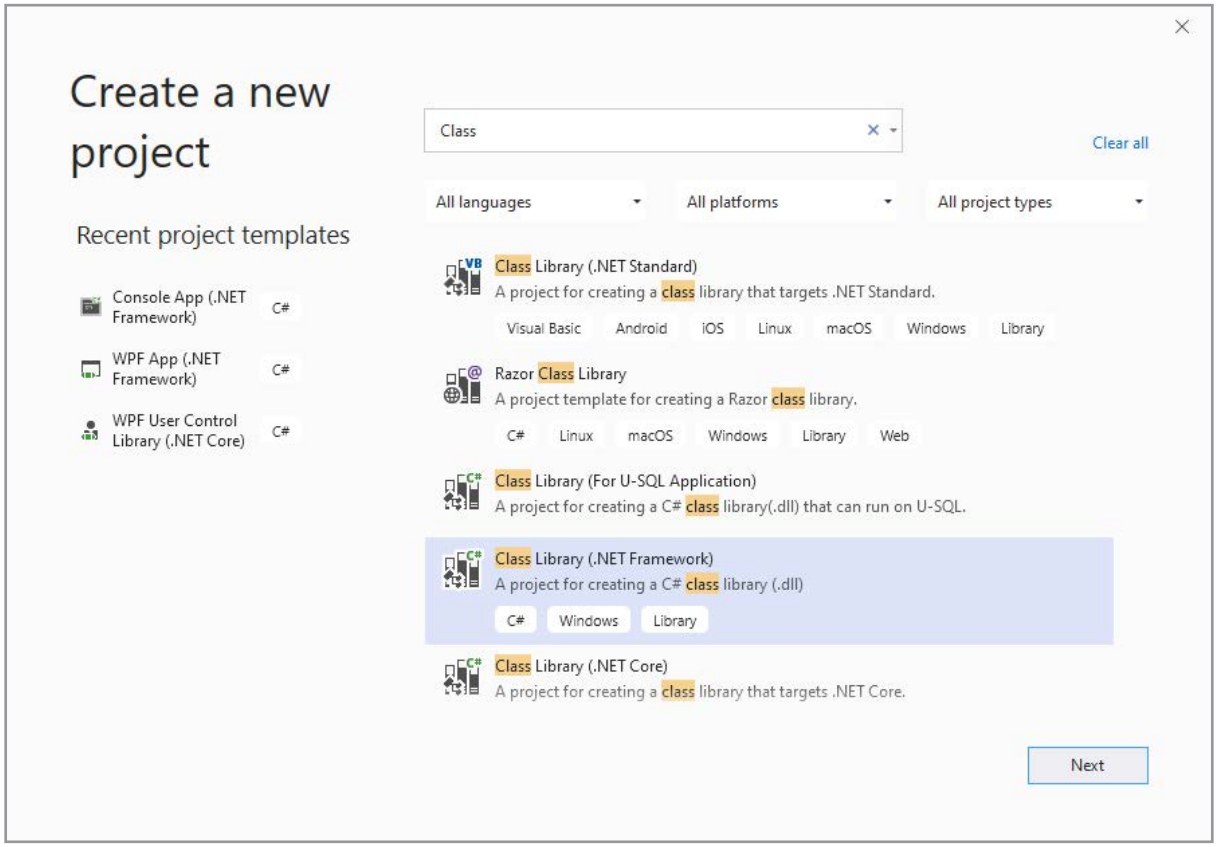
The class creates a random generator and a timer of the type *DispatcherTimer*. The type is explained later, but you should note that it requires an extra *using* statement. In the constructor is added an event handler to the timer, also called a timer function, and defined how often the timer must tick. As the last the timer is started.

The event handler selects a random images from the program's resources, and you must pay particular attention to the syntax to reference the resource.

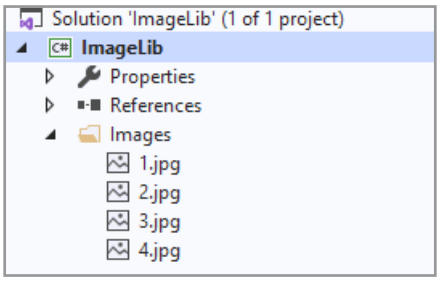
As an important comment on the above program one should examine the exe file. If you do, you will find that it is very large and the reason is that it now contains the 4 images.

6.2 RESOURCE LIBRARIES

Instead of embedding resources in the same assembly as the program code, resources can be embedded in another assembly, which can then be used from multiple examples. To show it, I will make another version of the program above. I start with a *Class Library ().NET Framework)* project:



which I have called *ImageLib*. It is a very simple library, consisting solely of a library of the four images:



It means that I have deleted the class that Visual Studio automatically creates. In addition, I manually set the *Build Action* property to *Resource* for all images. You should note that the result is an assembly (a *dll*) containing only 4 images, but no types.

Next, I created a new WPF project called *ShowImage1* where the XML code has an *Image* tag:

```
<Grid>
  <Image Name="img"
    Source="pack://application:,,,/ImageLib;component/Images/1.jpg"
    Stretch="Fill"></Image>
</Grid>
```

One should then refer to the image in the assembly *ImageLib*, which happens with a *pack Uri*, but with a slightly different syntax. It requires a reference to the assembly with the images, but that is also the only thing. To set a reference to the assembly you right click on *References* in Solution Explorer and select *Add Reference* and here you can browse to the above library: *ImageLib.dll*.

The C# code should be exactly the same, except that you need to change the reference to the image:

```
private void timer_Tick(object sender, EventArgs e)
{
  images.Source = new BitmapImage(new Uri(
  "pack://application:,,,/ImageLib;component/Images/" + rand.Next(1, 5)
  + ".jpg"));
}
```

6.3 CONTENT FILES

Of course, it is not necessary or always desirable to embed resources as images in the code, as they cannot then be changed without a recompilation. Instead, you can refer directly to the images files, and to avoid using absolute path names you can place the files in the program's folder, and one then talks about Content files.

With Visual Studio, it's easy. Similar to the above, I created in the *Images* folder and added the four images to it. For each image, set two of its properties:

Build Action: Content

Copy to Output Directory: Copy always

When you then build the program, a folder named *Images* is automatically created in the program folder and the four images are copied to it. Back there is just to see how to reference the image. In XML this happens as follows:

```
<Image Name="img" Source="pack://siteOfOrigin:,,,/Images/1.jpg"
  Stretch="Fill"></Image>
```

thus as a *pack Uri*, and the important thing here is *siteOfOrigin*, which indicates that references is made to the program folder. In the code, the reference is as follows:

```
img.Source = new BitmapImage(
  new Uri("pack://siteOfOrigin:,,,/Images/" + rand.Next(1, 5) + ".jpg"));
```

The project is called *ShowImage2*.

6.4 OTHER RESOURCES

In the above examples, resources have been images, but a resource can be anything, and in particular a resource can be a file that cannot be represented by a *pack Uri*. Instead, you can treat the resource as a file.

