Poul Klausen

C# 10

More on WPF

Software Development

bookboon
LEARNING

POUL KLAUSEN

C# 10: MORE ON WPF

SOFTWARE DEVELOPMENT

C# 10: More on WPF: Software Development
1st edition
© 2021 Poul Klausen & bookboon.com
ISBN 978-87-403-3701-3

CONTENTS

FOREWORD

This book is the tenth in a series of books on software development. The programming language is C#, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process, and how to develop good and robust applications. This book is a continuation of the previous book on WPF. The book deals with issues that one typically encounters in connection with the development of Windows programs. Examples are implementing undo / redo and printing to a physical printer. In addition, as a continuation of the previous book, the book focuses on the development of custom components, but components that are larger and more complex than those shown in the previous book. The book, together with the previous book, is the end of the treatment of WPF, but also of the development of classic PC programs, and the following books will focus on WEB applications and phone apps.

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#. It can be learned 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 being treated in the books. All books in the series is 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 that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn 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 it is a program that you quickly become familiar 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

The book C# 2 is an introduction to writing programs with a graphical user interface using WPF and the previous books has shown many examples of applications with a graphical user interface. This book is a continuation as WPF includes many things that are not included in the previous books, and the book deals with the following topics, which there was no room for in the previous book

1. Undo / redo
2. Enter text
3. Print
4. A grid control
5. A class library

These topics do not directly relate to each other, but are topics that are necessary to know in order to develop GUI programs with a complex user interface, but to some extent the first four topics relate to data entry.

When you write programs where the user constantly has to manipulate data, most users today will demand an undo function where the user can undo an action and preferably several steps back. Which operations the user wants to be able to undo of course depends on the program, and an undo function can therefore not be a feature, which can generally work on all programs, and it is therefore left to the programmer to implement this functionality, and how to do it is the subject for the next chapter in this book.

To enter text in a WPF application is simple as a *TextBox* is all you need. At least as long as you do not need to enter formatted text as in a real word processor. However, WPF has a component for that purpose, which is called a *RichTextBox*. On the other hand, it is not quite simple to use, which is the topic of Chapter 3. The chapter ends with user control which acts as a simple editor for formatted text and thus in principle a very simple word processing program.

In the past, being able to print to a physical printer was an important component of almost any program. Even though this is still the case to some extent, being able to print a document does not play the same role as in the past, as we simply do not print as much on paper as before. Still, it is necessary to be able to expand a program with a print function and if you have to print complex data such as text documents or spreadsheet-like documents, it is actually not simple, especially because the result must be adapted to the current margins and page size. All that is the subject of chapter 4.

In many programs there is a need to present data in rows and columns and including also to be able to edit this data. For this purpose, WPF has a *DataGrid*, which is perfect as long as it concerns data from a data model for e.g. a database table. However, a *DataGrid* is not perfect in all cases. It is simply not flexible enough. There are several third party components that you can purchase and that provide all the features that you may want, but chapter 5 is an example of how to write your own as a custom component with some of the most important features.

2 UNDO / REDO

Most commercial applications support Undo / Redo, and in fact, most would expect that to be the case. WPF actually supports undo and redo to some extent, for example in connection with entering text, but if you think a little more about it and on more complex operations in a user program, it is clear that it cannot be a built-in operation in WPF. WPF cannot know what it means to undo an operation and what it is for an operation. At first glance, this may sound like a simple task, but it is not necessarily so and there are several factors to be aware of. In addition, it's worth the work to plan a strategy that can be used in several applications.

At a given time, a program has a state, and to implement undo, in connection with an operation that changes the state of the program, one must save the previous state, or at least such a large part of the state for the program that the state can be recreated. You are then able to perform an undo which, for the sake of redo, the saved state also must consists of saving the current state of the program. It is in principle quite simple to implement undo / redo in that way and only requires a stack for undo states and a stack for redo states.

As an example I will used the application *Puzzles* from the book C# 2 and build undo / redo facilities in that program. You should note that for practical use, it does not make much sense to build undo / redo in this program, and the program is chosen only because it is a simple program with simple operations. First I have to decide which operations it must be possible to undo and there is only two:

Move / click a piece
Select square size

You should note that it is decided that it should not be possible to undo a *New game* operation. When this operation is performed, the current undo operations should instead be cleared. Also it should not be possible to undo an operation, where a user is added to the high score list. You should note that these decisions regarding what it should be possible to undo belong under the analysis as a decision for how the program should work.

To implement the feature the project is expanded with a new model class:

```
public class UndoRedo<T>
{
  private Stack<T> undoStates = new Stack<T>();
  private Stack<T> redoStates = new Stack<T>();

  public void ToUndo(T state)
  {
    undoStates.Push(state);
  }

  public bool CanUndo
  {
    get { return undoStates.Count > 0; }
  }

  public bool CanRedo
  {
    get { return redoStates.Count > 0; }
  }

  public T Undo()
  {
    T state = undoStates.Pop();
    redoStates.Push(state);
    return state;
  }

  public T Redo()
  {
    T state = redoStates.Pop();
    undoStates.Push(state);
    return state;
  }

  public void Clear()
  {
    undoStates.Clear();
    redoStates.Clear();
  }
}
```

The class is quite simple and consists of two stacks, one for undo operations and one for redo operations. There is three important methods. The method *ToUndo()* save an operation for later undo, and the method *Undo()* move the operation to the redo stack and return the operation to the program. The method *Redo()* works in the same way, but move the operation from the redo stack back to the undo stack.

The class *UndoRedo* is generic and the parameter is the type of an operation. How to implement this type depends on the application and what operations there are. In this case there are two operations and often it is appropriate to implement these operations as a class hierarchy. The base class is

```
abstract class Operation
{
  public Cell BlankBefore { get; set; }
  public Cell BlankAfter { get; set; }
}
```

which has two properties that represents the position of the blank piece before the operation is performed and the position after the operation is performed. The class is defined *abstract* as it should not be possible to instantiate *Operation* objects. Then a move operation can be defined as:

```
class MoveOperation : Operation
{
  public int State { get; set; }
}
```

To undo and redo a move operation the program must know the positions from the base class as well as the value of the piece that are moved.

An operation for resize of the puzzle is more complicated as it is necessary to save the full state of the program both before and after the resize:

```
class ResizeOperation : Operation
{
  public int[,] StateBefore { get; set; }
  public int[,] StateAfter { get; set; }
  public int CountBefore { get; set; }
  public int CountAfter { get; set; }
}
```

To implement the feature the model class *Game* must be changed. The methods for move a piece and for resize the square must be modified:

```
public void Move(Cell cell)
{
  if (Solved()) return;
  if (CanMove(cell))
  {
    Cell cell1 = new Cell(row, col);
    int count1 = count;
    int value = state[cell.Row, cell.Col];
    int r1 = row;
    int c1 = col;
    Swap(cell.Row, cell.Col);
    ++count;
    UR.ToUndo(new MoveOperation {
      BlankBefore = cell1, BlankAfter = cell, State = value });
    ...
}

public bool Resize(int size)
{
  if (!SIZE.Contains(size) || size == this.size) return false;
  Cell cell1 = new Cell(row, col);
  int count1 = count;
  int[,] state1 = state;
  this.size = size;
  Initialize();
  UR.ToUndo(new ResizeOperation { BlankBefore = cell1,
    BlankAfter = new Cell(row, col), CountBefore = count1,
    CountAfter = count, StateBefore = state1, StateAfter = state });
  return true;
}
```

The the model must be expanded with two methods for undo and redo:

```
public void Undo()
{
  if (UR.CanUndo)
  {
    Operation operation = UR.Undo();
    if (operation is MoveOperation)
    {
      MoveOperation opr = (MoveOperation)operation;
      state[row, col] = opr.State;
      row = opr.BlankBefore.Row;
      col = opr.BlankBefore.Col;
      state[row, col] = 0;
      --count;
    }
    else
    {
      ResizeOperation opr = (ResizeOperation)operation;
      state = opr.StateBefore;
      size = state.GetLength(0);
      count = opr.CountBefore;
      row = opr.BlankBefore.Row;
      col = opr.BlankBefore.Col;
    }
    if (PuzzlesRefresh != null)
      PuzzlesRefresh(this, new PuzzleRefreshEventArgs(operation));
  }
}

public void Redo()
{
  if (UR.CanRedo)
  {
    Operation operation = UR.Redo();
    if (operation is MoveOperation)
    {
      MoveOperation opr = (MoveOperation)operation;
      state[row, col] = opr.State;
      row = opr.BlankAfter.Row;
      col = opr.BlankAfter.Col;
      state[row, col] = 0;
      ++count;
```

```
    }
    else
    {
      ResizeOperation opr = (ResizeOperation)operation;
      state = opr.StateAfter;
      size = state.GetLength(0);
      count = opr.CountAfter;
      row = opr.BlankAfter.Row;
      col = opr.BlankAfter.Col;
    }
    if (PuzzlesRefresh != null)
      PuzzlesRefresh(this, new PuzzleRefreshEventArgs(operation));
  }
}
```

Both methods fire an event as the user interface has to be notified after an undo and a redo.

As the last thing the user interface must be updated for a user gesture, and standard is Ctrl+Z (for undo) and Ctrl+Y (for redo).

In this case, it is relatively simple to implement Undo / Redo as there are only a few operations, but with a user dialog with many operations it can be more comprehensive. However, the above method can be used as a skeleton.

One must be aware that with a large user interaction, there can be many operations stored on the undo stack. It is of course fine, if it has to be possible to undo all operations, but conversely it naturally fills. Sometimes it may be appropriate to place restrictions on the stack and, for example, use a circular stack with a fixed size and where the oldest operations are automatically overwritten. In this case, the problem is not that big as the two stacks are cleared each time the user starts a new game.

The program *Puzzle1* is exactly the same program as the above with only one difference that Undo / Redo is implemented with a stack of fixed size and which in the case where the stack is full automatically overwrites the oldest element.

Such a stack could be implemented as:

```
public class Stack<T>
{
  private T[] buff;
  private int top = 0;
  private int count = 0;

  public Stack(int n)
  {
    buff = new T[n];
  }

  public int Count
  {
    get { return count; }
  }

  public bool IsEmpty
  {
    get { return count == 0; }
  }

  public T Peek()
  {
    if (IsEmpty) throw new Exception("The buffer is empty");
    return buff[Prev(top)];
  }

  public T Pop()
  {
    if (IsEmpty) throw new Exception("The stack is empty");
    --count;
    return buff[top = Prev(top)];
  }

  public void Push(T elem)
  {
    buff[top] = elem;
    top = Next(top);
    if (!IsFull) ++count;
  }
```

```
  public void Clear()
  {
    top = count = 0;
  }

  private int Next(int n)
  {
    return (n + 1) % buff.Length;
  }

  private int Prev(int n)
  {
    --n;
    return n < 0 ? buff.Length - 1 : n;
  }

  private bool IsFull
  {
    get { return count == buff.Length; }
  }
}
```

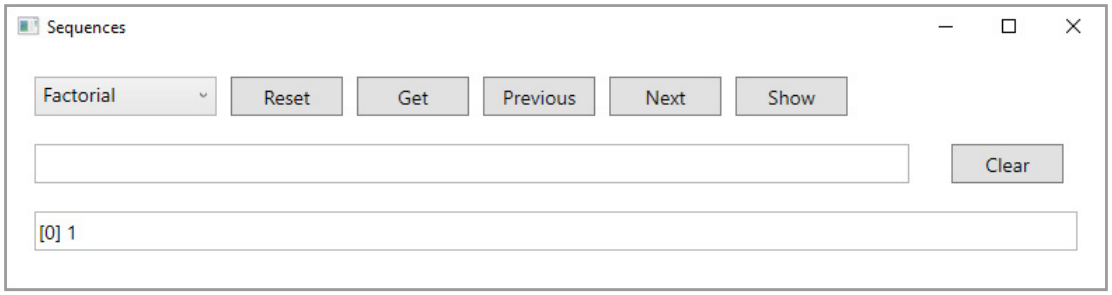
Then the class *UndoRebo* is changed to use the above stack class:

```
public class UndoRedo<T>
{
  private Stack<T> undoStates;
  private Stack<T> redoStates;

  public UndoRedo(int size)
  {
    undoStates = new Stack<T>(size);
    redoStates = new Stack<T>(size);
  }
```

and when the model (the class *Game*) create a *UndoRedo* object it defines that it should only be possible to undo 10 operations. That is all and the program can run again.

2.1 SEQUENCES

In the book C# 3 problem 1 I have shown a program to navigate number sequences. The user can select between four sequences: Factorial, Fibonacci, Power and Power2. The sequences are implemented as *BigIntegers*. Perhaps it makes sense to read the problem's text and recall how the finished program is written and works. In this section I will add Undo and Redo to the program, but the user interface is changed a bit:



The last field is an ordinary *TextBox*. The field was before a multi-line *TextBox*. It means that the event handlers also must be modified. The output field (the last *TextBox*) now always shows the result of the last operation. Also I have removed the button to clear the output field as it is no longer needed.

Free eBook on
Learning & Development
By the Chief Learning Officer of McKinsey
Download Now
bookboon
Prof. Dr. Nick H.M. van Dam
21st Century Corporate Learning & Development
Trends and Best Practices

To add Undo and Redo there are five operation which are the operations for the first four buttons as when the user select a new sequence. The state to store is this time the state of a sequence, and it can be done simple by each time store the current sequence. There is one problem as the current sequence is an object which change state when the user navigates the sequence. It is then necessary to store a copy of the sequence. To do that I have expanded the interface *ISequence* with a new method:

```
ISequence Clone();
```

defined the method *abstract* in *Sequence*, and implemented the method in the classes *Factorial*, *Fibonacci* and *Power*.

After these preparations, I am ready to add Undo and Redo to the program. The feature should be implemented using a circular stack as in the previous example, and I have added the class *Stack* from this example. I have also added the class *UndoRedo*, but modified a bit:

```
namespace Sequences
{
  public class UndoRedo<T>
  {
    ...

    public T Undo(T state)
    {
      redoStates.Push(state);
      return undoStates.Pop();
    }

    public T Redo(T state)
    {
      undoStates.Push(state);
      return redoStates.Pop();
    }

    ...
  }
}
```

The two methods *Undo()* and *Redo()* are changed such they have the current state of the program as parameter. *Undo()* place this state on the redo stack and pop the undo stack and return the value. *Redo()* works the same way, but push on the undo stack and pop the redo stack. Then it is simple to implement the functionality where the event handlers for the four buttons as well as the event handler for *SelectionChanged* for the combo box must call the method

```
UR.ToUndo(seq.Clone());
```

to store the current state. The code behind must also have methods for undo and redo and as an example:

```
public void Undo()
{
  if (UR.CanUndo)
  {
    seq = UR.Undo(seq);
    txtRes.Text = seq.ToString();
    SelectSeq();
  }
}
```

The method is quite simple, but as the last statement it calls *SelectSeq()*, which is a method used to reset the combo box when the undo is a state for selecting a sequence. It result in a WPF problem, when the *SelectionChanged* for the combo box can occurs in two situations: When the user select a new sequence in the combo box and when the selected item is changed in code (and also when the program starts). One way to solve this problem is to implements handlers for *DropDownOpened* and *DropDownClosed*:

```
private void lstSeq_DropDownClosed(object sender, EventArgs e)
{
  userInteraction = false;
}

private void lstSeq_DropDownOpened(object sender, EventArgs e)
{
  userInteraction = true;
}

private void lstSeq_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
  if (userInteraction)
  {
    UR.ToUndo(seq.Clone());
    switch (((ComboBoxItem)((ComboBox)sender).SelectedItem).Tag.
ToString())
    {
      case "1":
        seq = new Factorial();
        break;
      case "2":
        InputBox dlg = new InputBox();
        dlg.ShowDialog();
        if (dlg.DialogResult == true) seq = new Power(dlg.Value);
        break;
      case "3":
        seq = new Power2();
        break;
      case "4":
        seq = new Fibonacci();
        break;
    }
    seq.Reset();
    txtRes.Text = seq.ToString();
  }
}
```

Here *userIntercation* is an instance variable and the variable can for the event *SelectionChanged* be used to test if the event is fired for a user action.

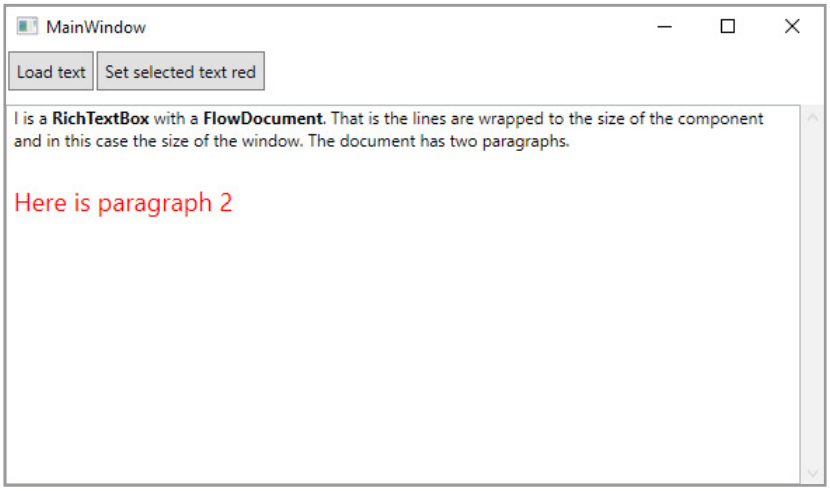
All that remains is to implement an event handler that captures events for Ctrl + Z and Ctrl + Y:

```
public MainWindow()
{
  InitializeComponent();
  KeyDown += KeyDownHandler;
}

private void KeyDownHandler(object sender, KeyEventArgs e)
{
  if (e.Key == Key.Z &&
    (Keyboard.Modifiers & ModifierKeys.Control) == ModifierKeys.Control)
  {
    Undo();
    e.Handled = true;
  }
  else if (e.Key == Key.Y &&
    (Keyboard.Modifiers & ModifierKeys.Control) == ModifierKeys.Control)
  {
    Redo();
    e.Handled = true;
  }
}
```

3 ENTER TEXT

In most programs there is a need to be able to enter text, and in WPF it is quite simple using a *TextBox*, which has all the facilities that are typically needed. You can go a long way with this component, but it does not solve all tasks and think, for example, of a word processing program where there is a need to be able to format text and work with text written with different fonts. In fact, entering and editing text has been one of the major issues in all the time one has developed computer programs. WPF has a component for that called a *RichTextBox*. It is a complex control, and it is also complex to use the control which is the subject for this chapter.

To show what it is all about I will start with a simple program which opens the following window:



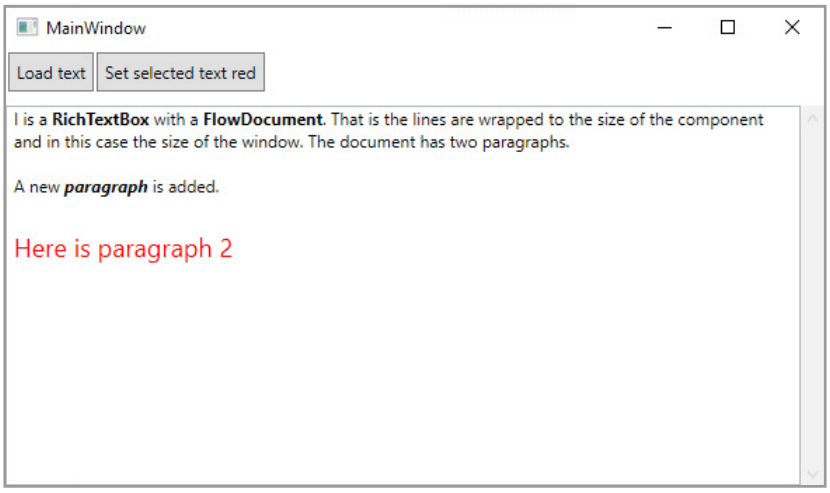
The window has two buttons and a *RichTextBox* in a *ScrollViewer*. You should note that the window contains some text where two words are bold, and the text in the last line is red and written with a large font. When you run the program you must also note, that the line wraps as you change the size of the window. This is this kind of things you can do with a *RichTextBox*, where you can assign attributes to text elements. You can use the same attributes in a *TextBox*, but here they have effect on all the content of the component.

The window is created as shown below:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="30"/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <StackPanel Orientation="Horizontal">
    <Button x:Name="load" Content="Load text" Margin="2,2,0,0"
Padding="5,0,5,0"
     Click="load_Click"/>
    <Button Content="Set selected text red" Margin="2,2,0,0"
Padding="5,0,5,0"
     Click="Red_Click"/>
  </StackPanel>
  <ScrollViewer Grid.Row="1" Margin="0,10,0,0">
    <RichTextBox Name="editor">
      <FlowDocument>
        <Paragraph>I is a <Bold>RichTextBox</Bold> with a
          <Bold>FlowDocument</Bold>. That is the lines are wrapped
          to the size of the component and in this case the size of the
          window.
          The document has two paragraphs.
        </Paragraph>
        <Paragraph Foreground="Red" FontSize="18">Here is paragraph 2</
        Paragraph>
      </FlowDocument>
    </RichTextBox>
  </ScrollViewer>
</Grid>
```

When you read the code, you can easily understand the meaning but of course you should especially note the definition of the *RichTextBox* component. Such a component has a content, which is a *FlowDocument* that consists of *Paragraph* elements. In this case there is two. Also note that the text can be decorated with inline attributes and above two words are defined as bold. You can also assign attributes to a paragraph element, and for the last paragraph the foreground color is defined as red.

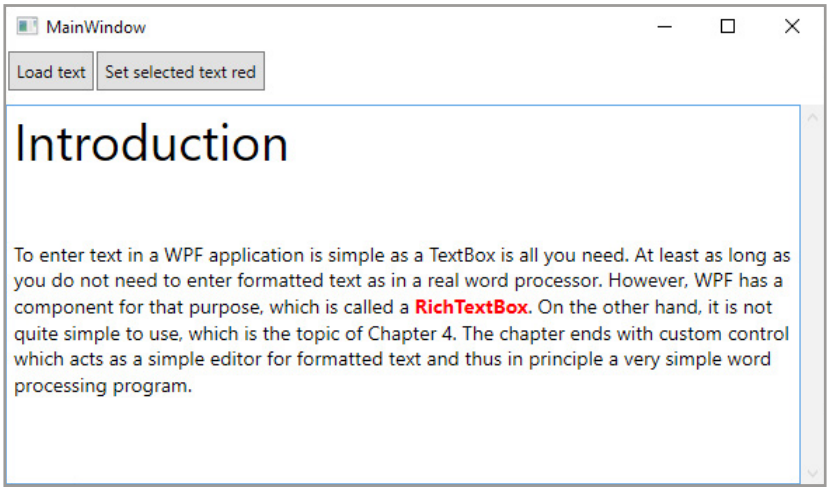
If you run the program you can enter text and edit already entered text and below is added a new paragraph:



Here you should note the word *paragraph* is bold and italic. It is done by selecting the text and enter Ctrl + B and Ctrl + I. That is the control has some of the usual formatting commands build in.

If you click on the first button the content of the window is changed to:

when the event handler has initialized the component with another text. If I select the text *RichTextBox* and enter Ctrl + B and then click on the second button in the toolbar the result is:



That is the button change the color of the selected text to red. The code for the two event handlers is:

```
private void Red_Click(object sender, RoutedEventArgs e)
{
  TextSelection text = editor.Selection;
  if (!text.IsEmpty) text.ApplyPropertyValue(ForegroundProperty,
  Brushes.Red);
}

private void load_Click(object sender, RoutedEventArgs e)
{
  FlowDocument doc = new FlowDocument();
  doc.FontSize = 14;
  Paragraph p1 = new Paragraph();
  p1.Inlines.Add(new Run("Introduction"));
  p1.FontSize = 36;
  Paragraph p2 = new Paragraph();
  p2.Inlines.Add(new Run("To enter text in a WPF application ... "));
  doc.Blocks.Add(p1);
  doc.Blocks.Add(p2);
  editor.Document = doc;
}
```

The last handler is performed when you click the first button. The handler creates a new *FlowDocument* and set the font size for this document. The a *Paragraph* is created and for this paragraph is added a *Run* with a text. A *Run* element represents text in a paragraph

which can be formatted. In this case the method defines a font size for the paragraph, and this size will be used for all text in the paragraph. The handler also creates another paragraph, and the two paragraphs are added to the document's *Block* collection. The last statement assign the new document to the control.

The other event handler is used to color the selected text red. A *RichTextBox* control has a *Selection* property which returns a *TextSelection* representing the selected text. If something is selected this object is used to set a dependency property for foreground color.

3.1 DOCUMENTS

Before I show more on how to use a *RichTextBox* it is necessary with a few words about documents in WPF, which is actually a large area.

WPF supports a number of facilities regarding processing of documents and including special facilities for displaying documents. In general, documents are divided into two categories, which are called respectively

1. fixed documents
2. flow documents

Fixed documents are intended for applications that require a precise "what you see is what you get" presentation, independent of the screen, printer or other used hardware. Typical uses for fixed documents include word processing, where the page design is critical. As part of its layout, a fixed document maintains the precise positional placement in the document of content elements. For example, a fixed document page viewed on 96 dpi display will appear exactly the same when it is output to a 600 dpi printer as when it is output to a 4800 dpi phototypesetter. The page layout remains the same in all cases, while the quality maximizes to the capabilities of the device.

By comparison, flow documents are designed to optimize viewing and readability and are best utilized when reading the document is the most important. Rather than being set to one predefined layout, flow documents dynamically adjust and reflow their content based on run-time variables such as window size, device resolution, and optional user preferences. A Web page is a simple example of a flow document where the page content is dynamically formatted to fit the current window.

The .NET Framework has a set of controls that simplify using fixed documents, flow documents, and general text within an application. The display of fixed document content is supported using the control *DocumentViewer*, while display of flow documents is supported by

1. *FlowDocumentReader*
2. *FlowDocumentPageViewer*
3. *FlowDocumentScrollViewer*

The control *DocumentViewer* is designed to display the content of a *FixedDocument*, and the component has support for copy - paste, zoom, search functions and print, and the component provides access to the pages content through scrolling. The control can alone be used to display the content read-only, but editing or modification is not supported. In this chapter I will only look at *FlowDocuments*, but I will return to *FizedDocuments* in the next chapter about print.

The controls *FlowDocumentReader*, *FlowDocumentPageViewer*, and *FlowDocumentScrollViewer* are all used for flow documents. The first has features that enable the user to dynamically choose between various viewing modes, including single-page, two-page and a continuous scrolling. If you do not need the ability to dynamically switch between different viewing

modes you can use a *FlowDocumentPageViewer* or *FlowDocumentScrollViewer*. This three components can only be used to view a flow document, and if you need to edit at flow document you must use a *RichTextBox*. If you just add a *RichTextBox* directly to the window, it will automatically create a *FlowDocument* instance that you will be editing, but you can also as shown in the above example create a *FlowDocument* instance and use this document in the component.

A FlowDocument can consist of various elements including text, images, tables, and even *UIElement* derived classes like controls. To understand how to create complex flow content, it is important to know and understand the classes used to represent the content of the document. Each class used in flow content has a specific purpose. In addition, the hierarchical relation between flow classes helps to understand how they are used. Classes derived from the *Block* class are used to contain other objects while classes derived from *Inline* contain objects that are displayed.

Block-derived classes are derived from the class *Block* and are used to group elements under a common and to assign attributes to all elements in the block. The classes are:

1. *Paragraph* is the most common used block class and is used to represent a text block (a paragraph) in a document. A *Paragraph* will normally contains one or more *InLine* objects. A *FlowDocument* will always consist of one or more *Paragraph* objects.
2. *Section* is a class which can contains other *Block* elements as *Paragraph* objects and is typical uses to assign attributes applied by all paragraph elements of the *Section* object.
3. *BlockUIContainer* enables *UIElement* elements (for example a *Button*) to be embedded in block-derived flow content. Another class *InlineUIContainer* (see below) is used to embed UIElement elements inline.
4. *List* is a block-derived class used to inserts ordered or un-ordered lists in a document. It is a block class as it contains *ListItem* objects.
5. *Table* is a class which represents a table in a *FlowDocument*, and to build a table are used other classes as *TableRowGroup*, *TableRow* and *TableCell*.

Inline-derived classes are classes that inherit from the class *Inline* are either contained within a *Block* element or another *Inline*. Inline Elements are often used as the direct container of content that is rendered to the screen. For example can a *Paragraph* contain a *Run* object which represents some text. The most important classes are:

1. *Run* which is a class that represent unformatted text. It is the content of a *Run* object to which you assign formatting.
2. *Span* that is an inline object which groups other inline objects. Derived classes are *HyperLink*, *Bold*, *Italic* and *Underline*.

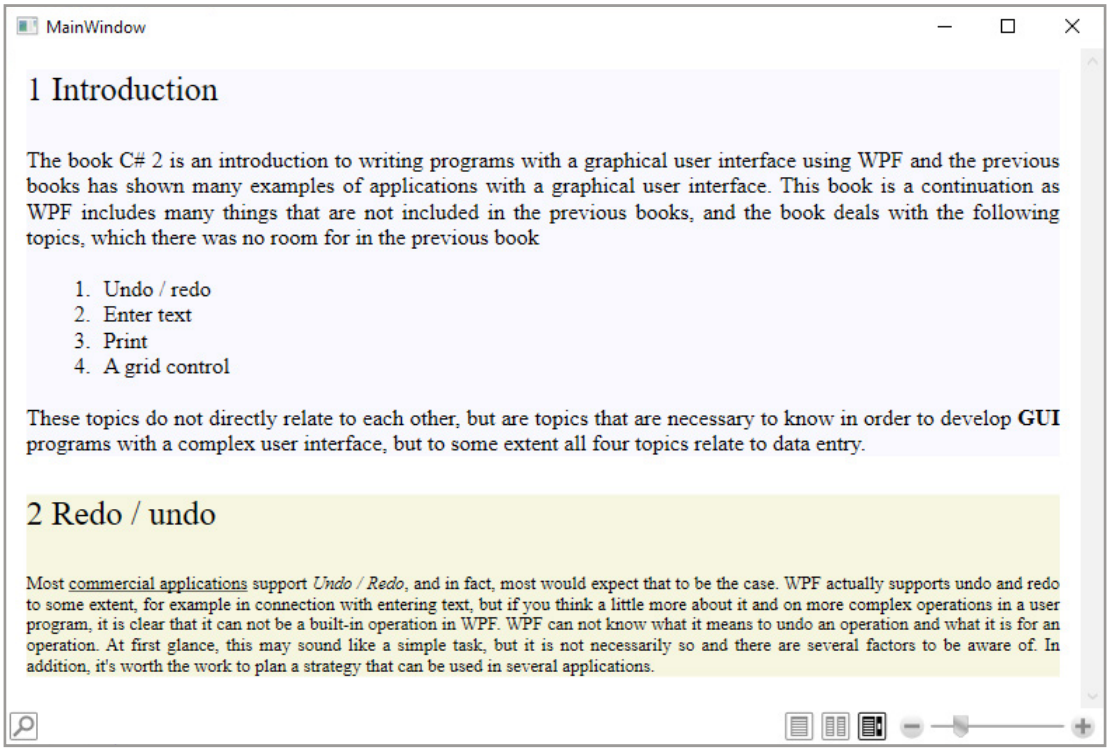
3. *InlineUIContainer* that enables *UIElement* objects to be embedded in another inline object.
4. *LineBreak* to insert a line break in a *FlowDocument*.

3.2 DISPLAYING FLOW DOCUMENTS

To see more on flow documents in action I will show some examples which create a *FlowDocument* and show the document in a *FlowDocumentReader*. If you run the program *ShowText1* it opens the window shown below. The window has only one component which is a *FlowDocumentReader* that is a component used to show a *FlowDocument*. You should note that the component has a built-in toolbar that appears at the bottom of the window. The toolbar has a search function and a zoom utility, and it also has three buttons where the user can can select how the document is displayed:

1. As pages
2. As two pages side by side
3. As a scroll view where the content fills all the window

If you look at the window below the mode is the last. The document has 5 paragraphs and a list, and you should note that the paragraphs uses different font sizes and also that the middle paragraph has a word that is bold while the last paragraph has some text underlined and some text in italic.



The document is defined in XML (I have not shown all the text):

