



**Vlaamse Dienst voor Arbeidsbemiddeling en
Beroepsopleiding**

C#
2017

WPF

Inhoudsopgave

1	INLEIDING	1
1.1	Vereisten	1
1.2	Theorie	1
1.2.1	Designer-gericht	1
1.2.2	Developer-gericht	3
2	ALGEMENE STRUCTUUR	4
2.1	Eerste voorbeeld	4
2.2	FrameworkElement	6
2.3	Control	9
2.4	Layout	10
2.4.1	StackPanel	10
2.4.2	WrapPanel	12
2.4.3	DockPanel	12
2.4.4	Grid	12
2.4.5	Canvas	14
2.5	Code-behind	15
2.5.1	Click-event per Button	15
2.5.2	Dezelfde Click-EventHandler voor meerdere Buttons	16
2.5.3	Het Click-event activeren met het klavier	16
3	WERKEN MET TEKST	18
3.1	TextBlock	18
3.2	Label	20
3.3	TextBox	22
3.4	PasswordBox	23
4	BUTTONS	24
4.1	Button	24
4.2	ToggleButton	24
4.3	RepeatButton	26
4.4	RadioButton	28
4.5	CheckBox	29
4.6	Images in Buttons	31
4.7	Relatieve en Absolute URI's	32
5	LISTBOX – COMBOBOX	33
5.1	Simpele inhoud(tekst) in de Boxen	33

5.2	Complexe inhoud in de Boxen	37
6	MESSAGEBOX	40
7	MENU – TOOLBAR – STATUSBAR	43
7.1	Menu	44
7.1.1	Werkwijze	44
7.1.2	Command-items	46
7.1.3	Toevoegen van ShortCut Keys en HotKeys	47
7.2	ToolBar	49
7.3	StatusBar	52
8	BESTANDSOPERATIES	54
8.1	Common dialog box	54
8.1.1	SaveFileDialog	55
8.1.2	OpenFile Dialog	57
8.1.3	PrintDialog	61
8.2	PrintPreview - DocumentViewer	64
8.3	Afsluiten	66
9	LAYOUT MOGELIJKHEDEN	68
9.1	StaticResources	68
9.2	Colors en Brushes	69
9.2.1	Color	69
9.2.2	Brush	70
9.3	Voorbeeldapplicatie	76
9.3.1	De functionaliteit van het voorbeeld	76
9.3.2	De XAML-code	76
9.3.3	De klasse Kleur	77
9.3.4	De code-behind	77
9.4	Styles	79
9.5	Templates	80
9.5.1	Data Templates	80
9.5.2	Control Templates	81
9.6	Themes	84
10	DATABINDING	85
10.1	Binding to ... een Control	85
10.1.1	UpdateSourceTrigger	87
10.2	Binding to ... een Collection	88
10.2.1	DataContext	89
10.3	Binding to ... een klasse	89
10.3.1	INotifyPropertyChanged	90
10.4	Binding parameters	92

10.4.1	StringFormat	93
10.4.2	ConverterCulture	94
11	DRAG AND DROP	95
11.1	Een simpel object	95
11.1.1	Drag-fase	95
11.1.2	Drop-fase	96
11.1.3	Testen op kleurvolgorde	98
11.2	Een complex object	99
12	RIBBON	104
12.1	ApplicationMenu	105
12.1.1	RibbonApplicationMenuItem	105
12.1.2	RibbonSeparator:	106
12.1.3	RibbonApplicationSplitMenuItem	106
12.1.4	FooterPaneContent	107
12.1.5	AuxiliaryPaneContent	107
12.2	RibbonTab	109
12.3	ContextualRibbonTab	111
12.4	QuickAccessToolBar (QAT)	113
12.5	QAT-items bijvoegen	113
12.6	HelpPaneContent	114
12.7	SuperToolTip	114
13	CONVERTERS	116
14	USER EN APPLICATION SETTINGS	119
14.1.1	Opslaan van toegevoegde QAT-items	119
14.1.2	Opslaan MRU-bestanden	121
15	WPF PROJECT MET MVVM-PATTERN	124
15.1	Toevoegen van MVVM Framework aan VS	124
15.2	Simpel voorbeeld	124
15.2.1	Model	126
15.2.2	ViewModel	126
15.2.3	View	130
15.3	Voorbeeld met data uit een database	132
15.3.1	Model	133
15.3.2	ViewModel	133
15.3.3	View	136
16	DEPLOYMENT	140
17	COLOFON	142

1 INLEIDING

Windows Presentation Foundation (WPF) is een nieuwe API (application programming interface) om een grafische applicatie te maken op een desktop platform. Het is een alternatief voor WinForms, maar WPF is naar ontwikkelingsmogelijkheden veel krachtiger, want WPF is vector-based, resolutie onafhankelijk en maakt gebruik van de multimedia mogelijkheden van de hedendaagse computers.

Met WPF kan je 2 soorten applicaties ontwikkelen:
een Windows Applicatie (vergelijkbaar met een Winforms applicatie) en een XAML Browser Application (verwant aan een Silverlight Applicatie).

1.1 Vereisten

Voorkennis : de cursus C# basiscursus.

PC-configuratie : .NET Framework 3.0 of hoger, Windows 7 of 8.

1.2 Theorie

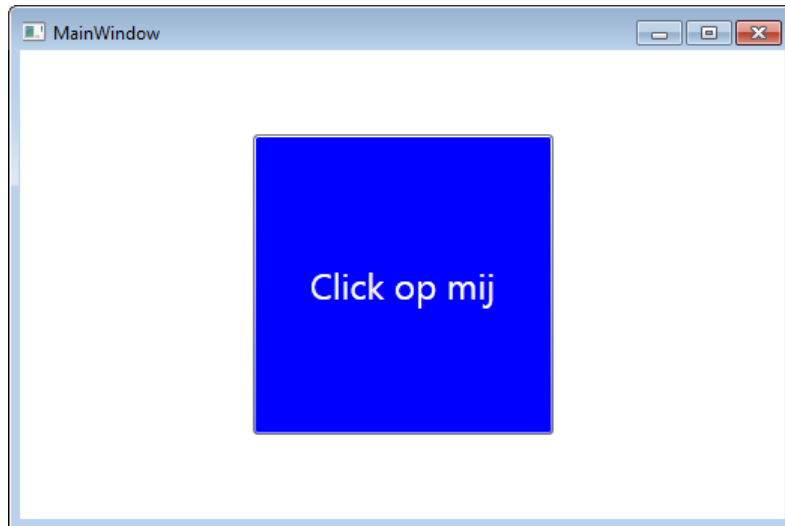
Binnen WPF zijn er 2 aspecten aanwezig:
het uiterlijk (bv. hoe ziet een button eruit) of designer-gericht en het gedrag (bv. wat gebeurt er als je op de button klikt) of developer-gericht. Hierdoor is het mogelijk om deze rollen van de designer en developer totaal gescheiden te houden.

1.2.1 Designer-gericht

Het uiterlijk wordt voornamelijk gespecificeerd in een XAML-bestand. XAML staat voor Extensible Application Markup Language. Je mag XAML als een subset van XML beschouwen en grof stellen dat XAML de HTML is van een Windows applicatie.

Het document bestaat uit *tags*, waarvan de *root-tag* bepaalt of het een windows of een browser pagina wordt, en *sub-tags* die op zich elk een instance van een (control) object voorstellen, en waarvan de attributen de properties en events voorstellen.

Dit voorbeeld kan je nu gewoon even doorlezen, daarna ga je dit uitwerken in praktijk: een Windows-venster met een grote blauwe knop erin. Als je er op klikt, verandert de tekst op de knop naar "Je hebt geklikt".



```
<Window x:Class="WpfCursus.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Button Name="ButtonKlik" Click="ButtonKlik_Click"
            Background="Blue" Foreground="White" Width="200" Height="200"
            Content="Click op mij" FontSize="24" />
</Window>
```

Window = een instance van de klasse System.Windows.Window

Button = een instance van de klasse System.Windows.Controls.Button

- Name = property : om de naam van de control te bepalen
- Click = event : de naam van de procedure die wordt opgeroepen bij het klikken op de knop (de procedure zelf wordt in het code-behind venster geschreven)
- Background = property : om de achtergrondkleur te bepalen
- Foreground = property : om de voorgrondkleur = tekstkleur te bepalen
- Width = property : de breedte van de button
- Height = property : de hoogte van de button
- Content = property : de tekst op de knop
- Fontsize = property : lettertype grootte van de tekst op de knop

Indien je nu de *Window-tag* vervangt door een *Page-tag*, dan krijg je een pagina die voor een browser bedoeld is:

```
<Page x:Class="WpfCursus.Page1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
      Title="Page1">
    <Button Name="ButtonKlik" Click="ButtonKlik_Click"
            Background="Blue" Foreground="White" Width="200" Height="200"
            Content="Click op mij" FontSize="24" />
</Page>
```

XAML kan zowel “gerenderd”-vertaald (in een browser) als gecompileerd (in een window) worden. Indien er achterliggende code (C# of VB) gebruikt is, dan zal dit natuurlijk alleen werken als alles gecompileerd is.

1.2.2 Developer-gericht

Wat een bepaald event teweegbrengt in de applicatie, wordt geschreven in een code-behind venster, dus in een C# of VB bestand.