The next example shows a resource that is a text file. The resource is added to the program in the same way as other resources, and the only thing to be aware of is to set the *Build Action* property to *Resource* (if Visual Studio has not already done) so that the file is embedded in the program's assembly. The text file is called *Kings.txt* and contains lines with names of Danish Kings. If you run the program, the result is a list box with names for Danish Kings, and there is not much to say about the user interface, as it consists solely of a *ListBox*:

The code consists of a single method that extracts lines from the resource and adds them to the list box:

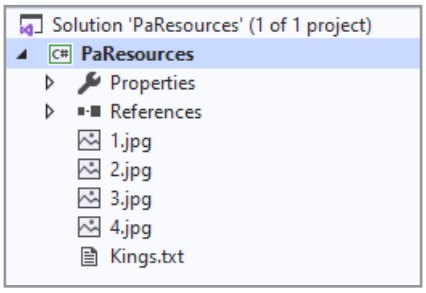
```
private void FillList()
{
  StreamResourceInfo info =
    Application.GetResourceStream(new Uri("Konger.txt", UriKind.
    Relative));
  StreamReader reader = new StreamReader(info.Stream, Encoding.Default);
  for (string line = reader.ReadLine(); line != null; line = reader.
  ReadLine())
  {
    int p = line.IndexOf(',');
    if (p > 0) lstKonger.Items.Add(line.Substring(0, p));
  }
}
```

Here you should note how the type *StreamResourceInfo* refers to an embedded resource, and since the resource in this case is a text file, the file is treated as a regular file with a *Reader*.

6.5 LOADING RESOURCES MANUALLY

It is possible to manually load resources into the code. It may not be that often this strategy is needed and it requires a little more code.

I'll start by creating a dll with the five resources I've used above. That is the four images and the text file. I have created a class library project named *PaResources* and deleted the class that Visual Studio creates. Next, I have added the 5 resources:



For each resource file the property *Build Action* must be set to *Embedded Resource.*

After that, the project can be translated and an assembly is created which contains only 5 resources. The example should show that a resource library can contain anything and in this case both images and text.

I will then write a program where the user interface has an *Image* and a *ListBox*:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition Width="150"/>
  </Grid.ColumnDefinitions>
  <Image Name="billede" Grid.Column="0" Stretch="UniformToFill" />
  <ListBox Name="lstKonger" Grid.Column="1" />
</Grid>
```

and if you run the program (*ShowResources*) it opens the window:

where the image shifts every 5 seconds.

The five resources must then be loaded from their assembly, which I will assume is called:

C:\Libs\PaResources.dll

and the code is:

```
using System;
using System.Text;
using System.Windows;
using System.Windows.Media.Imaging;

using System.IO;
using System.Reflection;
using System.Windows.Threading;

namespace ShowResources
{
  public partial class MainWindow : Window
  {
    private BitmapFrame[] bitmap = new BitmapFrame[4];
    private static Random rand = new Random();
    private DispatcherTimer timer = new DispatcherTimer();

    public MainWindow()
    {
      InitializeComponent();
      LoadResources();
      timer.Tick += new EventHandler(timer_Tick);
      timer.Interval = new TimeSpan(0, 0, 5);
      timer.Start();
    }

    private void LoadResources()
    {
      AssemblyName name =
        AssemblyName.GetAssemblyName("C:\\Libs\\PaResources.dll");
      Assembly asm = Assembly.Load(name);
      string[] names = asm.GetManifestResourceNames();
      int n = 0;
      for (int i = 0; i < names.Length; ++i)
      {
        int p = names[i].LastIndexOf('.');
        if (p > 0 && n < bitmap.Length &&
          names[i].Substring(p + 1).ToLower().Equals("jpg"))
        {
          Stream stream = asm.GetManifestResourceStream(names[i]);
```

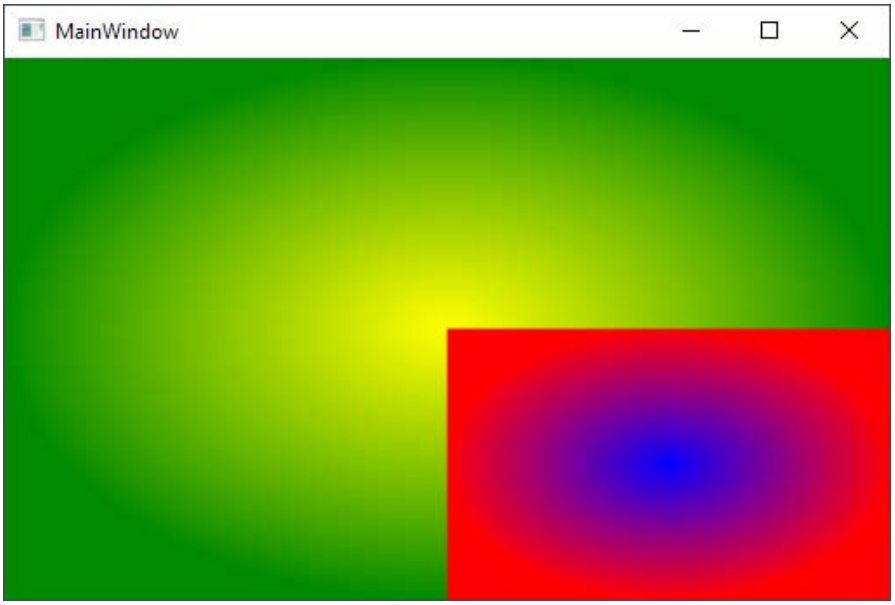
```
          bitmap[n++] = BitmapFrame.Create(stream, BitmapCreateOptions.
          None,
            BitmapCacheOption.OnLoad);
        }
        else if (p > 0 && names[i].Substring(p + 1).ToLower().Equals("txt"))
        {
          Stream stream = asm.GetManifestResourceStream(names[i]);
          StreamReader reader = new StreamReader(stream, Encoding.Default);
          for (string line = reader.ReadLine(); line != null;
            line = reader.ReadLine())
          {
            int q = line.IndexOf(',');
            if (q > 0) lstKings.Items.Add(line.Substring(0, q));
          }
        }
      }
      img.Source = bitmap[0];
    }

    private void timer_Tick(object sender, EventArgs e)
    {
      img.Source = bitmap[rand.Next(bitmap.Length)];
    }
  }
}
```

The program basically works like the previous examples in this chapter. You should note which using statements are used. The most important is the method *LoadResources()*, which loads the five resources from the dll's assembly. The first line defines an object that represents the assembly. This object contains most relevant data regarding that assembly. Note that the static method *GetAssemblyName()* as a parameter has the full name of the assembly. The next line loads that assembly into memory. Finally, the third line retrieves the names of all assembly resources.

The following for loop uses the names to extract the resources from the assembly as a byte stream. In this case, the extension of the name is used to determine the content of the resource in question, and thus whether it is an image or a text file, and the loop is not particularly flexible, and is based on a knowledge that the assembly contains images and one text file. If it is the text file, the list box is initialized as in the previous example.

6.6 LOGICAL RESOURCES

It is possible to define objects that can be used directly in XML, and it will typically be objects to build the user interface such as for example a *Brush*, but all objects can in principle be defined as a logical resource. In this way you can define objects that can be used by several elements in the user interface, and you can obtain a more readable XML. A WPF element defines a resource collection that contains resources defined for that element, and thus multiple resources can be defined for a particular element. As an example the program *ShowBrushes* opens the window:



The background color for the entire window is defined by a logical resource, and the same applies to the lower right rectangle. All defined in XML:

```
<Window x:Class="ShowBrushes.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
  <Window.Resources>
    <RadialGradientBrush x:Key="MainBrush">
      <GradientStop Color="Yellow" Offset="0" />
      <GradientStop Color="Green" Offset="1" />
    </RadialGradientBrush>
  </Window.Resources>
  <Grid Background="{StaticResource MainBrush}">
    <Grid.Resources>
      <RadialGradientBrush x:Key="PaBrush">
        <GradientStop Color="Blue" Offset="0" />
        <GradientStop Color="Red" Offset="1" />
      </RadialGradientBrush>
    </Grid.Resources>
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Rectangle Grid.Row="1" Grid.Column="1" Fill="{StaticResource PaBrush}"/>
  </Grid>
</Window>
```

The window itself is a WPF element and therefore resources can be defined for the window. Such a resource will be available to all other window elements. In order to reference a resource, it must have a name, and in this case a *Window* resource with the name *MainBrush* is defined. You should note that a resource is defined using XML alone. In this case, it's a *Brush* and exactly a *RadialGradientBrush*. This is explained later, but it is a brush which in this case mixes two colors.

A resource is used as a static resource as follows:

```
<Grid Background="{StaticResource MainBrush}">
```

In particular, note that a resource must be defined before it is used.

The program also defines a resource for the *Grid* object. It is called *PaBrush* and it is used by the rectangle object as a static resource.

As shown above, a resource is defined by a name, key and it does not have to be unique within the entire application, but it must be unique within the same resource collection. That is both of the above resources could well be called the same. When referring to a resource with a given name in a WPF element, the first one you encounter is used in the visual tree for that element: First is searched in the element's resource collection for a key that match. If it is not found, the resource collection for the element's parent is searched, and this is continued until a resource with the correct name is found.

The above is a static resource, but it is also possible to define a dynamic resource, for example

```
<Grid Background="{DynamicResource MainBrush}">
```

Immediately one will not experience the big difference, but it has something to do with how to manipulate the resource in code. I will return to that later, but in the vast majority of cases static resources are used for performance reasons.

6.7 A RESOURCE DICTIONARY

In the above example, the resources are defined in the XML code and as such can only be used within the current window. However, you can create a so-called *Resource Dictionary*, which is a collection of resources in a special XML file, and the resources in question can then be used anywhere within the same solution. The project *ShowResources1* is an example that uses a resource dictionary.

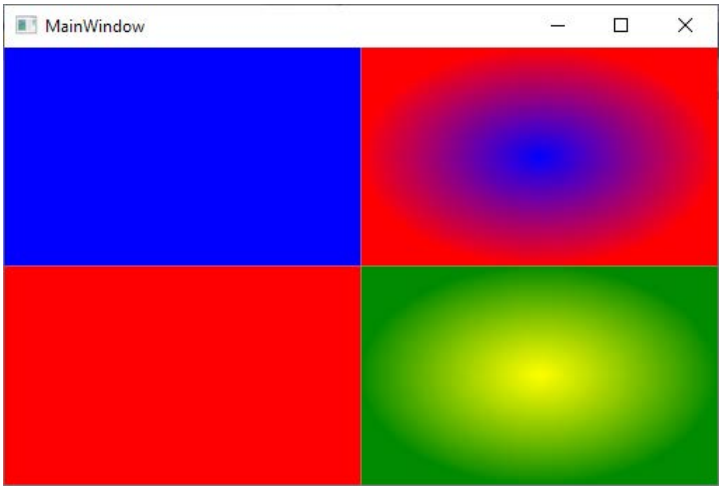
You create a Resource Dictionary from the Project menu in Visual Studio, giving it a name. In this example, I've called it *PaResources*. The result is a file named *PaResources.xaml*, and after the file is created, you can add resources:

```
<ResourceDictionary
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <SolidColorBrush x:Key="RedBrush" Color="Red"/>
  <RadialGradientBrush x:Key="RedBlueBrush">
    <GradientStop Color="Blue" Offset="0" />
    <GradientStop Color="Red" Offset="1" />
  </RadialGradientBrush>
</ResourceDictionary>
```

In this case I have defined two resources which are called respectively *RedBrush* and *RedBlueBrush*. In order for these resources to be used in a program, the dictionary must be merged into a resource collection, for example *Window.Resources* where they are merged with possibly other resources. The result could for example be the following where the above dictionary is merged with two other resources:

```
<Window.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="PaResources.xaml" />
    </ResourceDictionary.MergedDictionaries>
    <SolidColorBrush x:Key="BlueBrush" Color="Blue"/>
    <RadialGradientBrush x:Key="GreenYellowBrush">
      <GradientStop Color="Yellow" Offset="0" />
      <GradientStop Color="Green" Offset="1" />
    </RadialGradientBrush>
  </ResourceDictionary>
</Window.Resources>
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Grid Grid.Row="0" Grid.Column="0"
    Background="{StaticResource BlueBrush}"></Grid>
  <Grid Grid.Row="0" Grid.Column="1"
    Background="{StaticResource RedBlueBrush}"></Grid>
  <Grid Grid.Row="1" Grid.Column="0" Background="{StaticResource
  RedBrush}"></Grid>
  <Grid Grid.Row="1" Grid.Column="1"
    Background="{StaticResource GreenYellowBrush}"></Grid>
</Grid>
```

The program thus has four resources available. If the program is run, you will get the following window:



In practice, resources are defined in three places. The most common is in the *Window.Resources* collection as resources that can only be used for that window. As another option, resources can be defined in a *Resource Dictionary* that allows them to be used within the same solution. Finally, in *App.xaml*, you can define resources in *Application.Resources*, which define resources that can be used throughout the application. The next example *ShowResources2* shows how to do it.

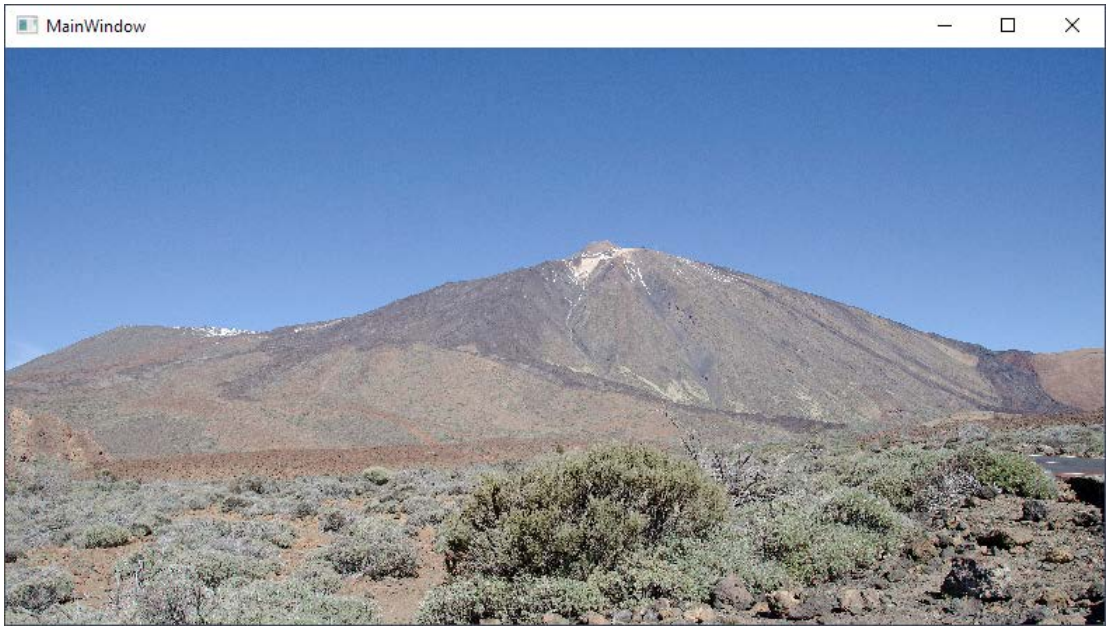
First, an image has been added to the project. By default, Visual Studio sets the Build Action to *Resource* so that the image is embedded in the application's assembly.

Next, I have defined a logical resource in *App.xaml*:

```
<Application x:Class="Exam0135.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
             presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
  <Application.Resources>
    <ImageBrush x:Key="MainBrush" ImageSource="Teide.jpg"
    Stretch="UniformToFill">
    </ImageBrush>
  </Application.Resources>
</Application>
```

This resource is then used in the program window, and the result is that the background of the window is filled in with that image:

```
<Grid Background="{StaticResource MainBrush}">
</Grid>
```



6.8 USING RESOURCES FROM CODE

It is possible to reference a logical resource in the code at runtime, and there are at least two ways to do that. The following program called *ShowResources3* sets the window background to blue, and then the program changes the background color to respectively yellow and red, changing color every 10 seconds.

The window defines a single resource associated with the *Grid* object as a static resource:

```
<Window.Resources>
  <SolidColorBrush x:Key="MainBrush" Color="Blue"/>
</Window.Resources>
<Grid Name="MainGrid" Background="{StaticResource MainBrush}">
</Grid>
```

The code is as follows:

```
public partial class MainWindow : Window
{
  private DispatcherTimer timer = new DispatcherTimer();
  private bool yellow = false;

  public MainWindow()
  {
    InitializeComponent();
    timer.Tick += new EventHandler(timer_Tick);
    timer.Interval = new TimeSpan(0, 0, 10);
    timer.Start();
  }

  private void timer_Tick(object sender, EventArgs e)
  {
    switch (yellow = !yellow)
    {
      case false: Red(); break;
      case true: Yellow(); break;
    }
  }

  private void Yellow()
  {
    SolidColorBrush brush = (SolidColorBrush)MainGrid.
    FindResource("MainBrush");
    brush.Color = Colors.Yellow;
  }

  private void Red()
  {
    SolidColorBrush brush = (SolidColorBrush)Resources["MainBrush"];
    brush.Color = Colors.Red;
  }
}
```

Here you should note the last two methods that change the color, but this is done by referring to the resource in two different ways. Also note that although the resource is associated with the WPF element *Grid* as a static resource, you can still change the color.

7 STYLES

In this chapter, I will show how to use styles that define how certain elements of the user interface should be displayed, and using styles can give controls a different visual appearance than they have by default. The idea for styles originates from the World Wide Web, where the concept is known as cascading styles. In fact, I have already used styles in the program *Lotto* to define that the columns in a *DataGrid* should be right aligned.

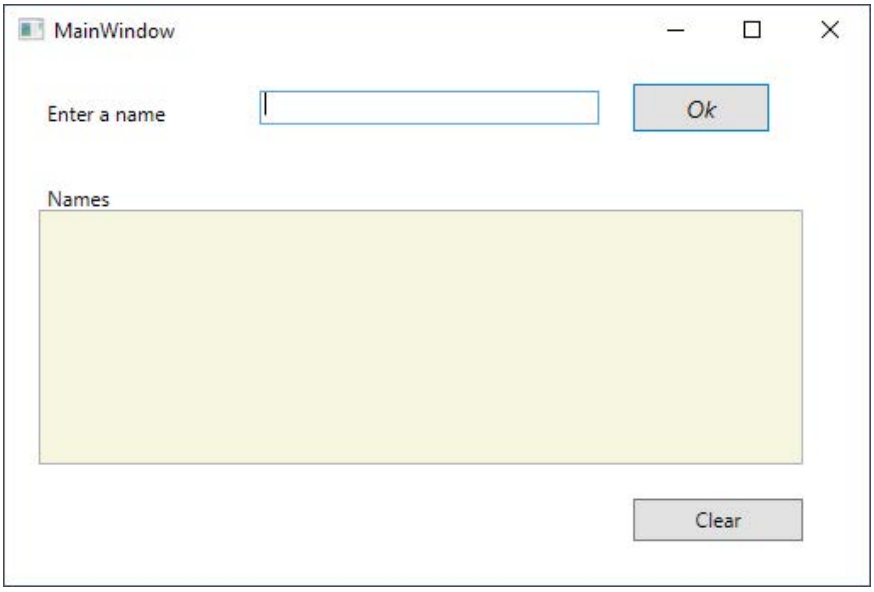
Styles in WPF is based on a class called *Style*, which assign styling data to properties. A style can be assigned to a single element, to all elements of a particular type, but also multiple types, and some of the most important properties of the class *Style* are:

1. *BasedOn*, which refers to another style on which this style is based and corresponds to that a style that can be inherited
1. *Resources*, which is a collection of local resources used by this style
2. *Setters*, which is a collection of *Setter* or *EventSetter* that defines the properties or events to which this style relates
3. *TargetType*, which defines which element type this style can be used for
4. *Triggers*, which is a collection of *Trigger* objects

Of these properties, *Setters* is the most important.

7.1 STYLES IN XML

The example *Styling* show how to define and apply styles. If you run the program you will get a window like below that contains 6 controls. If you enter a text in the top input field and click the *Ok* button, the text is inserted in the list box and clicking on the bottom button deletes the list box.



The program defines six styles. You can define styles in several places. In this case, a style is defined in *App.xaml*:

```
<Application.Resources>
  <Style x:Key="AppStyle" TargetType="Window">
    <Setter Property="FontFamily" Value="Verdana" />
    <Setter Property="FontSize" Value="12" />
    <Setter Property="FontWeight" Value="Bold" />
  </Style>
</Application.Resources>
```

This style is then available to the entire program. It is defined as a resource and is named *AppStyle*. It associates values to three properties, all of which relate to the font used. Each property is defined by a *Setter*. In this case, *TargetType* defines a style for a *Window* object.

The XML code for the window is as follows:

```
<Window x:Class="Styling.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525"
        Style="{StaticResource AppStyle}" Activated="Window_Activated">
  <Window.Resources>
    <Style x:Key="MainStyle" TargetType="FrameworkElement">
      <Setter Property="HorizontalAlignment" Value="Left" />
      <Setter Property="VerticalAlignment" Value="Top" />
    </Style>
    <Style x:Key="TextStyle" BasedOn="{StaticResource MainStyle}"
      TargetType="TextBox">
      <Setter Property="FontWeight" Value="Normal"/>
      <Setter Property="FontSize" Value="10"/>
    </Style>
    <Style x:Key="CmdStyle" BasedOn="{StaticResource MainStyle}"
      TargetType="Button">
      <Setter Property="Width" Value="100"/>
      <Setter Property="Height" Value="25"/>
      <EventSetter Event="Click" Handler="cmd_Click" />
    </Style>
  </Window.Resources>
  <Grid Style="{StaticResource MainStyle}">
    <Grid.Resources>
      <Style x:Key="ListStyle" BasedOn="{StaticResource MainStyle}"
        TargetType="ListBox">
        <Setter Property="FontWeight" Value="Normal"/>
        <Setter Property="Background" Value="Beige"/>
        <Setter Property="Foreground" Value="Gray"/>
      </Style>
    </Grid.Resources>
    <Label Content="Indtast et navn" Margin="20,20,0,0" />
    <TextBox Name="txtNavn" Margin="150,20,0,0" Width="200" Height="20"
      Style="{StaticResource TextStyle}" />
    <Button Name="cmdOk" Content="Ok" Margin="370,16,0,0" IsDefault="True"
      Click="cmd_Click">
      <Button.Style>
        <Style BasedOn="{StaticResource MainStyle}" TargetType="Button">
          <Setter Property="Button.FontSize" Value="14"/>
          <Setter Property="Button.FontStyle" Value="Italic" />
```

```
        <Setter Property="Width" Value="80" />
        <Setter Property="Height" Value="28" />
      </Style>
    </Button.Style>
  </Button>
  <Label Content="Navne" Margin="20,70,0,0" />
  <ListBox Name="lstNavne" Margin="20,90,0,0" Width="450" Height="150"
    Style="{StaticResource ListStyle}" />
  <Button Content="Slet" Style="{StaticResource CmdStyle}"
  Margin="370,260,0,0"/>
 </Grid>
</Window>
```

Here are some details here. First, note that the style from *App.xaml* is used by the *Window* definition and the syntax is the same as for a static resource:

```
Style="{StaticResource AppStyle}"
```

Next, three styles are defined as *Window* resources. They are thus available within the entire window. The first is called *MainStyle* and is defined for a *FrameworkElement* and can therefore be used by any control. It only defines how a control should be adjusted in the container. The next one is called *TextStyle* and is defined for a *TextBox* control, but it inherits *MainStyle*, and defines two properties regarding the font for the component. In particular, you should note that styles can inherit each other and the syntax used. The third style is called *CmdStyle* and also inherits *MainStyle*, and it defines two properties for a button, but it also defines an *EventSetter*, which I will return to in a bit.

MainStyle is used in the *Grid* for the window:

```
<Grid Style="{StaticResource MainStyle}">
```

and this means that all the components in the container have this style and thus are adjusted to the upper left corner.

The above three styles are defined as *Window* resources, and it is a very common place to define styles and thus styles that can be used for all elements in that window. However, styles can also be defined for a particular element. As an example, the style *ListStyle* for a *ListBox* is defined above and it is defined as a resource for the *Grid* container. In this case, there is no particular reason for defining the style object at this location beyond showing that it is possible. The style in question is used like other styles.

Finally, there is the button *cmdOk* that defines its own style as part of the defining XML. There are rarely justifications for defining inner styles in that way, and in this case one could simply initialize the properties directly, but if a property itself is a composite object such as a gradient brush, it may be helpful to know the syntax.

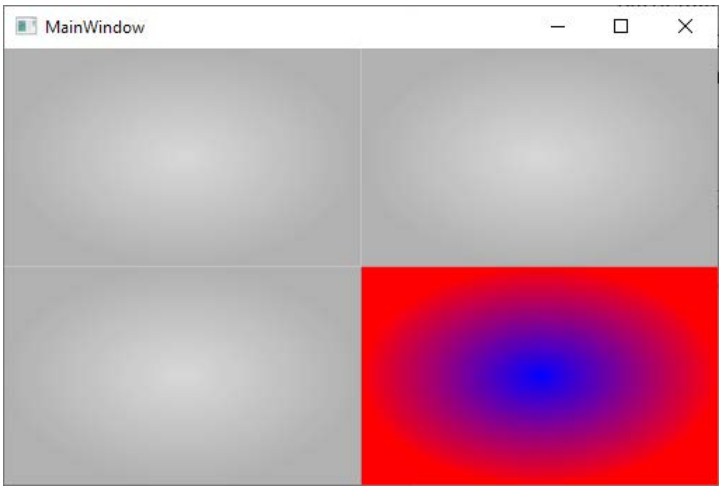
Sometimes it can be difficult to see which style applies to the properties of a particular element, but generally it is the style that is defined at the inner level that applies, and if there is none, it is the default value. For example, if looking at the two *Label* controls, no style is specified for them, but you can see that they get the style defined in the *App.xaml* attached to the window. All window controls will automatically have that style unless otherwise defined. The two components are also aligned to the upper-left corner of the container, similar to the *MainStyle* applies to a *FramewordElement* and thus specifically a *Label*.

The *TextBox* component has a *TextStyle*, and here you should especially notice that the properties defined in *TextStyle* override the properties defined in *AppStyle*.

Back there is the event handling. As usual, an event handler is associated with the *Ok* button, but the same event handler is associated with the other button. It happens with an *EventStyle*. There is no particular reason for doing so in addition to show the syntax, and it is an option that is rarely used, but if a window contains many buttons, which everyone should call the same event handler, it can be useful to associate the event handler with a style.

7.2 STYLES IN C#

The next example is called *Styling1*. If you run the program it opens the window:



In XML is defined a style for a brush:

```
<Window.Resources>
  <Style TargetType="Rectangle">
    <Setter Property="Fill">
      <Setter.Value>
        <RadialGradientBrush>
          <GradientStop Color="LightGray" Offset="0"/>
          <GradientStop Color="DarkGray" Offset="1"/>
        </RadialGradientBrush>
      </Setter.Value>
    </Setter>
  </Style>
</Window.Resources>
```

Here you should note that it has no name and that it is a style defined for a *Rectangle*. There is a single *Setter* for a *Fill* property and you should note how to define its value.

The window is divided into 4 areas and in each area there is a *Rectangle*:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Rectangle Grid.Row="0" Grid.Column="0" />
  <Rectangle Grid.Row="0" Grid.Column="1" />
  <Rectangle Grid.Row="1" Grid.Column="0" />
  <Rectangle Name="rect" Grid.Row="1" Grid.Column="1" />
</Grid>
```

Note in particular that the four *Rectangle* objects automatically get the above style, except the last one. That's because it gets its style defined in the code just to show how to do it:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    rect.Style = Create();
  }

  private Style Create()
  {
    Style style = new Style();
    Setter setter = new Setter();
    setter.Property = Rectangle.FillProperty;
    setter.Value = new RadialGradientBrush(Colors.Blue, Colors.Red);
    style.Setters.Add(setter);
    return style;
  }
}
```

8 FINAL EXAMPLE: A PUZZLE

In this chapter I will show the development of a simple GUI program. It's not a large program, but larger than the examples that have otherwise been presented to this place. The goal is to show the process used to write the program, but will also include concepts that will first be discussed in detail in the next two books.

Above I mentioned MVC as a pattern for architecture of a GUI program, and through the problems in chapter 6 small steps have been taken towards this architecture. The goal is to separate the code into multiple layers or modules to make it easier to view and maintain the program. The following example will use MVC more precisely, and exactly as a variant, which is called MVVM for *model-view-viewmodel*, and which better supports the architecture used by WPF.

The example will also describes the activities analysis, design and test, which are basic activities in system development and activities that is part of every software development project.

8.1 THE TASK

The task is to write a program that simulates a simple game, which consists of a square of 5 x 5 pieces arranged in random order. One of the pieces is empty, and you can move a piece by moving a neighbor to the empty piece. In the case below (the square on the left) you can then move the pieces H, E, L and J (you can not move diagonally). The game is solved when the pieces are arranged as shown in the example to the right.

K	G	V	M	R
I	S	X	U	F
C	O	A	E	T
Q	Y	H		L
N	D	P	J	B

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	

The program should have a high score list where to save the three best results. A result consists of the number of moves, used to solve the square, and it is therefore to solve the square by moving as few pieces as possible.

The square is basically a 5 x 5 square, but it must be possible to configure the program so that you can also play with a 3 x 3 square, a 7 x 7 square and a 9 x 9 square.

8.2 ANALYSIS

In this case, the analysis is simple, since the task is simple, and since the formulation of the task extensively describes the task. There are only few issues to be clarified.

In the project description, the individual pieces are symbolized with letters. It's fine by a 5 x 5 square, but is not sufficient by respectively a 7 x 7 and 9 x 9 square. It is therefore decided to symbolize the pieces by integers from 1 onward.

With regard of the high score list it should be attached a high score list for each of the four levels of difficulty, as it does not make sense of comparing results from different levels of difficulty.

As the program's philosophy is entertainment and results can not be regarded as significant, it is agreed that the high score lists are saved as regular files in the user's home directory. This means that anybody can delete the high score lists.

The program's idea is comparable with other entertainment programs like solitaire programs, which also has associated a kind of high score list. It is therefore decided that the high score list should be developed general in order to be used in other applications. A high score will consist of

- a name (the user who has obtained the score)
- a score, that is an integer
- a time which indicates when the high score is achieved

Moreover, one must be able to specify an ordering that indicates where great values of the score is positive or negative. A high score list must have room for a certain number of high scores, as should be defined when the list is created.

The Requirements Specification

The program should basically have the following functions:

1. *Play* where the user has an interaction with the program and rearrange the pieces until the square is resolved, or the user provides.
2. *New game* where the user selects a new game. This means that the pieces are shuffled and the state of the program is set to start.

3. *Update high score list*, a feature which is triggered by the program when the square is resolved and the user's point indicates that the user should be on the list. The user must then enter an identification in the form of the name or another text.
4. *Show high score list*, where the user chooses to view the high score list.
5. *Choosing the difficulty level* where the user must select a square size. This will automatically result in a new game.

Regarding the high score list there are the following requirements:

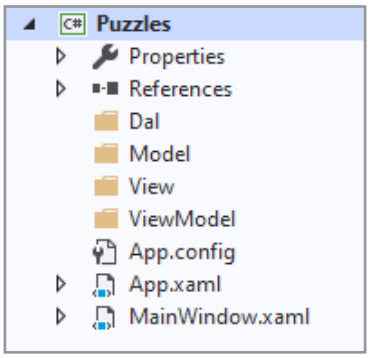
1. The high score list must show the top three and thus the players who have solved the square by moving as few pieces as possible.
2. Each high score must consists of a name, a date and time and the number of pieces that are moved.
3. It should not be possible (from the program) to delete the high score list.

8.3 A PROTOTYPE

When you must write a GUI program it is often appropriate to start with a project that to some extent establishes the program architecture and presents a prototype. The prototype can be a good tool for clarifying the user interface of the program with the future users and thus ensuring that the task is understood correctly. In this case I have created a project *Puzzles* and designed the following *MainWindow*:

The window has a menu with the program's functions, but the menu items have no function yet, and the same goes for the window buttons.

I have also created 4 folders which should be used for types for the different MVVM layers. Here you should note the folder *Dal* (data access layer) used for classes to save data to the hard disk.



8.4 ITERATION 1: SOLVE A SQUARE

When developing a program, depending on the size and complexity of the program, one often chooses to divide the development into a number of steps or iterations, so that after each iteration, the program is in a stable state where it can in principle be used and at least tested with the functions implemented until this point. In the first iteration I will implement the playing function, which means that the user can shuffle and move the pieces.

To move the pieces I will swap there content, and when the user click a button I need to know which button is click. For that I have defined a class called *Cell* in the *Model* layer. The class has only two properties representing a row index and a column index. When then buttons are created in *MainWindow* a *Cell* object representing the row and the column is assign to the *Tag* property, which can be tested when the button is clicked.

The class *Game* (also in the *Model* layer) represents the current game:

```
class Game
{
  private static Random rand = new Random();
  private static int[] SIZE = { 3, 5, 7, 9 };

  public event EventHandler PuzzlesSwapped;
  public event EventHandler PuzzlesSolved;

  private int size = 5;          // the size of this square
  private int count = 0;         // the number of moves int this game
  private int[,] state = null;   // the current values for the buttons
  private int row = 0;           // the row number for the blank piece
  private int col = 0;           // the column number for the blank piece

  public Game()
  {
    Resize();
  }

  ...