```
<FlowDocumentReader ViewingMode="Scroll" >
  <FlowDocument FontFamily="Times New Roman" FontSize="13">
    <Section FontSize="16" Background="GhostWhite">
      <Paragraph FontSize="24">
        1 Introduction
      </Paragraph>
      <Paragraph>
        The book C# 2 is an introduction ...
      </Paragraph>
      <List MarkerStyle="Decimal" StartIndex="1" >
        <ListItem>
          <Paragraph>Undo / redo</Paragraph>
        </ListItem>
        <ListItem>
          <Paragraph>Enter text</Paragraph>
        </ListItem>
        <ListItem>
          <Paragraph>Print</Paragraph>
        </ListItem>
        <ListItem>
          <Paragraph>A grid control</Paragraph>
        </ListItem>
      </List>
      <Paragraph>These topics do not directly relate to each other,
      but are topics
        that are necessary to know in order to develop <Bold>GUI</Bold>
        programs...
      </Paragraph>
    </Section>
    <Section Background="Beige">
      <Paragraph FontSize="24">2 Redo / undo</Paragraph>
      <Paragraph>Most <Underline>commercial applications</Underline>
      support
        <Italic>Undo / Redo</Italic>, and in fact ...
      </Paragraph>
    </Section>
  </FlowDocument>
</FlowDocumentReader>
```

First note how is defined to use *Scroll* as *ViewingMode*. When the *FlowDocument* is defined the font and the font size are defined, and these values are used through the document if not other values are defined for these attributes. Then you should note that the document has two sections. The first section is defined with at font size and a background color, and this section has three paragraphs and a list. The other section is defined with a background color and has two paragraphs.

In the first section the first paragraph is a header and has a font size on 24. The text is automatic wrapped in a *Run* element and you do not need to write this element in XML. The second paragraph is just text, and when nothing is said the font size from the *Section* element is used. You must note how to define the list and here the attribute *MakerStyle* that defines the list to be an ordered list. Default is an un-ordered list. Else it is quite simple to define a list consisting of *ListItem* elements containing *Paragraph* elements. In the last paragraph in this section you should note how to indicate that some text must be bold.

For the last section there is not much to add, but you should note how to indicate that text should be underlined and italic.

As it turns out, it is easy enough to define a *FlowDocument* in XML, and when you see the code it looks like all other markup such as HTML. However, it is not so often that you need to define a *FlowDocument* in XML. The goal of this chapter is to be able to write and format a *FlowDocument* in a *RichTextBox* and thus a component where the user edits a *FlowDocument*. It requires that the document and its properties must be able to be maintained in code behind, and it is therefore more interesting how to define a *FlowDocument* in C#. The program *ShowText2* opens exactly the same window as above, but this time the same document is defined in C#:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    viewer.Document = CreateDocument();
  }

  private FlowDocument CreateDocument()
  {
    FlowDocument doc = new FlowDocument { FontSize = 13 };
    foreach (FontFamily font in Fonts.SystemFontFamilies)
      if (font.Source.Equals("Times New Roman"))
      {
        doc.FontFamily = font;
        break;
      }
    Section s1 = new Section { FontSize = 16, Background = Brushes.
    GhostWhite };
    Paragraph p1 = new Paragraph { FontSize = 24 };
    p1.Inlines.Add(new Run("1 Introduction")); ;
    Paragraph p2 = new Paragraph();
```

```
    p2.Inlines.Add(new Run("The book C# 2 is ... "));
    List ls = new List { MarkerStyle = TextMarkerStyle.Decimal,
    StartIndex = 1 };
    ls.ListItems.Add(new ListItem(new Paragraph(new Run("Undo /
    redo"))));
    ls.ListItems.Add(new ListItem(new Paragraph(new Run("Enter
    text"))));
    ls.ListItems.Add(new ListItem(new Paragraph(new Run("Print"))));
    ls.ListItems.Add(new ListItem(new Paragraph(new Run("A grid
    control"))));
    Paragraph p3 = new Paragraph();
    p3.Inlines.Add(new Run("These topics do not directly ... "));
    p3.Inlines.Add(new Bold(new Run("GUI")));
    p3.Inlines.Add(new Run(" programs with a complex user ... "));
    s1.Blocks.Add(p1);
    s1.Blocks.Add(p2);
    s1.Blocks.Add(ls);
    s1.Blocks.Add(p3);
    Section s2 = new Section { Background = Brushes.Beige };
    Paragraph p4 = new Paragraph { FontSize = 24 };
    p4.Inlines.Add(new Run("2 Redo / undo"));
    Paragraph p5 = new Paragraph();
    p5.Inlines.Add("Most ");
    p5.Inlines.Add(new Underline(new Run("commercial applications")));
    p5.Inlines.Add(" support ");
    p5.Inlines.Add(new Italic(new Run("Undo / Redo")));
    p5.Inlines.Add(" and in fact, most would ... ");
    s2.Blocks.Add(p4);
    s2.Blocks.Add(p5);
    doc.Blocks.Add(s1);
    doc.Blocks.Add(s2);
    return doc;
  }
}
```

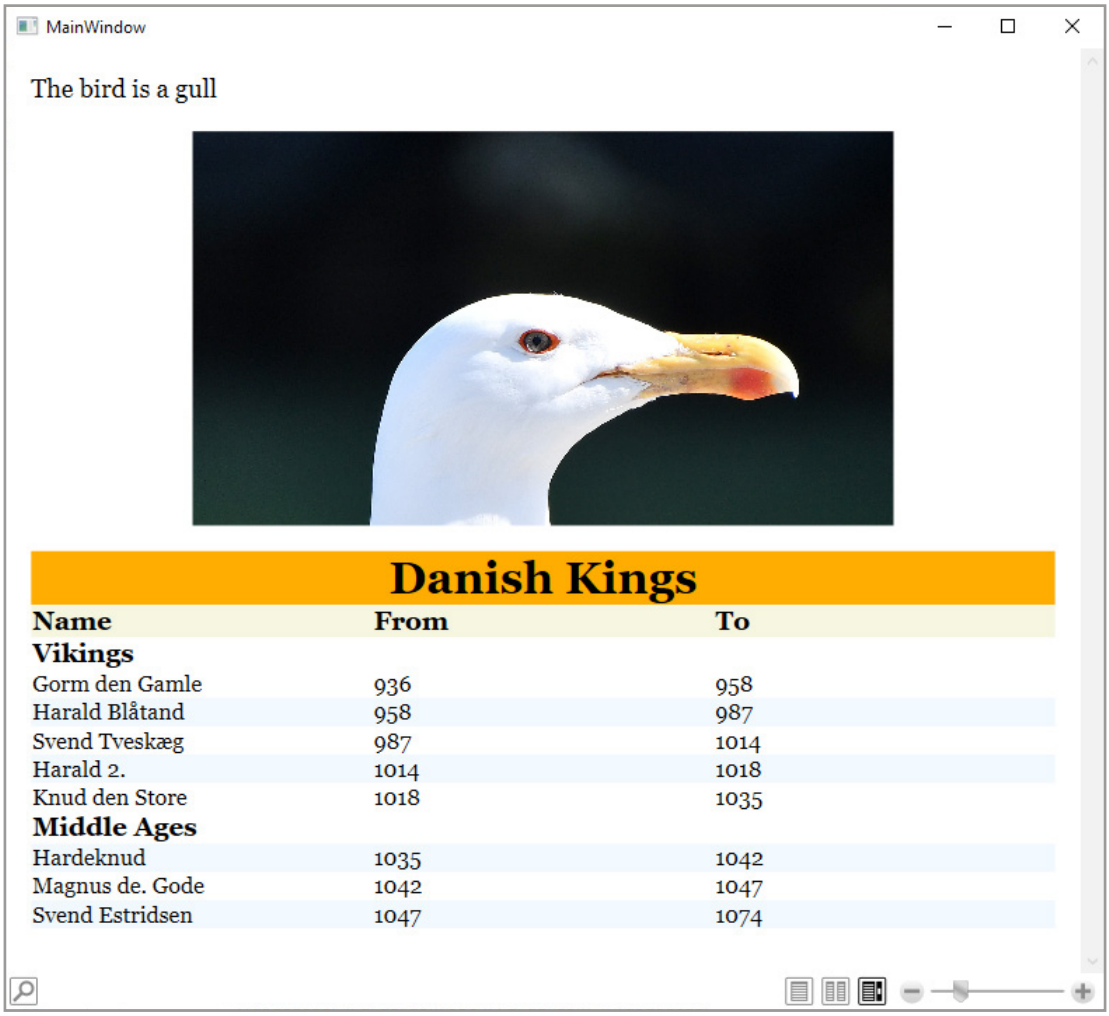
The method *CreateDocument()* create a *FlowDocument* and sets the font size and the font family. Then it starts to build the document, and I thing it is easy to follow what is happening.

As an example note *p3* that is a *Paragraph* object. To the object's *InLines* collection are added three elements. The first is a *Run* object representing text. Then follows another *Run* object (the text *GUI*), but this object is wrapped in a *Bold* object, which is a *Span* object that is added to the collection *InLines*. Last a third *Run* object is added.

Also note how to add elements to a *Section*. The first *Section* object has three *Paragraph* objects and one *List* object.

The example should show how to create a *FlowDocument* in C#. As it turns out, it can be comprehensive, but is in principle simple enough, and if you need to write a text editor using a *RichTextBox*, this is what is needed.

The next example called *ShowText3* also show a *FlowDocument* in a *FlowDocumentReader*. The document is written as XML and it should show how to insert an image in a document and a table. The table shows eight Danish kings, and the important thing is to show how to arrange data in rows and columns with a table. If you run the program it opens a window as shown below:



The document is defined as:

```
<FlowDocumentReader ViewingMode="Scroll">
  <FlowDocument>
    <Paragraph FontSize="18">
      The bird is a gull
    </Paragraph>
    <BlockUIContainer><Image Source="/062a.jpg" Width="500" /></
    BlockUIContainer>
    <Table>
      <Table.Columns>
        <TableColumn/>
        <TableColumn/>
        <TableColumn/>
      </Table.Columns>
      <TableRowGroup>
        <TableRow Background="Orange">
          <TableCell ColumnSpan="3" TextAlignment="Center">
            <Paragraph FontSize="24pt" FontWeight="Bold">Danish Kings</
            Paragraph>
          </TableCell>
        </TableRow>
        <TableRow Background="Beige">
          <TableCell><Paragraph FontSize="14pt"
            FontWeight="Bold">Name</Paragraph></TableCell>
          <TableCell><Paragraph FontSize="14pt"
            FontWeight="Bold">From</Paragraph></TableCell>
          <TableCell><Paragraph FontSize="14pt"
            FontWeight="Bold">To</Paragraph></TableCell>
        </TableRow>
        <TableRow>
          <TableCell ColumnSpan="3"><Paragraph FontSize="14pt"
          FontWeight="Bold">Vikings</Paragraph></TableCell>
        </TableRow>
        <TableRow>
          <TableCell><Paragraph>Gorm den Gamle</Paragraph></TableCell>
          <TableCell><Paragraph>936</Paragraph></TableCell>
          <TableCell><Paragraph>958</Paragraph></TableCell>
        </TableRow>
        <TableRow Background="AliceBlue">
          <TableCell><Paragraph>Harald Blåtand</Paragraph></TableCell>
          <TableCell><Paragraph>958</Paragraph></TableCell>
          <TableCell><Paragraph>987</Paragraph></TableCell>
```

```
        </TableRow>
        <TableRow>
          <TableCell><Paragraph>Svend Tveskæg</Paragraph></TableCell>
          <TableCell><Paragraph>987</Paragraph></TableCell>
          <TableCell><Paragraph>1014</Paragraph></TableCell>
        </TableRow>
        <TableRow Background="AliceBlue">
          <TableCell><Paragraph>Harald 2.</Paragraph></TableCell>
          <TableCell><Paragraph>1014</Paragraph></TableCell>
          <TableCell><Paragraph>1018</Paragraph></TableCell>
        </TableRow>
        <TableRow >
          <TableCell><Paragraph>Knud den Store</Paragraph></TableCell>
          <TableCell><Paragraph>1018</Paragraph></TableCell>
          <TableCell><Paragraph>1035</Paragraph></TableCell>
        </TableRow>
        <TableRow>
          <TableCell ColumnSpan="3"><Paragraph FontSize="14pt"
            FontWeight="Bold">Middle Ages</Paragraph></TableCell>
        </TableRow>
        <TableRow Background="AliceBlue">
          <TableCell><Paragraph>Hardeknud</Paragraph></TableCell>
          <TableCell><Paragraph>1035</Paragraph></TableCell>
          <TableCell><Paragraph>1042</Paragraph></TableCell>
        </TableRow>
        <TableRow>
          <TableCell><Paragraph>Magnus de. Gode</Paragraph></TableCell>
          <TableCell><Paragraph>1042</Paragraph></TableCell>
          <TableCell><Paragraph>1047</Paragraph></TableCell>
        </TableRow>
        <TableRow Background="AliceBlue">
          <TableCell><Paragraph>Svend Estridsen</Paragraph></TableCell>
          <TableCell><Paragraph>1047</Paragraph></TableCell>
          <TableCell><Paragraph>1074</Paragraph></TableCell>
        </TableRow>
      </TableRowGroup>
    </Table>
  </FlowDocument>
</FlowDocumentReader>
```

It's easy enough to understand the code and you must primarily note how to define a table as a

1. *Table* element with *TableColumn* elements
2. a *TableRowGroup* element with *TableRow* elements where each *TableRow* contains one or three *TableCell* elements

The program *ShowText4* performs the same, but the document is defined in code:

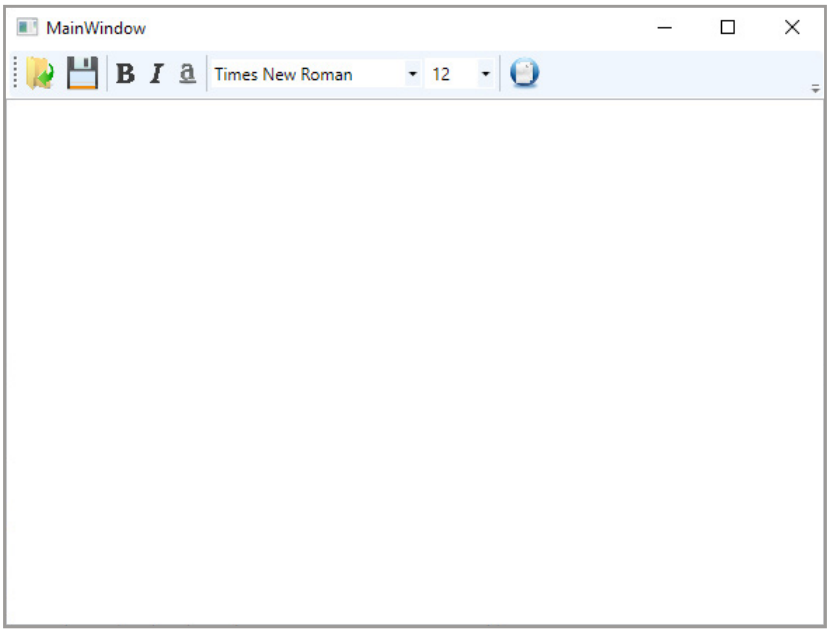
```
private FlowDocument CreateDocument()
{
  FlowDocument doc = new FlowDocument();
  Paragraph p = new Paragraph { FontSize = 18 };
  p.Inlines.Add("The bird is a gull");
  Image img = new Image { Width = 500 };
  BitmapImage bmp = new BitmapImage();
  bmp.BeginInit();
  bmp.UriSource = new Uri("062a.jpg", UriKind.Relative);
  bmp.EndInit();
  img.Source = bmp;
  Table table = new Table();
  table.Columns.Add(new TableColumn());
  table.Columns.Add(new TableColumn());
  table.Columns.Add(new TableColumn());
  TableRowGroup rows = new TableRowGroup();
  TableRow row = new TableRow { Background = Brushes.Orange };
  TableCell c1 = new TableCell { ColumnSpan = 3,
    TextAlignment = TextAlignment.Center };
  Paragraph p1 = new Paragraph { FontSize = 24, FontWeight =
  FontWeights.Bold };
  p1.Inlines.Add("Danish Kings");
  c1.Blocks.Add(p1);
  row.Cells.Add(c1);
  rows.Rows.Add(row);
  rows.Rows.Add(CreateRow(Brushes.Beige, 14, FontWeights.Bold, "Name",
    "From", "To"));
  rows.Rows.Add(CreateRow(Brushes.White, 14, FontWeights.Bold, "Vikings"));
  rows.Rows.Add(CreateRow(Brushes.White, 12, FontWeights.Normal,
    "Gorm den Gamle", "936", "958"));
  rows.Rows.Add(CreateRow(Brushes.AliceBlue, 12, FontWeights.Normal,
    "Harald Blåtand", "958", "987"));
  rows.Rows.Add(CreateRow(Brushes.White, 12, FontWeights.Normal,
    "Svend Tveskæg", "987", "1014"));
  rows.Rows.Add(CreateRow(Brushes.AliceBlue, 12, FontWeights.Normal,
    "Harald 2.", "1014", "1018"));
  rows.Rows.Add(CreateRow(Brushes.White, 12, FontWeights.Normal,
    "Knud den Store", "1018", "1035"));
  rows.Rows.Add(CreateRow(Brushes.White, 14, FontWeights.Bold, "Middle
  Ages"));
```

```
  rows.Rows.Add(CreateRow(Brushes.AliceBlue, 12, FontWeights.Normal,
    "Hardeknud", "1035", "1042"));
  rows.Rows.Add(CreateRow(Brushes.White, 12, FontWeights.Normal,
    "Magnus den Gode", "1042", "1047"));
  rows.Rows.Add(CreateRow(Brushes.AliceBlue, 12, FontWeights.Normal,
    "Svend Estridsen", "1047", "1074"));
  table.RowGroups.Add(rows);
  doc.Blocks.Add(p);
  doc.Blocks.Add(new BlockUIContainer(img));
  doc.Blocks.Add(table);
  return doc;
}

private TableRow CreateRow(Brush brush, int size, FontWeight weight,
  params string[] text)
{
  TableRow row = new TableRow { Background = brush };
  foreach (string s in text)
  {
    Paragraph p = new Paragraph { FontSize = size, FontWeight = weight };
    p.Inlines.Add(s);
    TableCell c = new TableCell();
    c.Blocks.Add(p);
    row.Cells.Add(c);
  }
  return row;
}
```

You should notice how an auxiliary method that creates a *TableRow* is defined. When working with a *FlowDocument* in C#, it can be beneficial to define that kind of auxiliary method.

3.3 A RICHTEXTBOX

The example *EditText2* opens the following window:



It is a window with a toolbar and a *RichTextBox*. The toolbar has functions to load and save a document and to set selected text bold, italic or underline, and in the same way the user can select a font and a font size for the selected text. The last button will be explained below. After entered some test the result could be

The text is a *FlowDocument* with 12 paragraphs as empty lines counts as a paragraph. If the content is as above, and you click on the last button in the toolbar, you get the following window:

It looks complicated and it is, but it is the contents of the current *FlowDocument* converted to *Xaml*. Sure, it looks complicated, but it is text and readable, and you can rediscover the text of the document including its structure in the form of paragraphs and attributes.

To write the program I must write the XML for the user interface

```
<DockPanel>
  <ToolBar DockPanel.Dock="Top">
    <Button Command="ApplicationCommands.Open">
      <Image Source="/Images/Open.png" Width="24" Height="24" />
    </Button>
    <Button Command="ApplicationCommands.Save">
      <Image Source="/Images/Save.png" Width="24" Height="24" />
    </Button>
    <Separator />
    <ToggleButton Command="EditingCommands.ToggleBold" Name="cmdBold">
      <Image Source="/Images/Bold.png" Width="16" Height="16" />
    </ToggleButton>
```

```
    <ToggleButton Command="EditingCommands.ToggleItalic"
    Name="cmdItalic">
      <Image Source="/Images/Italic.png" Width="16" Height="16" />
    </ToggleButton>
    <ToggleButton Command="EditingCommands.ToggleUnderline"
    Name="cmdUnderline">
      <Image Source="/Images/Underline.png" Width="16" Height="16" />
    </ToggleButton>
    <Separator />
    <ComboBox Name="cmbFamily" Width="150" SelectionChanged="Family_
    Changed" />
    <ComboBox Name="cmbSize" Width="50" IsEditable="True"
      TextBoxBase.TextChanged="Size_Changed" />
    <Separator />
    <Button Command="{Binding ShowCommand}">
      <Image Source="/Images/Copy.png" Width="24" Height="24" />
    </Button>
  </ToolBar>
  <RichTextBox Name="editor" VerticalScrollBarVisibility="Auto"
    SelectionChanged="editor_SelectionChanged" />
</DockPanel>
```

The code defines primarily the window's toolbar. The two first buttons are bound to pre-defined commands which call methods defined in code behind. The next three buttons, which are *ToogleButtons* are also bound to commands that is pre-defined commands supported by a *RichTextBox* component. The two combo boxes has assigned event handlers and the same goes for the *RichTextBox*. The first combo box fires an event when the user selects another font. The other combo box fires an event when the value is changed which happens if the user select a value or enter a value. The last event (the *RichTextBox*) fires when then the text changes, that is when the user enter a character. The last button which opens the window with the *FlowDocument* in text is bound to a user defined command.

The two commands for the *Open* and *Save* buttons references methods which opens a file with a *FlowDocument* and updates the *RichTextBox*, and save the content of the *RichTextBox* in a file:

```
private void Save(object sender, ExecutedRoutedEventArgs e)
{
  SaveFileDialog dlg = new SaveFileDialog();
  dlg.Filter = "Rich Text Format (*.rtf)|*.rtf|All files (*.*)|*.*";
  if (dlg.ShowDialog() == true)
  {
    FileStream fileStream = new FileStream(dlg.FileName, FileMode.Create);
    TextRange range = new TextRange(editor.Document.ContentStart,
      editor.Document.ContentEnd);
    range.Save(fileStream, DataFormats.Rtf);
  }
}

private void Open(object sender, ExecutedRoutedEventArgs e)
{
  OpenFileDialog dlg = new OpenFileDialog();
  dlg.Filter = "Rich Text Format (*.rtf)|*.rtf|All files (*.*)|*.*";
  if (dlg.ShowDialog() == true)
  {
    FileStream fileStream = new FileStream(dlg.FileName, FileMode.Open);
    TextRange range = new TextRange(editor.Document.ContentStart,
      editor.Document.ContentEnd);
    range.Load(fileStream, DataFormats.Rtf);
  }
}
```

The first method opens a usual *Save File* dialog box, and the user enter / select a file the file is created. You must then note how to reference all text in *RichTextBox* component, and the text is save in the file in RTF format. That is the *FlowDocument* is converted to RTF. The other method opens an *Open File* dialog box, and if the user selects a file it is opened. Again the content of the *RichTextBox* is referenced with a *TextRange* object, which is filled reading the content of the fill.

If you select some text in the component and clicks on one of the buttons for bold, italic and underline you do not have to do something in the code as these functions are built-in for a *RichTextBox* component.

The event handlers for the two combo boxes are as follows, and there is not much to explain other than noting the syntax for how to change the font and font size of the selected text:

```
private void Family_Changed(object sender, SelectionChangedEventArgs e)
{
  if (cmbFamily.SelectedItem != null)
    editor.Selection.ApplyPropertyValue(Inline.FontFamilyProperty,
      cmbFamily.SelectedItem);
}

private void Size_Changed(object sender, TextChangedEventArgs e)
{
  if (cmbSize.SelectedItem != null)
  {
    editor.SetValue(Inline.FontSizeProperty, cmbSize.SelectedItem);
    editor.Selection.ApplyPropertyValue(Inline.FontSizeProperty,
      cmbSize.SelectedItem);
  }
}
```

The event handler for *RichTextBox* fills and I will not show the code here, but it ensures that the selections made in the toolbar are used for the text entered.

The result of the program is that you can edit a *FlowDocument*, but also perform some formatting, and you can save the document in a file and open the file again. There is of course use for several formatting options, which is the topic of section 3.5.

3.4 SOME PREPARATIONS

To write an editor using a *RichTextBox* you need a *ColorPicker* which is a component used to select a color. In the editor this component should be used to color a selected text. You should note that there are many such components on the internet but I have chosen for the sake of example to write my own. As the component can be used in other programs it is placed in a class library, a library which I in the rest of this book will expand with other components. As so I start creating a class library project called *PaGUI*, but it is important that it is a

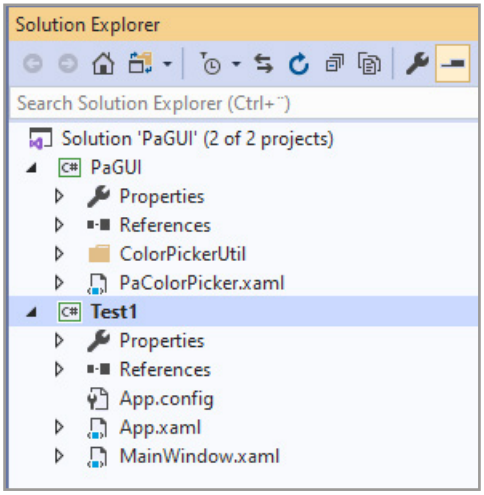
WPF User Control Library (.NET Framework)

project. When Visual Studio creates the project it automatic add a User Control to the project. I have renamed this control to *PaColorPicker*. I have also added a folder called *ColorPickerUtil* to the project, a folder which should be used for the files for this component.

Next I have for the same *Solution* added a new project (a *WPF App (.NET Framework)* project) called *Test1* which should be used to test the component. I also set the new project as

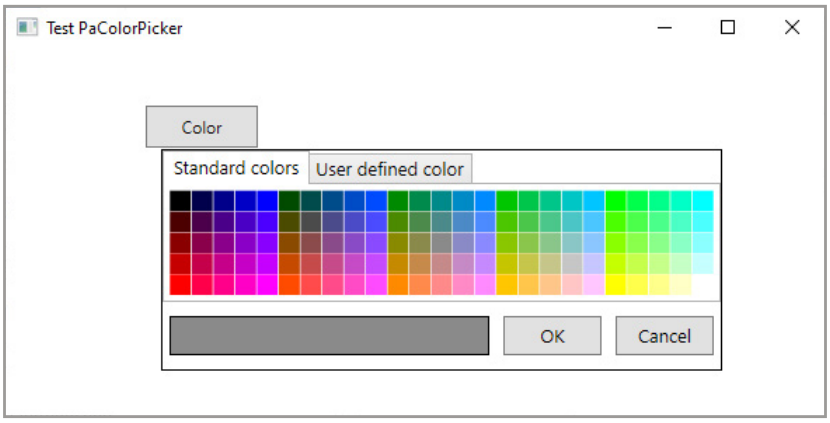
Set as Startup Project

which means that Visual Studio starts this program when you select *Start* in the menu. Then the *Solution Explorer* shows:

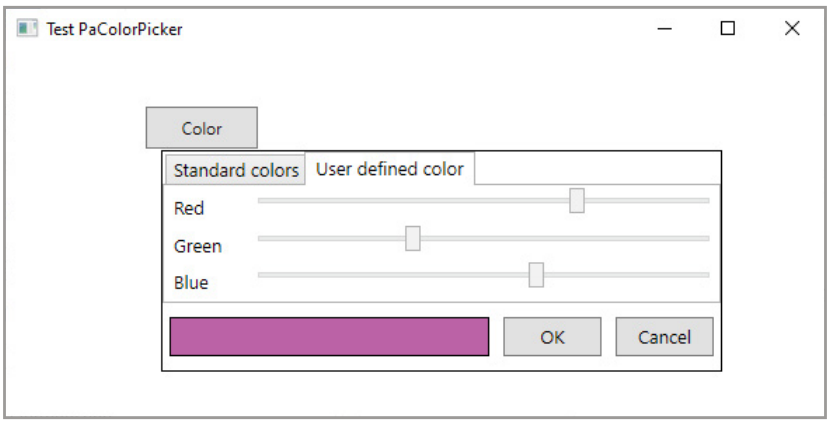


If you run the program it opens the following window:

If you clicks the button the program opens the color picker:



and that is the program shows the user control, where the can choose between 125 standard colors. The control has another tab (see below), where the user using three *Slider* components can create any color. If the user clicks *OK* or *Cancel* the user control disappears, and if it is the *OK* button the background color of the window is changed to the selected color.



The code for the user control (the folder *ColorPickerUtil*) consists of two other user controls, one for each of the two tabs. The user control for the first tab is called *ColorPicker*. The XML is trivial and consists only of a *UniformGrid*, and I will not show the XML here. The C# code is also quite simple:

```
public partial class ColorPicker : UserControl
{
  public event EventHandler<ColorEventArgs> ColorChanged;

  public ColorPicker()
  {
    InitializeComponent();
    byte[] bytes = { 0, 64, 128, 192, 255 };
    foreach (byte r in bytes)
      foreach (byte g in bytes)
        foreach (byte b in bytes) grid.Children.Add(CreateRect(r, g, b));
  }

  private Rectangle CreateRect(byte r, byte g, byte b)
  {
    Brush brush = new SolidColorBrush(Color.FromRgb(r, g, b));
    Rectangle rect = new Rectangle { Fill = brush, Width = 15, Height
    = 15 };
    rect.Tag = brush;
    rect.MouseDown += RectClicked;
    return rect;
  }

  private void RectClicked(object sender, MouseEventArgs e)
  {
    if (ColorChanged != null) ColorChanged(this,
      new ColorEventArgs((Brush)((Rectangle)sender).Tag));
  }
}
```

The class defines an event which is fired every times the user clicks on a rectangle. The constructor create the 125 rectangles and add them to the user interface. Note that each rectangle has an event handler for *MouseDown*, a handler that fires a *ColorChanged* event with a *Brush* for the color which is clicked.

The XML for the other user control used for the second tab is also simple and defines primarily there *Slider* components bound to dependency properties in code behind. This time the code fills more as it defines four dependency properties:

```
public partial class ColorMixer : UserControl
{
  public static readonly DependencyProperty CurrentBrushProperty = ...
  public static readonly DependencyProperty RedProperty = ...
  public static readonly DependencyProperty GreenProperty = ...
  public static readonly DependencyProperty BlueProperty = ...

  public event EventHandler<ColorEventArgs> ColorChanged;

  public ColorMixer()
  {
    InitializeComponent();
    DataContext = this;
  }

  ...

  private static void RedChanged(DependencyObject obj,
    DependencyPropertyChangedEventArgs args)
  {
    int red = (int)args.NewValue;
    int green = ((ColorMixer)obj).Green;
    int blue = ((ColorMixer)obj).Blue;
    Brush brush = new SolidColorBrush(Color.FromRgb((byte)red,
      (byte)green, (byte)blue));
    if (((ColorMixer)obj).ColorChanged != null)
      ((ColorMixer)obj).ColorChanged(obj, new ColorEventArgs(brush));
  }

  ...
}
```

The *Value* property in three sliders are bound to a dependency properties for the three colors red, green and blue. When the sliders are moved the *PropertyChangedCallback()* method is called for the actual slider, and a method fires an event to notify about the new color.

The XML for the main user control is:

```
<Grid>
  <Border BorderBrush="Black" BorderThickness="1" Margin="1">
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="48"/>
      </Grid.RowDefinitions>
      <TabControl x:Name="tabs" SelectionChanged="TabControl_
      SelectionChanged">
        <TabItem Header="Standard colors">
          <tools:ColorPicker ColorChanged="ColorPicker_ColorChanged"/>
        </TabItem>
      </TabControl>
      <Grid Grid.Row="1">
        <Grid.ColumnDefinitions>
          <ColumnDefinition/>
          <ColumnDefinition Width="90"/>
          <ColumnDefinition Width="75"/>
        </Grid.ColumnDefinitions>
        <Rectangle x:Name="rect" Grid.Row="1" StrokeThickness="1"
          Stroke="Black" Margin="5,10,0,10" Fill="{Binding Color}" />
        <Button Grid.Column="1" Content="OK" Margin="10" Click="Ok_
        Click" />
        <Button Grid.Column="2" Content="Cancel" Margin="0,10,5,10"
          Click="Cancel_Click" />
      </Grid>
    </Grid>
  </Border>
</Grid>
```

and defines a layout consisting of a *TabControl*, a *Rectangle* and two *Button* controls. There is not mush new, but you should note that the *TabControl* only defines one *TabItem* as the other is defined in code behind:

```
public partial class PaColorPicker : UserControl
{
  public static readonly DependencyProperty ColorProperty = ...
  public event EventHandler<ColorEventArgs> ColorChanged;
  private ColorMixer mixer = new ColorMixer();

  public PaColorPicker()
  {
    InitializeComponent();
    tabs.Items.Add(new TabItem { Header = "User defined color", Content
    = mixer });
    mixer.ColorChanged += ColorPicker_ColorChanged;
    Width = 402;
    Height = 160;
    DataContext = this;
  }

  public Brush Color
  {
    get { return (Brush)GetValue(ColorProperty); }
    set { SetValue(ColorProperty, value); }
  }

  private void ColorPicker_ColorChanged(object sender,
    ColorPickerUtil.ColorEventArgs e)
  {
    Color = e.Color;
  }

  private void TabControl_SelectionChanged(object sender,
    SelectionChangedEventArgs e)
  {
    TabControl tabControl = (TabControl)sender;
    if (tabControl.SelectedIndex == 1)  mixer.CurrentBrush = Color;
  }

  private void Ok_Click(object sender, RoutedEventArgs e)
  {
    if (ColorChanged != null) ColorChanged(this, new ColorEventArgs(Color));
    Visibility = Visibility.Collapsed;
  }

  private void Cancel_Click(object sender, RoutedEventArgs e)
  {
    Visibility = Visibility.Collapsed;
  }
}
```

You show note that the component for the second tab is created and added to the *TabControl*. Also note that the the *TabControl* has an event handler. The purpose is to update the sliders in the second tab, when this tab is selected. The event handlers for the buttons are simple, but not they set the visibility to *Collapsed*, which means the control is not visible on the screen. The control is a user control, and you cannot close a user control.