Om in dit voorbeeld de button zijn functie te laten doen, wordt er in het code-behind venster volgende code toegevoegd:

```
private void ButtonKlik_Click(object sender, RoutedEventArgs e)
{
    ButtonKlik.Content = "Je hebt geklikt";
}
```

In deze cursus behandelen we alleen de Windows-applicatie kant van WPF (wat in praktijk de opvolger of vervanger van Winforms zal zijn).

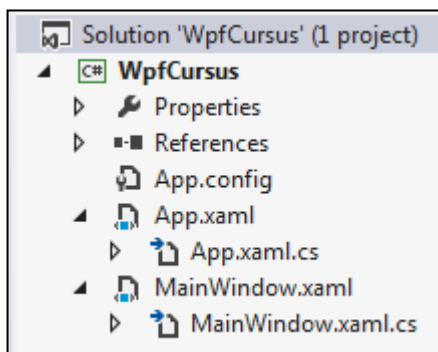
2 ALGEMENE STRUCTUUR

2.1 Eerste voorbeeld

Een WPF applicatie bestaat uit één of meerdere pagina's.

Je maakt een eerste WPF applicatie:

- Kies in het menu File de opdracht New en dan Project.
- Kies links bij Project Types voor Visual C#.
- Kies rechts bij Templates voor WPF App (.NET Framework).
- Tik bij Name de naam van het project WpfCursus, verander eventueel de Location en de Solution Name en kies OK.



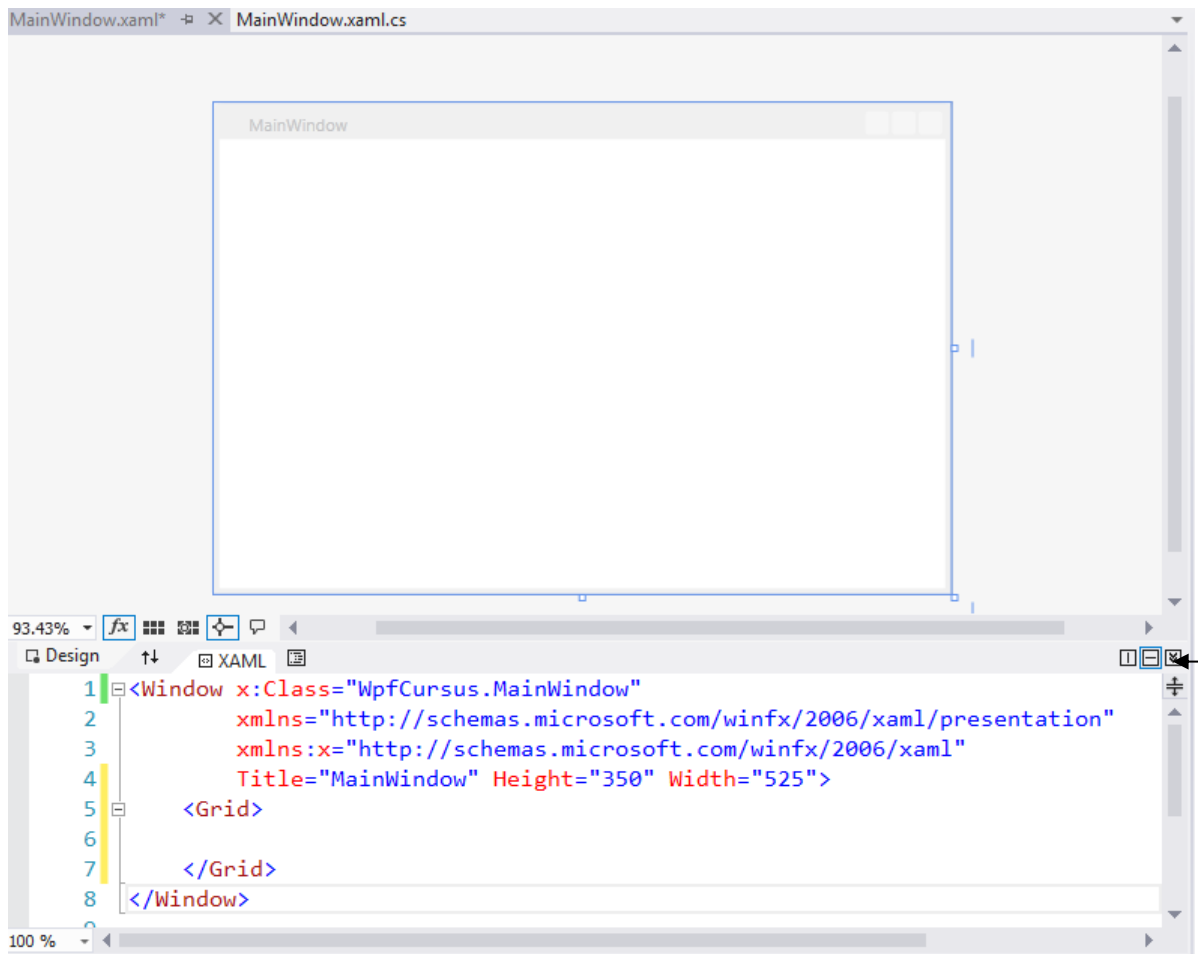
Je ziet in de Solution Explorer dat het project reeds vijf bestanden bevat, MainWindow.xaml (het xaml-bestand zelf), MainWindow.xaml.cs (het code-behind bestand), App.xaml (applicatie bestand), App.xaml.cs (het code-behind bestand en App.config (het configuratie-bestand).

Het App.xaml bestand is niet grafisch visualiseerbaar, maar qua inhoud omvat het vooral resources die door het programma gebruikt worden en de naam van het opstartscherm. (`StartupUri="MainWindow.xaml"`).

Je ziet in MainWindow.xaml.cs dat MainWindow een class is, die erft van de class Window (uit het .NET Framework).



In het midden van Visual Studio.NET zie je het MainWindow in horizontale split-view (op de scheiding aan de rechterkant kan je de andere mogelijkheden instellen)



Omdat de standaardbenaming van het venster, de klasse enz. vrij algemeen zijn, is het nuttig om alles een betekenisvolle naam te geven. Hierbij moet je wel opletten dat alle verbanden intact blijven.

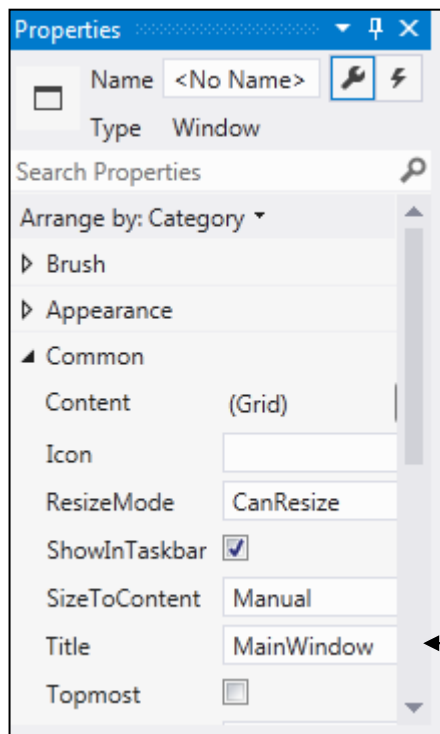
Je moet volgende stappen ondernemen:

- Selecteer MainWindow.xaml in de Solution Explorer, en kies voor Rename
- Verander de naam naar ButtonWindow. Het code-behind bestand wordt automatisch mee aangepast.
- Open het App.xaml bestand en verander de `StartupUri="MainWindow.xaml"` naar `StartupUri="ButtonWindow.xaml"`.
- Open het code-behind bestand en klik met de rechtermuisknop op MainWindow in de regel `public partial class MainWindow : Window`
- Kies Rename..., verander de naam hier ook in de ButtonWindow, vink alles aan en klik OK
Bevestig ook de veranderingen.

Om inhoudelijk veranderingen aan te brengen, kan je op drie manieren te werk gaan:

- Via de xaml-view (code-view).
- Via de design-view (niet aan te raden)

- Via het Properties venster met zowel de Properties als de Events tab.

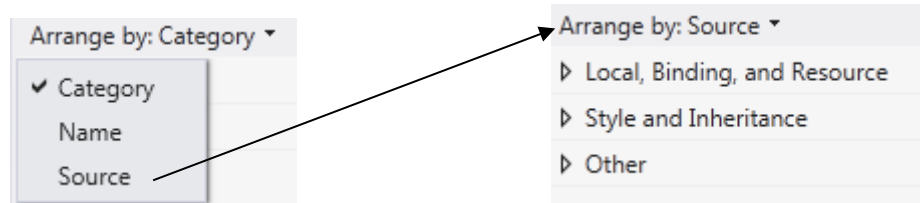


Om in de titelbalk de naam van de applicatie te wijzigen, verander je de **Title – Property** in bv. Button Applicatie.