  // Test where the square is solved
  public bool Solved()
  {
  }

  // Move a button (int the model) if possible
  public void Move(Cell cell)
  {
  }

  private void Shuffle()
  {
  }

  private void Swap(int r, int c)
  {
    state[row, col] = state[r, c];
    row = r;
    col = c;
    state[row, col] = 0;
  }
}
```

The class represents the game and keep track of its state. The class will also include most of the program logic. The most important method is *Move()* that try to move a button. The method must test if the button assigned with the parameter *cell* is a neighbor to the blank piece, and if so perform a logical swap in the model of the two pieces. If it happens the user interface (*MainWindow*) needs to know to update the components, and for that the model fires an event to notify the user interface. The method must also test where the square is solved, and if so it fires another event to notify the user interface.

The algorithm to shuffle the pieces is not obvious. One can not simply shuffle the pieces randomly (using a random number generator) and if done, the square could not necessarily be solved. It is therefore necessary with a strategy (an algorithm) to shuffle the pieces. The idea is to start with a square with the pieces arranged, and then simulate random moves one of the pieces that can be moved (there are 2, 3 or 4 options). Repeats it many times, you get a square where the pieces that are shuffled.

Above I mentioned the MVVM pattern. If you look at a WPF window it consists of two part:

1. the XML which defines the user interface, the layout and the controls
2. the C# code (also called code behind) which contains the event handlers and other initialization code

If you not do anything there is a tendency for code behind to become comprehensive and complex and therefore according to the MVC pattern, significant parts of the program logic must be moved to a control layer. Later I will introduce new concepts regarding WPF which means a tight coupling between the model and the user interface and for those reasons, WPF recommends a variation of the pattern, where replacing the control layer with a View Model layer and thus a model for the user interface. In this program I will start using this pattern, although this is not quite obvious and seems a bit over killed. I have added a class *MainViewModel*:

```
namespace Puzzles.ViewModel
{
  class MainViewModel
  {
    private Game model = new Game();
    public event EventHandler PuzzlesRefreshHandler;
    public event EventHandler PuzzleSolveHandler;

    public MainViewModel()
    {
      model.PuzzlesSwapped += SwappedHandler;
      model.PuzzlesSolved += SolveHandler;
    }

    private void SolveHandler(object sender, EventArgs e)
    {
      if (PuzzleSolveHandler != null) PuzzleSolveHandler(this, e);
    }

    private void SwappedHandler(object sender, EventArgs e)
    {
      if (PuzzlesRefreshHandler != null) PuzzlesRefreshHandler(this, e);
    }

    public int Count
    {
      get { return model.Count; }
    }

    public int Size
    {
      get { return model.Size; }
    }

    public int this[int r, int c]
    {
      get { return model[r, c]; }
    }

    public void Button_Click(object sender, RoutedEventArgs e)
    {
      Cell cell = (Cell)((Button)sender).Tag;
```

```
      model.Move(cell);
    }

    public void Reset()
    {
      model.Reset();
    }
  }
}
```

You should note that it is the class that knows the *Model* layer (the class *Game*), and the class creates an instance of the model. The user interface and that is, the class *MainWindow* does not need to know the model layer, instead it needs to know the view-model layer. When the user interface wants something, for example because the user does something, the user interface in situations where it does not directly have to do with components in the user interface, it will delegate the task to the view-model layer which then performs the task. It is not the intension that the view-model layer should know the user interface and thus the class *MainWindow* and the communication from the view-model layer to the user interface happens either by a return value from a method or by the view-model layer firing an event that the user interface can then listen to. Similarly, it is for the model layer that does not know the view-model layer (nor does it know the view layer), and the communication from the model to the view-model layer occurs similarly as return values or the model fires an event.

All it is to separate the program's classes and can increase clarity, but there is a price. As can be seen from the above class, the view model layer can easily become a thin layer for the sole purpose of passing a method call to the model layer and sending a return value or firing an event to the view layer. Thus, it may be difficult to see the meaning of the class *MainViewModel*, but the reason is that I still lack some concepts regarding WPF (discussed in a later book), and until then I will mostly apply the MVVM pattern for the sake of principle than because there are visible benefits.

To conclude this iteration there is also the code behind for *MainWindow*:

```
public partial class MainWindow : Window
{
  private MainViewModel model = new MainViewModel();

  public MainWindow()
  {
    InitializeComponent();
    Create();
    model.PuzzlesRefreshHandler += RefreshHandler;
    model.PuzzleSolveHandler += SolveHandler;
  }

  private void Create()
  {
    panel.Children.Clear();
    panel.Rows = model.Size;
    panel.Columns = model.Size;
    for (int i = 0; i < model.Size; ++i)
      for (int j = 0; j < model.Size; ++j)
      {
        Button cmd = new Button();
        cmd.Tag = new Cell(i, j);
        cmd.Click += model.Button_Click;
        cmd.Content = model[i, j] == 0 ? "" : "" + model[i, j];
        cmd.FontSize = 24;
        panel.Children.Add(cmd);
      }
    Width = model.Size * 60 + 36;
    Height = model.Size * 60 + 78 + menu.Height + lblCount.Height;
    lblCount.Content = "" + model.Count;
  }

  private void New_Click(object sender, RoutedEventArgs e)
  {
    ...
  }

  private void Reset()
  {
    ...
  }
```

```
  private void SolveHandler(object sender, EventArgs e)
  {
    ...
  }

  private void RefreshHandler(object sender, EventArgs e)
  {
    ...
  }

  ...
}
```

First you should note that the class has an instance of the class *MainViewModel* and as so a reference to the view-model layer. Also note that the class is a listener for events from the view-model layer, events that is fired when the view-model layer is notified from the model layer that a piece is moved or the square is solved. The method *Create()* which creates the buttons is relative to the prototype updated and the buttons are initialized from values in the model (reference trough the view-model).

After this iteration 1 is finished, the user can start a new game (rearrange the pieces) and move them until the square is solved. When it is the case, the buttons are disabled to show the user, that the square is solved. Note that this means that the program is in a stable state and can be used in principle.

8.5 ITERATION 2: CHOOSE DIFFICULTY LEVEL

In the same way as in the previous iteration I have create a copy of the project called *Puzzles1*. It is then a version of the program as it was after completion of iteration 1.

In this iteration I will expand the program with the function to select the square size and then the difficulty level. It is a very short iteration as most of what is needed are done.

The model is change where

1. the set part of the property *Size* is deleted
2. the method *Resize()* if modified as it now is *public* and has a return value, and it has a parameter which is the new size
3. the method Initialize() is modified slightly
4. the constructor is changed

To the class *MainViewModel* is added a method *Resize()* which calls the corresponding method in the model.

The XML which defines the menu items is changed:

```
<MenuItem Header="Size" >
  <MenuItem Header="3 x 3" Tag="3" Click="Size_Click" />
  <MenuItem Header="5 x 5" Tag="5" Click="Size_Click" />
  <MenuItem Header="7 x 7" Tag="7" Click="Size_Click" />
  <MenuItem Header="9 x 9" Tag="9" Click="Size_Click" />
</MenuItem>
```

The event handler for the menu items must be implemented in code behind:

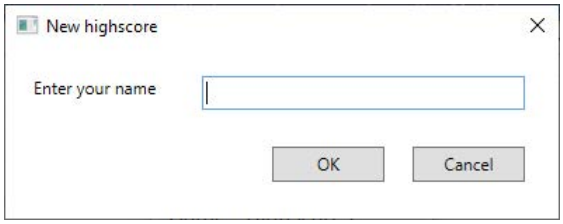
```
private void Size_Click(object sender, RoutedEventArgs e)
{
  int size = int.Parse(((MenuItem)sender).Tag.ToString());
  if (model.Resize(size)) Create();
}
```

That's all, and you can run the program again and test it for then four square sizes. Note that the program again is in a stable state and can be used to play the game.

8.6 ITERATION 3: THE HIGH SCORE LIST

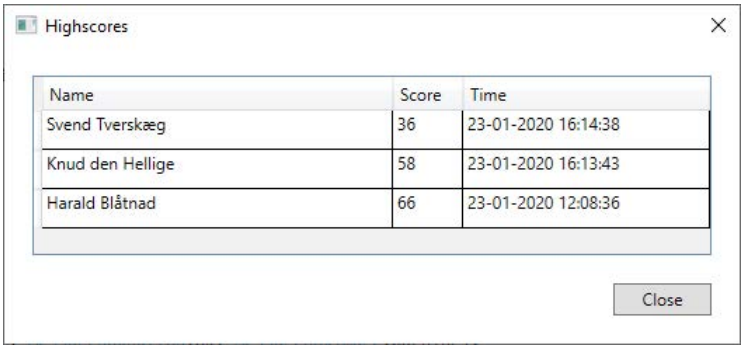
In this iteration I will implement the high score list, and it requires a little more. First I have added a class *HighScore* to the model layer. It is a simple class representing a high score. Then I have added a class *Repository* to the dal layer, which is a class that serialize and deserialize an array of *HighScore* objects to a file in the program directory for the program. The class also has methods which check where a high score must be added to the list and if so a method which adds a high score to the list. Every time you creates a new *Repository* object the high score list is deserialized, and every time you add a new high score to the list is serialized.

To add a high score to the high score list for the current square size is add e new Wpf window called *NewScoreWindow* to the view layer. It is a very simple window where the user must enter a name:



This window should only be opened if the users score means the user must be added to the list. You should note that the window also has it's own view-model as a class in the view-model layer, a class that is largely trivial.

In the same way is added a WPF window to show the high scores, and also here a class in the view-model layer:



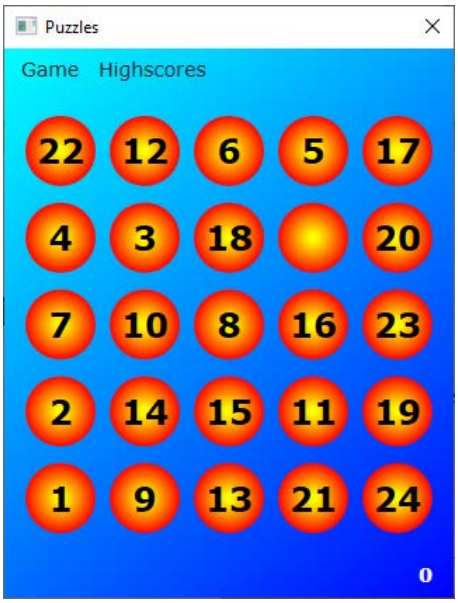
To show the high scores is used a *DataGrid* component, but this time I know the number of columns, and the table can be defined in XML:

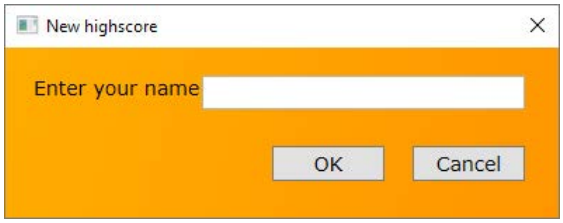
```
<DataGrid Name="table" AutoGenerateColumns="False" IsReadOnly="True"
  CanUserAddRows="False" RowHeight="28">
  <DataGrid.Columns>
    <DataGridTextColumn Header="Name" Width="250" Binding="{Binding
    Name}" />
    <DataGridTextColumn Header="Score" Width="50" Binding="{Binding
    Score}" />
    <DataGridTextColumn Header="Time" Width="*" Binding="{Binding Date}"
    />
  </DataGrid.Columns>
</DataGrid>
```

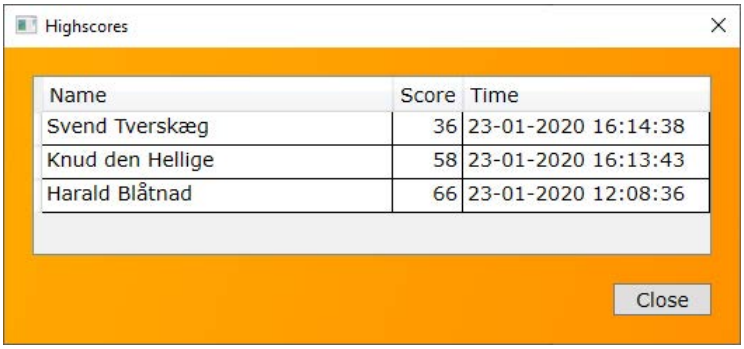
You may not know that the table should be created that way, but when you see the code it is easy enough to understand, close to the last *Binding* attribute in the column definitions. This means that the table should show objects which has properties named as the binding says, and it is the values of these properties that is shown in the table.

8.7 ITERATION 4: THE LAST TINGS

After the above iteration all functions are implemented and the program is ready for use, but to create a nicer user interface, you can try to associate styles with the individual components. For example, the result could be as shown below:







I will not show how, but you are encouraged to study the code and how the individual styles are written. You should note how the buttons have been changed using styles.

8.8 TEST

When a program is finished, it must be tested in real environments and of any other than the programmer. In this case there is not much to do in connection with the finally test, because the program does not have to communicate with other programs, but the program must of course be tested, and it must be in correct user environment.

In this case, you can create a folder somewhere on your hard drive and then copy the program's exe file to that folder and you can start the program by double click the program name. Then there is not much else to do than to play and try to solve the squares. Here you must pay particular attention to the high score list, and the players are correctly added to the lists. In addition, you must have an eye on the visual, and if all looks as it should do. Specifically, you can use the following test cases:

1. Set the level to a 3 x 3 square
2. Solve the square and add the user to the high score list
3. Solve the square again, but this time with many swap, such that the user get a small score
4. Solve the square a third time
5. Show the high score list and examine whether the three players are correctly inserted
6. Solve squares until another player is added to the high score list
7. Display the high score list to ensure that the last player is properly inserted
8. Repeat the above with a 5 x 5 square
9. Manually delete the two files to high score lists
10. Display the high score lists for both a 3 x 3 and 5 x 5 square and ensure that they are empty

If these test cases are performed satisfactorily, I will consider the program as tested and ready to use.