Now the control is finish and is ready for use in a program. The test program only has a main window, and the XML is:

```
<Grid>
  <Button Content="Color" Width="80" Height="30" VerticalAlignment="Top"
    HorizontalAlignment="Left" Margin="100,40,0,0" Click="Button_Click"/>
  <tools:PaColorPicker Name="pick" Color="Gray" VerticalAlignment="Top"
    HorizontalAlignment="Left" Margin="110,70,0,0" Visibility="Collapsed"/>
</Grid>
```

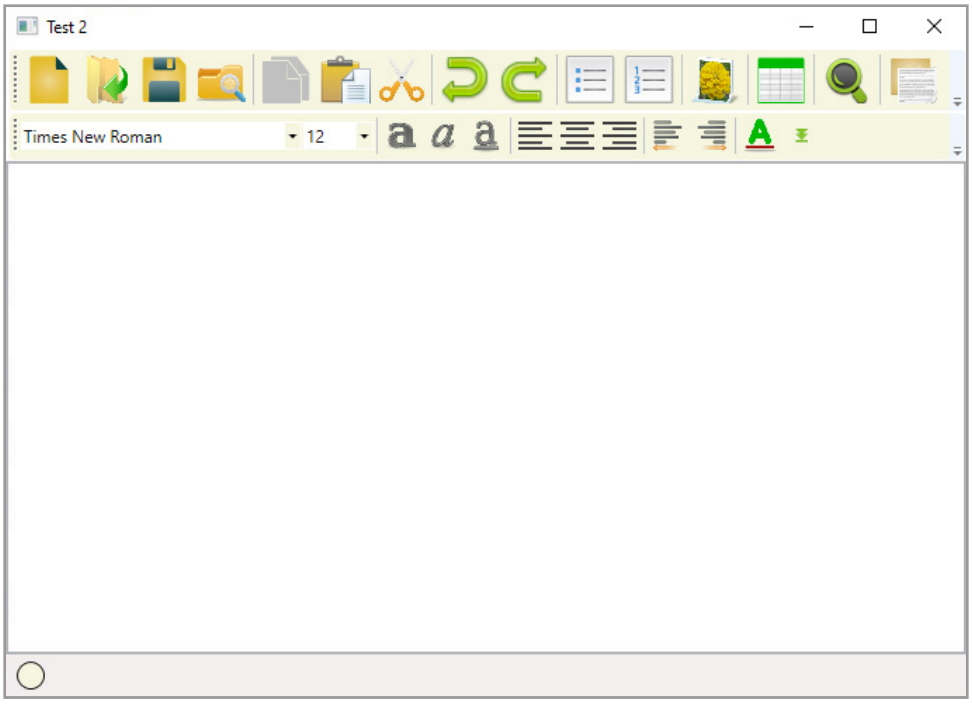
You must note how to use the user control in the design, and the control is used like all other controls. Also note, that control is defined to start its life as collapsed. The code behind consists of two event handles, one for the button which sets the user control to visible, and one for the event fired when the user clicks *OK* in the user control:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    pick.ColorChanged += ColorPicker_ColorChanged;
  }

  private void Button_Click(object sender, RoutedEventArgs e)
  {
    pick.Visibility = Visibility.Visible;
  }

  private void ColorPicker_ColorChanged(object sender, ColorEventArgs e)
  {
    Background = e.Color;
  }
}
```

3.5 AN EDITOR

As the last about enter text I will show a user control which can be used to edit formatted, and is most of all an expanded version of the program from section 3.3. The control is called *PaTextEditor* and is added to the class library *PaGUI* as a user control and the project is expanded is also expanded with a new folder called *TextEditorUtil* which contains the types used by this control. To the solution is also added a new test project called *Test2*. It is a simple project which only opens a window with the new user control, and the result is as shown below, and you have a program you can used to edit a *FlowDocument*.

In principle, the program is a simple word processing program, but it is far from a "real" word processing program. If you compare with a program like Word you will see that the services that the current component makes available are in no way close to the services that Work has for editing text. The component can therefore be used for two things. Firstly, to get an impression of how comprehensive it is to write a program as Word, but also as a control that can be used in practice in programs where there is a need to edit formatted text.



The control is a primarily a component with to toolbars, a *RichTextBox* and a status bar. All the editor function are defined in the toolbars and are as follows.

The first toolbar:

1. Create a new blank document (Ctrl + N)
2. Open document (Ctrl + O)
3. Save document. If it is a new document the function is the same as Save as (Ctrl + S)

4. Save as, which save the document under a new name
5. Copy (Ctrl + C)
6. Paste (Ctrl + V)
7. Cut (Ctrl + X)
8. Undo (Ctrl + Z)
9. Redo (Ctrl + Y)
10. Insert bullet list
11. Insert numbered list
12. Insert image (Jpg and Png images)
13. Insert table
14. Search text (Ctrl + F)
15. Show current *FlowDocument* as text

The second toolbar:

1. Select font family from a combo box
2. Select font size from a combo box (only sizes from a predefined list)
3. Set bold text (Ctrl + B)
4. Set italic text (Ctrl + I)
5. Set underlined text (Ctrl + U)
6. Left align text in paragraph (Ctrl + L)
7. Center align text in paragraph (Ctrl + E)
8. Right align text in paragraph (Ctrl + R)
9. Right indentation (Tab)
10. Left indentation (Shift + Tab)
11. Foreground color to current color
12. Select current color

The component also has a status bar which has only two components. The first is an *Ellipse* which is colored red when the content of a document is changed. The *Shape's Fill* property is bound to a property *DocState*, which indicates where the document is changed. The other component is a *Label* component which is bound to a property *Filename* and as so shows the name a file opened in the editor.

You should note that the component actually has a large number of other functions that are built into a *RichTextBox* and are activated using short keys. When I mentioned above that the component lacks a lot in being a real word processing program, it is partly because programs such as Work have many functions that are not supported in this component and partly because several of the functions do not support the flexibility that you often want require. As an example are the functions to insert lists limited as you only can inserts lists with a

fixed bullet and a fixed number format and only in one level. More important is the table function limited as the table always has the same format (look and feel), and you cannot add or remove columns, merge cells and so on. Another limitation is that the component should have some kind of settings where you could define default settings and more.

All that means that there is room for improvements, and if you have the desire and time, just take it. The current component should show how.

The toolbars have 25 buttons and two combo boxes. All buttons are bound to a command and in the following I will mention the most important of this commands.

The first button in the first toolbar clear the document and the button must be bound to a command. The class *ApplicationCommands* defines many commands used for standard operations in a program. To use such a command the command should be defined in XML:

```
<UserControl.CommandBindings>
  <CommandBinding Command="ApplicationCommands.New" Executed="New" />
  <CommandBinding Command="ApplicationCommands.Open" Executed="Open" />
  <CommandBinding Command="ApplicationCommands.Save" Executed="Save" />
  ...
```

where you must defined the method which should be performed when the command is activated. In this case the method is:

```
private void New(object sender, ExecutedRoutedEventArgs e)
{
  if (DocState.Equals(Brushes.Beige) ||
    MessageBox.Show(...) == MessageBoxResult.Yes)
  {
    Filename = null;
    editor.Document.Blocks.Clear();
    DocState = Brushes.Beige;
  }
}
```

If the document is modified it opens a message box and ask the user if the document should be saved before it is cleared. You should note how to clear the document be removing all blocks. As the next command I will show the command *SaveAS* which also is an *ApplicationCommands* command:

```
private void SaveAs(object sender, ExecutedRoutedEventArgs e)
{
  SaveFileDialog dlg = new SaveFileDialog();
  dlg.Filter = "Text base64 Format (*.b64)|*.b64|All files (*.*)|*.*";
  if (dlg.ShowDialog() == true)
  {
    string data = GetContent(editor.Document);
    StreamWriter writer = new StreamWriter(dlg.FileName);
    writer.WriteLine(data);
    writer.Close();
    Filename = dlg.FileName;
    DocState = Brushes.Beige;
  }
}

public string GetContent(FlowDocument document)
{
  MemoryStream ms = new MemoryStream();
  TextRange range = new TextRange(document.ContentStart, document.
  ContentEnd);
  range.Save(ms, dataFormat);
  return Convert.ToBase64String(ms.ToArray());
}
```

The method opens a *Save File* dialog box to select where to save the document. Note the filter as I want to save the document in Base 64 formatted document. The reason is that a document can contain images, and in some situations it can cause problems that a document can contain binary data. If the user select / enter a filename the method *GetContent()* is used the content of the document to a string and base 64 encoded. Note that as data format is used

```
private string dataFormat = DataFormats.XamlPackage;
```

and the document as so is formatted in a XML format. In some situations it is a better format than RTF. When the content of a document is converted to string it is written to the file.

The command to open a file are in principle

```
private void Open(object sender, ExecutedRoutedEventArgs e)
{
  OpenFileDialog dlg = new OpenFileDialog();
  dlg.Filter = "Text base64 Format (*.b64)|*.b64|All files (*.*)|*.*";
  if (dlg.ShowDialog() == true)
  {
    StreamReader reader = new StreamReader(dlg.FileName);
    string data = reader.ReadToEnd();
    SetContent(data, editor.Document);
    Filename = dlg.FileName;
    reader.Close();
    DocState = Brushes.Beige;
  }
}

private void SetContent(string data, FlowDocument flowDocument)
{
  byte[] content = Convert.FromBase64String(data);
  MemoryStream ms = new MemoryStream(content);
  TextRange range =
    new TextRange(flowDocument.ContentStart, flowDocument.ContentEnd);
  range.Load(ms, dataFormat);
}
```

The most important to note is the method *SetContent()* which convert the base 64 encode string read in the file and update to document.

There is also the command for save, but I will note show the code for that command here. When creating a component like the current one, one can discuss whether commands for file operations should be part of the component or not. One can easily imagine using the component where it does not make sense to save the document in a file. The component therefore has a property *FileAccess* of the type *Visibility* and the three buttons for the file operations have their *Visibility* property bound to the property *FileAcces*. This way you can specify whether the three buttons should be displayed or not.

To add commands to the next 5 buttons in the first toolbar trivial is trivial as they are defined as *ApplicationCommands* commands and a *RichTextBox* implements all this commands. In XML the commands are defined in the same way as above and bound to the current button. As an example the method for the *Copy* button is defined in code behind as:

```
private void Copy(object sender, ExecutedRoutedEventArgs e)
{
  editor.Copy();
}
```

where *editor* is the name of the *RichTextBox* component.

The next button to insert a bullet list is more complicated as I here has to create a new command. The command is called *BulletsCommand* and executes a method *InsertList()*. First I have to decide how the function should work:

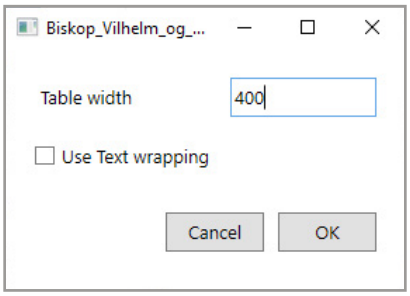
- If one or more paragraphs are selected these paragraphs must be changed to a list.
- If the paragraphs is always a list, the markers for the list must be removed.
- If the cursor is on an empty line the result should be an empty list.
- The case where the document is empty is a special case.

It requires a lot of code, and I will not show the code here as it takes up a lot of space and primarily consists of a number of details regarding how to reference elements in the document. The code is documented so that it should be possible to follow what is happening. The method is defined as:

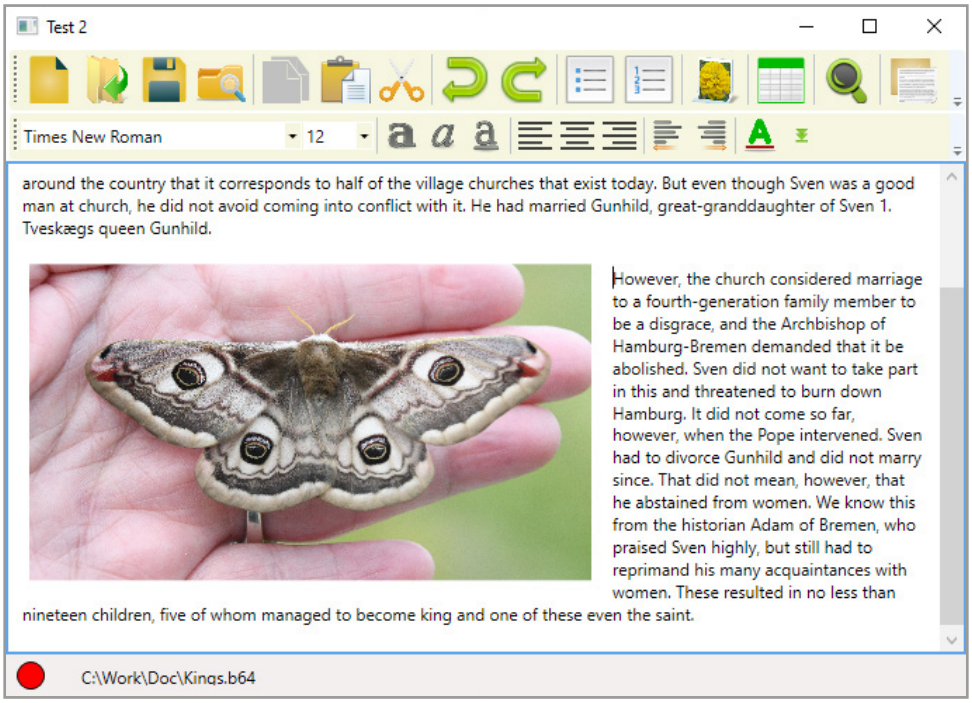
```
private void InsertList(TextMarkerStyle marker)
{
  ...
```

where the parameter defines the bullet type used for the list. This means that the method can be used both an unnumbered and a numbered list.

The next button is used to insert an image in the document. If you clicks the button the components opens a file browser where you can select the image, and if you do the program opens the following dialog box:



You must select the width of the image. You can also select text wrapping, and the image is inserted in the next paragraph in the upper left corner, but such the text wraps around the image:



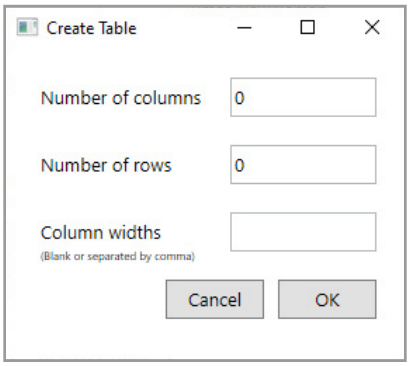
If you do not select text wrapping the image is inserted below the paragraph with the caret and center in the paragraph. To implement this function I have added a new command called *ImagesCommand*. The command calls a method *LoadImage()* which opens the file browser, and if the user selects a file, the method fires an event, and the event handler has the job to insert the image into the document:

```
public void SelectedImage(object sender, ImageEventArgs e)
{
  // use data from the dialog box to create an Image object.
  ImageModel data = e.Model;
  Image img = new Image { Width = data.Width };

  // load the image as a BitmapImage and use the bitmap as Source
  // for the image object
  BitmapImage bmp = new BitmapImage();
  bmp.BeginInit();
  bmp.UriSource = new Uri(data.Filename, UriKind.Absolute);
  bmp.EndInit();
  img.Source = bmp;
  // find the block for the caret position
  TextPointer tp = editor.CaretPosition;
  Block block = editor.Document.Blocks.Where(x => x.ContentStart.
  CompareTo(tp) ==
    -1 && x.ContentEnd.CompareTo(tp) == 1).FirstOrDefault();
  // if no caret position form the image is inserted last in the document
  if (block == null)
  {
    if (editor.Document.Blocks.Count == 0)
      editor.Document.Blocks.Add(new Paragraph());
    block = editor.Document.Blocks.Last();
    if (data.Wrap)
      editor.Document.Blocks.InsertBefore(block, WraphImage(img, data.
      Width));
    else editor.Document.Blocks.InsertBefore(block, new
    BlockUIContainer(img));
  }
  // the image is inserted in the block after the block with the caret
  else
  {
    if (data.Wrap)
      editor.Document.Blocks.InsertAfter(block, WraphImage(img, data.
      Width));
    else editor.Document.Blocks.InsertAfter(block, new
    BlockUIContainer(img));
  }
}
```

```
private Paragraph WraphImage(Image img, int width)
{
  Floater floater = new Floater { Width = width + 10,
  HorizontalAlignment =
    HorizontalAlignment.Left, Margin = new Thickness(0, 0, 10, 10) };
  floater.Blocks.Add(new BlockUIContainer(img));
  Paragraph p = new Paragraph();
  p.Inlines.Add(floater);
  return p;
}
```

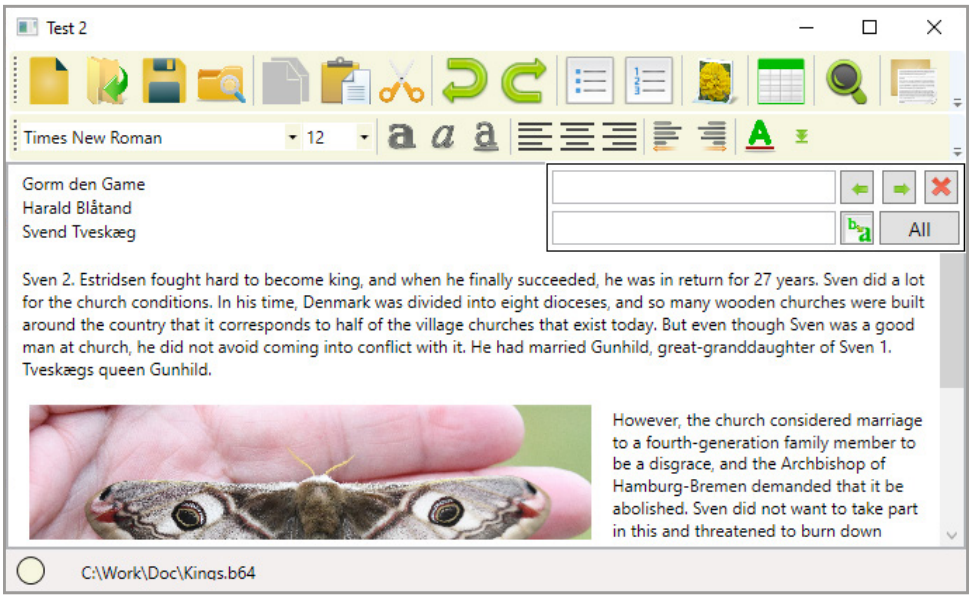
You must primarily note the use of the type *Floater* used to wrap an element.

The next button is used to insert a table. If you clicks the button the program opens the following dialog box:



You must enter the number of columns and the number of rows. If you do not enter something for column widths the current width of the window, is divided equally between the columns of the table. Otherwise you can enter column widths as a comma-separated list, where 0 means that the remaining space must be divided equally. The program inserts as default a border on 1 pixel around each cell, but the control has a dependency property which can be used to modify this value. I will not show the code to insert a table as it in principle works in the same way as the code which inserts an image.

The next function is search, and if you click on the button the control opens a search box:



It is a user control called *SearchControl*. The upper *TextBox* is used to enter the search text, and you can search forward and backward. The other *TextBox* is used to enter a text for replace the selected text, and the button *All* replaces all instances of search text with the replace text.

The user control is added to the editor in XML:

```
<tools:SearchControl x:Name="find"  ... Visibility="Collapsed" />
```

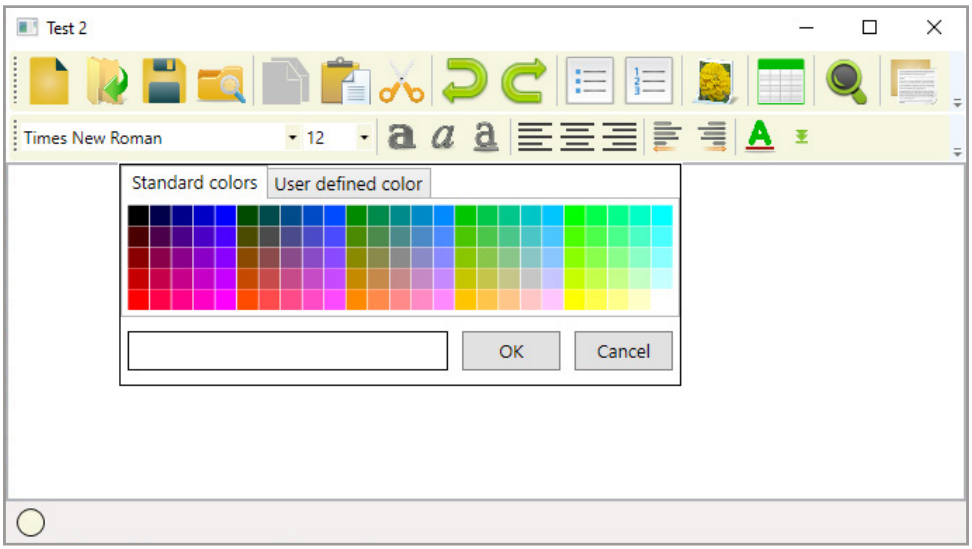
where it starts its life as collapsed. The user control fires an event when the user clicks on one of the two search buttons (the green arrows), and another event when the user clicks one of the two replace buttons. The code for the user control *SearchControl* is simple and I should not show it here, but the two event handlers for events from the controller fills, as it must be possible to navigate through the document, find the text searched and maybe replace it. It's not quite simple and requires several methods related to a *FlowDocument*, but once you see the code it's easy enough to figure out what's going on. Since it all takes up a lot of space, I will just show the code that is used to search forward.

```
public void TextSearched(object sender, SearchEventArgs e)
{
  if (e.IsNext)  // indicateto search foreward
  {
    // current is an instance variable that hold the position of the
    paragraph
    // where to start the search
    // if current is null it indicate a new search operation
    if (current == null)
    {
      TextRange text = new TextRange(editor.Document.ContentStart,
        editor.Document.ContentEnd);
      current = text.Start.GetInsertionPosition(LogicalDirection.Forward);
    }
    else current = current.GetNextContextPosition(LogicalDirection.
    Forward);
    // search for a paragraph which contains the search text
    while (current != null)
    {
      string textInRun = current.GetTextInRun(LogicalDirection.Forward);
      if (!string.IsNullOrWhiteSpace(textInRun))
      {
        int index = textInRun.IndexOf(e.Text);
        if (index != -1)
        {
          TextPointer selectionStart =
            current.GetPositionAtOffset(index, LogicalDirection.Forward);
          TextPointer selectionEnd =
            selectionStart.GetPositionAtOffset(e.Text.Length,
              LogicalDirection.Forward);
          // find the search text, select the text and ensure that the
          text is
          // visibile by scrolling the document
          TextRange selection = new TextRange(selectionStart, selectionEnd);
          editor.Selection.Select(selection.Start, selection.End);
          ScrollTo(selectionStart);
          // break the search and you are ready for next search
          break;
```

```
        }
      }
      current = current.GetNextContextPosition(LogicalDirection.Forward);
    }
    // the text is not found
    if (current == null) MessageBox.Show(...);
  }
  else
  {
    ...
  }
}
```

The last button in the first toolbar opens a text box which shows the current *FlowDocument* as text (formatted to XML). I doubt that a "real" editor should have such a function, but in connection with development it can be good to be able to see what the *FlowDocument* you are working on looks like. I do not want to show the code here, but it is basically simple and consists mostly of formatting to *FlowDocument* to XML.

Then there is the last toolbar, but I should not associate many comments to this toolbar as there is not much new relative to what is explained in the previous program and as the component *RichTextBox* has most of the functions built-in. The only exception if the two buttons for set a foreground color. If you select some text and clicks the color button the text is colored by the selected color which as default is gray. If you click on the green arrow the controls opens the color picker from the previous section, and you can select a new color.



The color picker is in the same way as the search user control added as a component which is collapsed until the user clicks the arrow.

The project *PaGUI* has a test project which use the component. It is a simple project which only instantiates and object of the type *PaTextEditor* and add this component to a *Grid*:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    grid.Children.Add(new PaTextEditor() { FileAccess = Visibility.
    Visible });
  }
}
```

Of course, you can also instantiate the object directly in XML.

4 PRINT

The topic of this chapter is add print facilities to a program so that one can print a document on a physical printer. As mentioned earlier, it is a topic that does not have the same meaning as before simply because today we read more online and on the computer screen than before. Still, there are still programs that need to be able to write to the printer so hence this chapter on what it takes.

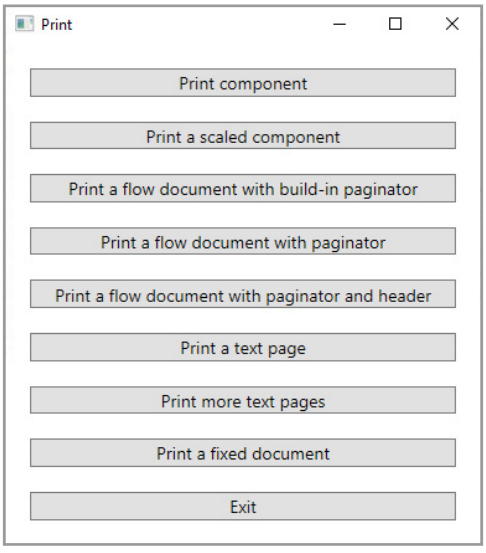
When developing programs with printing facilities, it is recommended to install a PDF printer which is a program that prints a document as a PDF file instead of as paper pages on a real physical printer. This saves paper and you is free to run to the printer all the time to check the result. An example is *CutePDF*.

There are several types of print jobs, and in the following I will look at three types of jobs each with their challenges:

1. Print a drawing or figure
2. Print of text
3. Print a table

Printing a figure is relatively simple, while printing both text and data in tabular form even with the support of the framework can be quite complex. The complexity is due to the fact that you have to control how much can be on a line, and that the text may continue on the next line or cut off, and one must also ensure that a new page starts in an appropriate place.

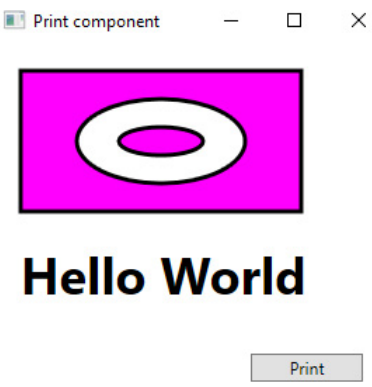
The project *PrintProject* is a WPF program that shows a number of examples of print jobs and thus print of various components and documents. If you run the program, you get the following window:



where each of the top 8 buttons shows an example of a print problem.

4.1 PRINT COMPONENT

I want to start simple. The first example opens the following window:



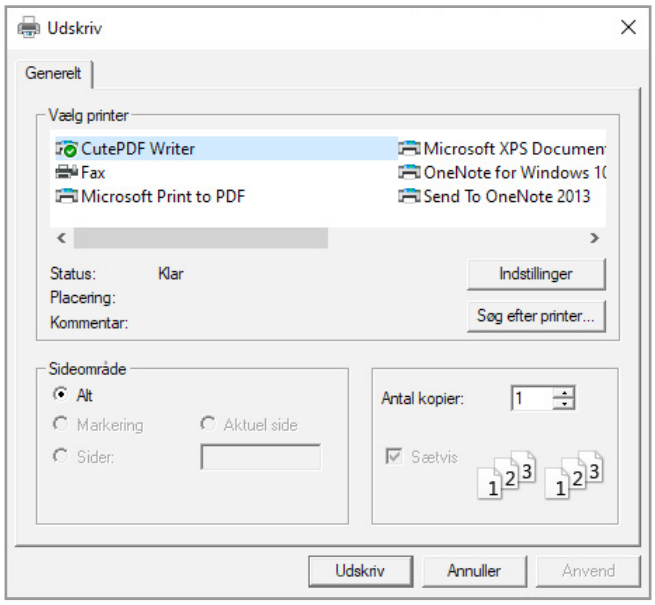
where there is a figure consisting of a rectangle with an ellipse inside - again with an ellipse inside and then a text:

```
<Grid Margin="20">
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <Canvas Name="page">
    <Path Fill="Magenta" Stroke="Black" StrokeThickness="3" Canvas.Top="0"
      Canvas.Left="0" >
      <Path.Data>
        <GeometryGroup>
          <RectangleGeometry Rect="0 0 200 100"/>
          <EllipseGeometry Center="100 50" RadiusX="60" RadiusY="30"/>
          <EllipseGeometry Center="100 50" RadiusX="30" RadiusY="10"/>
        </GeometryGroup>
      </Path.Data>
    </Path>
    <TextBlock Canvas.Top="120" Canvas.Left="0" FontSize="36"
      FontWeight="Bold" Text="Hello World"/>
  </Canvas>
  <Button Grid.Row="2" Click="cmdPrint_Click" Width="80"
    HorizontalAlignment="Right" Content="Print"/>
</Grid>
```

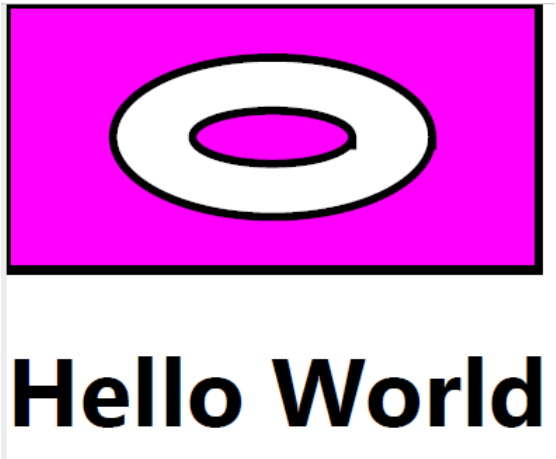
Finally, the window has a button, and the event handler is:

```
private void cmdPrint_Click(object sender, RoutedEventArgs e)
{
  PrintDialog printDialog = new PrintDialog();
  if (printDialog.ShowDialog() == true) printDialog.PrintVisual(page,
  "Print 1");
}
```

If you run the program and click on the button, you get a usual print dialog box:



The result is that the figure is printed on the selected printer, where the figure is printed relative to the upper left corner:

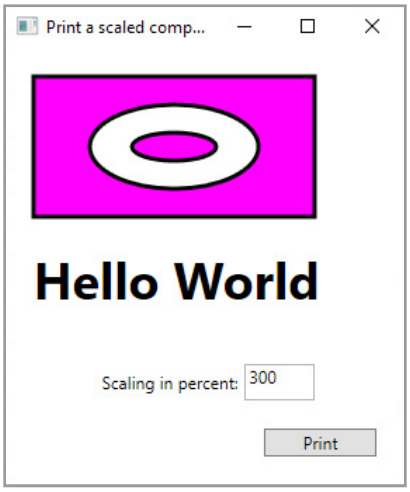


Here you must, among other things note that the name of the *Canvas* object defined in XML is *page*. When you click on the button the event handler creates a *PrintDialog* object, which is the dialog box for the print window. If the user in the dialog box clicks OK (the *ShowDialog()* method returns true), then the *PrintDialog* object's *PrintVisual()* method is used to print the component *page*. The last parameter is simply the name of the print in the print queue.

It cannot be simpler, and it can be stated that direct printing of a component is extremely simple, but of course there is also a need to take into account e.g. margin, and whether paper has room for all the figure.

4.2 PRINT SCALED COMPONENT

The next example is similar to the above, and the difference is that there is an input field where you can enter a scaling percentage, which by default is 300. When you click the *Print* button, the figure is printed as in the previous example, but before that it is scaled. In addition, this time a margin of 100 is defined.



The window is called *Print2Window* and in terms of the XML part, there is not much new. On the other hand, the event handler is more comprehensive:

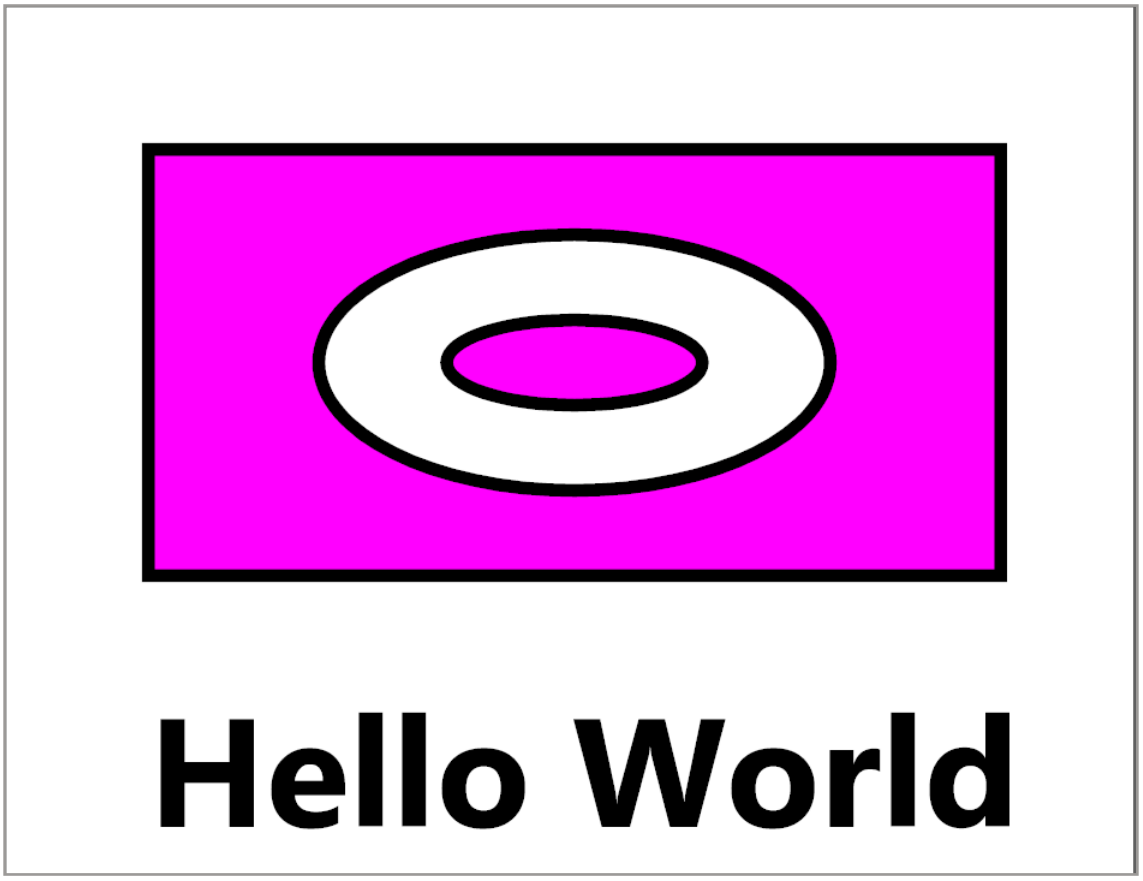
```
private void cmdPrint_Click(object sender, RoutedEventArgs e)
{
  PrintDialog printDialog = new PrintDialog();
  if (printDialog.ShowDialog() == true)
  {
    double scale;
    if (Double.TryParse(txtScale.Text, out scale))
    {
      double width = page.ActualWidth;
      double height = page.ActualHeight;
      SetPage(new Point(100, 100),
            new Size(printDialog.PrintableAreaWidth - 200,
                     printDialog.PrintableAreaHeight - 200),
            new ScaleTransform(scale / 100, scale / 100));
      printDialog.PrintVisual(page, "Print 2");
      SetPage(new Point(0, 0), new Size(width, height), null);
    }
    else
    {
      MessageBox.Show("Invalid scale value.");
    }
  }
}

private void SetPage(Point point, Size size, Transform trans)
{
  page.LayoutTransform = trans;
  page.Measure(size);
  page.Arrange(new Rect(point.X, point.Y, size.Width, size.Height));
}
```

Notice first the last method which lays out the figure *page* in its container. First, a possible transformation. Next, the size of the figure is stated using the *Measure()* method, and finally the figure is laid out in the container using the *Arrange()* method.

Otherwise, in the same way as in the first example, a *PrintDialog* object is created, and if the user clicks *OK* and has entered a legal scaling, the figure is printed. First, the original size of the figure is saved in two variables. This is because after the figure is scaled and moved, these operations will have an effect on the screen, and the change must therefore be set back after the figure is printed. In the next step *SetPage()*is called with three parameters:

1. The coordinates of the upper left corner of the figure on the paper. It is here (100, 100), which corresponds to a margin of 100.
2. The size of the figure, which is determined by the size of the paper and the current margin. You should note how to determine the paper size as properties of the *PrintDialog* object.
3. How to scale the figure and here with a *ScaleTransform*.

Next, the page is printed, and as the last is called *SetPage()* again, so that the shape of the figure on the screen is unchanged.