Synchroom wordt dan in de xaml-view de Title attribuut van Window naar de nieuwe waarde veranderd.

```
<Window x:Class="WpfCursus.ButtonWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Button Applicatie"
Height="350" Width="525">
```

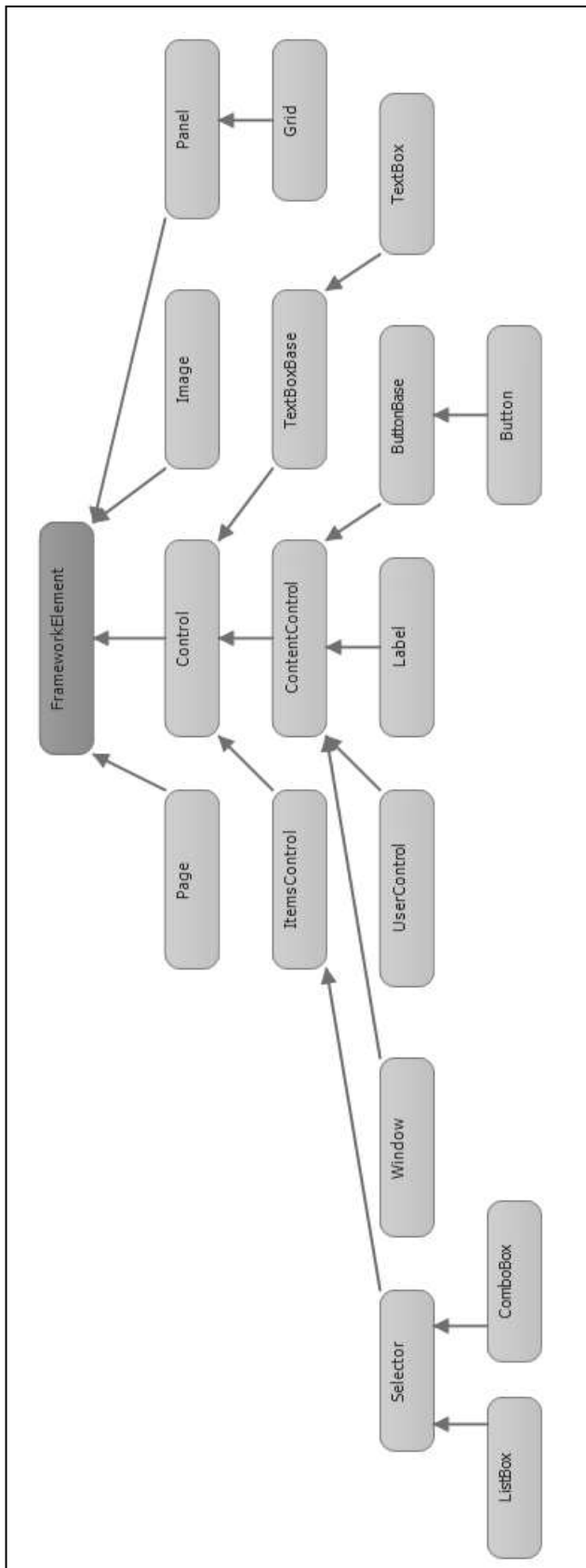
Binnen het *Properties Window* heb je voor de *Properties* 3 manieren om alle eigenschappen te tonen:



Je kan XAML-code in commentaar zetten door bij het begin `<!--` te zetten, en aan het einde van de commentaar `-->`

2.2 FrameworkElement

Bijna alle elementen die je gaat gebruiken zijn afgeleide klassen van de klasse `FrameworkElement` of van `Control`. Hieronder vind je een deeldiagram van de klassen en hun relatie tot elkaar.



Dit FrameworkElement omvat al een aantal *Properties* die je dus bij elk element kan terugvinden :

Name : De naam geeft je een identificatie van het element. Hierdoor is het mogelijk om in het code-behind venster (C# of VB venster) deze

referentie te gebruiken om er code aan vast te hangen.
Via het properties-venster kan je rechtstreeks een naam invullen waardoor in XAML het attribuut *Name="naam"* wordt gegenereerd. Indien je rechtstreeks in XAML het name-attribuut maakt, kan je zowel *Name* als *x:Name* gebruiken. *x:Name* is in dit geval de juiste benaming, maar het framework heeft ervoor gezorgd dat *Name* een alias van *x:Name* is, en deze dus als gelijkwaardig behandeld worden.

Als je te maken krijgt met groottes en breedtes of andere afmetingen, dan zijn alle ingegeven waarden default in px (pixels) wat overeenkomt met 1/96th inch per eenheid. Hierdoor kan je ze device-onafhankelijk definiëren.

Je kan ook gebruik maken van andere eenheden zoals :

in is inches; 1in=96px

cm is centimeters; 1cm=(96/2.54) px

pt is points; 1pt=(96/72) px

Margin : Hiermee stel je de hoeveelheid witruimte in aan de buitenkant van het element (left, right, top en bottom).

1 waarde = alle vier de kanten hebben dezelfde waarde

2 waarden = eerste → left en right, tweede → top en bottom

4 waarden = left, top, right, bottom

Een veelgebruikt object die deze waarden ook definieert is *Thickness*.

Dit object heeft 2 constructors nl.

new Thickness(waarde) → definieert voor alle marges dezelfde waarde

new Thickness(w1, w2, w3, w4) → definieert respectievelijk een left, top, right en bottom marge.

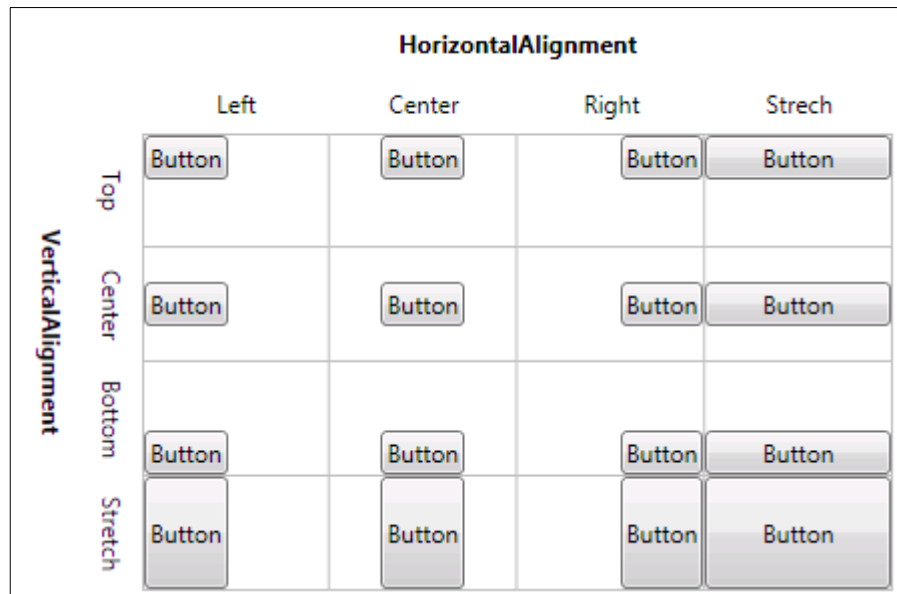
Height : Hiermee stel je de hoogte van een element in. Er zijn twee aanverwante properties die in sommige gevallen handiger zijn om mee te werken nl. **MinHeight** en **MaxHeight**. Via de property **ActualHeight** kan je de echte hoogte van het element opvragen.

Width: Hiermee stel je de breedte van een element in. Er zijn twee aanverwante properties die in sommige gevallen handiger zijn om mee te werken nl. **MinWidth** en **MaxWidth**. Via de property **ActualWidth** kan de echte breedte van het element opvragen.

ToolTip : als je met de muis over de TextBlock gaat, verschijnt de tekst gespecificeerd in deze property

HorizontalAlignment : Zet de horizontale uitlijning van het element t.o.v. het panel waarin het zich bevindt. Standaard staat deze op *Stretch* (= het uitrekken van het element tot de volledige ruimte in horizontale zin gevuld is.) Mogelijke andere waarden: *Left*, *Center* en *Right*.

VerticalAlignment : Zet de verticale uitlijning van het element t.o.v. het panel waarin het zich bevindt. Standaard staat deze op *Stretch* (= het uitrekken van het element tot de volledige ruimte in verticale zin gevuld is.) Mogelijke andere waarden: *Top*, *Center* en *Bottom*.



Tag : Hierin kan je een willekeurige waarde stockeren voor dit bepaald element (dat eventueel in code verwerkt kan worden)

2.3 Control

Een Control afgeleid van het FrameworkElement geeft meer mogelijkheden naar bv. layout toe. Dus naast alle geërfde properties komen er nog zeer interessante properties bij:

Alle FrameworkElement properties (Height, Width, Margin, ...)

Background	de achtergrond van het TextBlock kan in een volle kleur gezet worden
BorderBrush	bepaalt de kleur van de rand van de control
BorderThickness	bepaalt de dikte van de rand van de control
Foreground	de tekst zelf kan in een volle kleur gezet worden
FontFamily	soort lettertype
FontSize	grootte van het lettertype: getal groter dan 0 kan in verschillende eenheden wordt uitgedrukt: px = 1/96 inch in = in inches cm = in centimeters pt = in punten = 96/72 px
FontStretch	hoeveel moet een karakter worden uitgerekt (tot 200%) of versmald (tot 50%)
FontStyle	normal, <i>italic</i> (= schuin designed) of <i>oblique</i> (= kunstmatig schuingezet)

FontWeight	de zwaarte van de karakters: normal = 400, thin = 100, extrablack of ultrablack = 950 of verschillende waarden ertussen
IsEnabled	kan de control door de gebruiker worden aangesproken
IsVisible	is de control te zien op het scherm
Opacity	de doorzichtigheid (0% niet te zien → 100% niet doorzichtig)
Padding	de witruimte die gelaten wordt aan de binnenkant van de control

2.4 Layout

Een deel van de elementen waar je mee werkt, wordt afgeleid van de *ContentControl* (= Control met Content). Zij laten maar 1 Content (inhoud) toe bv. één knop in een Window. Om meerdere controls in een bepaald element te krijgen bv. 5 knoppen in een window, moet je gebruik maken van één of ander *Panel*.

De basisklasse van deze *Panels* is de abstracte klasse *Panel* die zelf is afgeleid van *FrameworkElement*. De belangrijkste property van *Panel* is:

Children : geeft een Collection terug van alle Child-elementen binnen dit panel. Deze collection is van het type *UIElementCollection*, dus als je specifieker wilt werken, moet je de elementen nog *casten* naar de juiste afgeleide klassen.

Om de verschillende soorten *Panels* duidelijk uit te leggen, ga je gebruikmaken van een basisvoorbeeld: in het nieuw aangemaakte project, zorg je ervoor dat alleen de <Window>-tag in het XAML-venster nog aanwezig is.

Voeg binnen de <Window>-tag volgende buttons toe:

```
<Window x:Class="WpfCursus.ButtonWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Button Applicatie" Height="350" Width="525">
  <Button Name="ButtonRed" Margin="10"
    Content="Rode achtergrond"></Button>
  <Button Name="ButtonGreen" Margin="10"
    Content="Groene achtergrond"></Button>
  <Button Name="ButtonBlue" Margin="10"
    Content="Blauwe achtergrond"></Button>
</Window>
```

Dit geeft een foutmelding tot gevolg, want <Window> kan maar één *Content* bevatten. Dus ga je een *Panel* rond de *Buttons* opbouwen.

2.4.1 StackPanel

De *StackPanel* is een simpele en goed bruikbare manier om zijn *Children* horizontaal of verticaal te stapelen.

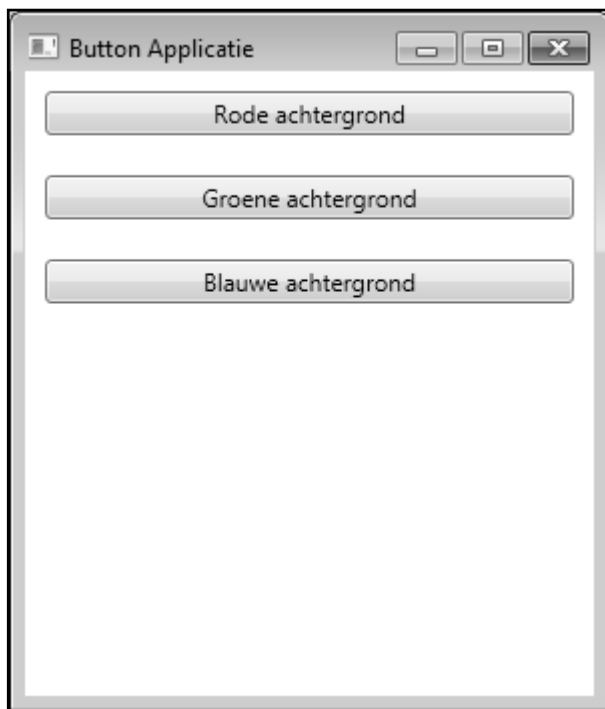
Voeg deze tag toe rond de Buttons zodat dit Panel de enige Content wordt van <Window>.

```
<Window x:Class="WpfCursus.ButtonWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Button Applicatie" Height="350" Width="525">
    <StackPanel>
        <Button Name="ButtonRed" Margin="10"
            Content="Rode achtergrond"></Button>
        <Button Name="ButtonGreen" Margin="10"
            Content="Groene achtergrond"></Button>
        <Button Name="ButtonBlue" Margin="10"
            Content="Blauwe achtergrond"></Button>
    </StackPanel>
</Window>
```

Misschien toch in je achterhoofd houden wat de *default* waarde van bepaalde properties zijn:

- HorizontalAlignment : Stretch
- VerticalAlignment : Stretch
- Orientation : Vertical → dit wil zeggen dat alles in 1 kolom onder elkaar gezet wordt, en wat niet binnen het venster kan, valt er gewoon buiten.

De applicatie is nu foutvrij en kan gecompileerd worden. In uitvoering ziet het resultaat er als volgt uit:



Door in de begin-tag van het StackPanel een paar attributen toe te voegen bv.

```
<StackPanel HorizontalAlignment="Left" VerticalAlignment="Bottom"
    Orientation="Horizontal">
```

verander je het zicht en de plaatsing van de *Buttons* totaal. Let op : als bv. het *Panel* te smal is voor de 3 *Buttons* dan valt dit gewoon buiten het venster ! Probeer alle combinaties i.v.m. *HorizontalAlignment*, *VerticalAlignment* en *Orientation* maar eens uit, en let vooral op het resultaat !

2.4.2 WrapPanel

Deze vorm van *Panel* is iets flexibeler dan de *StackPanel* omdat dit *Panel* meerdere rijen of kolommen aankan. Met andere woorden: is het venster te smal, dan wordt bv. de volgende *Button* onder de eerste gezet, enz. Dit voorkomt dat er elementen buiten het venster vallen.

Probeer volgende combinatie bv. maar eens uit:

```
<WrapPanel HorizontalAlignment="Left" VerticalAlignment="Bottom"
Orientation="Horizontal">
```

2.4.3 DockPanel

Een andere manier om elementen bv. *Buttons* te schikken binnen een *Panel* is gebruikmaken van een *DockPanel*. Dit soort panel laat toe om per *Child* in dit geval dus per *Button* in te stellen aan welke zijde van het *Panel* deze geplakt (=docked) moet worden.

Verwijder de <StackPanel>-tags en plaats een <DockPanel>-tag in de plaats.

```
<DockPanel>
. . .
</DockPanel>
```

Probeer maar uit.

De *Children* worden van links naar rechts in *Stretch*-vorm getoond, en de laatste *Button* wordt uitgerekt tot de hele oppervlakte bedekt is. Dit komt omdat de property *LastChildFill* standaard op *True* staat. Verander dit naar *False* en bekijk het resultaat.

```
<DockPanel LastChildFill="False">
. . .
</DockPanel>
```

De plaatsing van de *Children* wordt altijd t.o.v. de resterende ruimte binnen het *Panel* voorzien. Per *Child* kan je bepalen aan welke van de 4 kanten (Left, Right, Top of Bottom) hij geplaatst moet worden met de property *DockPanel.Dock*. bv.

```
<DockPanel LastChildFill="False">
    <Button DockPanel.Dock="Right" Name="ButtonRed"
        Margin="10" Content="Rode achtergrond"></Button>
    <Button DockPanel.Dock="Left" Name="ButtonGreen"
        Margin="10" Content="Groene achtergrond"></Button>
    <Button DockPanel.Dock="Bottom" Name="ButtonBlue"
        Margin="10" Content="Blauwe achtergrond"></Button>
</DockPanel>
```

Experimenteer maar met de verschillende mogelijkheden.

2.4.4 Grid

Als je nood hebt aan een soort tabelstructuur, een raster om bv. een formulier netjes te vormen, dan kan je gebruikmaken van de *Grid*-tag.

Om het noorden niet kwijt te geraken, heeft *Grid* een property *ShowGridLines* om een idee te hebben in wel stukje *Grid* je bezig bent. De enige visuele vorm is een stippellijn.

De *Grid*-tag moet helemaal gedefinieerd worden: hoeveel *Rows* (rijen), hoeveel *Columns* (kolommen) en per *cel* moet worden bepaald in welke vakje deze thuishoort.

Naar het voorbeeld toe: wis alle info over DockPanel en start juist onder de <Window>-tag.

Je gaat een Grid maken van 3 rijen op 2 kolommen:

```
<Grid ShowGridLines="True">
    <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Button Name="ButtonRed" Margin="10"
        Content="Rode achtergrond"></Button>
    <Button Name="ButtonGreen" Margin="10"
        Content="Groene achtergrond"></Button>
    <Button Name="ButtonBlue" Margin="10"
        Content="Blauwe achtergrond"></Button>
</Grid>
```

Resultaat: je ziet alleen de laatste *Button* staan, want ze staan alle 3 boven elkaar.

Nu moet je dus per Button gaan bepalen in welke cel van de grid hij thuishoort bv.

```
<Button Grid.Row="0" Grid.Column="0" Name="ButtonRed" Margin="10" Content="Rode
achtergrond"></Button>
<Button Grid.Row="1" Grid.Column="1" Name="ButtonGreen" Margin="10"
Content="Groene achtergrond"></Button>
<Button Grid.Row="2" Grid.Column="0" Name="ButtonBlue" Margin="10"
Content="Blauwe achtergrond"></Button>
```

Zolang er bij de *RowDefinitions* en *ColumnDefinitions* niets gespecificeerd is met properties, blijven ze zich gelijkmatig verdelen, en wordt het *Grid* zelf over het ganse *Window* uitgestreken (*Stretch*).