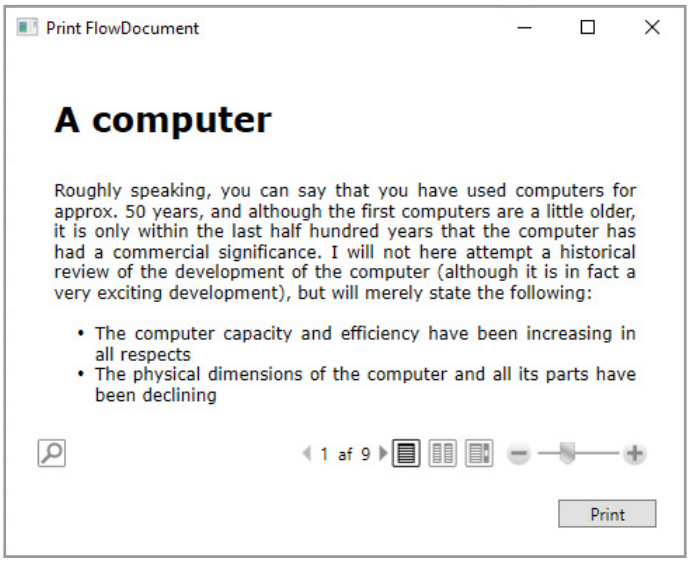


4.3 PRINT A FLOWDOCUMENT WITH BUILD IN PAGINATOR

The next example shows how to print a document - in this case a *FlowDocument*. It is a document defined as *xaml* and the project has a flow document called *TheComputer.xaml*, and the beginning of the file is:

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation" ColumnWidth="400" FontSize="12" FontFamily="Verdana">
  <Paragraph FontSize="24" FontWeight="Bold">
    A computer
  </Paragraph>
  <Paragraph>
    Roughly speaking, you can say that you have used computers for
approx. 50 years, and although the first computers are a little older,
it is only within the last half hundred years that the computer has
had a commercial significance. I will not here attempt a historical
review of the development of the computer (although it is in fact a
very exciting development), but will merely state the following:
  </Paragraph>
  <List>
    <ListItem>
      <Paragraph>The computer capacity and efficiency have been
      increasing in all respects</Paragraph>
    </ListItem>
```

You are encouraged to study the contents of the file. If you select the third print function in the program, you get the following window:



The text is displayed with a *FlowDocumentReader*. It allows you to display the text one page at a time, as shown above and by default. You can also display the text as two columns or as one long page that can be scrolled. It is also possible to zoom and search in the text. Clicking the *Print* button opens a print dialog box and the text can be printed.

The window's XML is:

```
<Grid Margin="20">
  <Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
  </Grid.RowDefinitions>
    <FlowDocumentReader Name="reader"/>
  <Button Command="ApplicationCommands.Print"
    CommandTarget="{Binding ElementName=reader}" Margin="0,20,0,0"
    Grid.Row="1"
    Width="70" HorizontalAlignment="Right" Content="Print"/>
</Grid>
```

Here, of course, the most important is the component *FlowDocumentReader*, which here has the name *reader*. The document itself is assigned in the code, but since a flow document is XML, there is nothing to prevent the document from being written inline in XML. You should also notice the button, which uses a standard print command, which is bound to the reader object. This means, among other things, that it is this standard command that is responsible for the page division, and since no one is specified, there is e.g. no margin. Also note that the document is so large that it takes up two pages.

The code has no event handler, but the file with the text must be loaded. I have set its property so that it is automatically copied to the program library, and the code can therefore be written as follows:

```
public partial class Print3 : Window
{
  public Print3()
  {
    InitializeComponent();
    string path =
      System.IO.Path.Combine(Directory.GetCurrentDirectory(),
      "TheComputer.xaml");
    using (FileStream fs = File.Open(path, FileMode.Open))
    {
      FlowDocument doc = (FlowDocument)XamlReader.Load(fs);
      reader.Document = doc;
    }
  }
}
```

It is simple and consists only of setting the reader's *Document* property to a *FlowDocument*, which is initialized by the contents of the file.

It can hardly be simpler, but in return it is limited what options I have for managing the finished result. That is the subject of the next two examples.

4.4 PRINT A FLOW DOCUMENT WITH PAGINATOR

It is almost the same example as above. It prints the same document, and if you open the window, it is almost identical to the previous example. There are two differences. In XML, I have instead used a *FlowDocumentPageViewer*, and instead of a standard command, this time the button has an event handler:

```
<FlowDocumentPageViewer Name="viewer" />
<Button Margin="0,20,0,0" Grid.Row="1" Width="70"
HorizontalAlignment="Right"
  Content="Print" Click="cmdPrint_Click"/>
```

The latter means that it is the programmer who has to write the code for the event handler:

```
public partial class Print4 : Window
{
  public Print4()
  {
    InitializeComponent();
    string path =
      System.IO.Path.Combine(Directory.GetCurrentDirectory(),
      "TheComputer.xaml");
    using (FileStream fs = File.Open(path, FileMode.Open))
    {
      FlowDocument doc = (FlowDocument)XamlReader.Load(fs);
      viewer.Document = doc;
    }
  }
```

```
  private void cmdPrint_Click(object sender, RoutedEventArgs e)
  {
    PrintDialog printDialog = new PrintDialog();
    if (printDialog.ShowDialog() == true)
    {
      FlowDocument doc = viewer.Document as FlowDocument;
      double height = doc.PageHeight;
      double width = doc.PageWidth;
      Thickness padding = doc.PagePadding;
      double gap = doc.ColumnGap;
      double space = doc.ColumnWidth;
      doc.PageHeight = printDialog.PrintableAreaHeight;
      doc.PageWidth = printDialog.PrintableAreaWidth;
      doc.PagePadding = new Thickness(50);
      doc.ColumnGap = 25;
      doc.ColumnWidth = (doc.PageWidth - doc.ColumnGap - doc.
      PagePadding.Left -
                          doc.PagePadding.Right) / 2;
      printDialog.PrintDocument(
        ((IDocumentPaginatorSource)doc).DocumentPaginator, "Print 4");
      doc.PageHeight = height;
      doc.PageWidth = width;
      doc.PagePadding = padding;
      doc.ColumnGap = gap;
      doc.ColumnWidth = space;
    }
  }
}
```

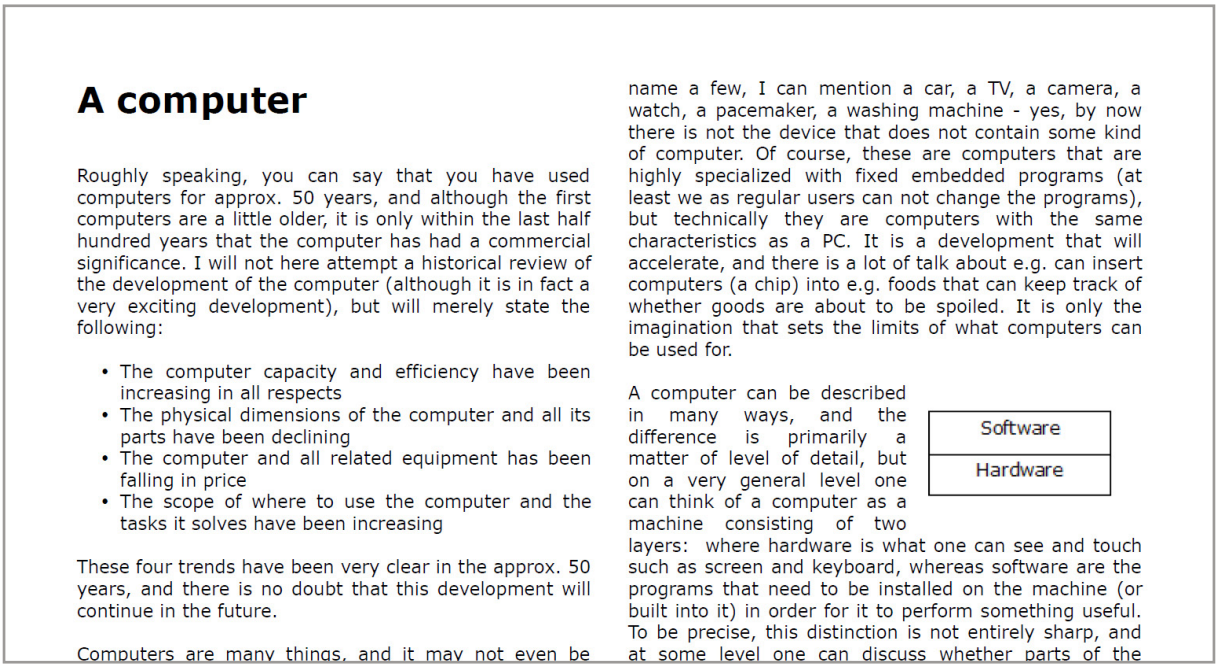
In the code, the following happens. The document must be loaded and linked to the viewer, which is done in exactly the same way as in the previous example. The event handler starts by setting a reference to the document. Next, the following values are saved in variables

1. height and width of the document
2. document margin in the form of the variable padding
3. the gap between the two columns (variable gap)
4. column width (space variable)

These 5 values are then defined according to the size of the paper, a margin of 50, a gap between columns of 50, while the column width is set to the remaining space divided by 2. Changing these values means changing values at the document paginator, which is an object used to perform page layout.

After the document is printed, the document settings must be returned to the original settings.

If you try the example, the result will be as shown below. Here you must, among other things note that there is a margin, as well as that text is wrapped around the image.



A computer

Roughly speaking, you can say that you have used computers for approx. 50 years, and although the first computers are a little older, it is only within the last half hundred years that the computer has had a commercial significance. I will not here attempt a historical review of the development of the computer (although it is in fact a very exciting development), but will merely state the following:

- The computer capacity and efficiency have been increasing in all respects
- The physical dimensions of the computer and all its parts have been declining
- The computer and all related equipment has been falling in price
- The scope of where to use the computer and the tasks it solves have been increasing

These four trends have been very clear in the approx. 50 years, and there is no doubt that this development will continue in the future.

Computers are many things, and it may not even be name a few, I can mention a car, a TV, a camera, a watch, a pacemaker, a washing machine - yes, by now there is not the device that does not contain some kind of computer. Of course, these are computers that are highly specialized with fixed embedded programs (at least we as regular users can not change the programs), but technically they are computers with the same characteristics as a PC. It is a development that will accelerate, and there is a lot of talk about e.g. can insert computers (a chip) into e.g. foods that can keep track of whether goods are about to be spoiled. It is only the imagination that sets the limits of what computers can be used for.

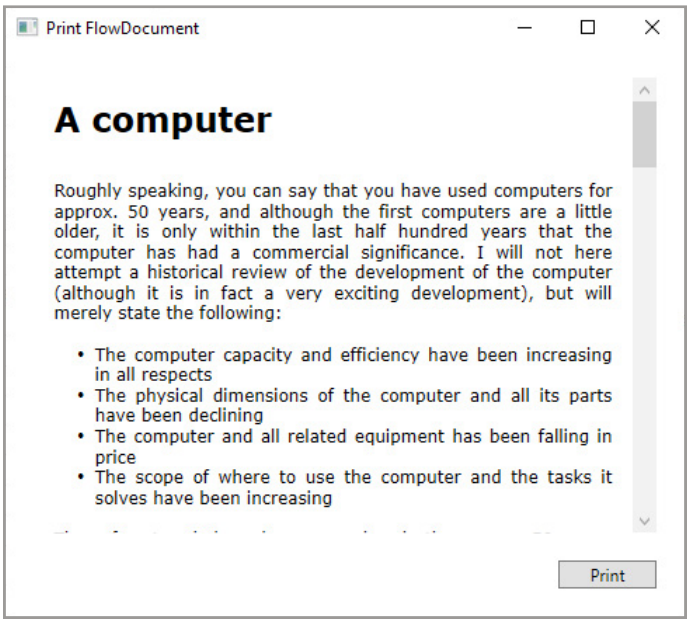
A computer can be described in many ways, and the difference is primarily a matter of level of detail, but on a very general level one can think of a computer as a machine consisting of two layers: where hardware is what one can see and touch such as screen and keyboard, whereas software are the programs that need to be installed on the machine (or built into it) in order for it to perform something useful. To be precise, this distinction is not entirely sharp, and at some level one can discuss whether parts of the

4.5 PRINT A FLOW DOCUMENT WITH PAGINATOR AND HEADER

The example is the same as the previous two, and it is the same document that is printed, but this time a FlowDocumentScrollViewer is used:

```
<FlowDocumentScrollViewer Name="viewer"/>
```

It provides a slightly simpler user interface:



where the only thing one can do is scroll the document.

The document is loaded in the same way as in the other examples, and the *Print* button's event handler is very similar to the above, but is simpler:

```
private void cmdPrint_Click(object sender, RoutedEventArgs e)
{
  PrintDialog printDialog = new PrintDialog();
  if (printDialog.ShowDialog() == true)
  {
    FlowDocument doc = viewer.Document;
    double height = doc.PageHeight;
    double width = doc.PageWidth;
    Thickness padding = doc.PagePadding;
    doc.PageHeight = printDialog.PrintableAreaHeight;
    doc.PageWidth = printDialog.PrintableAreaWidth;
    doc.PagePadding = new Thickness(50, 100, 50, 40);
    HeaderPaginator paginator = new HeaderPaginator(doc, doc.PagePadding);
    printDialog.PrintDocument(paginator, "Print 5");
    doc.PageHeight = height;
    doc.PageWidth = width;
    doc.PagePadding = padding;
  }
}
```

In particular, note that a paginator of the type *HeaderPaginator* is created that is used to print the document. That is instead of using a default paginator, it is the user who takes care of the page layout. A paginator is a class that inherits the class *DocumentPaginator*. The constructor has two parameters, the first being the flow document to be printed, while the second is the page margin in the form of a *Thickness* object:

```
public class HeaderPaginator : DocumentPaginator
{
  private DocumentPaginator paginator;
  private Thickness padding;

  public HeaderPaginator(FlowDocument document, Thickness padding)
  {
    paginator = ((IDocumentPaginatorSource)document).DocumentPaginator;
    this.padding = padding;
  }

  public override DocumentPage GetPage(int pageNumber)
  {
    DocumentPage page = paginator.GetPage(pageNumber);
    ContainerVisual visual = new ContainerVisual();
    visual.Children.Add(page.Visual);
    DrawingVisual header = new DrawingVisual();
    using (DrawingContext context = header.RenderOpen())
    {
      Typeface typeface = new Typeface("Verdana");
      FormattedText text = new FormattedText("Side " + (pageNumber +
      1).ToString(),
        CultureInfo.CurrentCulture, FlowDirection.LeftToRight, typeface,
        14,
        Brushes.Black);
      context.DrawText(text, new Point(padding.Left, padding.Top / 2));
    }
    visual.Children.Add(header);
    DocumentPage newPage = new DocumentPage(visual);
    return newPage;
  }

  public override bool IsPageCountValid
  {
    get { return paginator.IsPageCountValid; }
  }

  public override int PageCount
  {
    get { return paginator.PageCount; }
  }
```

```
  public override Size PageSize
  {
    get { return paginator.PageSize; }
    set { paginator.PageSize = value; }
  }

  public override IDocumentPaginatorSource Source
  {
    get { return paginator.Source; }
  }
}
```

The class must override the method *GetPage()*, as well as four properties that specify whether all pages are printed, the number of pages, the page size, and the document to be printed. The important thing is of course *GetPage()*, whose task in this case is to add a header with page numbers. The code may be a little difficult to figure out, but the method is performed for each page, and the parameter is the page number, and in short, the code means the following:

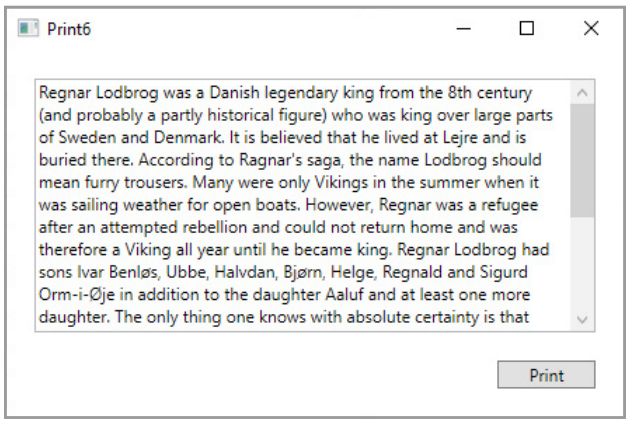
- First, a reference to the current page is set in the form of a *DocumentPage* object (called the encapsulated page's *GetPage()*).
- Next, a container for Visual objects is created and the current page is added to the container.
- As the next item, a header of the type *DrawingVisual* is created, which is to be used for the page header with the page number.
- The page number is then formatted with a 14-point Verdana font.

Finally, the header is added to that container, and the method returns a page based on the contents of the container.

4.6 PRINT A TEXT PAGE

If you select this function, you will get the following window, which only contains a *TextBox* and a button:

```
<Grid Margin="20">
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <TextBox Name="txtText" TextWrapping="Wrap" AcceptsReturn="True" Ver
  ticalScrollBarVisibility="Visible">
Regnar Lodbrog var en dansk sagnkonge fra det 8. århundrede ....
  </TextBox>
  <Button Grid.Row="1" Margin="0,20,0,0" Click="cmdPrint_Click"
Content="Print" Width="70" HorizontalAlignment="Right"/>
</Grid>
```



There is nothing to explain about user interface, but note that you can edit the text. The event handler for the button is

```
private void cmdPrint_Click(object sender, RoutedEventArgs e)
{
  PrintDialog printDialog = new PrintDialog();
  if (printDialog.ShowDialog() == true)
  {
    DrawingVisual visual = new DrawingVisual();
    using (DrawingContext dc = visual.RenderOpen())
    {
      FormattedText text = new FormattedText(txtText.Text,
        CultureInfo.CurrentCulture, FlowDirection.LeftToRight,
        new Typeface("Verdana"), 14, Brushes.Black);
```

```
      double margin = 96;
      text.MaxTextWidth = printDialog.PrintableAreaWidth - 2 * margin;
      Size size = new Size(text.Width, text.Height);
      Point point = new Point(margin, margin);
      dc.DrawText(text, point);
      dc.DrawRectangle(null, new Pen(Brushes.Black, 1),
        new Rect(margin, margin, printDialog.PrintableAreaWidth -
        margin * 2,
                 printDialog.PrintableAreaHeight - margin * 2));
    }
    printDialog.PrintVisual(visual, "Print 6");
  }
}
```

The example should show how to easily print text using a TextBox.

4.7 PRINT MORE TEXT PAGES

This example shows two things:

1. How to print directly without first displaying data in a view.
2. How to print data organized in rows and columns.

Name	From	To
Regnar Lodbrog		
Gorm den Gamle		958
Harald 1. Blåtand	958	986
Svend 1. Tveskæg	986	1014
Harald 2.	1014	1018
Knud 1. den Store	1018	1035
Hardeknud	1035	1042
Magnus den Gode	1042	1047
Svend 2. Estridsen	1047	1074

where I have used the list of Danish kings as an example. The important thing is that the list is so long that it requires two pages.

All the code is in *MainWindow*, and then of course there is the *King* class and the class *Kings* (both classes simplified a bit compared to before). The largest part of the code is the class *KingsPaginator*, which is the class that builds the pages:

```
public class KingsPaginator : DocumentPaginator
{
  private Kings kings;
  private Typeface face;
  private double size;
  private double margin;
  private int pageCount;
  private Size pageSize;
  private int rows;
  private double width;
  private double height;

  public KingsPaginator(Kings kings, Typeface face, double size,
  double margin,
    Size pageSize)
  {
    this.kings = kings;
    this.face = face;
    this.size = size;
    this.margin = margin;
    this.pageSize = pageSize;
    Paginate();
  }

  public override bool IsPageCountValid
  {
    get { return true; }
  }

  public override int PageCount
  {
    get { return pageCount; }
  }

  public override Size PageSize
  {
    get
    {
      return pageSize;
    }
    set
```

```
  {
    pageSize = value;
    Paginate();
  }
}

public override IDocumentPaginatorSource Source
{
  get { return null; }
}

private void Paginate()
{
  FormattedText text = GetFormattedText("A");
  width = text.Width;
  height = text.Height;
  rows = (int)((pageSize.Height - margin * 2) / height) - 1;
  pageCount = rows > 0 ? (int)Math.Ceiling((double)kings.Count /
  rows) : 1;
}

public override DocumentPage GetPage(int pageNumber)
{
  double col1 = margin;
  double col2 = col1 + width * 25;
  double col3 = col2 + width * 6;
  int min = pageNumber * rows;
  int max = Math.Min(min + rows, kings.Count);
  DrawingVisual visual = new DrawingVisual();
  Point point = new Point(margin, margin);
  using (DrawingContext dc = visual.RenderOpen())
  {
    Typeface headerFace = new Typeface(face.FontFamily, FontStyles.
    Normal,
      FontWeights.Bold, FontStretches.Normal);
    point.X = col1;
    FormattedText text = GetFormattedText("Name", headerFace);
    dc.DrawText(text, point);
    text = GetFormattedText("From", headerFace);
    point.X = col2;
    dc.DrawText(text, point);
```

```
      text = GetFormattedText("To", headerFace);
      point.X = col3;
      dc.DrawText(text, point);
      dc.DrawLine(new Pen(Brushes.Black, 2), new Point(margin, margin +
      height),
        new Point(pageSize.Width - margin, margin + height));
      point.Y += height;
      for (int i = min; i < max; i++)
      {
        point.X = col1;
        text = GetFormattedText(kings[i].Name);
        dc.DrawText(text, point);
        text = GetFormattedText(kings[i].From);
        point.X = col2;
        dc.DrawText(text, point);
        text = GetFormattedText(kings[i].To);
        point.X = col3;
        dc.DrawText(text, point);
        point.Y += height;
      }
    }
    return new DocumentPage(visual, pageSize, new Rect(pageSize),
         new Rect(pageSize));
  }

  private FormattedText GetFormattedText(string text)
  {
    return GetFormattedText(text, face);
  }

  private FormattedText GetFormattedText(string text, Typeface face)
  {
    if (text == null) text = "";
    return new FormattedText(text, CultureInfo.CurrentCulture,
       FlowDirection.LeftToRight, face, size, Brushes.Black);
  }
}
```

First, there are all the variables:

- *kings* refers to the data source and the data to be printed
- *face* is the font to be used
- *size* is the font size
- *margin* is the margin (left, top, right and bottom) to the page
- *pageCount* is the number of pages
- *pageSize* is the paper size

- *rows* is the number of rows
- *width* is a measure of the width of a character (exactly the width of a capital A in the selected font)
- *height* is the line height

All these variables are initialized partly in the constructor and partly in the *Paginate()* method. Regarding the latter, note how the last variables are initialized, how to calculate the number of rows, and then calculate the number of pages.

Then there is the method *GetPage()*, as where it all happens - it is the method that forms the individual pages, and the method is called every time a page is to be printed. The code is long, but briefly the following happens:

- First, variables for starting positions are defined for the three columns.
- Next, a range of the rows (min and max) that must be included on this page is calculated.
- The next point defines a *DrawingVisual* object as the page to be "drawn" as well as a point for the upper left corner of the paper.
- Using the context of this object, the drawing process starts:
- First, create a font for the header.
- Next, the three texts for the heading are formatted one at a time and drawn in the correct column positions.
- As the next step, draw a horizontal line across the paper.
- Then a loop runs across all lines, printing three texts in the correct columns for each repetition. Note in particular how the y-position of each repetition is modified by the line height.
- Finally, the method returns the current page in the form of a *DocumentPage* object.

Back there is the event handler, which does not contain anything new:

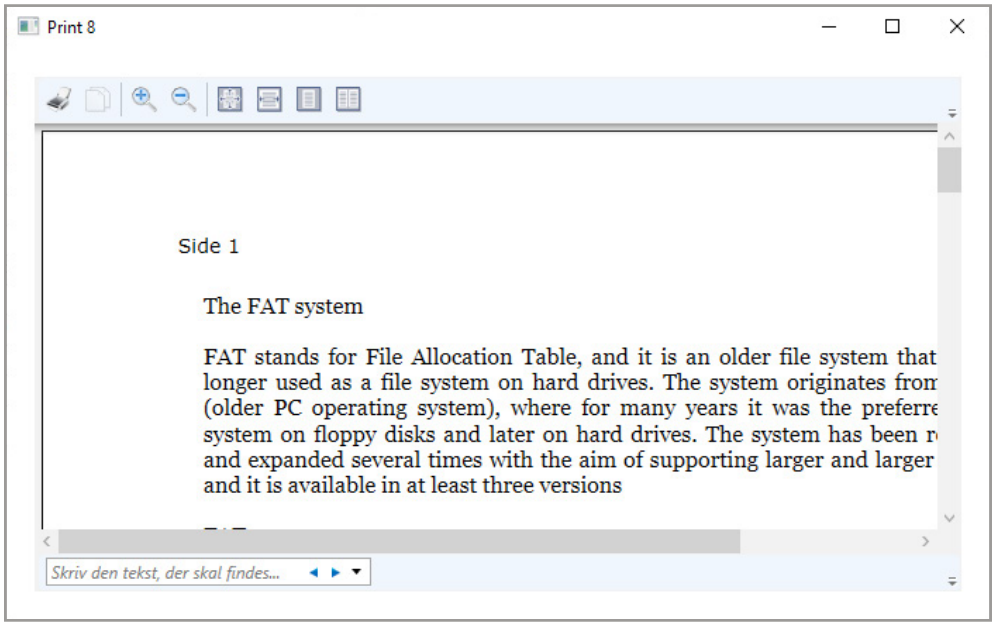
```
private void cmdPrint7_Click(object sender, RoutedEventArgs e)
{
  PrintDialog printDialog = new PrintDialog();
  if (printDialog.ShowDialog() == true)
  {
    Kings kings = new Kings();
    KingsPaginator paginator = new KingsPaginator(kings, new
    Typeface("Verdana"),
      24, 96 * 0.75, new Size(printDialog.PrintableAreaWidth,
      printDialog.PrintableAreaHeight));
    printDialog.PrintDocument(paginator, "Print 8");
  }
}
```

4.8 PRINT A FIXED DOCUMENT

The last example in this project contains several things:

1. How to use a *DocumentViewer*
2. How to print a plain text document
3. How to create an *Xps* document

The example opens the following window, which can be perceived as a print preview for a fixed document:




```
<Grid Margin="20">
  <DocumentViewer Name="docViewer"/>
</Grid>
```

The window is extremely simple and has only a single component, which is a *DocumentViewer*. The viewer shows the contents of a plain text file called *FAT.txt*, and the only interesting thing about it is that it is so large that it takes up two pages. The viewer itself provides several services, including facilities for zooming and searching, and as the most important option for printing the content. Thus, it is simple to print a document using a *DocumentViewer*, but it requires that the document is first converted to an *Xps* document, and this in turn is not entirely simple. In this case, the conversion takes place when the window is loaded. That is the conversion happens every time the window is opened, and this is of course not necessary.

The code is as follows:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  string filename = "Fat";
  Paragraph paragraph = new Paragraph();
  paragraph.Inlines.Add(File.ReadAllText(filename + ".txt"));
  FlowDocument flowDoc = new FlowDocument(paragraph);
  CreateXps(filename, flowDoc);
  XpsDocument doc = new XpsDocument(filename + ".xps", FileAccess.
  ReadWrite);
  docViewer.Document = doc.GetFixedDocumentSequence();
  doc.Close();
}

private void CreateXps(string filename, IDocumentPaginatorSource doc)
{
  using (Package container = Package.Open(filename + ".xps", FileMode.
Create))
  {
    using (XpsDocument xps = new XpsDocument(container,
    CompressionOption.Maximum))
    {
      XpsSerializationManager rsm =
        new XpsSerializationManager(new XpsPackagingPolicy(xps), false);
      DocumentPaginator paginator =
        ((IDocumentPaginatorSource)doc).DocumentPaginator;
      paginator = new DocumentPaginatorWrapper(paginator, new
      Size(793.7, 1122.5),
          new Size(96, 96));
      rsm.SaveAsXaml(paginator);
    }
  }
}
```

The content of the file (which here is just plain text) is loaded and encapsulated in a *Paragraph* object, and it is subsequently used to create a *FlowDocument*. The flow document is then passed over to a method that serializes the document as an *Xps* document. This process uses a custom *DocumentPaginatorWrapper* paginator. I will not show the type of this paginator here, but they are similar to other paginator classes that I have shown above. The result is a *Fat.xps* file, which is opened and assigned to the *DocumentViewer* object. A bit of a long road, but a lot actually happens, and in addition to how to use a *DocumentViewer* to print a document, you need to notice how to create an *Xps* file.

5 ANOTHER GRID

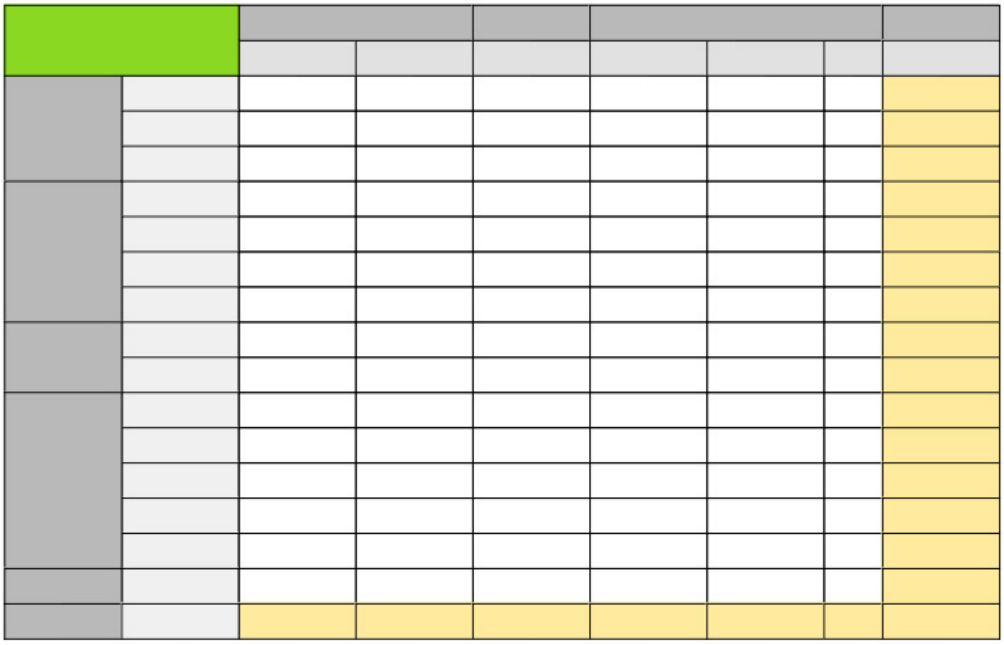
When you write programs with a graphic user interface you often need to arrange data in rows and columns. For that you in WPF has a *Grid* which is a layout control and you have a *DataGrid*. If your task for example is to show data extracted from a database, the *DataGrid* component is perfect, and it is exactly what the purpose of the component is, but even though the component is flexible with many properties and you have rich options to define using styles, how the individual elements should be shown, the component is not ideal for any kind of presentation of data on tabular form. If you for example must write a program which show a table of numbers as a spreadsheet the component does not offer everything you need. You could then try to solve the problem by arranges components in a *Grid*. Here it is possible to achieve the necessary flexibility, but for a table with many rows and columns performance would be a problem. As so there is a need for a more light weight grid, and you can online find a lot of examples of that kind of grids written as a component to use in a WPF application. In this chapter I will add yet another example of such a grid, partly because the component can be useful, but especially as yet another example of non trivial user component.

It is not simple to write such a component, and a solution will typically be a balancing as at least three considerations:

1. the component must be sufficiently flexible to have value in practice
2. use of the component must be simple enough for anyone who want to use it
3. the component must be effective, also for a grid with many rows and columns

5.1 AN OVERALL DESCRIPTION

The component should be designed as a custom control which divides the screen area for the component into 6 areas:



1. Center, which is the center area consisting of the white cells and is the actual grid. This area can scroll both horizontal and vertical.
2. Top, which are column headers. There can be several rows of column headers as the headers may be ordered in a hierarchy of column headers. In the figure above there are two header rows, where the first header in the first row spans two columns, the next only one and the third at least three columns. In principle the top area can contains all row headers needed. The top area can scroll horizontally.
3. Left, which are row headers. There can be several columns of row headers, as the headers may be ordered in a hierarchy of row headers. In the figure there are two header columns, where the first header in the first column spans three rows, the next four rows and so on. In principle the left area can contains all column headers needed. The left area can scroll vertically.
4. Right, which can contain one or more columns (the yellow area). It is normal columns like the columns in the center area and the only difference is that these columns are fixed and only scroll vertically.
5. Bottom, which can contain one or more rows. It is normal rows like the rows in the center area and the only difference is that these rows are fixed and only scroll horizontally.
6. The upper left corner and then the green area. It can in principle contain any visual object, but are as default empty.

Each of the four areas top, left, bottom and right may be empty, when the grid has no headers or fixed rows or columns, and if so the components does not shows these areas. If one of the header areas are empty, the corner will also be empty and are not shown.

The cells contained in both the center, right and bottom areas are called data cells and may span more columns and rows.

It is clear from the above description that the idea behind the component is a spreadsheet, but it is in no way a spreadsheet. It is a lightweight component on a much more basic level and is intended solely as a component for organizing data into rows and columns. That the component can then be used to implement a spreadsheet is a completely different matter.

5.2 THE COMPONENT'S DESIGN

The primary design element is a cell which represents both a data cell and a header cell, where

1. a cell in the top area, the left area and the corner is called a header cell
2. a cell in the center area, the right area and the bottom area is called a data cell

I expect that in most applications of the component both the right area and the area at the bottom will be empty, just as I expect that the top area will only have one header line and the left area one header column, but all options must of course be possible.

A cell should in principle consists of

1. an object representing the cell's value, and object that can be drawn on the screen
2. attributes that defines how the cell should be drawn, primarily used if the cell contains text

A cell is the defined as the following abstract class, where I think the names should explains what the properties are used for:

```
public abstract class Cell : FrameworkElement
{
  static Cell()
  {
    SelectionColor = Brushes.LightSkyBlue;
  }

  public int Row { get; set; }
  public int Col { get; set; }
  public int RowSpan { get; set; }
  public int ColSpan { get; set; }
  public TableAttributes Attribute { get; set; }
  public TableModel Model { get; set;}
  public virtual bool IsEditable { get; set; }

  public abstract void Draw(Graphics g, Rect rect);
  public abstract object GetValue();
  public abstract bool SetValue(object value);
  public abstract Cell Create(TableModel model);
  public abstract Cell Clone();
}
```

The class is defined as *FrameworkElement*, and you should note that the class has a reference to the row and column for its position in the grid. Here are used the following convention:

1. If both *Row* and *Col* are greater than or equal 0 it is a data cell.
2. If both *Row* and *Col* are -1 it means the upper left corner.
3. If *Row* is -1 and *Col* is greater than or equal 0 it means the first column header line, the column just above the data cells, which defines number of columns in the grid as well as the width for each column.
4. If *Row* is less than -1 and *Col* is greater than or equal 0 it means the next column header line with decreasing index upwards.
5. If *Col* is -1 and *Row* is greater than or equal 0 it means the first row header line, the row just to the left of the data cells, which defines number of rows in the grid as well as the height for each row.
6. If *Col* is less than -1 and *Row* is greater or equal 0 it means the next row header column with decreasing index to the left.