Pas het voorbeeld aan naar volgende XAML:

```
<Grid ShowGridLines="True" Height="250">
    <Grid.RowDefinitions>
        <RowDefinition Height="auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
        <RowDefinition Height="2*"></RowDefinition>
    </Grid.RowDefinitions>
    . . .
```

De hoogte van de *Grid* is nu 250 eenheden. De *RowDefinitions* bepalen de hoogte van alle rijen:

auto : de hoogte nodig voor de *Button*

*** en 2*** : de overschot van de hoogte wordt verdeeld tussen de 2 overige rijen waarbij de onderste dubbel zo hoog is als de andere.

waarde : je hebt ook nog de mogelijkheid om een getal in te vullen als hoogte. Hierdoor wordt deze rij exact zoveel eenheden hoog.

Dit principe kan je ook toepassen op de *ColumnDefinition* maar dan wel met de *Width* property.

Voor de *Buttons* staan zowel voor *HorizontalAlignment* en *VerticalAlignment* op *Stretch*. Hierdoor wordt de cel volledig opgevuld door de *Button*.

Door te experimenteren met deze properties krijg je nog verschillende effecten te zien binnen een bepaalde cel bv.

```
<Button Grid.Row="0" Grid.Column="0" Name="ButtonRed" Margin="10" Content="Rode
achtergrond"></Button>
<Button VerticalAlignment="Top" Grid.Row="1" Grid.Column="1" Name="ButtonGreen"
Margin="10" Content="Groene achtergrond"></Button>
<Button VerticalAlignment="Bottom" Grid.Row="2" Grid.Column="0"
Name="ButtonBlue" Margin="10,10,10,0" Content="Blauwe achtergrond"></Button>
```

Het samenvoegen van kolommen kan je verwezenlijken met de property *Grid.ColumnSpan*, en het samenvoegen van rijen doe je door de property *Grid.RowSpan* te gebruiken.

Voeg bij de *RowDefinitions* onderaan `<RowDefinition></RowDefinition>` toe en plaats voor de *Grid*-eindtag volgende code

```
<TextBlock Grid.Row="3" Grid.ColumnSpan="2">Dit is tekst over de hele breedte
van het grid</TextBlock>
```

Bekijk het resultaat.

Je kan de gebruiker toelaten om zelf bv. de breedte van de kolommen aan te passen. Hiervoor gebruik je de *GridSplitter*. Dit element moet binnen de *Grid*-tags staan zoals bv.

```
<GridSplitter Grid.Row="0" Grid.RowSpan="3" BorderBrush="Black"
BorderThickness="2"></GridSplitter>
```

Wat resulteert in een verticale zwarte splitter vanaf de bovenste rij tot aan de samengevoegde rij.

2.4.5 Canvas

De *Canvas Panel* is de simpelste van de layoutpanels. Alle elementen binnen de *Canvas* moet met coördinaten expliciet gepositioneerd worden.

Deze vorm wordt weleens gebruikt voor 2D-tekeningen, maar naar een gebruikersinterface toe, is dit geen geschikt element.

Zet alles in verband met het *Grid* even in commentaar en voeg volgende code toe:

```
<Canvas Margin="50" Background="Yellow">
    <Rectangle Canvas.Left="10" Canvas.Top="10" Width="50"
        Height="25" Fill="Red"></Rectangle>
    <Ellipse Canvas.Bottom="20" Canvas.Right="20" Width="50"
        Height="25" Fill="Blue"></Ellipse>
</Canvas>
```

Resultaat : een geel vlak dat een marge van 50 eenheden heeft ten opzichte van het venster. Binnenin een rechthoek die linksboven staat op 10 eenheden van de

Canvas linkerbovenhoek en een ovaal die rechtsonder staat op 20 eenheden van de *Canvas* rechteronderhoek.

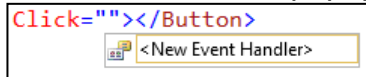
Dus alle ingestelde marges (*Canvas.Left*, *Canvas.Top*, *Canvas.Right* en *Canvas.Bottom*) worden relatief (vanuit de rand van de *Canvas* zelf) berekend.

2.5 Code-behind

2.5.1 Click-event per Button

Om de *Buttons* nu nog iets te laten doen, heb je het code-venster nodig. Er zijn meerdere manieren om daar te geraken bv.

- Een eenvoudige manier is bv. dubbelklikken op de *Button* zelf bv.
dubbelklikken op de *Button* met de tekst "Rode achtergrond".
Hierdoor kom je in het code-behind-venster terecht of beter gezegd in het bestand *ButtonWindow.xaml.cs*.
- In de XAML-code van de *Button* binnen de tag *Click* te tikken (en/of te kiezen) waardoor dat er een popup komt



Door te kiezen voor *<New Event Handler>* wordt de event-handler gemaakt, benoemd en klaargezet in het code-behind venster. Je kan eventueel ook een reeds bestaande *EventHandler* kiezen als er al één bestaat.

Aan de hand van de benaming kan je zien dat je in de *Click-EventHandler* terechtgekomen bent van de *ButtonRed*.

De code zelf ziet er uit als volgt:

```
private void ButtonRed_Click(object sender, RoutedEventArgs e)
{
    this.Background = new SolidColorBrush(Colors.Red);
}
```

(1)

- (1) *this* → de klasse zelf = *ButtonWindow*
 Colors → verzameling van alle voorgedefinieerde kleuren
 SolidColorBrush → kleurt een gebied met een volle kleur

Zo ook de code voor de twee andere *Buttons*:

```
private void ButtonGreen_Click(object sender, RoutedEventArgs e)
{
    this.Background = new SolidColorBrush(Colors.Green);
}

private void ButtonBlue_Click(object sender, RoutedEventArgs e)
{
    this.Background = new SolidColorBrush(Colors.Blue);
}
```

2.5.2 Dezelfde Click-EventHandler voor meerdere Buttons

Deze code gaan we optimaliseren, want uiteindelijk doe je 3x hetzelfde.

Eerst voegen we in de XAML in de *Button*-tag een *Tag*-attribuut toe nl. de naam van de kleur bv.

```
<Button Grid.Row="0" Grid.Column="0" Name="ButtonRed" Margin="10"
        Content="Rode achtergrond" Click="ButtonRed_Click"
        Tag="Red">
</Button>
```

Dan ga je in het code-behind-venster de event van één van de drie Buttons hernoemen naar *ButtonKleur_Click*...

In het XAML-venster verander je de 3 *Click*-attributen naar *ButtonKleur_Click* en voeg je een "Tag" toe moet de juiste kleur.

In het code-behind-venster wis je de 2 originele *Click-EventHandlers*.

Nu ga je gebruikmaken van de *Tag*-property van de *Button* om de juiste kleur te tonen bij het klikken op de *Button*.

Hiervoor verander je de code naar:

```
private void ButtonKleur_Click(object sender, RoutedEventArgs e)
{
    Button knop = (Button)sender;                                (1)
    SolidColorBrush kleur = (SolidColorBrush)new BrushConverter().
    ConvertFromString(knop.Tag.ToString());                        (2)
    this.Background = kleur;                                      (3)
}
```

- (1) de *sender* is het element dat het event teweegbrengt. Dit element kan naar een *Button* gecast worden.
- (2) in *knop.Tag* staat de kleur met zijn voorgedefinieerde naam als een object gedefinieerd. Met de *ToString()* wordt dit geconverteerd naar een *String*. Via een *BrushConverter.ConvertFromString* wordt dit omgevormd tot een *Brush* die dan op zijn beurt gecast wordt naar een *SolidColorBrush*.
- (3) De *Background* van het *Window* wordt in die nieuwe kleur gezet.

2.5.3 Het Click-event activeren met het klavier

Je kan in de *Content*-property een onderlijningsteken gebruiken om een letter van de tekst in combinatie met de *Alt*-toets te associëren met de *Click-EventHandler* van die *Button*.

Content="_Rode achtergrond", *Content="_Groene achtergrond"* en *Content="_Blaue achtergrond"*

Hierdoor krijg je bij de toetscombinatie *Alt R* een rode achtergrond, enz.

De *ESC*- en de *Enter*-toets zijn 2 speciale toetsen op het klavier, die meestal staan voor *Cancel* en *Enter (of Default)*.

In XAML hebben *Buttons* een *IsDefault* en een *IsCancel* attribuut die op *True* gezet kunnen worden, zodat deze de *Click-EventHandlers* activeren bij de aanslag van respectievelijk de Enter en de Escape toets.

Je kan dit uitproberen.



Opdracht: Verkeerslicht

3 WERKEN MET TEKST

Werken met tekst kan 2 dingen omvatten: je toont tekst aan de gebruiker die hij niet kan wijzigen (TextBlock en Label), of je laat de gebruiker toe om tekst in te tikken (TextBox en PassWordBox).

Aan de hand van een voorbeeld ga je de meest bruikbare attributen(properties) van deze elementen leren kennen.

Het uiteindelijke resultaat ziet eruit als volgt:



Je maakt een nieuwe WPF Application met de naam TekstVerwerken. Zorg ervoor dat MainWindow is hernoemd naar TextWindow (zowel het bestand als de interne klasse).