The class also has a property for a *TableAttributes* object which defines the attributes that can be used. Here it is decided that following attributes can be used:

1. Font family
2. Font size
3. Bold
4. Italic
5. Foreground
6. Background
7. Horizontal alignment (left, right, center)
8. Vertical alignment (top, bottom, center)
9. Decimals, as number of decimals for numeric data

Many other attributes could be defined, but here it has been decided to settle for the above. The class also has a property, that indicates where a cell should be editable. As the last property the class has a reference to a data model of the type *TableModel* which is explained below, but it is a class which represents the data that the grid must show.

The class defines five abstract methods. The first is the method which draws the cell on the screen. The first parameter is an encapsulation of a *DrawingContext* packed with some utility methods, and the second parameter defines the area within this cell must be drawn. The second method must return the data value for this cell while the third must be used to update the cell. This method must return where the cell is updated. The method *Create()* is used to create a new *Cell* object of the same type as the current concrete cell. Finally the last method creates and return a clone of this cell.

The class is a basic class for the program, and the idea is to write a subclass for all kinds of objects that it should be possible to show in the grid. The control creates three concrete classes:

1. *TextCell* which is a class for a cell which contains a text and is the default cell type.
2. *NumberCell* which is a class for a cell which contains a number of the type *double*.
3. *ImageCell* which is a class for a cell which contains an image defined as an *ImageSource* object.

Another important class is *TableModel* which is the component's data model. It is a comprehensive class but basically the model is defined as:

```
public class TableModel : INotifyPropertyChanged
{
  private List<List<Cell>> cells;

  public TableModel(string[] rows, string[] cols, Cell[,] cells,
    string[] rowHeaders, string[] colHeaders, int fixedRows = 0,
    int fixedCols = 0, bool hasRows = true, bool hasCols = true)

  public int FixedCols { get; set; }
  public int FixedRows { get; set; }
  public Cell Corner { get; set; }
  public TableCols TableCols { get; private set; }
  public TableRows TableRows { get; private set; }
  public List<ColHeader> ColHeaders { get; private set; }
  public List<RowHeader> RowHeaders { get; private set; }
  public bool HasPopups { get; set; }
  public Cell this[int r, int c]
```

The data cells are defined as a list of lists. When I have not used a 2-dimensional array, it is because it should be possible to add and remove rows and columns.

The class has more constructors to make it easy to create typical grids, as the main constructor has many parameters, where the last 5 has default values. If an array is *null* or the length is 0 the length of the array is perceived as infinity, and the size of the table is then defined as:

1. number of rows = Min(rows.Length, cell.GetLength(0)
2. number of cols = Min(cols.Length, cell.GetLength(1)

If one of these two values are less than 1 or is infinity, it indicates an error and the constructor throws an exception. Note that this means that a table must have at least one column and one row. If cell is *null* or empty all data cells are created as objects representing text, and that is as cells of the type *TextCell*.

A stacked row header or column header is defined as a semicolon separated string and consists of pairs as a text (the cell label) and an *int* indicating the span of items relative to the previous level (the lowest level is the row or column header). If there is an error when parsing, the rest of the header is used for an empty cell which spans the rest of the columns. If the string defines more cells or spans more columns than needed, they are ignored. The

two parameters *rowHeaders* and *colHeaders* are handled in the same way, and it is up to the user to ensure that the header cells matches the number of rows and columns, and also if there is fixed rows or columns.

I think the other properties are self-explanatory, but several help types are used:

TableCols defines the primary column header. The type is simple, but important as the type that defines the number of columns and the width of each column. The type also defines the height of the header which may be 0, if the header should not be shown. The class *TableRows* has the same purpose, but for rows.

ColHeaders is a list of *ColHeader* objects and represents the other column header lines. A *ColHeader* consists of a height and a cell. The width of these cells are calculated as it is determined of the span and the width of the columns in the previous header line. *ColHeader* is also a simple class. The class *RowHeaders* has the same purpose, but for row headers.

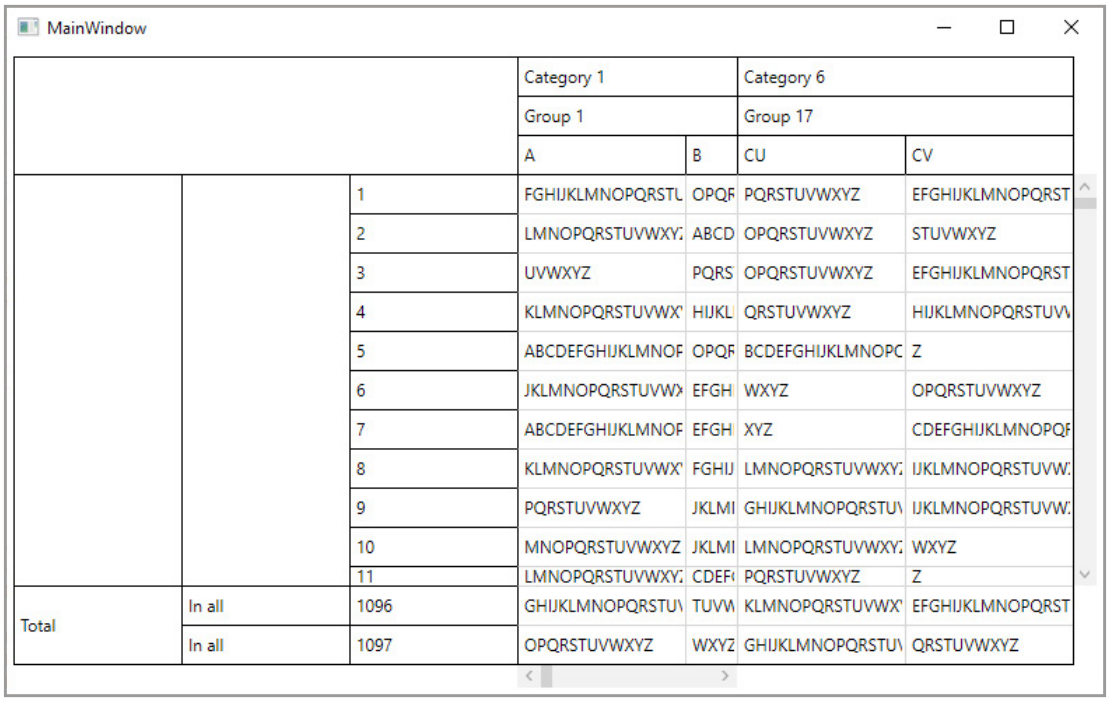
The model class has a property *HasPopups* which indicates where there should be assign popup menus to the grid. Finally the class has an override of the index operator which references the individual cells both for data cells and header cells.

The class *TableModel* is in principle simple, but the class has many other properties used to draw the user interface. The class has also other methods and here methods to insert and remove rows and columns. This methods are defined virtual as it must be possible to override the methods in a subclass.

The component is defined as a user control called *PaTable*. The XML is simple as the user interface is drawn using a custom control, but the class's code behind is comprehensive as the main application of the class is to make the component usable and including to define a number of properties.

As an example of a program which uses the control the following window has a *PaTable* in a grid, where the table has

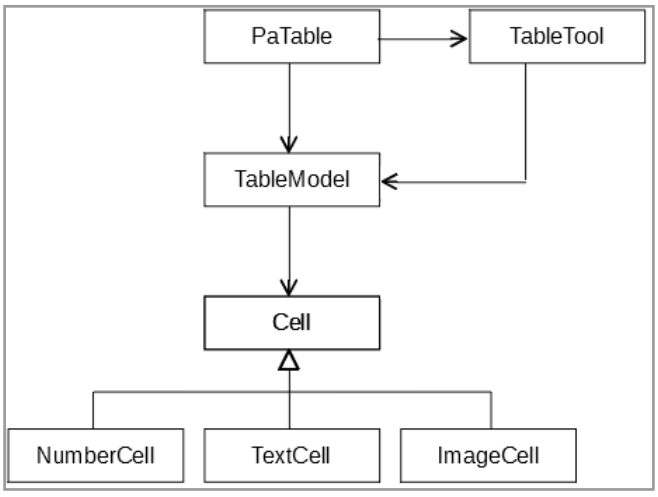
- a column header
- a row header
- 1097 data rows
- 100 columns
- 2 fixed columns
- 2 fixed rows
- 2 stacked column headers
- 2 stacked row headers



The component has two scroll bars, a vertical scroll bar to scroll the rows except the fixed rows, and a horizontal scroll bar to scroll the columns except the fixed columns.

The class *TableTool* is a custom control. It is the class which draw the table, and it is the most complex of the program's classes.

With the above remarks the overall design of the component can be outlined as:



5.3 THE CODE

Then there is the code and there is a lot of it. I will not show the code here, but mention what you should primarily be aware of when studying the code as well as the most important decisions made in connection with the programming.

TableModel

I will start with the model which I already has mentioned above. The class is called *TableModel* and the most complicated is the constructor as it must create all the necessary data structures and ensure they all are initialized correct. The constructor must

- Calculate the number of rows and the number of columns and throw an exception if the values are not correct.
- Create the cell for the corner which is an empty *TextCell*.
- Create column definitions as a *TableCols* object and row definitions as a *TableRows* object. These operations includes to determine where to uses user defined header names or auto generated names (as in a spreadsheet). Each cell contains a *TextCell* object.
- Create or use user defined data cells. If the user has specified the cells as a parameter to the constructor, they are used. Else the constructor creates cells of the type *TextCell*.
- Create stacked headers for columns and rows. These operations are not quite simple as they include parsing input strings and calculating row and column span.

The class has many properties and implements the interface

```
INotifyPropertyChange
```

to notify observers about changes of the two properties *HScroll* and *VScroll*. These properties are bound to the scroll bars in *PaTable*. The class also has an event *UpdateView* which is fired when the two above properties is changed to notify the class *TableTool* about the change. The class has many other properties and I should not note them here, but you must notify, that several of them are calculated, and may be they should have been written as methods as the performance is not always as one would expect from a property.

You should also be aware of 9 methods which has to do with insertion of columns and rows and remove of columns and rows. 3 of the methods has to do with copy / paste. When a cell references the row and column where it is placed, insertions and remove of columns and rows leads to many updates of these references, and the primary purpose of for the 6 methods is to ensure consistency of these row and column references. When cell types could require more (for example a spreadsheet where a cell could contain an expression with references to other cells) these methods are all defined *virtual* to be overwritten in a subclass.

TableTool

It is clearly the most comprehensive and complex class of the component. Actually, the component only needs to do three things:

1. Drawing the component using the model
2. Capture the mouse
3. Implement the user interaction with the component

and none of the three functions are in any way trivial.

I will start with class's data definitions. The class defines 4 dependency properties:

```
public static readonly DependencyProperty HScrollMaxProperty =
  DependencyProperty.Register("HScrollMax", typeof(int),
  typeof(TableTool),
  new UIPropertyMetadata(0, HScrollMaxChanged));
public static readonly DependencyProperty VScrollMaxProperty =
  DependencyProperty.Register("VScrollMax", typeof(int),
  typeof(TableTool),
  new UIPropertyMetadata(0));
public static readonly DependencyProperty VScrollVisibleProperty =
  DependencyProperty.Register("VScrollVisible", typeof(Visibility),
  typeof(TableTool), new UIPropertyMetadata(Visibility.Visible));
public static readonly DependencyProperty HScrollVisibleProperty =
  DependencyProperty.Register("HScrollVisible", typeof(Visibility),
  typeof(TableTool), new UIPropertyMetadata(Visibility.Visible));
```

which are properties to bound to the scroll bars in the class *PaTable*. Next four events are defined:

```
public event EventHandler<EditorEventArgs> EditorHandler;
public event EventHandler<SelectionEventArgs> StartHandler;
public event EventHandler<SelectionEventArgs> MoveHandler;
public event EventHandler EndHandler;
```

These events are all attached to the mouse and fired when

1. the user double click in a cell
2. the user hold down the mouse and starts a drag operation
3. the user move the mouse with the mouse button held down and the drag the mouse
4. the user release the mouse and end the drag operation

There are also 6 objects for popup menus:

```
private ContextMenu topLeftMenu = new ContextMenu();
private ContextMenu colHeadMenu = new ContextMenu();
private ContextMenu colColsMenu = new ContextMenu();
private ContextMenu rowHeadMenu = new ContextMenu();
private ContextMenu rowRowsMenu = new ContextMenu();
private ContextMenu celGridMenu = new ContextMenu();
```

The component opens a popup menu when the user right clicks on (if the area is visible):

1. The upper left corner
2. The stacked header lines for columns
3. The column header line
4. The stacked header lines for rows
5. The row header line
6. The data cells

The class has other variables:

```
// Reference to the data model implemented as a read only property
private TableModel model; private TableModel model;

// Variables used to implements mouse logic
private Timer ClickTimer;    // timer used to implement double click
private int clickCounter;    // indicates how many times the mouse is
clicked
private Position start;      // Mouse position when the user hold the
mouse down
private bool mouseDown = false;   // indicates where the mouse is held
down
private Position cell1;      // the first selected cell in a selected area
private Position cell2;     // the second selected cell in a selected area
```

You should note the timer which is used to implement logic for double click with the mouse. I show how below, but the problem is that the class *FrameworkElement* does not support double click.

The constructor is

```
public TableTool(TableModel model)
{
  this.model = model;
  model.UpdateView += UpdateView;
  ClickTimer = new System.Timers.Timer(300);
  ClickTimer.Elapsed += new ElapsedEventHandler(EvaluateClicks);
  CreateMenus();
}
```

Note that the constructor has a reference to the model, and the model then must be created before this class is created. The constructor registers the current object as an observer for *UpdateView* events, which receive notifications from then model and tells to redraw the component. The constructor creates a timer which ticks every 300 milliseconds, a relatively long period is required as many calculations have to be made. As the last the constructor calls the method *CreateMenus()* which creates all the popup menus.

When it is time to draw the component the following method is performed:

```
protected override void OnRender(DrawingContext drawingContext)
{
  Graphics g = new Graphics { Dc = drawingContext, DpiFactor = 1 /
PresentationSource.FromVisual(this).CompositionTarget.
TransformToDevice.M11 };
  Draw(g, Width, Height);
}
```

which create a *Graphics* object (a helper class) and call the method *Draw()* to draw the component:

```
private void Draw(Graphics g, double width, double height)
{
  double left = model.Left;
  double right = model.Right;
  double center = model.Width;
  double top = model.Top;
  double bottom = model.Bottom;
  double middle = model.Height;
  Pen pen1 = g.CreatePen(Brushes.Black, 1);
  Pen pen2 = g.CreatePen(Brushes.LightGray, 1);
  DrawCorner(g, left, top, pen1);
  DrawColHeaders(g, width, height, left, center, right, pen1);
  DrawRowHeaders(g, width, height, left, top, middle, bottom, pen1);
  DrawCells(g, width, height, left, center, right, top, middle,
  bottom, pen2);
  DrawFrame(g, width, height, left, center, right, top, middle,
  bottom, pen1);
}
```

The method determines the sizes of the 6 areas using properties in the model and the component is drawn using 5 other methods. Here are the three methods *DrawColHeaders()*, *DrawRowHeaders()* and *DrawCells()* complex, as it must be taken into account whether the component is scrolled, that fixed rows and columns are visible to us so on. The result is many lines of code that I will not show here. To draw the individual cells the respective cell classes have everything needed and here the attributes to be used, and the only thing that must happens in the above drawing methods is to calculate a rectangle for the actual cell. A particular problem is cells that span multiple columns or rows. Such cells are perceived as

a special case, and are saved during the drawing process in a list, and they are then drawn after the ordinary drawing process. This means that such cells may override other cells.

Drawing the component is a very important process as it may happens many times. It is therefore absolutely crucial that the above drawing methods are effective, and it is a place where it is worth the sacrifice of effort, if one feels that the component is being updated too slowly. Here you have to concentrate on not drawing more than necessary and not performing more calculations than necessary and especially with regard to the latter, you can probably do better.

As the next problem is the mouse what in principle is simple and consists of implementing three event handlers. The control *FrameworkElement* implements handlers for mouse events that you just has to overwrite in you subclass, but the mouse position is relative to the window which contains the component. This means the position must be converted to a position relative to the current component (the *TableTool* object) and then to the cell om which the mouse points. Below is the event handler for mouse down, and the comments should explain the most important of what is going on:

```
protected override void OnMouseDown(MouseButtonEventArgs e)
{
  // convert the mouse position relative to this component
  Point point = MousePosition(e.GetPosition(null));
  double x = point.X;
  double y = point.Y;

  // calculate on which cell the mouse point
  // note the row and column index may be negative if the mouse
  points on a
  // row or column header
  double top = model.Top;         // height of the top area
  double left = model.Left;       // width of the left area
  double bottom = model.Bottom;   // height of the bottom area
  double right = model.Right;     // width of the right area
  Position pos = Calculate(x, y, left, top, right, bottom);

  start = pos;

  // if it is a right click and the control must opens a context menu
  if (e.ChangedButton == MouseButton.Right)
  {
```

```
    if (pos.row == -1 && pos.col == -1) topLeftMenu.IsOpen = true;
    else if (pos.row == -1)
    {
      colColsMenu.Tag = pos;
      colColsMenu.IsOpen = true;
    }
    else if (pos.col == -1)
    {
      rowRowsMenu.Tag = pos;
      rowRowsMenu.IsOpen = true;
    }
    else if (pos.row < -1)
    {
      colHeadMenu.Tag = pos;
      colHeadMenu.IsOpen = true;
    }
    else if (pos.col < -1)
    {
      rowHeadMenu.Tag = pos;
      rowHeadMenu.IsOpen = true;
    }
    else
    {
      celGridMenu.Tag = pos;
      celGridMenu.IsOpen = true;
    }
  }

  // the general case where a cell other than the upper left corner
  is clicked
  // two things must happen
  // 1) start a selection of cells if the cell clicked on is not a
        header cell
  // 2) catch a double click to open the cell for editing
  else
  {

    // if it is the upper left corner where current selection should
    be cleared
    if (pos.row == -1 && pos.col == -1)
    {
```

```
      cell1 = cell2 = null;
      ReDraw();
    }

    // the component does not support double click and must be
    simulated
    // using a timer
    // if the user double clicks this event handler is performed two times
    // the first time the handler stops a running timer, count the counter
    // and start the timer
    // the second time the handler do the same, but if the user does
    not click
    // the mouse again the timer function is executes after some delay
    ClickTimer.Stop();
    ++clickCounter;
    ClickTimer.Start();

    // if it is a data cell a drag operation is started
    if (start.row >= 0 && start.col >= 0)
    {
      cell2 = cell1 = start;
      mouseDown = true;
      if (StartHandler != null)
        StartHandler(this, new SelectionEventArgs(0, 0, start.x, start.y));
    }
  }
}
```

The timer function is

```
private void EvaluateClicks(object sender, EventArgs e)
{
  ClickTimer.Stop();
  if (clickCounter > 1 && model[start.row, start.col].IsEditable)
  {
    if (EditorHandler != null) this.Dispatcher.Invoke(() => OpenEditor());
  }
  clickCounter = 0;
}
```

When the timer function is executed it simulates a double click event handler. The method fires an event to notify an observer (the class *PaTable*) about the double click. When this method is running as a timer method and then in its own thread the event must be fired using a dispatcher.

The event handler for mouse down also fires a *StartHandler* event to notify observers about start of a drag operation. The observer is *PaTable*. The last thing to note is the call of the method *Calculate()*. It is a comprehensive method as it performs a lot of calculations, but the purpose is to determine to which cell the mouse point and return the position:

```
private class Position
{
  public int row;       // row index for the cell clicked
  public int col;       // column index of the cell clicked
  public double x;      // x-coordinate for the upper left corner of
  the cell
  public double y;      // y-coordinate for the upper left corner of
  the cell
```

The event handler for mouse move is written a bit in the same way, and it has to calculate the current mouse position and send a notification to *PaTable* about the position. The same goes for mouse up, but this handler must also tell the model which cells are selected and redraw the window to show the selected cells.

You should note that the component here has another limitation. Selecting cells by dragging the mouse can only select the cells that are visible on the screen, but the component should scroll if you move the mouse outside the visible area. So another place where there is room for improvement.

As the last concerning the class *TableTool* is the user interaction and there are three things to solve:

1. Edit the content of a cell
2. Show selected cells when dragging the mouse
3. Implement the popup menus

I will start with last, and there is 6 popup menus which is used depending on where the user right-clicks:

Right click on the upper left corner:

1. Attributes for Top / Left corner

Right click on the stacked columns header are:

1. Attributes for headers
2. Header height
3. Attributes for header
4. Attributes for this header cell

Right click on a column header:

1. Header height
2. Attributes for header
3. Attributes for this header cell
4. Show hidden columns
5. Column width
6. Insert column before this column
7. Insert column after this column
8. Delete this column
9. Hide this column
10. Attributes for all cells in this column
11. Select all cells in this column

Right click on the stacked rows header are:

1. Attributes for headers
2. Header width
3. Attributes for header
4. Attributes for this header cell

Right click on a row header:

1. Header width
2. Attributes for header
3. Attributes for this header cell

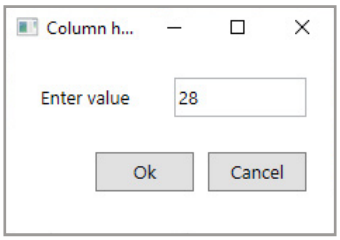
4. Show hidden rows
5. Row height
6. Insert row before this row
7. Insert row below this row
8. Delete this row
9. Hide this row
10. Attributes for all cells in this row
11. Select all cells in this row

Right click on a data cell:

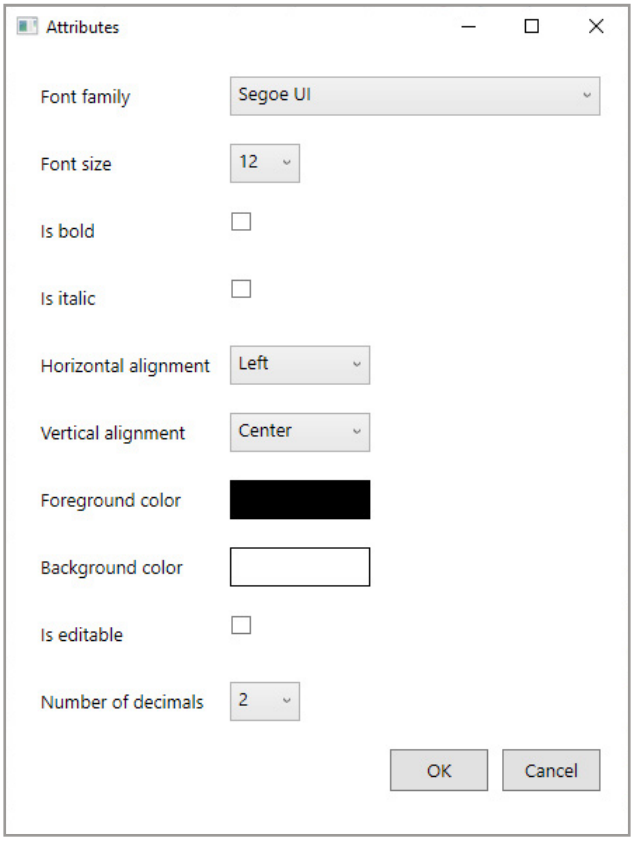
1. Attributes for data cells
2. Attributes for this or selected cells

I think the names tell the purpose of the individual popup menus, but they all mean you update the model and the component must be redrawn. If the component has stacked row or column headers the popup menus does not shows the items to insert and remove rows and columns. If it should be possible the methods in the model to insert and remove rows and columns must be rewritten as they must take into account that headers span properties must be updated. Another place where there is room for improvement.

To implement the popup menus they must be created, which happens in a method called from the constructor. When creating a menu item an event handler must be associated, but in general these handlers are simple. Some of them opens a dialog box where the user has to enter something. For example, if the user needs to enter a value for a width or a height, the following dialog box opens:



As another example many of the menu items relates to maintenance of attributes and they all open a dialog box:

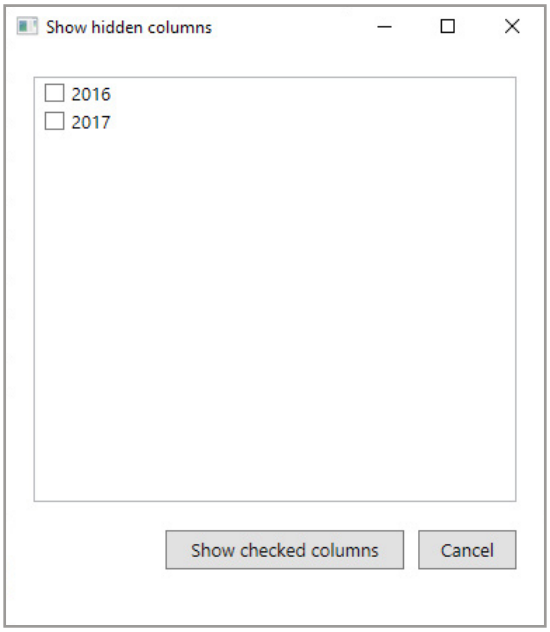


The fields for colors is *Rectangle* controls and if you click on one of these controls they opens a *PaColorPicker* control. The other controls are *ComboBox* and *CheckBox* controls.

There is a problem how to initialize this dialog box. If the dialog box is used to maintain attributes for a single cell there is no problem, but if the dialog box is used to maintain attributes for more cells, for example a column the problem is which cell to use to initialize the combo box. It has been decided to always use the first of these cells. Another challenge is to overwrite a cell's attributes. For example, if you change the attributes of a particular cell and then change the attribute of the cell's column, the first changes are lost. It can thus be stated that the order in which attributes are changed is not irrelevant. The problem is, in fact, that a *TableAttribute* object is attached to all cells. It simplifies programming, but it would actually be better to let a cell search up the hierarchy and find a default attribute. Incidentally, it would also improve performance, so another place for improvements in the next version of the component.

Then there is the menu items to insert and delete rows and columns. It is simple to implement this functions as the model has everything you need. The only thing to be aware of is to update the scroll area, as there will be more or fewer columns / rows.

Also the menu items to hide a column or row are trivial as the only thing needed is to set the width or height to 0. Another thing is how to show hidden columns or rows. If you use the component in a program and two columns is hidden and you click the menu item to show hidden columns you get the dialog box below. Here you can check which columns should be visible again.



The above user interaction when using popup menus can be fine, but it is far from given that the option is desired. In many contexts and even probably the vast majority, such a component will be used simple to display data, where data is organized in rows and columns. Therefore, the model class has a property *HasPopups* which by default is *false* and which indicates whether popup menus can be opened. It is then the user of the component which must active popup menus.

A cell can be editable and that goes for both data cells and header cells, it has the greatest interest for data cells. If a cell is editable and the user double click the cell the class sends a notification including which cell and its position to *PaTable*, and it is up to this class to take care of what is to happen hereafter. The same goes for dragging the mouse, and it is up to the class *PaTable* to take care of what visual should happen.

PaTable

It is the main class for the component. The component is a user control and the XML is quite simple:

```
<Grid x:Name="grid">
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition Height="auto"/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition Width="auto" />
  </Grid.ColumnDefinitions>
  <ScrollBar x:Name="lowerBar" Grid.Row="1" Orientation="Horizontal"
    Margin="{Binding Model.HorizontalMargin}" Minimum="0"
    Maximum="{Binding Table.HScrollMax}"
    Value="{Binding Model.HScroll}"
    Visibility="{Binding Table.HScrollVisible}" />
  <ScrollBar x:Name="rightBar" Grid.Column="1" Orientation="Vertical"
    Margin="{Binding Model.VerticalMargin}" Minimum="0"
    Maximum="{Binding Table.VScrollMax}"
    Value="{Binding Model.VScroll}"
    Visibility="{Binding Table.VScrollVisible}" />
</Grid>
```

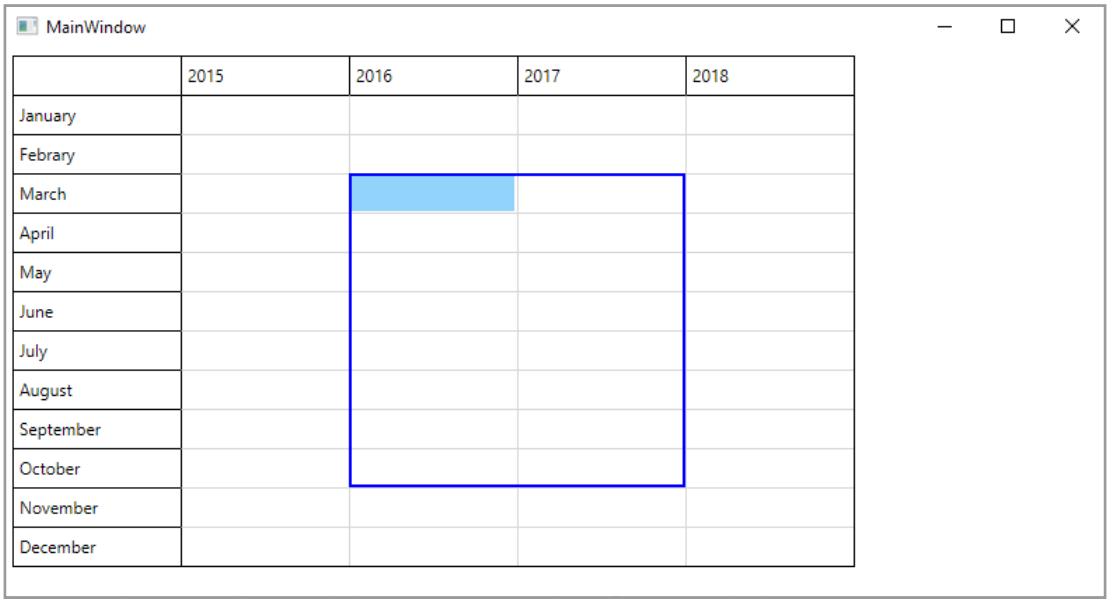
There is not much to note and the layout is a simple *Grid* with 2 rows and 2 columns. The last row and the last column has a *ScrollBar* component which are used to scroll the component in the upper left corner. It is defined in code, but the component is a *TableTool* component. You must note that each scroll bar has four bound properties in either *TableModel* or *TableTool*. In particular, you should note the binding of *Visibility* as a scroll bar does not need to be visible if all the table can be shown in the window.

Then there is the code, which is also a very comprehensive class, but compared to *TableTool*, it is a very simple class, and most of the code are used to make all the necessary properties available to the user. If you look at the constructor it has a data model as parameter, and for that model it creates a *TableTool* component (it is a custom control) and add this control to the *Grid*:

```
public PaTable(TableModel model)
{
  InitializeComponent();
  Model = model;
  grid.Children.Add(Table = new TableTool(model));
  CreateEditor();
  CreateSelection();
  grid.Children.Add(cellEditor);
  grid.Children.Add(selection);
  DataContext = this;
}
```

Next two methods are called. The first creates a *TextBox* component, and add the component to the *Grid*, but as an invisible component. When the user double click a cell the class get an event with the size of and the position for the clicked cell. The size of the *TextBox* is defined to the same size as the cell and moved to the same position, and then the *TextBox* is made visible. For the user it look likes it is the cell that is opened, and the user can edit the content. When the user enter ESC or ENTER the *TextBox* is again made invisible.

The other method works in the same way and creates an invisible *Rectangle* control and add it to the *Grid*. When a drag operation starts the rectangle is made visible and change size when the user move the mouse:



and when the user release the mouse the rectangle disappears and the cells are shown as selected.

The rest of the classes should not be mentioned here, but there are others and here specifically for the dialog boxes which the component uses.

5.4 THE KEY BOARD

It must also be possible to use the key board:

1. If the user select a cell it should be possible to navigate the cell using the arrow keys and Page Up and Page Down (and Home and End).
2. If the user select a cell and hit the Enter key it should be possible to edit the cell if the cell is editable.
3. The control should support copy / paste using Ctrl+C, Ctrl+V and Ctrl+X.

To implements these functions the class *PaTable* must have an event handler for *KeyDown*, and the handler must for each key call a method in *TableTool* to perform the action. In general, it is simple to implement these features. An exception, however, is Ctrl + Z and Ctrl + Y for undo / redo.

The implementation of redo-undo is in principle quite simple and is done in the same way as shown in chapter 2. The first thing to do is to decide which operations it should be possible to undo. In many uses of a component like *PaTable* it will be used only to show data, and in such uses there is nothing to undo. So in the only cases where where the can be undo operations is when a table is editable or the popup menus are activated. To undo an operation it means that the old state must be saved. If you for example change the attributes for all data cells they could in principle all be different and to undo such an operation it would be necessary to save all the old attributes. For a big table the result would be that a lot of memory would be used to save undoable states. I have then only implemented undo (and redo) for the following operations:

1. Edit a cell
2. Change column width
3. Change column height
4. Change column header height
5. Change row height
6. Change row width
7. Change row header width
8. Insert column before
9. Insert column after
10. Remove column

11. Hide column
12. Insert row before
13. Insert row after
14. Remove row
15. Hide row

The implementation is simple, and for each of the 15 operations are added a class (in the file *TableTool.cs*) and when the operation is performed and object of the operations type is pushed on the undo stack.

When editing cells the component opens (makes it visible) a *TextBox* component defined in the class *PaTable*. That's fine enough, but in some contexts it might be of interest to replace this *TextBox* with a custom component. To the class *PaTable* is then added a property for this *TextBox*.

As the last thing the class *TableModel* should be expanded with two new methods:

```
public string Serialize(int indent) { ... }
public static TableModel Serialize(string json) { ... }
```

where the first method serialize the model as JSON and returns the result as a *string*. The other method should deserialize such a *string* and return the deserialized string as a *TableModel* object, but to implement these functions I will first mentioned what JSON is.

5.5 JSON

JSON is a standard for data exchange primarily between client and server in web applications, but in principle, JSON can be used in many other contexts. In many contexts, JSON is used as an alternative to XML. JSON is characterized by being easy to write and read (there are very few and simple rules), being text based and being platform independent.