Klik met je rechtermuisknop op de naam van het project in de Solution Explorer en kies "Set as Startup Project" om dit project te kunnen starten.

Voeg volgende XAML toe:

```
<Window x:Class="TekstVerwerken.TextWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Werken met tekst" Height="400" Width="525">
    <StackPanel Margin="25">
        <TextBlock>Dit is een voorbeeld van het gebruik van WERKEN MET TEKST
    </TextBlock>
    </StackPanel>
</Window>
```

3.1 TextBlock

TextBlock is rechtstreeks afgeleid van de FrameworkElement klasse en hierdoor geen "echte" Control. Toch zijn de meeste Layout-properties van de Control klasse geïmplementeerd.

Het is de meest eenvoudige manier om een kleine hoeveelheid tekst op het scherm te tonen. Mits dat het element geen Content-property heeft, kan je de tekst tussen de begin- en eindtag van het element plaatsen of gebruikmaken van de **Text**-property.

Interessante Properties:

TextAlignment	De horizontale schikking van de tekst t.o.v. het TextBlock. Left : linkerkant Right : rechterkant Center : gecentreerd Justify : uitgelijnd tegen linkse en rechtse kant
TextDecorations	Extra opmaak van het lettertype Underline : een lijn onder de tekst OverLine : een lijn boven de tekst Strikethrough : een lijn door de tekst Baseline : een lijn ter hoogte van de basis van de tekst
TextWrapping	Wat gebeurt er als de tekst breder is dan de breedte van het TextBlock: NoWrap (default) : een gedeelte staat buiten beeld. Wrap : tekst op meerdere regels : als er niet gesplitst kan worden, dan komt het woord op 2 regels terecht WrapWithOverflow : tekst op meerdere regels : als er niet gesplitst kan worden, dan valt het woord deels buiten beeld.

Toegepast in het voorbeeld, wordt dit bv.

```
<TextBlock Background="Aqua"
           Foreground="Blue"
           FontSize="24"
           TextWrapping="Wrap"
           ToolTip="Voorbeeldtekst">
    Dit is een voorbeeld van het gebruik van WERKEN MET TEKST
</TextBlock>
```



Indien je binnen het *TextBlock* nog specifieke opmaak wil, dan kan je dat bekomen door met *Inline Elements* te werken.

Enkele mogelijkheden:

Bold : vetdruk tussen begin- en eindtag

- Italic :** schuindruk tussen begin- en eindtag
- Underline :** tekst wordt onderlijnd tussen de tags
- Span :** om plaatselijk de opmaak aan te passen (tussen begin- en eindtag)
- LineBreak :** een nieuwe regel beginnen

Een voorbeeld hiervan kan je zien als je de XAML verandert naar:

```
<TextBlock TextAlignment="Center" FontSize="24" TextWrapping="Wrap"
Background="Aqua" Foreground="Blue" ToolTip="Voorbeeldtekst">
    Dit is een <Italic>voorbeeld</Italic> van <LineBreak /> het
    <Bold>gebruik</Bold> van <Underline>WERKEN MET</Underline>
    <Span Background="Yellow" FontSize="42">TEKST</Span>
</TextBlock>
```

Met als resultaat:



Naar het voorbeeld toe, mag je de veranderingen tenietdoen.

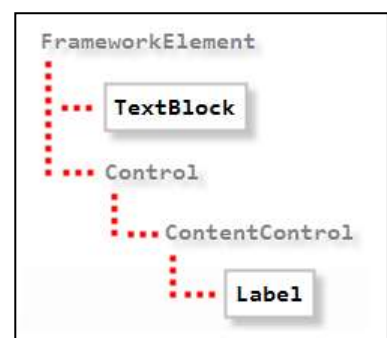
3.2 Label

Label is afgeleid van de *ContentControl*. Dit element biedt dus zeker veel meer opmaakmogelijkheden dan het *TextBlock*.

In het voorbeeld ga je dit uitwerken met een aanmeldingsscherm waarbij *Gebruikersnaam* en *Paswoord* de *Labels* zullen zijn. De *TextBoxen* worden ook al aangemaakt om het verband tussen beide te kunnen leggen.

Om de schikking netjes te houden, maak je gebruik van een *Grid*.

Je vult de XAML aan met volgende code:
(Binnen de *StackPanel*-tag, onder het *TextBlock*)




```
<Grid Margin="25">
  <Grid.RowDefinitions>
    <RowDefinition Height="40"></RowDefinition> (1)
    <RowDefinition Height="40"></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition></ColumnDefinition> (2)
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>
  <Label Grid.Row="0" Grid.Column="0" Margin="5" (3)
    Content="_Gebruikersnaam" ></Label>
  <Label Grid.Row="1" Grid.Column="0" Margin="5" (4)
    Content="_Paswoord" ></Label>
  <TextBox Grid.Row="0" Grid.Column="1" Margin="5" (5)
    Name="textBoxGebruikersnaam"></TextBox>
  <PasswordBox Grid.Row="1" Grid.Column="1" Margin="5" (6)
    Name="psdBox"></PasswordBox>
</Grid>
```

- (1) je definieert 2 rijen met een hoogte van 40 eenheden
- (2) je definieert 2 kolommen met dezelfde breedte over de breedte van de *Grid*
- (3) je maakt een *Label* in de eerste rij en eerste kolom met een marge van 5 eenheden en als inhoud (*Content*) “_Gebruikersnaam”.
- (4) je maakt een *Label* in de tweede rij en eerste kolom met een marge van 5 eenheden en als inhoud (*Content*) “_Paswoord”.
- (5) je maakt een *TextBox* in de eerste rij en tweede kolom met een marge van 5 eenheden, en je geeft deze de naam “textBoxGebruikersnaam”
- (6) je maakt een *TextBox* (*PasswordBox*) in de tweede rij en tweede kolom met een marge van 5 eenheden, en je geeft deze de naam “psdBox”



De *Content* van de *Labels* wordt begonnen met een *underscore* `_`. In de uitvoering van de applicatie wordt het karakter dat erna komt onderlijnd. (Als dit niet zichtbaar is, dan moet je even op de *Alt*-toets drukken). Dit heet de *Access Key*. Via een *Target*-attribuut kan je het *Label* koppelen aan bv. een *TextBox* waardoor je met de

toetscombinatie *Alt* + onderlijnd karakter de focus geeft aan het gekoppelde element.

Pas de *Labels* als volgt aan:

```
<Label Grid.Row="0" Grid.Column="0" Margin="5" Content="_Gebruikersnaam"
      Target="{Binding ElementName=textBoxGebruikersnaam}"></Label>
<Label Grid.Row="1" Grid.Column="0" Margin="5" Content="_Paswoord"
      Target="{Binding ElementName=psdBox}"></Label>
```

Het *Target*-attribuut bevat een *markup extention* (= de accolades { }) met een *Binding* naar een *Elementname*. In dit geval dus de naam van de bijhorende *TextBox*. (meer over *Bindings* later in de cursus).

Doordat een *Label* een *ContentControl* is, heeft het automatisch (geërfd) de mogelijkheid om een kader rond zich te hebben (*BorderBrush* en *BorderThickness*).

Breidt de *Labels* uit als volgt:

```
<Label Grid.Row="0" Grid.Column="0" Margin="5" Content="_Gebruikersnaam"
      Target="{Binding ElementName=textBoxGebruikersnaam}"
      BorderBrush="Black" BorderThickness="1"></Label>
<Label Grid.Row="1" Grid.Column="0" Margin="5" Content="_Paswoord"
      Target="{Binding ElementName=psdBox}"
      BorderBrush="Black" BorderThickness="1"></Label>
```



3.3 TextBox

Ondertussen heb je al gebruikgemaakt van een *TextBox*. Dit element is afgeleid van een *Control*. De ingegeven tekst komt in de *Text* property terecht.

Daar zijn echter weer een paar interessante properties:

AcceptsReturn	laat toe om een Return te tikken en als gevolg een meerlijnige <i>TextBox</i> te maken
AcceptsTab	laat toe om de <i>TAB</i> -toets te gebruiken
HorizontalContentAlignment VerticalContentAlignment	het uitlijnen van de tekst binnen de <i>TextBox</i> .
HorizontalScrollBarVisibility	instellingen scrollbars :

VerticalScrollBarVisibility	Disabled : niet Auto : als't nodig is Hidden : verborgen Visible : zichtbaar
MaxLines MinLines	maximum en minimum aantal lijnen in de <i>TextBox</i>
SelectionBrush SelectionOpacity	kleur en doorzichtigheid van een selectie
TextAlignment TextDecorations TextWrapping	zoals bij <i>TextBlock</i> .

3.4 PassWordBox

De *PassWordBox* kan je zien als een *TextBox* met gemaskeerde inhoud. Het is gebruikelijk dat de inhoud van de box getoond wordt als sterretjes of bolletjes. Het gebruikte karakter is instelbaar met de property *PassWordChar*.