A JSON document has in principle the same syntax as an object in *JavaScript*, and an example could be:

```
danish = {
“kings”: [
{
“name”:”Gorm den Gamle”,
“to”: “958”
},
{
“name”:”Harald Blåtand”,
“from”:”958”,
“to”: “987”
},
{
“name”:”Svend Tveskæg”,
“from”:”987”,
“to”: “1014”
}
]
};
```

The basic syntax is that data are represented as key / value pair separated by a colon. Several data elements (key / value pairs) can be gathered as objects in a collection where the elements are separated by commas. Finally, the value of a data element may be an array. In general, JSON supports the following data types:

1. *Number* that is a signed decimal number and must include an exponential notation
2. *String* that is double-quoted Unicode with backslash as escaping
3. *Boolean* that is *true* or *false*
4. *Array* which is an ordered sequence of values in square brackets separated by comma and where the values can be of any type
5. *Object* that is an unordered collection of key / value pairs
6. *Null*, that is an empty value

The above can therefore also be written as follows, where there are no quotation marks around the years:

```
danish = {
"kings": [
{
"name": "Gorm den Gamle",
"to": 958
},
{
"name": "Harald Blåtand",
"from": 958,
"to": 987
},
{
"name": "Svend Tveskæg",
"from": 987,
"to": 1014
}
]
};
```

The above is a JSON data structure called *danish*, which consist of one element *kings* that is an array with three elements where each element is an object. Each object has two or three properties, that have the type string and number.

With regard to escaping of characters in strings, the same symbols are used as in C#. Put a little differently, a JSON data structure has the same syntax as a JavaScript object consisting solely of properties (JavaScript is explained in a later book).

As another example, is shown a JSON structure that defines a 2-dimensional array:

```
numbers = {
"primes" : [
[2, 3, 5, 7],
[11, 13, 17, 19],
[23, 29],
[31, 37],
[41, 43, 47]
]
};
```

Of course, writing JSON data correctly requires some practice, but using NuGet you can download a library that under certain conditions makes it quite automatic. The library can serialize objects as JSON objects and deserialize the JSON objects again and instantiate the

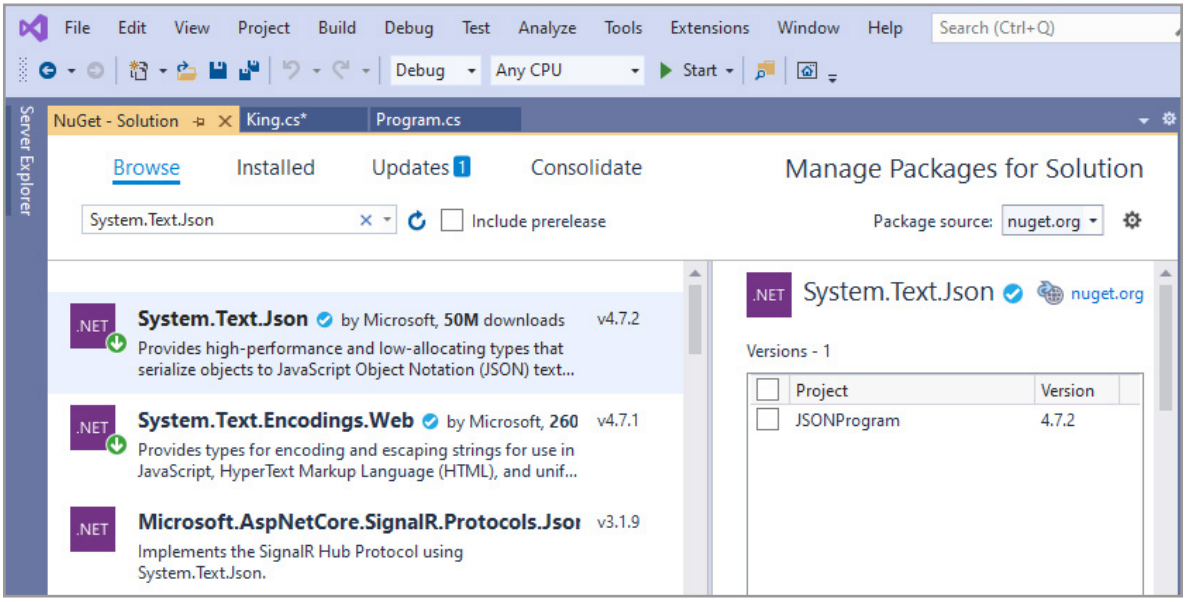
serialized objects. The requirement is quite simple, that an object can be serialized to JSON and deserialized again, if the object's class has read / write properties to these data elements which must be serialized. Note that if you just want to serialize objects is it enough that the object has read only properties.

The project *JSONProgram* is a console application which shows how to serialize an object as JSON and deserialize it again. As the object I will use a *Kings* object, where *Kings* is a class I have used before in this book, but in a modified form:

```
namespace JSONProgram
{
  public class Kings
  {
    private List<King> list;

    public Kings()
    {
    }

    public List<King> List
    {
      get
      {
        if (list == null) list = Load();
        return list;
      }
      set
      {
        list = value;
      }
    }

    public IEnumerator<King> GetEnumerator()
    {
      return List.GetEnumerator();
    }

    private List<King> Load() { ... }
  }

  public class King
  {
    public string Name { get; set; }
    public string From { get; set; }
    public string To { get; set; }
  }
}
```

The class *King* is exactly as before, but the class *Kings* is modified as it now has a read / write property for the list of *King* objects. The *get* property tests if *list* is *null*, and if so it calls the method *Load()* to initialize the list from the XML file. The method *Load()* is unchanged and is not shown above.

The program *JSONProgram* serializes a *Kings* object as JSON, and to do that you need a class with some tools. You can install the library using NuGet:



and the result is two name spaces with classes:

System.Text.Json
System.Text.Json.Serialization

The program can then be written as:

```
namespace JSONProgram
{
  class Program
  {
    static void Main(string[] args)
    {
      WriteKings1();
      ReadKings1();
      Console.ReadLine();
    }

    private static void WriteKings1()
    {
      Kings kings = new Kings();
      var options = new JsonSerializerOptions
      {
        WriteIndented = true,
      };
      string json = JsonSerializer.Serialize<Kings>(kings, options);
      File.WriteAllText("kings.json", json);
    }

    private static void ReadKings1()
    {
      string json = File.ReadAllText("kings.json");
      Kings kings = JsonSerializer.Deserialize<Kings>(json);
      foreach (King king in kings) Console.WriteLine(king);
    }
  }
}
```

The method *WriteKings()* create a *Kings* object. The method next define a property which is used as a parameter to the serialization process. The parameter means that the result is formatted with indentation and so on and such the result is readable. In an operating environment, this parameter will typically be omitted as the result takes up more space due to many indentations. Next the object *kings* is serialized to a *string* and the *string* is written to a file in the application directory.

The method *ReadKings()* read the file and deserialize the result to a *Kings* object and then print all *King* objects on the screen.

It can hardly be simpler. Below is shown some of the saved JSON:

```
{
  "List": [
    {
      "Name": "Regnar Lodbrog",
      "From": null,
      "To": null
    },
    {
      "Name": "Gorm den Gamle",
      "From": null,
      "To": "958"
    },
    {
      "Name": "Harald 1. Bl\u00E5tand",
      "From": "958",
      "To": "986"
    },
    ...
  ]
}
```

The program *JSONProgram* has two other methods which should show the performance of JSON library:

```
private static void WriteKings2()
{
  List<Kings> kings = new List<Kings>();
  for (int i = 0; i < 10000; ++i) kings.Add(new Kings());
  var options = new JsonSerializerOptions
  {
    WriteIndented = true,
  };
  string json = JsonSerializer.Serialize<List<Kings>>(kings, options);
  File.WriteAllText("kings1.json", json);
}

private static void ReadKings2()
{
  System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();
  string json = File.ReadAllText("kings1.json");
  sw.Start();
  List<Kings> kings = JsonSerializer.Deserialize<List<Kings>>(json);
  sw.Stop();
  Console.WriteLine(sw.ElapsedMilliseconds);
}
```

The first method creates a list of 10000 *Kings* objects and serialize this list to a file using JSON serialization. The result is a file which fills a little over 53 MB. If you want you can open the file in *NotePad* and you can see that the content is well formed JSON and the file has 2740002 lines. It is thus a large file. The other method read the file and deserialize from JSON to a C# object. The method measure how many milliseconds used to deserialize the file, and on my machine it is about 1300 milliseconds. You should note that the deserialization process must create 10000 *Kings* objects and for each of these objects 54 *King* objects and then 550000 objects. The bottom line is that the JSON library is quite efficient.

5.6 SERIALIZING A TABLEMODEL

It should be simple using the above JSON library, but there are two problems. First not all types can immediately be serialized as JSON, and especially are the types used in the class *TableAttributes* a problem. This means that special actions are needed to convert objects of these types to or from a *string*. Another challenge is that the JSON library does not support inheritance, and thus a work around is needed to serialize the *Cell* classes. One solution is to try another JSON library, and there are others, but another solution is to write all the serialization / deserialization code yourself.

I want to choose the last solution, and the advantage is that I in that way have full control over, that the objects are serialized correctly, but also that my component is not dependent on a third-party library. Of course, there are also disadvantages which, among other things, include that I have to write a significant part of the code, but especially that the solution is not based on finished and thoroughly tested code and thus a correspondingly high risk of errors. Another disadvantage is efficiency as it is difficult to implement such a solution with sufficient performance.

The task is to serialize a *TableModel* object. Such an object depends on other types:

- *TableRow*
- *TableCol*
- *TableRows*
- *TableCols*
- *RowHeaders*
- *ColHeaders*
- *TableAttributes*
- *Cell*

These types must be expanded with two new methods to serialize and deserialize an object of the type. To serialize the last type the class is expanded with a new property which represents the value of a concrete *Cell* as a *string*. This property is especially necessary in connection with an *ImageCell* where an image must be serialized as text. Here the image is serialized using a base 64 encoding.

When the methods to serialize and deserialize a *TableModel* are implemented you can get the model serialized to a string as a JSON object and then save the string in a file. You can then read the content of the file and use it as a parameter to the static method *Deseralize()* which creates a *TableModel* object from the string. The result is that you can save the state of a *PaTable* component in a file, and restore the state again.

The state is saved as text in JSON format, and you can as so read the content and you can in principle also edit the content. In the test program there is a test method *Test05()*. If you run the method it opens a table with 1097 rows and 100 columns and with both row headers and column headers. If you serialize the state and save the result in a file, the file fills 48642897 bytes and then 48 MB. It is one of the problems with JSON as the result (and the same for XML) takes up a lot of space. In this case you can set the parameter to *Serialize()* to -1 which means the JSON text is unformatted and all indentation and newlines are removed, and even if it means changing the file size by about 1 MB, it does not change the problem significantly. In the present case, this means that it takes a long time to deserialize the object, at least as in this case for a relatively large table.

Parsing of such a large object must necessarily take a long time as the process involves creating and initializing many objects, but you can do it better (take the JSON library from the previous section as an example), and the problem with my parsing is that it performs too many string operations.

However, one can go another way and instead save the state of the component in a simpler form, and when it sometimes can have interest I will show how you in this case can save the state using a comma separated file. That is, that fields in every object must be saved as strings separated with some separation character. It is quite simple and in the same way as above all model classes must be expanded with to new methods:

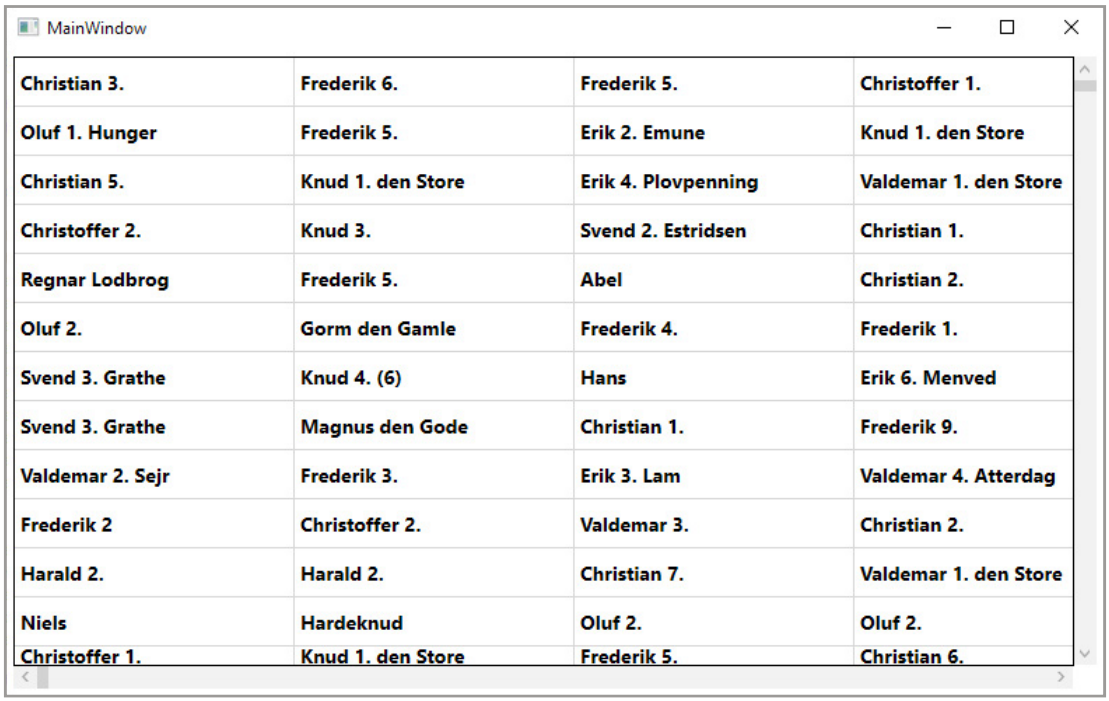
```
public string Serialize() { ... }
public static TableModel Parse (string text) { ... }
```

The only thing to solve is to select the separation characters which must be characters which cannot occur in the strings representing the individual fields. I have used characters from the lower ASCII table and characters with HEX codes between 0x11 and 0x1F.

If I implements these methods and serialize the same table as above it fills 11573708 and thus a significant reduction and the parsing of a serialized object is correspondingly reduced. You should note that the state of the control is still serialized as text, and you can open it in for example Notepad and read it, but in practice it is impossible to change the content as the order of the fields plays a crucial role. The biggest problem, however, is that the content does not meet any standard.

5.7 USING THE COMPONENT

With the component ready the next step is to show how to use the component in an application. The solution for the class library has a project called *Test3* which is a *WPF Application* project. The XML for *MainWindow* is trivial and defines a *Grid*. In code behind is defined 17 test methods, where each method defines a *PaTable* component and add the component to the *Grid*. I will not show the result of all test methods, but if you perform the first method the result is:

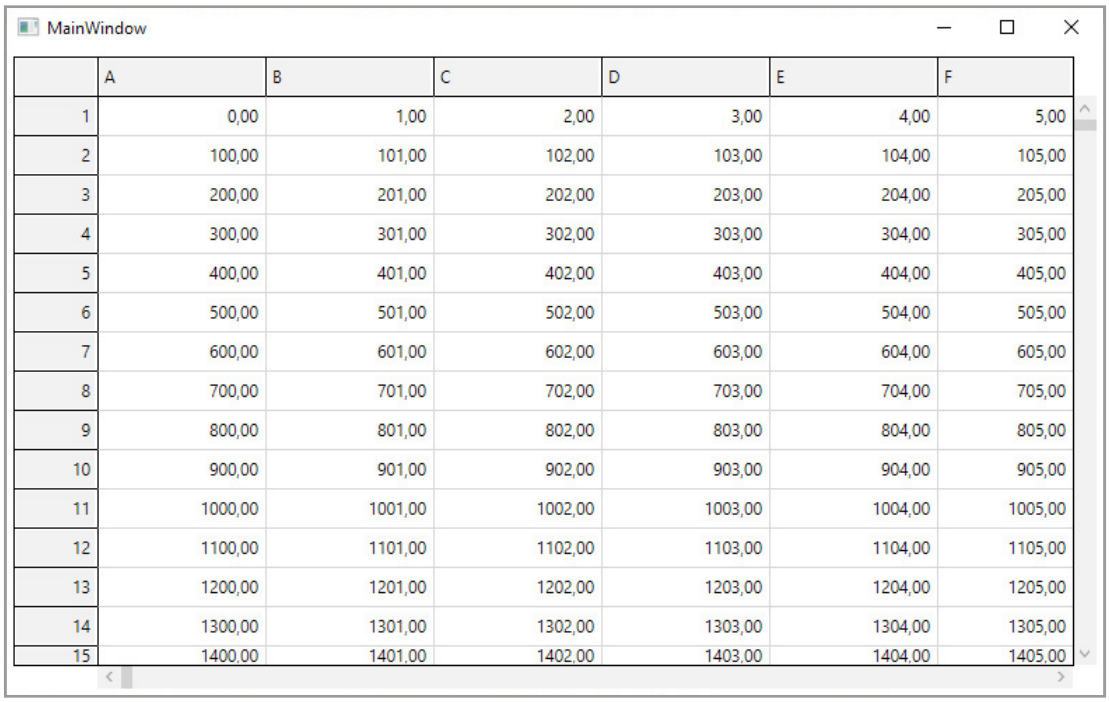


The method creates a 1000 x 100 table without headers and fills the cells with random names of Danish kings. As a result the table has 100000 cells where each cell contains a name. The method also programmatic change the columns width and the row height, change the font size and set the font to bold. The table is created as:

```
private void Test01()
{
  table = new PaTable(new TableModel(1000, 100));
  Kings kings = new Kings();
  for (int r = 0; r < table.Rows; ++r) for (int c = 0; c < table.
  Cols; ++c)
    table[r, c] = kings[rand.Next(kings.Count)].Name;
  for (int c = 0; c < table.Cols; ++c) table.SetColWidth(200, c);
  for (int r = 0; r < table.Rows; ++r) table.SetRowHeight(35, r);
  table.SetCellsFontSize(14);
  table.SetCellsBold(true);
  grid.Children.Add(table);
}
```

Here *Kings* is a class representing the Danish kings, a class I also used in the project *PrintProject*. You should note how to create a model only from the number of rows and the number of columns, and the result is a table without any headers and then only data cells. All cells have the type *TextCell*. The first loop initialize the cells with random names on Danish kings, the second loop set the width of all columns to 200 while the last loop sets the height of all rows to 35. Then the font size is set for all cell as well as the font weight is set to bold.

The next example opens a window which look like a spreadsheet:



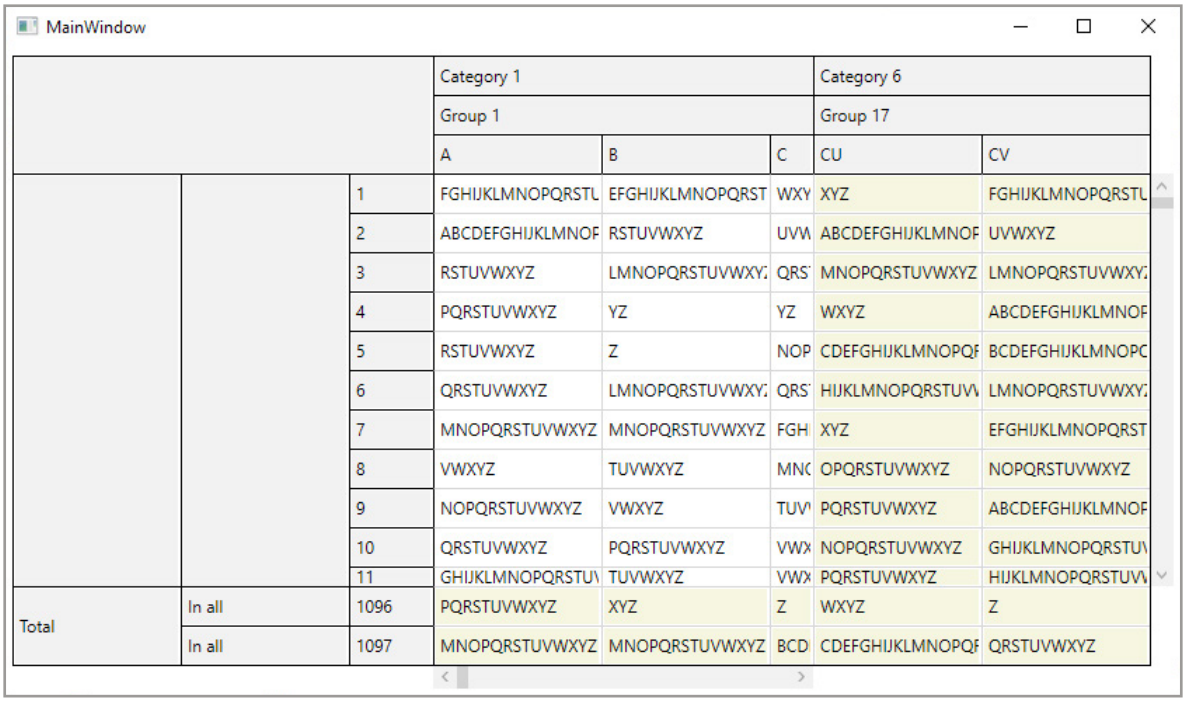
	A	B	C	D	E	F
1	0,00	1,00	2,00	3,00	4,00	5,00
2	100,00	101,00	102,00	103,00	104,00	105,00
3	200,00	201,00	202,00	203,00	204,00	205,00
4	300,00	301,00	302,00	303,00	304,00	305,00
5	400,00	401,00	402,00	403,00	404,00	405,00
6	500,00	501,00	502,00	503,00	504,00	505,00
7	600,00	601,00	602,00	603,00	604,00	605,00
8	700,00	701,00	702,00	703,00	704,00	705,00
9	800,00	801,00	802,00	803,00	804,00	805,00
10	900,00	901,00	902,00	903,00	904,00	905,00
11	1000,00	1001,00	1002,00	1003,00	1004,00	1005,00
12	1100,00	1101,00	1102,00	1103,00	1104,00	1105,00
13	1200,00	1201,00	1202,00	1203,00	1204,00	1205,00
14	1300,00	1301,00	1302,00	1303,00	1304,00	1305,00
15	1400,00	1401,00	1402,00	1403,00	1404,00	1405,00

You must note that it is not a spreadsheet but only a table which shows numbers, but there is no calculation functions. The component is created as:

```
private void Test02()
{
  Cell[,] cells = new Cell[1000, 100];
  for (int r = 0; r < cells.GetLength(0); ++r)
    for (int c = 0; c < cells.GetLength(1); ++c)
      cells[r, c] = new NumberCell() { Value = (r * 100 + c) };
  table = new PaTable(new TableModel(cells));
  table.SetCellsEditable(true);
  table.SetRowHeadsHorizontal(HorizontalAlignment.Right);
  table.SetRowHeaderWidth(60);
  table.SetRowHeadsBackground(new SolidColorBrush(Color.FromRgb(240,
  240, 240)));
  table.SetColHeadsBackground(new SolidColorBrush(Color.FromRgb(240,
  240, 240)));
  table.SetCornerBackground(new SolidColorBrush(Color.FromRgb(240, 240,
  240)));
  grid.Children.Add(table);
}
```

When you see the code I think it's easy enough to understand what's happening, but this time the model is created on the basis of a 2-dimensional array with the data cells to be used. You should also note how to set attributes for the headers and how to set the data cells editable.

The window below shows an example with a table which has stacked rows and columns as well as fixed rows and columns:



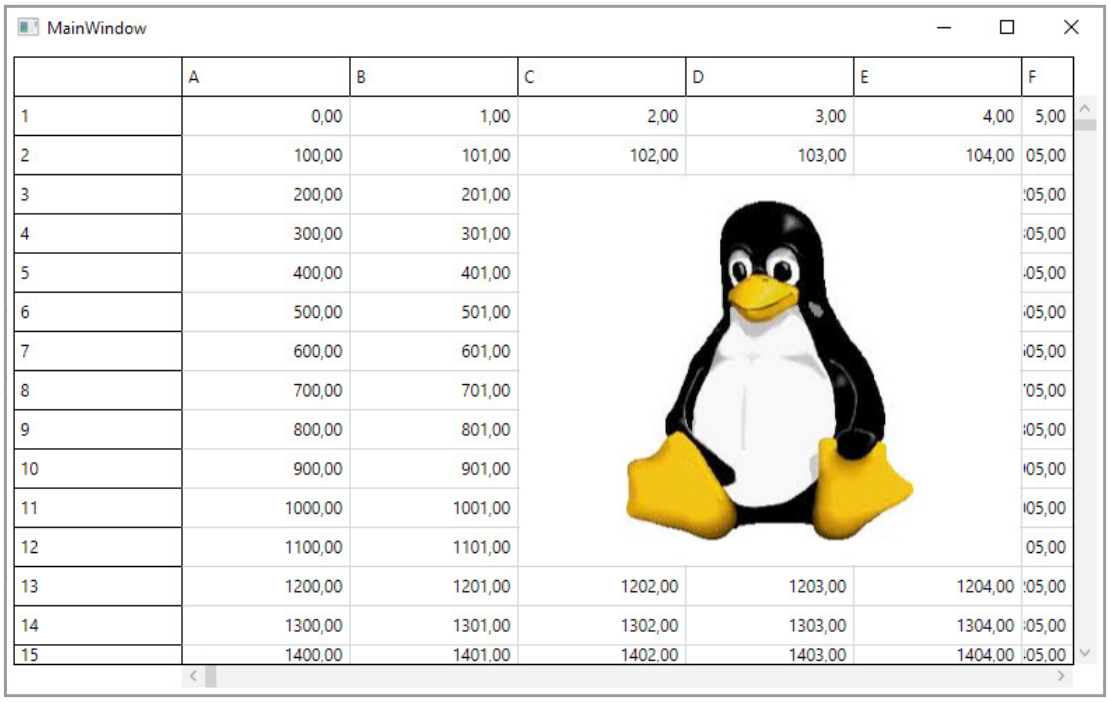
The result itself is uninteresting and all data cells show a random *string*. The interesting thing is how to create such a table and here the code to define the headers fills a lot:

```
private void Test05()
{
  StringBuilder builder = new StringBuilder();
  builder.Append("January;31");
  builder.Append(";Febrary;28");
  builder.Append(";March;31");
  builder.Append(";April;30");
  builder.Append(";May;31");
  builder.Append(";June;30");
  builder.Append(";July;31");
  builder.Append(";August;31");
  builder.Append(";September;30");
  builder.Append(";October;31");
  builder.Append(";November;30");
  builder.Append(";December;31");
  builder.Append(";January;31");
  builder.Append(";Febrary;28");
  builder.Append(";March;31");
  builder.Append(";April;30");
  builder.Append(";May;31");
  builder.Append(";June;30");
  builder.Append(";July;31");
  builder.Append(";August;31");
  builder.Append(";September;30");
  builder.Append(";October;31");
  builder.Append(";November;30");
  builder.Append(";December;31");
  builder.Append(";January;31");
  builder.Append(";Febrary;28");
  builder.Append(";March;31");
  builder.Append(";April;30");
  builder.Append(";May;31");
  builder.Append(";June;30");
  builder.Append(";July;31");
  builder.Append(";August;31");
  builder.Append(";September;30");
  builder.Append(";October;31");
  builder.Append(";November;30");
  builder.Append(";December;31");
  builder.Append(";In all;1");
  builder.Append(";In all;1");
```

```
  string[] rowHeaders = { "2017;12;2018;12;2019;12;Total;2", builder.
  ToString() };
  string[] colHeaders =
    { "Category 1;3;Category 2;2;Category 3;4;Category 4;5;Category 5;2;
      Category 6;1", "Group 1;5;Group 2;7;Group 3;15;Group 4;10;Group 5;
      13;Group 6;4;Group 7;16;Group 8;3;Group 9;3;Group 10;3;Group 11;3;
      Group 12;3;Group 13;3;Group 14;3;Group 15;3;Group 16;4;Group
      17;2" };
  table = new PaTable(new TableModel(1097, 100, rowHeaders,
  colHeaders, 2, 2));
  string text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  for (int r = 0; r < table.Rows; ++r)
    for (int c = 0; c < table.Cols; ++c)
      table[r, c] = text.Substring(rand.Next(text.Length));
  table.HasPopups = true;
  table.SetRowHeaderWidth(60);
  table.SetHeadsBackground(new SolidColorBrush(Color.FromRgb(240, 240,
  240)));
  for (int c = table.Cols - 1, i = 0; i < table.FixedCols; ++i, --c)
    table.SetColCellsBackground(Brushes.Beige, c);
  for (int r = table.Rows - 1, i = 0; i < table.FixedRows; ++i, --r)
    table.SetRowCellsBackground(Brushes.Beige, r);
  grid.Children.Add(table);
}
```

To get the expected result, it is important to carefully define stacked rows and column headers as semicolon separated strings.

As the last example I will show a window with a table containing numbers as well as an image:



You should especially note that the image spans multiple rows and columns.

```
private void Test09()
{
  BitmapImage img = new BitmapImage();
  img.BeginInit();
  img.UriSource = new Uri("penguin.jpg", UriKind.Relative);
  img.EndInit();
  img.Freeze();
  Cell[,] cells = new Cell[100, 20];
  for (int r = 0; r < cells.GetLength(0); ++r)
    for (int c = 0; c < cells.GetLength(1); ++c)
      cells[r, c] = new NumberCell() { Value = (r * 100 + c) };
  table = new PaTable(new TableModel(cells));
  Cell cell =
    new ImageCell() { Source = img, RowSpan = 10, ColSpan = 3,
    IsScaled = true };
  table.SetCell(cell, 2, 2);
  grid.Children.Add(table);
}
```

5.8 ABOUT THE RESULT

The aim of this example is to show from the bottom up the development of a relatively complex user control and show that it can be a very comprehensive task. The example shows that it can be the case, but also that many decisions are needed along the way.

When I start the development of this component I stated that

1. the component must be sufficiently flexible to have value in practice
2. the use of the component must be simple enough for anyone who want to use it
3. the component must be effective also for a grid with many rows and columns

The question then is in what degree these goals are met.

In terms of flexibility, I think the solution hits a suitable place, at least measured in relation to the places where I would use such a component. Not everything is possible and others will certainly be able to point to settings that are not supported or applications where the component is not sufficient at all. The choice is a kind of compromise between, on the one hand, the desire for flexibility and, on the other hand, manageability. It is at least as long as the component is used as it is, but the requirement for flexibility is supported by the fact that you can define your own cell types and specialize the model, but then it is no longer simple.

In terms of usability, the situation is a little different. If you want to use the component to display names or numbers organized in a rectangular schema, the component is quite easy to use, even if it must be possible to edit this data and even if you want a simple formatting. The latter, however, requires a little training and studio of the many properties that the component makes available. It is immediately different if you want to define stacked header lines for either rows or columns. It is not entirely simple and requires great precision on the part of the user and perhaps even some insight into how the component works internally. On the whole, the construction of the model can seem complex and difficult to understand.

Then there is performance, but here it goes wrong. The component creates simply too many objects: *Cell* objects, *TableAttributes* objects and so on. The problem is not to draw the component, as only the cells which falls within the window are used. The problem is when a new table must be created as it requires that all the objects must be created. One way to solve the problem could be to create the objects only when needed. If you use the component with for example 100000 cells there is no problem, and everything behaves nicely. If you have 1000000 cells it takes on my machine almost 2 seconds to create the table, and even though it does not sound like much, it is still noticeable. If you have 10000000 cells my machine uses almost 20 seconds to create the table, and now the problem is evident though it may also be a little extreme, but not if the component should be used to show content of a database table.

If a component like the above should be used in practice there is a long way. More users have to use the component and test it, and the result would be a very long to-do list. Some of the notes on the list will be errors and must be fixed. Others will be inconveniences where the component behaves differently than expected, and it is usually also things that need to be corrected. Finally, there will be a number of suggestions for improvements. Here you have to be more careful and not necessarily implement all the proposals as otherwise you risk ending up in a project that gets the character of one endless reprogramming and as aging is completed. In the case of small changes, the solution of which can be seen immediately and which can be implemented at low cost, you can take them with you immediately. Otherwise, they must be transferred to a wish list, and they can then possibly be implemented in a later version of the product.

I should not show the to-do list for this component, but just mention a few things which is a candidate for a future version:

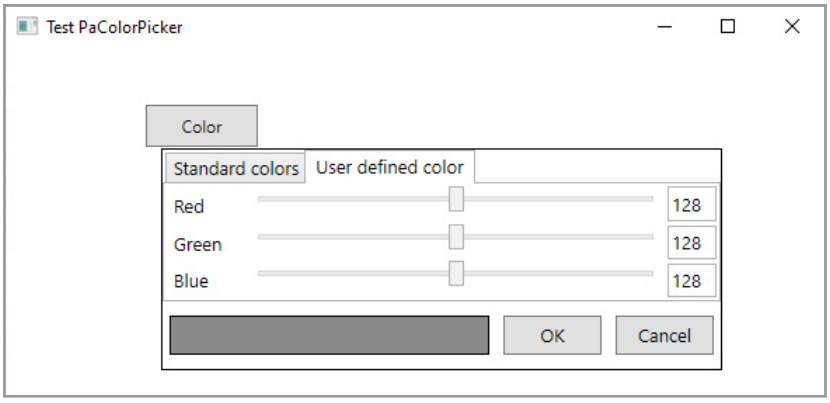
1. The attribute object assign to each cell should be removed and cells should only have an attribute when needed.
2. When selecting cells by dragging the mouse the table should scroll.
3. It should be possible to change the width of columns and height of rows by dragging the mouse.
4. It should be possible to insert and remove columns or rows also when the table has stacked rows or columns.
5. I should be possible to create and use a table without rows, and that is an empty table.
6. A refactoring of the table properties and maybe the component should be expanded with more properties, and maybe some of the properties should be defined as dependency properties.

6 A CLASS LIBRARY

In this book I have written a class library with three user controls. When you works as a software developer you will often write classes which you use in many projects, and it is a good idea to place such classes in a class library. The classes in the class library in this book are all used to build the user interface for a WPF application, and in this chapter I will finish the class library with some adjustments of two of the components, but also by adding another component to the library.

6.1 PACOLORPICKER

There is not much to add about this control, but to define a user defined color the control uses three *Slider* controls. It should also be possible to enter the value for the three colors, and the tab with the sliders is expanded with three new *TextBox* components:

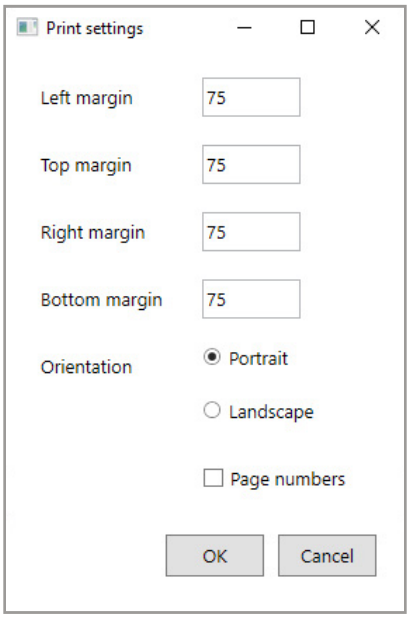


It can all be done in XML, where the layout must be changed and the *TextBox* components bound to the sliders.

6.2 PATEXTEDITOR

The component is used to edit formatted text. The component has, controlled by a setting, options to save edited text in a file and reload the file. If this feature is turned on, it may also be of interest to print the text.

To implement this feature the toolbar must be expanded with a button, and the command for this button opens the following dialog box, where you can enter some page settings. When you then clicks *OK* the components opens a usual print dialog box and one can print the document. You should note, that you can only activate the feature if the document is not empty.



6.3 PACHART

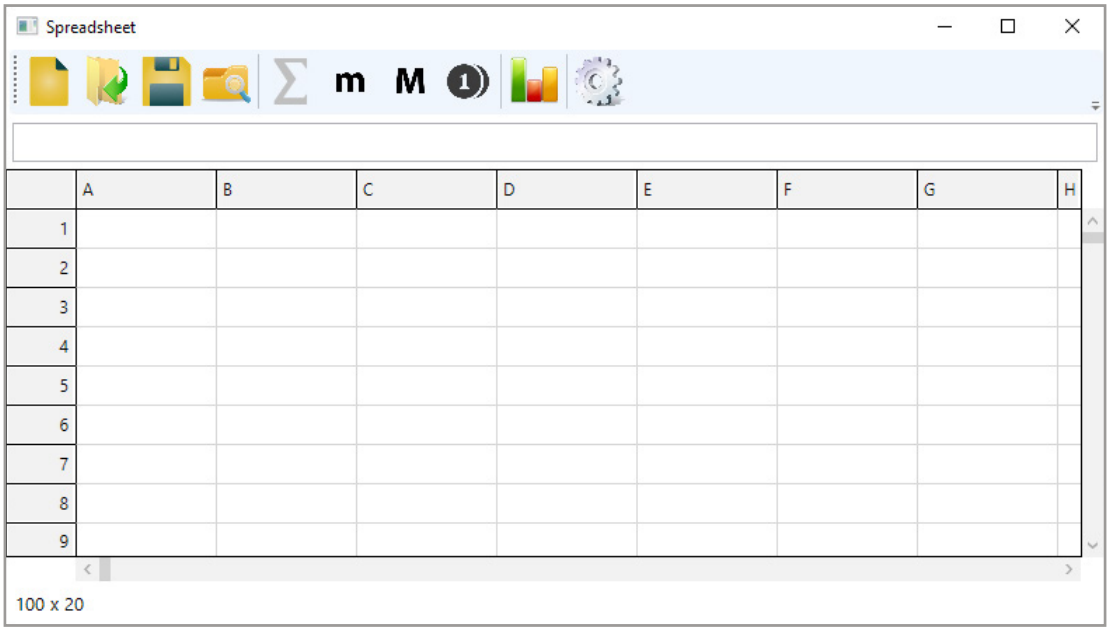
In the last book I write a class library *Charts* which implements a user control called *Chart*. It is a control which using a model can show numeric data as a graph or chart. I want to expand the class library *PaGUI* with this control.. It is simple and I have added a folder *ChartUtil* for all classes in the folder *Tools* in the library *Charts*. The class defines four other classes which are copied to the library *PaGUI*. All namespaces for the imported classes must be changed, and this is all needed. To have the same name convention used for the other classes in the library the name of the class *Chart* is changed to *PaChart*. Then the library can be compiled and can be used.

The book C# 9 also have a program called *GraphTest*. The project for this program is copied to C# 10 and the reference for the class library *Charts* is replaced with a reference to *PaGUI*. Some modifications in the code are needed because the namespaces are changed and the name of the custom control. When done the program can run again.

The component *PaChart* is expanded with 1 new method *CreateImage()* which returns the component as an *ImageSource*. The component draw charts, and in some contexts it may be of interest to save a chart as an image.

6.4 A SPREADSHEET

Now the class library is finish and is rebuild as a release version. Finally, I will show a program that uses the library. The program is a simple spreadsheet, and although the program probably has many of the features found in a spreadsheet, there is a very long way to go to a fully finished spreadsheet. The purpose is to show how to use the class library, and below I will focus on what is done. If you run the program you get the following window:



The design consists overall of 4 components:

A toolbar with 10 functions:

1. Create new spreadsheet
2. Open spreadsheet
3. Save spreadsheet
4. Save spreadsheet as
5. Insert sum of selected cells
6. Insert minimum of selected cells
7. Insert maximum of selected cells
8. Insert average selected cells
9. Insert a chart
10. Define number of rows and number of columns

A *TextBox* to edit an expression. The field is normally read only, but if you edit a cell in the spreadsheet it show the content of the cell and you can edit the content either directly in the cell or in the *TextBox* component.

At the bottom is a status bar which is only used to display the number of rows and the number of columns.

The last component fills the rest of the window and is a *PaTable* component.

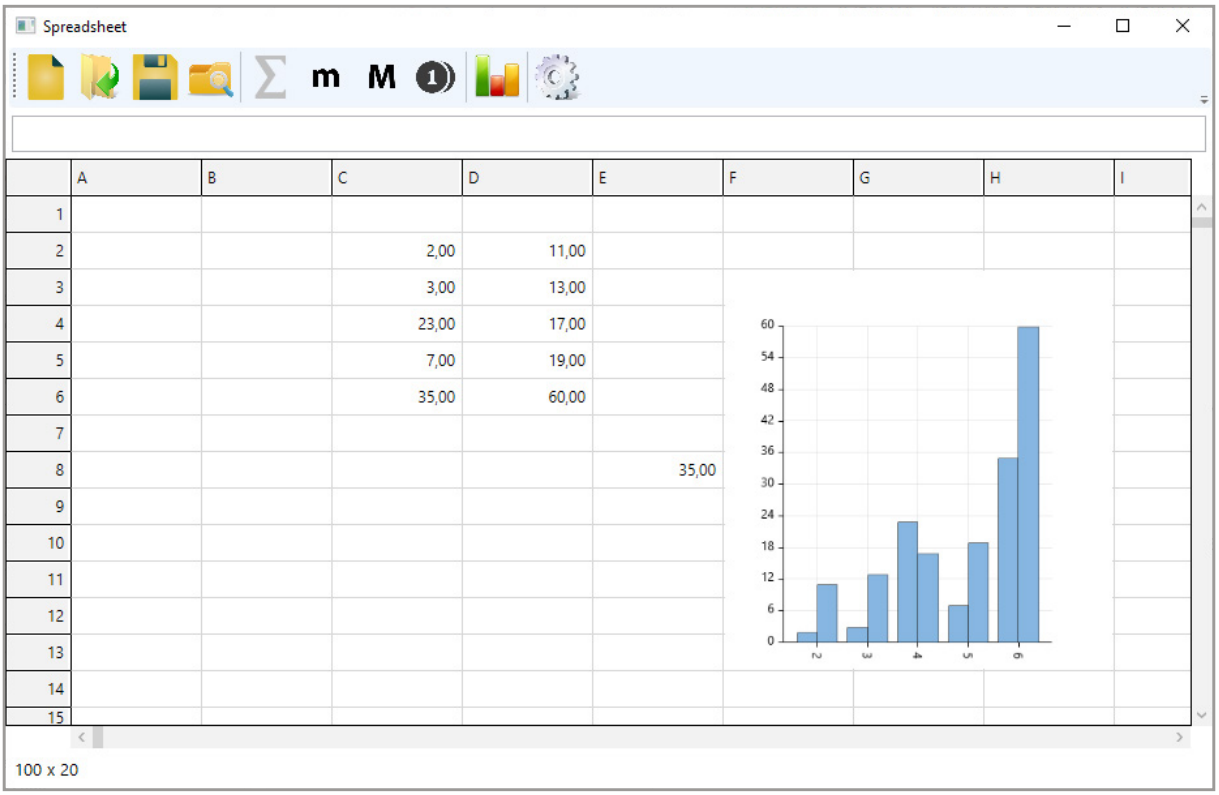
Code behind is quite simple as most of the program code is found in the class *MainViewModel*.

To implement spreadsheet function I need a new cell type as cells now must contains expressions. In the last example in the book C# 3 I write a simple calculator and here I write classes to represents tokens in an expression as well as a class *Expression* to represent an expression. These classes can be used in the current program and are copied to the current project. The classes for tokens are all in one file and these classes can be used with only a few modifications. One token *VarToken* is not needed longer and must be removed, but new tokens to reference a cell in the spreadsheet and an interval of cells must be added. I also needs tokens for the four aggregate functions defined in the toolbar. Then there is the class *Expression*. This class is renamed to *ExpCell* and the class inherits *Cell*. All the code can be reused or change slightly and the class must be expanded with what the abstract class *Cell* defines. After these changes I have a new cell type which can be used for cells in a spreadsheet, and I can create a *PaTable* component with cells of that type.

The component *PaTable* has popup menus enabled and as so the possibilities to set attributes and more. Especially one can insert rows and columns and remove rows and columns which presents a problem since cell references are then no longer necessarily correct. To solve these problems I have written a class *SheetModel* which inherits the class *TableModel*. Here I can override the virtual methods to insert rows and columns and then use an instance of this class as a model for the *PaTable* component.

It should be possible to select cells in a spreadsheet and show the numbers in a chart. To insert a chart in the spreadsheet it is in principle the same as insert an image. A chart must be defined in the same way as in the program *GraphTest* and the same dialog boxes as used in this program can be used to define the data structure for the chart. One thing is to insert a chart in the spreadsheet, but it must also be possible to edit the chart data and as so the type *ImageCell* cannot be used immediately. I have then defined a type *ChartCell* which inherits *ImageCell*.

It is most of what are needed, but there is of course a lot more and including implementing the features of the toolbar. I will not mention these functions here, but just mention that to save a spreadsheet in a file, serialization is used for a comma separated text. The reason is that it requires special measures to serialize an expression as JSON. Below is an example with numbers, expressions and a chart:



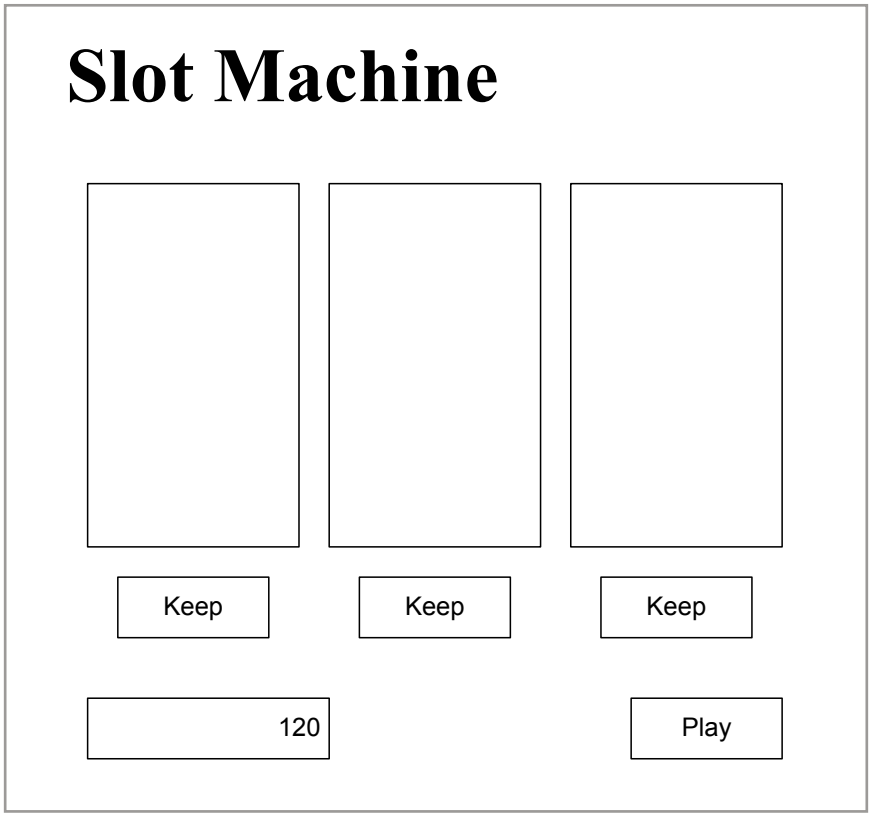
7 A SLOT MACHINE

As the final example of this book, I will show the development of a program that will simulate a slot machine. The aim is, of course, to show the use of some of the concepts introduced in this and the previous book, but where the focus is also on the process. A slot machine consists of (typical) three wheels on which some figures are placed. When you play the machine, the wheels rotate and the figure combination displayed when the wheels stops, determines whether there is a win and, if so, how much the winnings are.

As mentioned, I want to focus on the process. At least on analysis and design, but the program is relatively large and during the programming I will primarily describe what is made and what you should especially notice when you read the code, but not much about how it was made since the number of pages else will be very extensive.

7.1 TASK FORMULATION

Basically, the slot machine must have three wheels that shows a figure combination, and the starting point is the following layout:



There must be some form of user administration, and in order to play the machine, you must be created as a user with an account and have deposited money on the account. It must also be possible to receive an amount from the account if there is money on it which in practice will mean you have won.

You should not be able to play with "real" money, but the program must be able to simulate that the player puts in money and that the player gets paid money.

The machine must be configurable and you should be able to choose whether to play with 3 or 4 wheels. In terms of configuring the machine, one must be able to choose which and how many figures there should be on each wheel, which combinations should give a win and what the winnings should be, and finally it should also be possible to choose which images to use. It must be possible for the player to choose from several configurations. When setting up a configuration, ensure that in the long-term the configuration provides profits to the owner of the machine (the house), but at the same time, the machine must often make a gain to be sure that the machine gives so much back that there is someone who want to play on the machine.

There should also be an opportunity for some kind of statistics, where you can see when the machine gives a win and how big the winnings are.

7.2 ANALYSIS

The machine can be in one of two modes

1. User mode
2. Admin mode

where in the last mode besides playing, you can configure the machine and administrator users. The requirement specification is divided according to these two use patterns.

User mode

In user mode, it is a simple program with few features. When a user (a player) meets the program, the player must do one of the following:

1. Log in with a username (email address) and password
2. Register yourself as a user (if the player is not already a user)

If the player has forgotten his password, the player must contact the administrator, who may change the password. If the player is logged in, the player can change the password at any time.

In the second case (where a new player has to be created) the user must inform

1. email address
2. password
3. name
4. phone number
5. and optional deposit money into the account

After that, the user can play the machine if there is money on the account. In addition to play, the user can perform the following functions:

1. Log out
2. Deposit money into the account
3. Display account information and change these (but not the email address), and at the same time be able to get money paid if there is money on the account
4. View a summary of winning combinations for the current configuration and what the winnings are
5. View an account summary that shows transactions and winnings
6. Select another configuration

In terms of payment, the program must simulates that money is paid from the player's account, and if you put money on the account, the program must simulate the deposit of the player's account. All transactions must be recorded so that a player can always see how much has been inserted and paid.

Of course, the most important feature is to play, which is performed by clicking on a button or equivalent. It must be possible to hold the individual wheels similar to the following:

1. You cannot hold all wheels
2. It should not be possible to hold a wheel if it was held in the previous game
3. You cannot hold a wheel if there was a win in the previous game

When playing the machine, the result of a game must be registered, where the following information should be registered:

1. the user who has played
2. the time for the play
3. the configuration for play
4. the result as the wheels combination
5. the win (0, if there is no gain)

Admin mode

If you log in as administrator, besides playing the machine, you can configure it and manage user accounts.

Regarding the configuration of the machine, you must be able to maintain specific configurations where a configuration indicates:

1. A name so the configuration can be chosen by the players
2. Number of wheels (3 or 4)
3. Which figures are to be used on the wheels (typically in the neighborhood of 10)
4. The positions of the figures (order) on the wheels - the same figure may appear several times
5. Which combinations should give a win and the amounts of the winnings
6. What the cost is to play on the machine
7. If a configuration is active and can be selected by players

Regarding the administration of player accounts, you should be able to:

1. Change player's password (for example, if forgotten)
2. Delete a player where the player's balance is transferred to the player's account

Finally, the administrator should be able to print a statistic that can show how each configuration behaves. Here you can see, among other things

1. Whether a configuration gives positive returns or not
2. The distribution of the single winnings over time

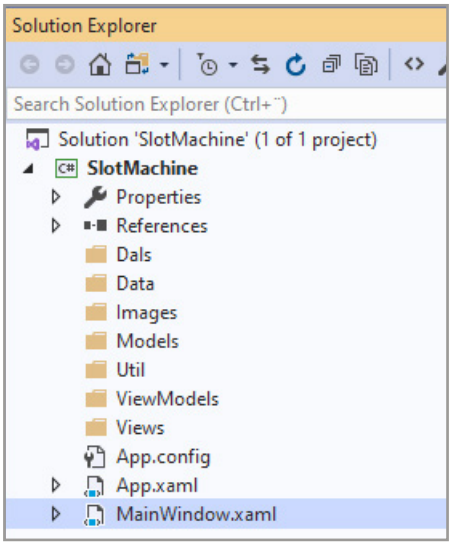
To write the program I will use the following iterations:

1. Create a project in Visual Studio and write a prototype
2. Design of the model

3. Create the main view so it is possible to play on the machine
4. Implement the user administration and logon
5. Configuration of the machine
6. The last features
7. Code review, test and installation

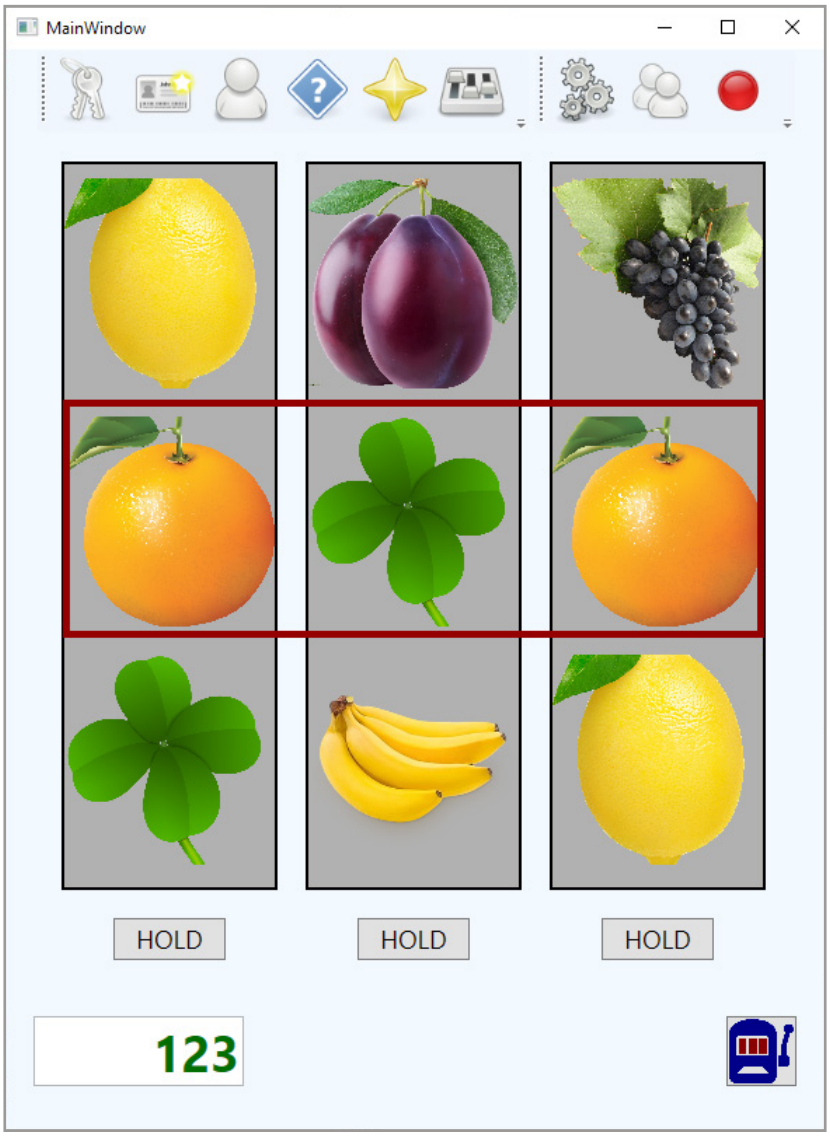
7.3 A VISUAL STUDIO PROJECT AND A PROTOTYPE

The program is a classical Windows application, and the program is developed using the MVVM pattern:



The proptotype

A prototype has been developed to illustrate the user interface and how it will be playing on the machine. If you click on the game button, the prototype selects random figures (there are 15 figures available) by performing an animation of the figures to simulate that the wheels rotates. The toolbar at the top of the window is the features available to the player, where the the toolbar at the right only are available when the user is login as the administrator. The prototype is called *SlotMashine0*.



I should not mentioned much concerning the prototype, and the most important is the animation of the wheels. The idea is to place the images in a *StackPanel*, and then animate the top margin and at the same time has a clip area for the container which contains the *StackPanel*.

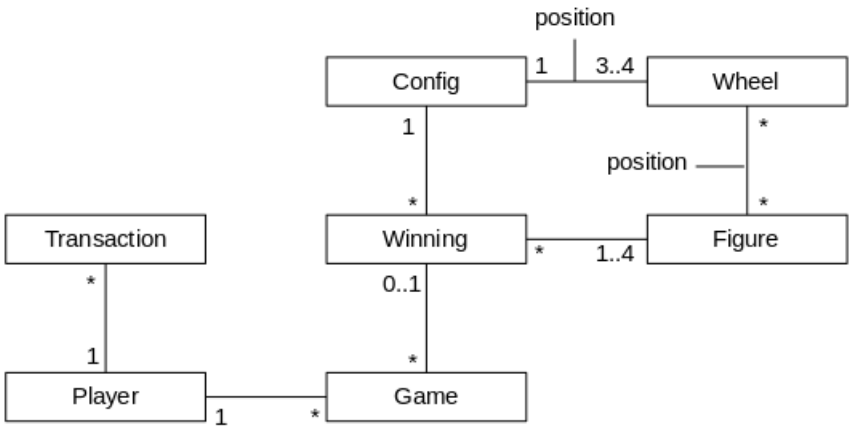
The prototype is implemented as a user control representing a wheel, and a user control representing three wheels. This user control is used in the main view to show the machine's wheels. For the time being, the goal is to show the layout of the program only, but the relevant components will be changed in the following iterations.

7.4 DESIGN OF THE MODEL

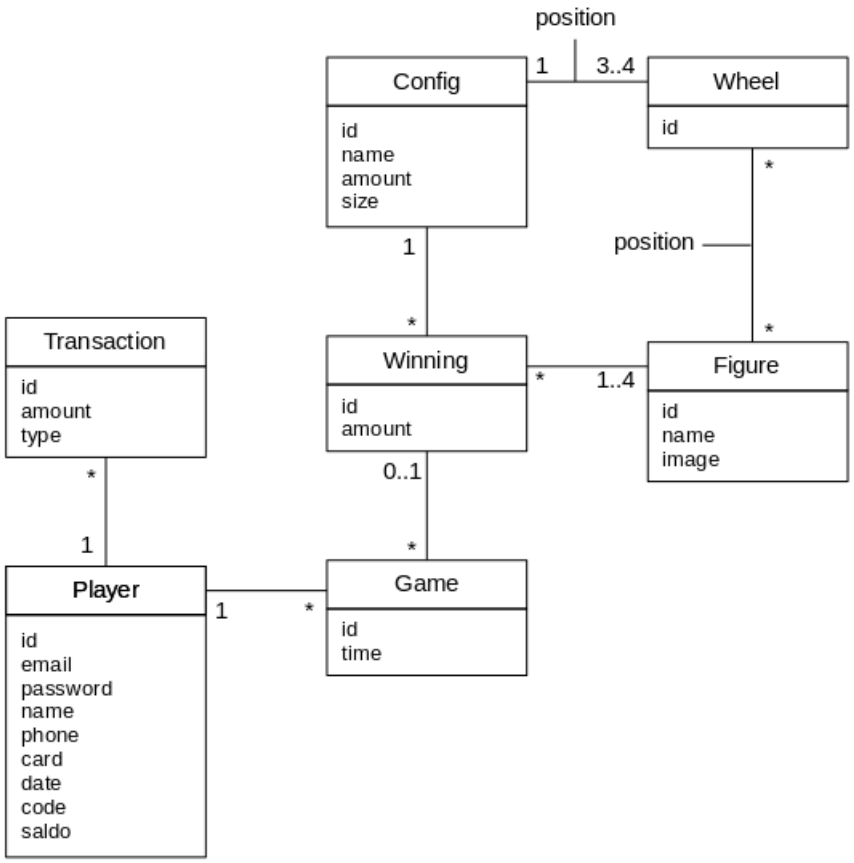
Information about players and configurations must be saved somewhere, and I will use a database. I will start with the design of this database, and the design is outlined in the following ER diagram, where there are the following entities:

1. *Config* that represents a concrete configuration of the machine
2. *Wheel* which represents a wheel for a configuration
3. *Figure*, that is a figure on a wheel
4. *Winning* that represents a gain with an amount for a given combination of figures
5. *Player* which represents a player
6. *Game* that represents a game on the machine for a player
7. *Transaction* that is an amount that the player has inserted or been paid on the account

The attribute between *Config* and *Wheel* indicates the position of the wheel, and in the same way, the attribute between *Wheel* and *Figure* indicate the position of a figure on a wheel.



If you add attributes to the diagram, the conceptual database design can be illustrated as show below. Here you should note the entity *Player* which has three attributes to simulate a credit card. Of course, this makes no sense for the current program, as one should not be able to play for real money, so the attributes are included only to illustrate what it could look like in the real world. The program does not apply these values to anything and it does not matter what values the user enters and the values are not validated.

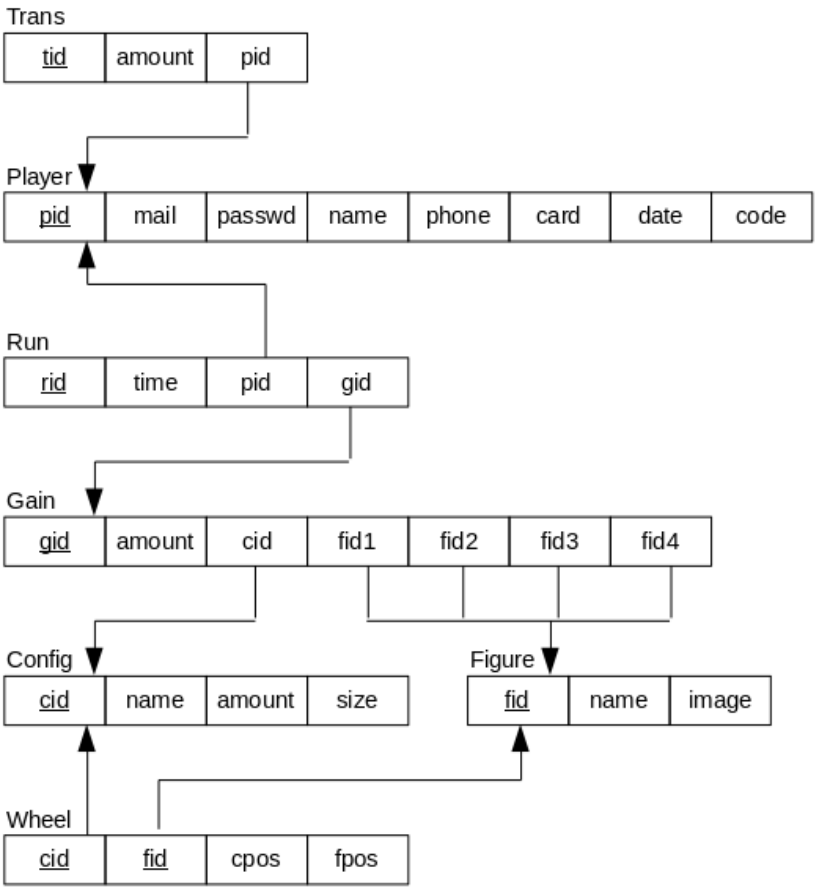


Data dictionary

Player		
id	INT autonumber	primary key
emal	VARCHAR(100)	must be unique
password	VARCHAR(200)	encrypted password
name	VARCHAR(50)	players name, NOT NULL
phone	VARCHAR(20)	players phone number
card	VARCHAR(20)	players cardnumber, NOT NULL
date	DATE	expiration date for card, NOT NULL
code	INT	card control code, NOT NULL
balance	DECIMAL	

Transaction		
id	INT autonumer	primary key
amount	INT	the amount of the transaction, NOT NULL
type	BOOLEAN	true = deposited, false = paid amount from
Config		
id	INT autonumner	primary key
name	VARCHAR(50)	the configurations name, NOT NULL
amount	INT	price to play with this configuration, NOT NULL
size	INT	number of wheels, must be 3 or 4
Figure		
id	INT autonumber	primary key
name	VARCHAR(50)	the name of the figure, NOT NULL
image	BINARY	the figure, NOT NULL
Wheel		
id	INT autonumber	primary key
Winning		
id	INT autonumber	primary key
amount	INT	the winning, may be 0
Game		
id	INT autonumber	primary key
time	DATETIME	date and time for the winning

The above ER model can be mapped to a relational model:



In addition to the names, there are only a few changes.

Regarding transactions, the type has been removed and it has been decided that a positive transaction means that it has been inserted into the account while a negative transaction means that the account has been paid to the user.

The *Wheel* entity has been changed so that it is a relation between *Config* and *Figure*, and the two positions are attributes in this relation, indicating the wheel number and the position of a figure, respectively.

All relations are simple and are in third normal form, and with reference to the above data dictionary, I can immediately write a script and create the database. The script is not shown here.

As database system I will use a simple SQLite database. To the *Dals* folder is added a class *DB* which is a singleton with a method that creates the database. Two additional changes have been added to the database:

1. The table *Figure* is expanded with a column *series*. The column contains a short text, and is used to categorizing the figures.
2. The database is expanded with a simple table *Admin* with two columns. The table must contain the password and email address of the administrator.

To the *Models* folder is added model classes for each table in the database. These classes are for the moment trivial and will all be expanded later, but the classes are used in the class *DB* to write four insert methods which inserts rows in four tables. There is also added a class *MainViewModel* to the folder *ViewModels*, and this class has a method used to create a default configuration. This configuration is used in the next iteration to implement the playing mechanism.

7.5 PROGRAMMING PLAYING ON THE MACHINE

If you have done a good and careful analysis and design and possibly combined with one or more prototypes, you should have all the requirements in place at the start of the programming phase. It is at least the theory and, with practice, it will also work for smaller projects. However, one must also acknowledge that no matter how careful you has been, then there during the programming can happen changes to the specification, including wishes for something that should work in a different way or maybe even new wishes. In fact, it is not strange, and during the development of a program, the understanding of the application's use and also the possibility of recognizing new features that the program should also have or something that is agreed in the specification that can be advantageously modified all may cause changes. Obviously, the larger a program, the greater the chances are that during the programming there will be a desire to change the requirement specification.

Should it happen - or when it happens - you must contact the customer and present the changes, including what the changes will mean in terms of resource usage and the time horizon for when the program can be completed. Then there must be an agreement as to whether the new wishes are included in the project or not, or they may have to be postponed to later. In practice, it is important that such wishes come true, but it is also important that, as a developer, you not just implements things that are not part of the requirement specification, since ultimately it is the customer who decides (and pays for) what should be part of the finished program.

The conclusion is that even after a thorough in-depth analysis, there is an opportunity for during the programming there will be wishes to change the requirement specification, wishes that may be clarified with the customer.

The same applies to the design, and here is more the rule than the exception. This is due to several factors, among other things, that the boundary between design and programming is not sharp. Thus, there will almost always be changes in the design, and usually changes that must be made by the developer without the customer's approval.

A typical design activity is database design, and changes will almost always occur during the programming phase. This does not mean that the database design is not important during the design phase, as programming typically just involves minor adjustments, but they will, almost certainly, also be there. This is also the case with the current program, where some columns have been added to some of the tables, as well as changes to the foreign keys. It is worth paying attention to the design of the database during the design phase to avoid this kind of changes during programming, but vice versa, with just a small size database, it is difficult to avoid any adjustments during programming. When such changes are problematic, it is because they sometimes means that the database data content needs to be recreated, which may be time consuming. In this case, some changes have meant that it has also been necessary to change the initialization of the database and it has been necessary to run the method *CreateDB()* again.

Another design activity is the design of the model layer and including the most important model classes. It is rarely associated with the major challenges, but you must be prepared to make changes to these model classes and to create new ones. It is not a problem and just states that during the design you are not at the same level of details as during programming, and that is also the idea, as the design should have focus on the overall concepts alone.

As mentioned, the boundary between design and programming is not sharp, which can result in significant decisions are postponed to the programming phase. In general, you should be careful and there is a tendency or the risk under the design not to find challenges and problems that should be solved. This indicates that during the design, sufficient time is not used, thus neglecting or underestimating problems that may occur during the programming. It's hard to put the right distinction between design and programming, but conversely, just under the design work with the big lines and not writing the program, but the result is, that if significant decisions are postponed for programming, there is a great risk that parts of the code that is written must be changed or, at worst, rewritten. Therefore, it is important that all major challenges are dealt with before you write the program.

In connection with the development of this program, there are actually examples of tasks that should have been treated better during the design and, for example, I would like to mention two.

It is part of the requirements that the administrator must be able to test the individual configurations as to how they earns and how much the house earns on the configuration. How it will happen is not treated in either the analysis or the design. That it has not been processed during the analysis is an indication that it allows developers to come up with a solution, which is not unusual, and therefore there should be a sketch of a solution under the design. For example, it should have been considered whether the database has the data needed to make a reasonable analysis of a given configuration.

Another issue is what should happen if you change or delete a configuration. Doing so, the data recorded for games with the deleted configuration is not necessarily longer valid, meaning that information about the individual player's use of the machine is no longer valid. The data in question can not only be deleted, as it also deletes how much a player has won or lost. It is also an example of a problem that should have been solved during the design.

After this iteration it should be possible to play on the machine, but without any user administration. That is when the program starts, it must read the default configuration from the database and connect the prototype from the analysis to this configuration. For that I have added the following model class:

```
public class Configuration
{
  public Configuration(string name)
  {
    Wheels = new List<List<Wheel>>();
    Gains = new List<Gain>();
    if (!DB.Instance.LoadConfig(this, name))
      throw new Exception("Illegal configuration");
  }

  public Config Config { get; set; }

  public List<List<Wheel>> Wheels { get; private set; }

  public List<Gain> Gains { get; private set; }
```

I have also moved the code to create the database and the default configuration as a static method to this class. The constructor call a method *LoadConfig()* in the class *DB* which load the configuration identified by the parameter *name*. The method initialize this object, and note that it means that the method must load the images from the database, the configurations of the wheels and gain combinations.

The class is instantiated in *MainViewModel*. This class has more properties and here properties for the hold logic for the wheels and property for the holdings (the user's amount). It is also this class which instantiates a *Slot3Control* with itself as parameter. The class is bound to the user interface, but for now it is only the *TextBox* for user's amount and a command for play button.

The result is a running machine where you can play. If you run the program you will note that it is too easy for the user to win.

7.6 USER ADMINISTRATION

It is decided to simplify user administration. All things about simulating a credit card are removed, and a player should then only have a user id, a name and an email address. At the same time it is decided that there should be one administrator and this administrator must be identified as the user with user id 1. The administrator must be created when the database is created, and the machine is automatic in administrator mode if the administrator logs in. It means that the table *Admin* can be removed from the database and also the model class *Admin*. The table *Player* must be changed as more columns should be removed, and the model class *Player* should be changed accordingly. The two tables *Trans* and *Run* are also changed as the type of the column *Time* in both tables are changed to *INTEGER*.

A player must have a password. When a new player is created the user must enter a password, and this password is send to the class *DB* together with a *Player* object:

```
public bool InsertPlayer(Player player, string passwd)
{
  try
  {
    using (SQLiteConnection connection = GetConnection())
    {
      connection.Open();
      using (SQLiteTransaction transaction = connection.BeginTransaction())
      {
        try
        {
          using (SQLiteCommand command = new SQLiteCommand(
"INSERT INTO Player (Name, Mail, Salt, Pass) VALUES (@Name, @Mail, @
Salt, @Pass)",
            connection))
          {
            string salt = Guid.NewGuid().ToString();
            command.Transaction = transaction;
            command.Parameters.AddWithValue("@Name", player.Name);
            command.Parameters.AddWithValue("@Mail", player.Mail);
            command.Parameters.AddWithValue("@Salt", salt);
            command.Parameters.AddWithValue("@Pass", ComputeHash(passwd,
            salt));
            if (command.ExecuteNonQuery() == 1)
            {
              player.Pid = (int)connection.LastInsertRowId;
              transaction.Commit();
            }
            else throw new Exception();
            return true;
          }
        }
        catch
        {
          transaction.Rollback();
          return false;
        }
      }
    }
  }
  catch
  {
    return false;
  }
}
```

The method opens a connection to the database and define a SQL command for an INSERT statement. It inserts values in four columns as the ID is auto generated. The last two columns are a unique string created as a *Guid*. This string is added to the created password to make the password stronger and increase security. The value is called a *salt*. The password is stored in the database as a hash of the user entered password and the salt. The hash is created using a stand algorithm:

```
public string ComputeHash(string password, string salt)
{
  byte[] bytes1 = Encoding.UTF8.GetBytes(salt);
  byte[] bytes2 = Encoding.UTF8.GetBytes(password);
  byte[] bytes3 = new byte[bytes1.Length + bytes2.Length];
  for (int i = 0; i < bytes2.Length; ++i) bytes3[i] = bytes2[i];
  for (int i = 0; i < bytes1.Length; ++i) bytes3[bytes2.Length + i] =
  bytes1[i];
  HashAlgorithm hash = new SHA256Managed();
  byte[] bytes4 = hash.ComputeHash(bytes3);
  byte[] bytes5 = new byte[bytes4.Length + bytes1.Length];
  for (int i = 0; i < bytes4.Length; ++i) bytes5[i] = bytes1[i];
  for (int i = 0; i < bytes1.Length; ++i) bytes5[bytes4.Length + i] =
  bytes1[i];
  return Convert.ToBase64String(bytes5);
}
```

You must note that this means that the password is only stored in the database table, but not as properties in objects somewhere in the code.

When a player login the user must enter the email address, as it is used as key, and the password. To test if the user is created and the password is correct the *DB* class reads the salt for this email and create the hash of the entered password and the salt and check if it is the same hashed value stored in the database.

This iteration then starts with some changes of the methods in the class *DB* which creates the tables, but also the method *CreateDB()* in the class *Configuration* as the method now must create the administrator. After these changes the rest of the iteration is simple and consists of:

The toolbar is updated where the button rightmost is removed, but a button is added to change the users password. All buttons in the toolbar have their *Visibility* property bound to a property in *MainViewModel* such the buttons are only visible when a user and for the last two the administrator is logged in. The same goes for the play button.

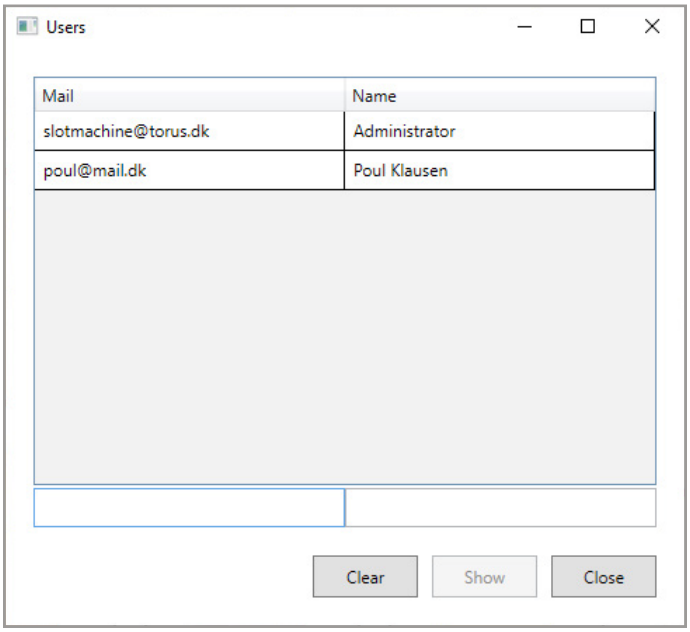
A dialog box is added for login where the user must enter the email address and the password. The class *DB* is expanded with a method to validate the user and return a *Player* object. The dialog box has a button to open another dialog box used to create a new player. The class *DB* has a method to insert a player, a method also used in the method *CreateDB()* to create the administrator.

When a new player is created the player cannot play on the machine as the holding is 0. To play the player must enter a deposit, and for that a dialog box is added, and *DB* is expanded with a method to insert a transaction. You should note that the administrator can play without any deposit, and it is possible for the administrator to have a negative holding.

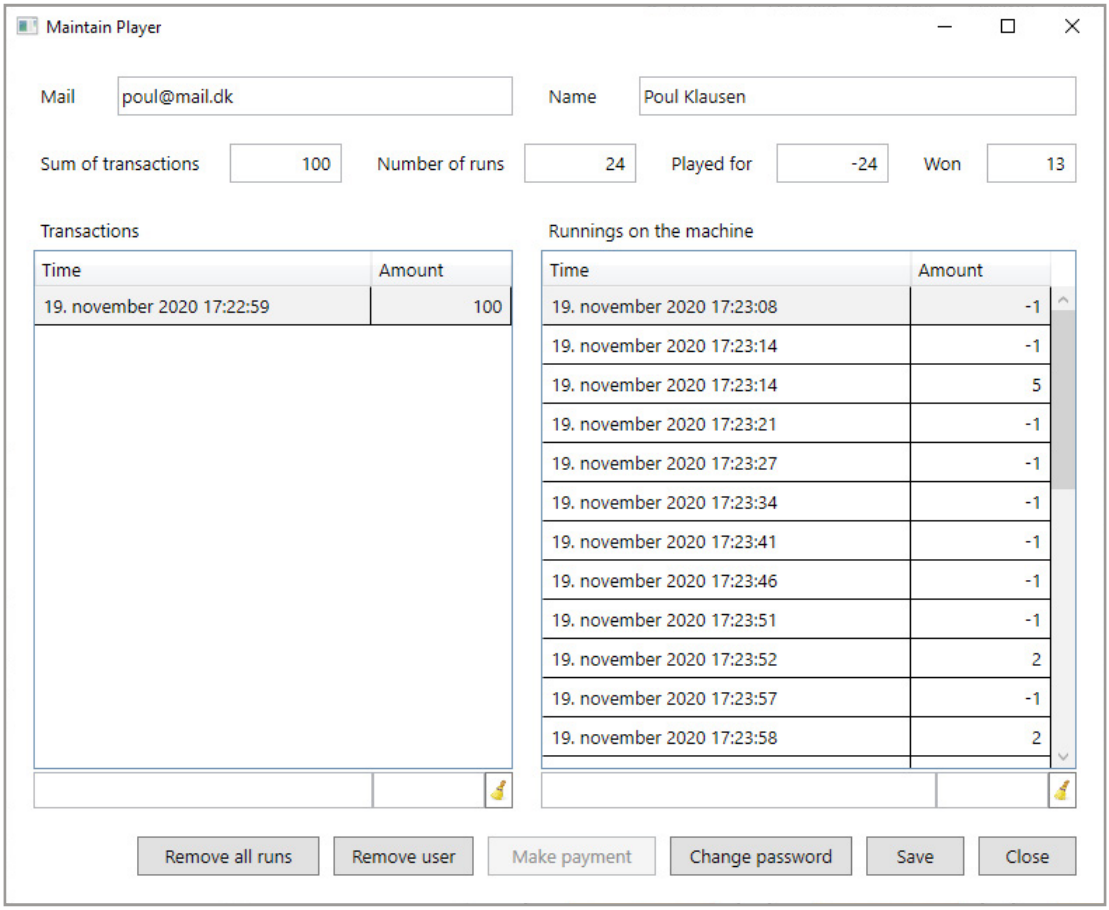
As the last is added a dialog box used to edit the user's name and mail address. The mail addres is used as key in the database and must then be unique. This dialog box also has a button which the player can use for logout.

Finally the program must maintain a user's holding in the database. When a user add a deposit a row must be added to the table *Trans*. When then play on the machine a row must be inserted in the table *Run* each the user clicks the play button. The same must happen, when the user wins. All these data should be used to analyze how how a configuration runs.

Regarding user administration, there is one function left where the administrator can manage existing users. The function is not particularly complex, but extensive nonetheless. If the administrator clicks on the icon for user administration in the toolbar, you will get the following window which shows an overview of all users of the program:



If you double click on a user in the table the program opens a window showing information about the user:

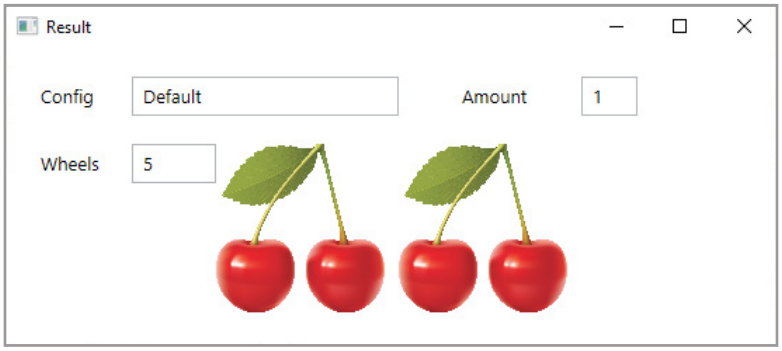


In this window you can edit *Mail* and *Name*, but the other fields are read only. The buttons are used for:

1. If you edit the fields for *Mail* and *Name* the changes are saved, and it means that the administrator can change these values for all users.
2. The administrator can also change the password for a user. If you click on the button *Password* you get a dialog box to change the password. The function should be used if a player has forgotten his password.
3. The button *Make payment* opens a dialog box where the administrator can enter an amount as a payback to the user. The function can only be used if the value of *Won* is greater than the value of *Payed for*, and the amount to payback must be less than the difference. The program does not use real money and the only thing that happens is that the money repaid to the player are inserted as a transaction with negative amount to indicate a payout.
4. The administrator can also remove an user, and if so both user, all transactions and all playing results for this user are removed.

5. As the last the administrator can remove all results for a user. If a player has used the machine for a long time there may be many results. and it may be appropriate to delete them. In that case, the sum of all runs is transferred as a transaction with the opposite sign, thus indicating that the user has either deposited money or been paid money.

If the administrator double click on a run with positive amount the program opens a window to show the result:



7.7 CONFIGURATION OF THE MACHINE

The next iteration should implement maintenance of configurations. That means the administrator should be able to create and maintenance existing configurations:

1. define the number of wheels for the machine (3 or 4)
2. select which the images to use
3. arrange the images on the wheels
4. define which figure combinations should yield a gain and the size of the gain

The images used for the wheels are now stored in the database in a base 64 encoding. It's fine in principle, but it takes some time to convert an image to a base 64 format, and it is a bit faster to store an image as a byte array instead. I have then started to change the database, but at the same time I have made three other changes:

1. The tables *Config* and *Figure* has a column *Series* which is not used. This column is therefore removed from both tables.
2. I have added another table called *Figs* which is a relation between *Config* and *Figure* which defines which figures a specific configuration uses.

3. The table column is expanded with a new column called *Length* which denotes the number of figures for each wheel. Each wheel must have the same number of figures.

These three changes are all simple to implements and means I have to update some methods in the class *DB* as well as some model classes. To store an image as a byte array the column in the database table must be defined as a *BLOB*:

```
CREATE TABLE Figure (..., Image BLOB NOT NULL)
```

and an image can be converted to a byte array using the method:

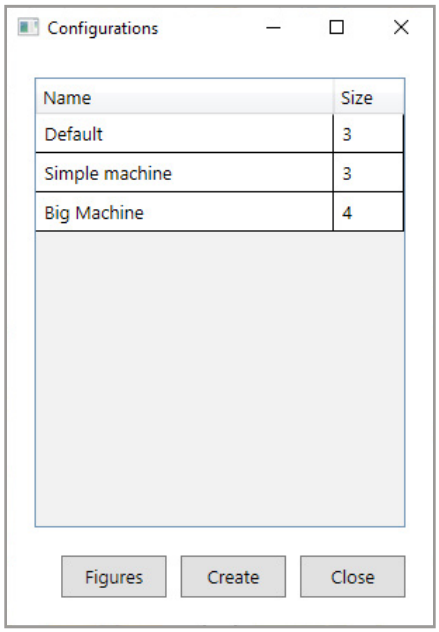
```
public byte[] ToBytes(BitmapEncoder encoder, ImageSource imageSource)
{
  byte[] bytes = null;
  var bitmapSource = imageSource as BitmapSource;
  if (bitmapSource != null)
  {
    encoder.Frames.Add(BitmapFrame.Create(bitmapSource));
    using (var stream = new MemoryStream())
    {
      encoder.Save(stream);
      bytes = stream.ToArray();
    }
  }
  return bytes;
}
```

To read an image from the database the following methods can be used:

```
private ImageSource ReadImage(SQLiteDataReader reader, int n)
{
  byte[] buffer = new byte[4096];
  long bytesRead;
  long fieldOffset = 0;
  using (MemoryStream stream = new MemoryStream())
  {
    while (
      (bytesRead = reader.GetBytes(n, fieldOffset, buffer, 0, buffer.
      Length)) > 0)
    {
      byte[] actualRead = new byte[bytesRead];
      Buffer.BlockCopy(buffer, 0, actualRead, 0, (int)bytesRead);
      stream.Write(actualRead, 0, actualRead.Length);
      fieldOffset += bytesRead;
    }
    return ToImage(stream.ToArray());
  }
}

public ImageSource ToImage(byte[] imageData)
{
  BitmapImage biImg = new BitmapImage();
  MemoryStream ms = new MemoryStream(imageData);
  biImg.BeginInit();
  biImg.StreamSource = ms;
  biImg.EndInit();
  ImageSource imgSrc = biImg as ImageSource;
  return imgSrc;
}
```

As the last thing the class *DB* must be expanded with a method to create the new relation table, and a method to create a new configuration is added.

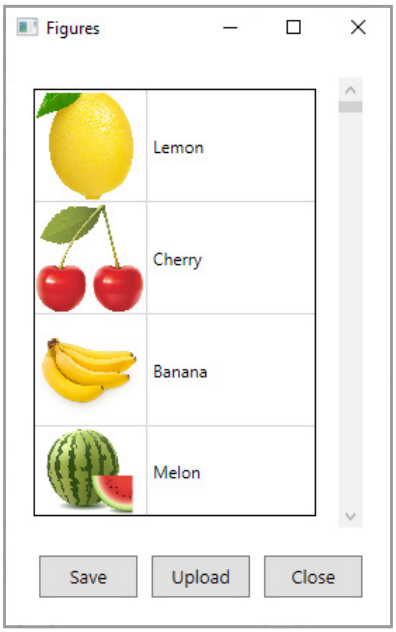
To configure the machine the user must be logged in as administrator and if so the toolbar has an icon for the configuration function. If you clicks the icon the program opens a window which shows all the created configurations:



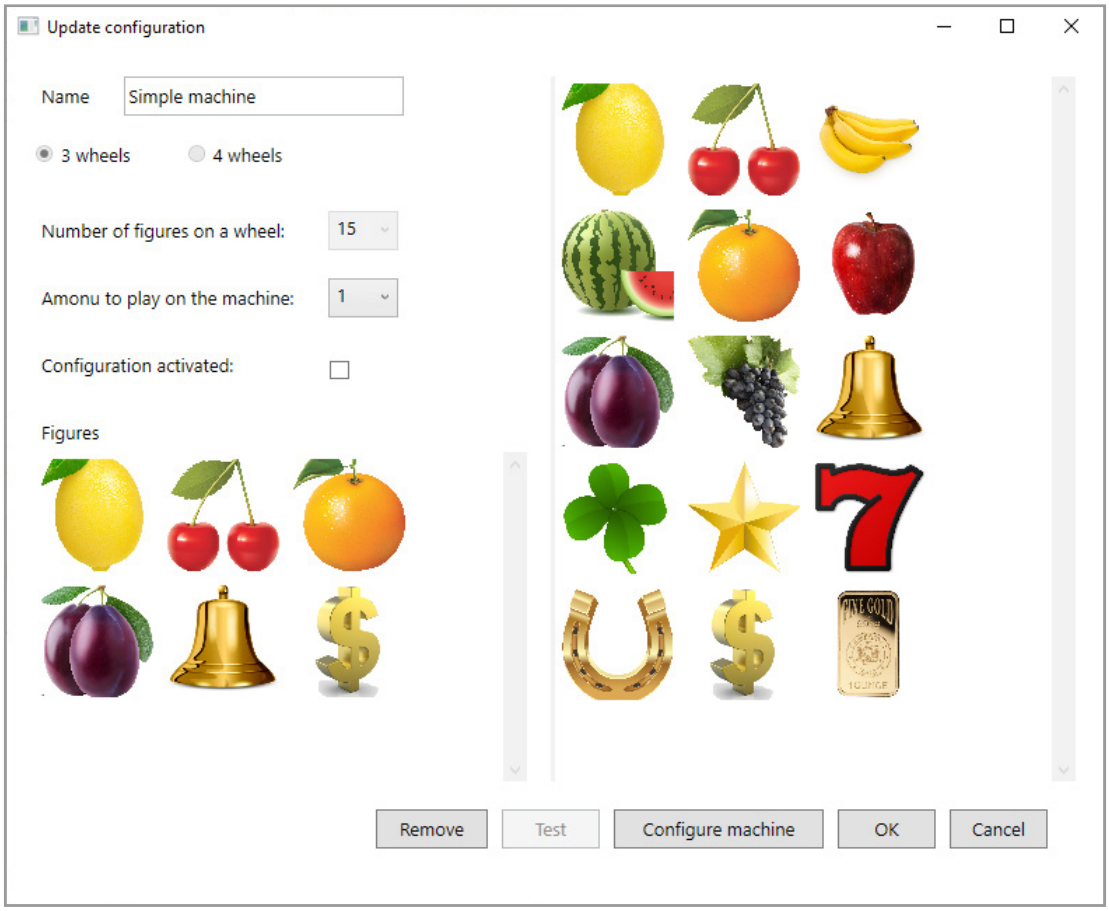
From this window you can select three functions:

1. Double click on a configuration to modify the configuration
2. Click the button *Create* to create a new configuration
3. Click the button *Figures* to add new figures to the database

If you select the last function you get the window below. Here you can add a figure when you clicks on *Upload*, and you get file browser to search the file to upload. It has been decided that a figure should be a square (same width and height) PNG image and it is the responsibility of the user to ensure it. If you want to remove a figure you must click on the image. When you upload an image the name is the filename. You can change the name if you double click the name. The changes you made to the list of figures are first stored in the database when you clicks *Save*. You should note that the function is primarily intended as an option to upload new figures, but deleting a figure the function does not test, if the figure is used in a configuration. You should therefore only delete figures that you are sure are not used.



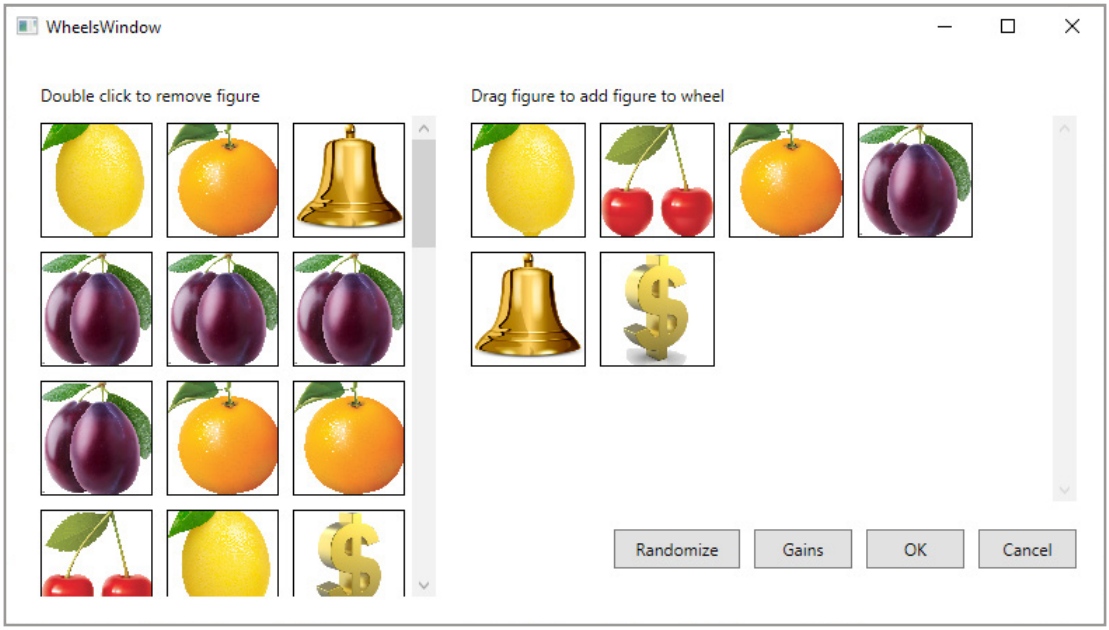
If you in the window which shows the configurations you double click on a configuration you get at window for that configuration:



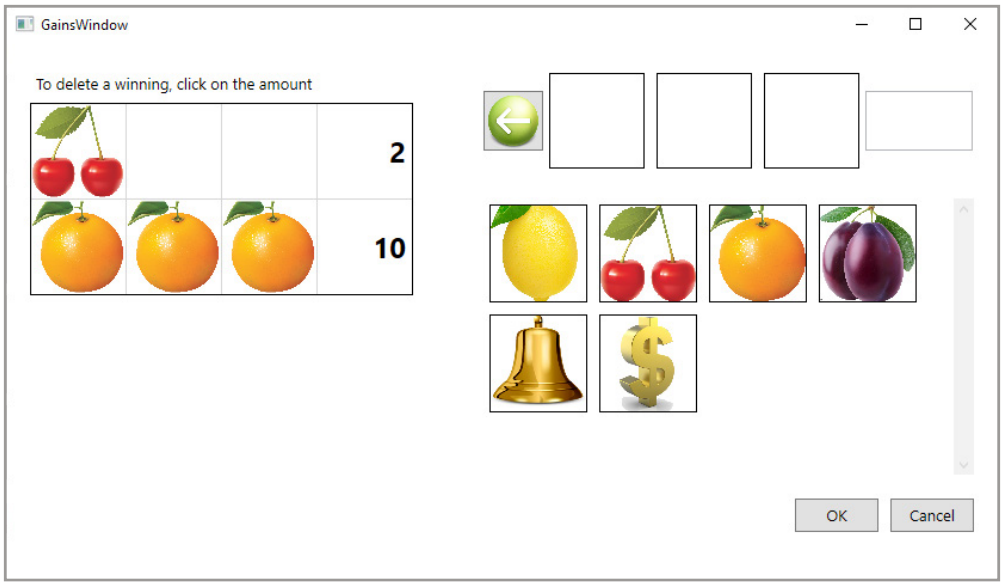
The right panel contains all figures uploaded to the database. The left panel contains information for this configuration:

1. The configuration name, which can be changed.
2. The number of wheels for this configuration. This value is defined when the configuration is created and cannot be changed.
3. The number of figures on each wheel. This value is defined when the configuration is created and cannot be changed.
4. The amount to play on the machine, a value which you can change.
5. Where the configuration is active, which means a user can select the configuration. Only a full configured configuration can be activated.
6. Below is a panel which contains the figures used by this configuration. You can drag configurations to this area from the figures in the right area.

If you click on the button *Configure machine* you get at window where you can assign figures to the wheels:

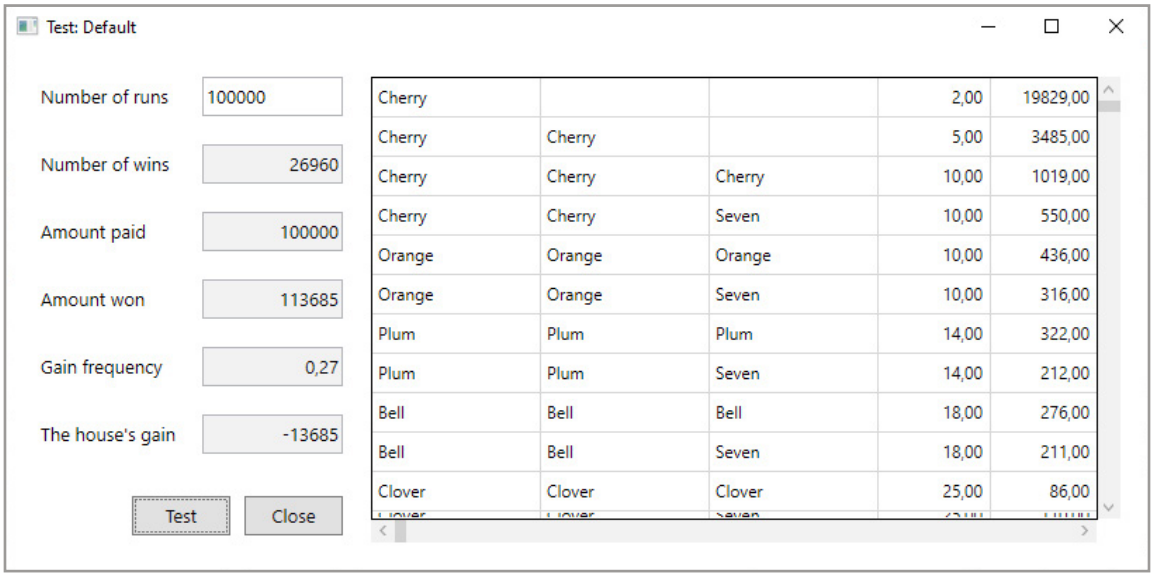


The left panel contains the figures used by this machine, and you add a figure to a wheel dragging the figure with the mouse. If you want to remove a figure you just double click the figure. The button *Randomize* is used to shuffle the figures, and the button *Gains* opens the following window where you can define winning combinations:



The lower right panel contains the figures used by this configuration. You define a winning combination by dragging figures from this panel to the fields above and when you have entered an amount you can click on the green button and the combination is added to the left table.

The window used to define a configuration has a button test. This button opens a window used to test the configuration. Below is shown a test of the default configuration:



The test simulates 100000 runs on the machine, and you get some information about the result. In this case you can see that the owner of the machine loses money equivalent to the machine giving too many gains. It is therefore necessary to adjust the machine.

7.8 THE LAST FEATURES

There is only three things back:

1. show winning combinations for the current configuration
2. show results for the current player
3. select another configuration

which are functions assigned to buttons in the toolbar. The first is no more than a simple table showing the different figure combinations and the amount. The next is is basically the same tool as used in managing users with the differences that the tool should not show user information and there should not be commands to manipulate the results. The tool only shows results for runs with wins.

The last function requires a little more. Basically it is the same function used when the administrator maintains configurations which starts with a window showing all configurations, but this time only active configurations. The window only has a *Close* button, but if the user double click on a configuration this configuration must be selected.

This requires some changes as the program only support a machine with 3 wheels. The class *Slot3Machine* is replaced with another class called *SlotMachine*, which is more dynamic and supports both 3 and 4 wheels. It is also a user control, and when the user selects another configuration the program must create another object of the type *SlotMachine* for the new configuration and replace the old control with the new control.

7.9 THE LAST THINGS

When the program is finished I have three things to do:

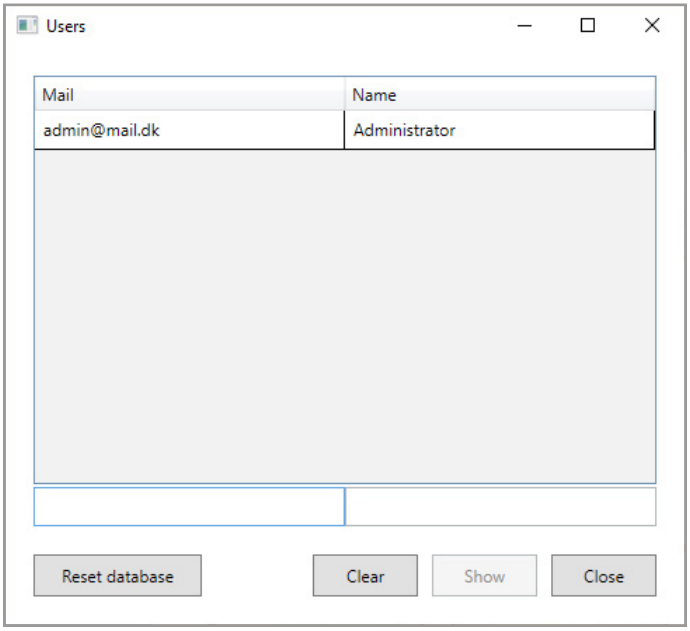
1. The usual code review and refactoring
2. Test
3. Create an install script

I should not list the result of the refactoring, just mention two things:

1. The class *DB* is extensive and several changes have been made here, corresponding to the fact that there are methods that perform almost the same thing and that there are methods that are no longer used. However, it is worth mentioning that it is not in itself a goal to make this class as simple as possible. It is more important that methods have a clear application and are easy to understand.

2. The program has many dialog boxes and not all have a view model. Many of the dialogs are simple and it can feel a bit overkilled to write a view model for all dialogs. On the other hand, there are benefits to being consistent and always writing a view model

There is one extension to the program. If you is logged in as administrator and select maintains of users the window has a new button:



Here the administrator can reset the machine and create a new database with only a default configuration. I do not think a real program should have such a function as it remove all data, but as part of a development it is a useful feature.

After the program is finished, it must be tested and it is not easy to test an application like this, with the primary emphasis on the user interface. When you (as part of the programming) have found that the program seems to work as expected, there is not much more to do than simply to play on the machine and in the following I will outline what you could do.

Before the test I have cleaned the database (executed the function *Reset database* as shown above). The result is a database without players and with a single default configuration.

Then I have performed a test for how to play at the machine:

1. Create a player and inserts 100 on the players account
2. Play at least 100 games with the player
3. Log out and create a second player and inserts 1000 on the players account
4. Play at least 50 games with that player
5. Close the program

6. Start the program again and create a third player
7. Log out and then log in as the first player and plays at least 50 games
8. Check the players gains by clicking the button in the toolbar
9. Check the overview of winnings by clicking the button in the toolbar
10. Close the program

To test the administrator functions I have change an event handler in *MainWindow* as:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  model.User = DB.Instance.Login(”admin@mail.dk”, ”1234567890”);
  int amount = DB.Instance.Holdings(model.User.Pid);
  model.Holdings = amount;
  //(new LoginWindow(model)).ShowDialog();
}
```

where the player is hardcoded as the administrator and so the machine is in admin mode (I do not the have to login all the time). Then I

1. Open the program
2. Change the administrator password
3. Log out as administrator
4. Log in as administrator again (with the new password)

Next I have created a configuration called *Big game*, that uses de same figures as the default configuration. The new configuration has fewer but bigger winnings, and the price to play is 2. After sawing the configuration, I opens it again and try to test it with 1000 and 10000 runs.

Then I in the same way creates another configuration called *Advanced*, but a configuration with 4 wheels.

Now I have three configurations, but only one is active and can be used by the players.

Then I have done the following:

1. From the administrator icon for users I opens the table with the users (there are three)
2. I changed the password for the user that above was created last
3. Log in as that user
4. Click the icon for select a configuration as a player - only the default configuration is active

5. Log in as administrator
6. In the window for configurations the two new configurations are defined as active
7. Login as the user again
8. Click the icon to select configuration again now there should be three configurations
9. Close the program

Next I have changed the *MainWindow* again, such I have to login. Then I opens the program and performs the following:

1. Log in as the first of the above three users
2. Play 20 games with the default configuration
3. Select the configuration *Big game*
4. Play at least 50 games with that configuration
5. Select the default configuration again
6. Play 20 games with the default configuration
7. Log out
8. Log in as the second player
9. Select the configuration *Advanced*
10. Play 100 games with that configuration
11. Log in as administrator
12. Open the configuration *Advanced*
13. Shuffle the wheels and save the configuration
14. Log out as administrator
15. Check the transactions for the current user.
16. Log out
17. Log in as the last user
18. Play 100 games with the *Old* configuration
19. Log out
20. Log in as the first user
21. Log in as administrator
22. Delete the *Old* configuration
23. Delete the two last users

If all goes as expected, I will consider the program to be completed - at least to be able to be used by house-selected test users.

As the last step the program must be assigned an icon and compiled as a release version. Then I must create a MSI script to install the program. I has as shown before used *Advanced Installer* and the only thing to be aware of is that the program uses SQLite and script thus has to include the required dll files and the same goes for *PaGUI.dll*.