Interessante properties:

MaxLength	bepaalt het maximum aantal karakters dat kan worden ingegeven
Password	is de property waar de inhoud van de box komt
PasswordChar	is de property waarin het karakter staat dat getoond wordt.
SecurePassword	is een read-only property en geeft de property <i>Password</i> als een <i>SecureString</i> weer. De <i>SecureString</i> is een gecodeerde versie van de <i>Password string</i> die enkel in het geheugen geschikt is.

Vb.

verander de *PassWordBox*-tag naar volgende waarden:

```
<PasswordBox Name="psdBox" Grid.Row="1" Grid.Column="1" Margin="5" MaxLength="8"
PasswordChar="?"></PasswordBox>
```

Resultaat bij het uitvoeren van het programma:



Het belangrijkste event i.v.m. de *PassWordBox* is het **PasswordChanged**-event. Dit event treedt op als de *Password*-property van inhoud verandert. Via de code-behind kan je eventuele testen of andere code uitvoeren.

4 BUTTONS

Alle soorten *Buttons* zijn gebaseerd op de klasse *ButtonBase*. Dit is een abstracte klasse die de basisfunctionaliteiten bevat. Deze klasse is een afgeleide klasse van de *ContentControl* klasse.

Voornaamste properties:

Content	bevat het gedeelte dat op de <i>Button</i> te zien is (tekst en/of image).
IsCancel	wordt de <i>Button</i> als Cancel-button gebruikt, en dus ook met de ESC-toets aanspreekbaar.
IsDefault	wordt de <i>Button</i> als default-button gebruikt, en dus ook met de ENTER-toets aanspreekbaar
ClickMode	de waarde bepaalt wanneer de <i>Click-EventHandler</i> wordt uitgevoerd: Release : is de defaultwaarde wordt uitgevoerd bij het loslaten van de muisknop Press : wordt uitgevoerd bij het indrukken van de muisknop Hover : wordt uitgevoerd als de muis over de <i>Button</i> beweegt

4.1 Button

De meeste bekende *Button* is de normale *Button*. Er wordt op geklikt en in de *Click-EventHandler* wordt gecodeerd wat van de *Button* verwacht wordt.

Om ons voorbeeld volledig te maken

Voeg onderaan in de XAML binnen de *StackPanel*-tag volgende tags toe:

```
<Button Content="Probeer maar aan te melden" Margin="0,10,0,10"
Click="Button_Click"></Button>
<TextBlock Name="textBlockAanmelding"></TextBlock>
```

Dubbelklik op de *Button* om in het code-behind venster toe komen waar de *Button_Click* routine automatisch wordt aangemaakt en vervolledig:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    textBlockAanmelding.TextWrapping = TextWrapping.Wrap;
    textBlockAanmelding.Text = "Je probeerde aan te melden met: " +
        textBoxGebruikersnaam.Text + " en paswoord: " + psdBox.Password;
}
```

Dit voorbeeld is nu afgewerkt, probeer maar uit!

4.2 ToggleButton

Om de specifieke *Buttons* uit te leggen, ga je met een nieuw voorbeeld van start:

Je maakt een nieuwe WPF Application. Zorg ervoor dat MainWindow is hernoemd naar ButtonGebruikWindow (zowel het bestand als de interne klasse).

Voeg volgende XAML toe:

```
<Window x:Class="ButtonGebruik.ButtonGebruikWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ButtonGebruik" Height="350" Width="525">
    <StackPanel>
        <Label BorderBrush="Black" BorderThickness="2" Name="LabelTekst"
              Height="50">Dit is de tekstblok die we gaan manipuleren</Label>
        <StackPanel Orientation="Horizontal">
            <ToggleButton Margin="10" Name="ButtonBold">Bold</ToggleButton>
            <ToggleButton Margin="10" Name="ButtonItalic">Italic</ToggleButton>
        </StackPanel>
    </StackPanel>
</Window>
```

Resultaat:



In de XAML-code is er gebruikgemaakt van de *ToggleButton*. Dit is een *ContentControl* die heel hard op een normale *Button* lijkt, maar met extra properties:

IsChecked	is de <i>Button</i> ingedrukt of niet Het type variabele hierbij is een Nullable Boolean m.a.w. false : uitgedrukt (default) true : ingedrukt null : geen waarde gespecificeerd; kan alleen via code op deze waarde gezet worden
IsThreeState	false : kan niet op <i>null</i> gezet worden true : kan wel programmatorisch op <i>null</i> gezet worden

Behalve het normale *Click*-event, heeft deze *Button*soort nog extra events:

Checked	Dit event treedt op als de IsChecked op true gezet wordt
Unchecked	Dit event treedt op als de IsChecked op false gezet wordt

Je kan dus op 2 manieren zorgen dat de nodige acties worden uitgevoerd, door deze twee events te gebruiken, of in de *Click-EventHandler* te testen op de waarde van *IsChecked*.

Ga op de *ButtonBold* staan, en dubbelklik in het *Properties*-venster bij *Events* op *Checked*. Je komt in het code-behind venster terecht bij de *Checked-EventHandler*. Voeg volgende code toe:

```
private void ButtonBold_Checked(object sender, RoutedEventArgs e)
{
    LabelTekst.FontWeight = FontWeights.Bold;
}
```

Doe een gelijkaardige actie om in de *Unchecked-EventHandler* te komen, en voeg daar volgende code toe:

```
private void ButtonBold_Unchecked(object sender, RoutedEventArgs e)
{
    LabelTekst.FontWeight = FontWeights.Normal;
}
```

FontWeights is een klasse die mogelijke waarden in verband met de dikte van het lettertype specificeert. Het is dus niet dat de *vetdruk* af- of aangezet wordt, maar je bepaalt de dikte van het lettertype.

Ga nu op de *ButtonItalic* staan, en dubbelklik bij de *Events* op het *Click*-event. Voeg volgende code toe:

```
private void ButtonItalic_Click(object sender, RoutedEventArgs e)
{
    if (ButtonItalic.IsChecked == true)
        LabelTekst.FontStyle = FontStyles.Italic;
    else
        LabelTekst.FontStyle = FontStyles.Normal;
}
```

Je moet nu zelf testen of de *Button* in- of uitgedrukt wordt om de stijl juist te zetten. **FontStyles** is een klasse die mogelijke waarden van schuindruk voor een lettertype bevat (zo is er ook nog *Oblique*).

4.3 RepeatButton

De *RepeatButton* is bedoeld om bij het blijvend indrukken van de *Button* een herhaling van de code-behind te krijgen.

Extra properties die hiervoor dienen:

Delay	Hiermee bepaal je het aantal millisecondes dat er gewacht wordt alvorens de herhaling in gang gezet wordt.
--------------	--

Interval	Hiermee bepaal je het aantal millisecondes dat tussen 2 herhalingen valt.
-----------------	---

Je voegt volgende tags aan de XAML toe juist na de *ToggleButton*:

```
<TextBlock Margin="10">Lettertype vergroten of verkleinen</TextBlock>
<RepeatButton Margin="10" Width="30" Name="RepeatButtonGroter" Interval="100"
    Delay="1000">+</RepeatButton>
<RepeatButton Margin="10" Width="30" Name="RepeatButtonKleiner" Interval="100"
    Delay="1000">-</RepeatButton>
```

Resultaat:



Het is de bedoeling dat je met de + *Button* de grootte van het lettertype in het *Label* met één verhoogt tot een maximum van 25, en dat je met de – *Button* de grootte verkleint tot minimaal 1. Pas na een seconde mag de herhaling in gang schieten (→ *Delay*="1000") en het mag zich herhalen om de 10^{de} seconde (→ *Interval*="100").

Zorg ervoor dat je in de *Click-EventHandler* terechtkomt van de *RepeatButtonGroter*, en voeg volgende code toe:

```
private void RepeatButtonGroter_Click(object sender, RoutedEventArgs e)
{
    if (LabelTekst.FontSize<25)
        LabelTekst.FontSize++;
}
```

Doe dit ook voor de *RepeaterButtonKleiner*.

```
private void RepeatButtonKleiner_Click(object sender, RoutedEventArgs e)
{
    if (LabelTekst.FontSize>1)
        LabelTekst.FontSize--;
}
```

Probeer het voorbeeld maar uit!

4.4 RadioButton

De *RadioButton* is een control die een afgeleide klasse is van de *ToggleButton* klasse, en heeft bij gevolg ook alle properties ervan.

Een beperking naar het gebruik van de *RadioButton* toe is dat je een keuze niet kan verwijderen met een muisklik. Dit kan alleen in code (programmatorisch).

Er is nog één specifieke property aan toegevoegd:

GroupName	Hierdoor is het mogelijk om meerdere <i>RadioButtons</i> aan elkaar te koppelen zodat maar één van hen geselecteerd kan worden.
------------------	---

Voeg onderaan de XAML, maar binnen de onderste *StackPanel* volgende tags toe:

```
<Label Margin="0 30 0 10" BorderBrush="Blue" BorderThickness="1">
    Tekstkleur</Label>
<StackPanel>
    <RadioButton Content="Blue" GroupName="kleur" Margin="5 0"></RadioButton>
    <RadioButton Content="Green" GroupName="kleur" Margin="5 0"></RadioButton>
    <RadioButton Content="Red" GroupName="kleur" Margin="5 0"></RadioButton>
    <RadioButton Content="Black" GroupName="kleur" Margin="5 0"></RadioButton>
</StackPanel>
```

Resultaat:



De bedoeling is om bij het kiezen van een kleur de tekst in de *Label* te wijzigen.

Mits dat alle *RadioButtons* dezelfde functie hebben, is het mogelijk om dezelfde routine voor deze opdracht te gebruiken.

Voeg in de XAML van de *RadioButtons* volgend attribuut toe:

```
Checked="Kleur_Checked"
```

In het code-behind venster moet er nu een procedure toegevoegd worden die overeenkomt met de naam van deze routine:

```
private void Kleur_Checked(object sender, RoutedEventArgs e)
{
    RadioButton knop = (RadioButton)sender;
    LabelTekst.Foreground = (SolidColorBrush)new BrushConverter().
                                                                    (1)
```



```
ConvertFromString(knop.Content.ToString());  
}
```

- (1) Eerst wordt de *sender* terug omgevormd naar de desbetreffende *RadioButton*
- (2) Daar dat de Content van de *RadioButton* een bestaande benaming is van een kleur, kan deze gebruikt worden om deze om te vormen *BrushConverter().ConvertFromString(knop.Content.ToString())* en dan als kleur voor de tekst te gebruiken.

Probeer maar uit!

4.5 CheckBox

De *CheckBox* is een control die ook een afgeleide klasse is van de *ToggleButton* klasse, en heeft bij gevolg ook alle properties ervan.

Deze control kan wel al klikkend gezet en gecleared worden.

Als laatste onderdeel in het voorbeeld wordt er onderaan een alternatieve methode toegevoegd om de tekst in de *Label* eventueel *Bold* en/of *Italic* te zetten of niet.

Voeg onderaan de XAML, maar binnen de onderste *StackPanel* volgende tags toe:

```
<Label Margin="0 30 0 10" BorderBrush="Blue" BorderThickness="1">Bold en  
Italic</Label>  
<StackPanel Orientation="Horizontal">  
    <CheckBox Name="CheckBoxBold" Content="Bold" Margin="10 0"></CheckBox>  
    <CheckBox Name="CheckBoxItalic" Content="Italic" Margin="10 0"></CheckBox>  
</StackPanel>
```

Resultaat:



Bij beide *CheckBoxen* ga je via de *Click-EventHandler* ervoor zorgen dat *Bold* en/of *Italic* wordt aan- of afgezet. Mits dat er al *ToggleButtons* bestaan die deze functionaliteit ook aanbieden, moet ervoor gezorgd worden dat deze ook de juiste status aangeven.

Voeg een *Click-EventHandler* toe aan de *CheckBoxBold*.

```
<CheckBox Name="CheckBoxBold" Content="Bold" Margin="10 0"  
    Click="CheckBoxBold_Click"></CheckBox>
```

En voeg in het code-behind de volgende code:

```
private void CheckBoxBold_Click(object sender, RoutedEventArgs e)
{
    ButtonBold.IsChecked = CheckBoxBold.IsChecked;
}
```

 (1)

- (1) Door de *Button.IsChecked* gelijk te stellen aan de *CheckBox.IsChecked* wordt de *Button* automatisch juist gezet.

Het omgekeerde moet ook gebeuren: als er op de *ButtonBold* geklikt wordt, moet de *CheckBoxBold* ook reageren.

Pas hiervoor de volgende code aan:

```
private void ButtonBold_Checked(object sender, RoutedEventArgs e)
{
    LabelTekst.FontWeight = FontWeights.Bold;
    CheckBoxBold.IsChecked = ButtonBold.IsChecked;
}

private void ButtonBold_Unchecked(object sender, RoutedEventArgs e)
{
    LabelTekst.FontWeight = FontWeights.Normal;
    CheckBoxBold.IsChecked = ButtonBold.IsChecked;
}
```

Om de *Italic* functionaliteit in te voeren, moet er anders gewerkt worden, omdat de *ButtonItalic* via het *Click*-event werd gemanipuleerd.

Voeg een *Click-EventHandler* van de *ButtonItalic* toe aan de *CheckBoxItalic*.

```
<CheckBox Name="CheckBoxItalic" Content="Italic" Margin="10 0"
    Click="ButtonItalic_Click"></CheckBox>
```

Omdat ze beide moeten reageren op de actie moeten er in de code nog aanpassingen gedaan worden:

```
using System.Windows.Controls.Primitives; (1)

private void ButtonItalic_Click(object sender, RoutedEventArgs e)
{
    ToggleButton knop = (ToggleButton)sender; (2)

    if (knop.IsChecked == true)
        LabelTekst.FontStyle = FontStyles.Italic;
    else
        LabelTekst.FontStyle = FontStyles.Normal;
    CheckBoxItalic.IsChecked = knop.IsChecked; (3)
    ButtonItalic.IsChecked = knop.IsChecked;
}
```

- (1) Om te kunnen casten naar een *ToggleButton* moet je eerst bovenaan een *using*-regel bijvoegen: `using System.Windows.Controls.Primitives;`

- (2) Door de *sender* te casten naar zijn klasse of moederklasse (want *CheckBox* is een afgeleide klasse van *ToggleButton*), kan je aan de *IsChecked* property.
- (3) Mits dat je niet weet welke van beide geklikt is, worden beide *IsChecked* properties correct gezet.

4.6 Images in Buttons

Mits dat alle *Buttons* van de klasse *ContentControl* zijn, hebben zij maar de mogelijkheid om één *Content* te hebben.

Nu kan de *Content* op zich niet alleen tekst bevatten, maar het kan om het even wat zijn zoals bv. een *Image*.

Maak in het project een map met de naam *Images*, en kopieer vanuit de aangeleverde bestanden "bold.png" naar deze map.

Verwijder de inhoud "Bold" van de *ToggleButton*, en vervang dit door een *Image*-tag met als *Source*-attribuut de juist gekopieerde *Image*:

```
<ToggleButton Margin="10" Name="ButtonBold" Checked="ButtonBold_Checked"
Unchecked="ButtonBold_Unchecked">
  <Image Source="Images/bold.png"></Image>
</ToggleButton>
```

Om ze nu beide (tekst en image) als *Button* te laten zien, moet er eerst een *StackPanel*-tag toegevoegd worden (want de *Content* mag maar één tag zijn).

Pas de XAML aan naar volgende code:

```
<ToggleButton Margin="10" Name="ButtonBold" Checked="ButtonBold_Checked"
Unchecked="ButtonBold_Unchecked">
  <StackPanel>
    <Image Source="Images/bold.png"></Image>
    <Label>Bold</Label>
  </StackPanel>
</ToggleButton>
```

Resultaat:



De combinatie-mogelijkheden zijn enorm.

4.7 Relatieve en Absolute URI's

Een URI (uniform resource identifier) mag je vrij vertalen als een string die een bronbestand identificeert en laadt. Eigenlijk heb je al een URI gebruikt in de XAML-code toen je de *Source* van de *Image* specificeerde:

```
<Image Source="Images/bold.png">.
```

Omdat er begonnen wordt met de naam van de map (Images) is dit een *Relatieve URI*, en wordt het totale padverwijzing gevormd vanaf de map waarin het project staat.

Nu is het niet altijd zeker dat dit na publicatie nog van toepassing is, waardoor je beter een padverwijzing kan gebruiken dat *absoluut* is en er voor zorgt dat het bestand bij het compileren in de assembly wordt opgenomen = pack. Dit *pack* wordt op volgende manier schematisch gevormd:

```
pack://authoriteit/pad
```

Voor een WPF-Window applicatie is deze autoriteit = application:,,,
(voor een Page-applicatie = siteoforigin:,,,))

Indien het bestand in hetzelfde project staat, wordt het pad gevormd door een slash gevolgd door eventuele submappen en de bestandsnaam.

In het voorbeeld zou dit geven :

```
<Image Source="pack://application:,,,/Images/bold.png">
```

Om een bestand te definiëren dat in een referenced assembly aanwezig is (dus in een ander project die een reference heeft naar het huidige project) is de syntax de volgende :

```
pack://authoriteit/referencedAssembly;component/pad  
bv. ="pack://application:,,,/CommonLibrary;component/Images/bold.png"
```

Je kan dit uitproberen bij de Image van de button hierboven (die natuurlijk in hetzelfde project aanwezig is).

Als je de *Image* in code wil koppelen, dan ga je als volgt te werk:

```
Image t = new Image();  
Uri bron = new Uri("pack://application:,,,/Images/bold.png", UriKind.Absolute);  
t.Source = new BitmapImage(bron);
```



Opdracht: Pizza bestellen

5 LISTBOX – COMBOBOX

Indien je met lijsten te maken krijgt, is het handig om deze al dan niet helemaal te tonen op het scherm. De *ComboBox* is een uitklapbare lijst die alleen de gemaakte keuze toont, terwijl de *ListBox* een uitgeklapte lijst is, die aangeeft welke keuze gemaakt is.

Interessante properties:

DisplayMemberPath	Als de inhoud van een <i>Item</i> meerdere elementen bevat, dan wordt hier gespecificeerd welk element gevisualiseerd wordt.
Items	De collectie van de inhoud van de <i>Box</i>
SelectedIndex	Het indexnummer van het selecteerde <i>Item</i>
SelectedItem	Het geselecteerde <i>Item</i>

Interessante events:

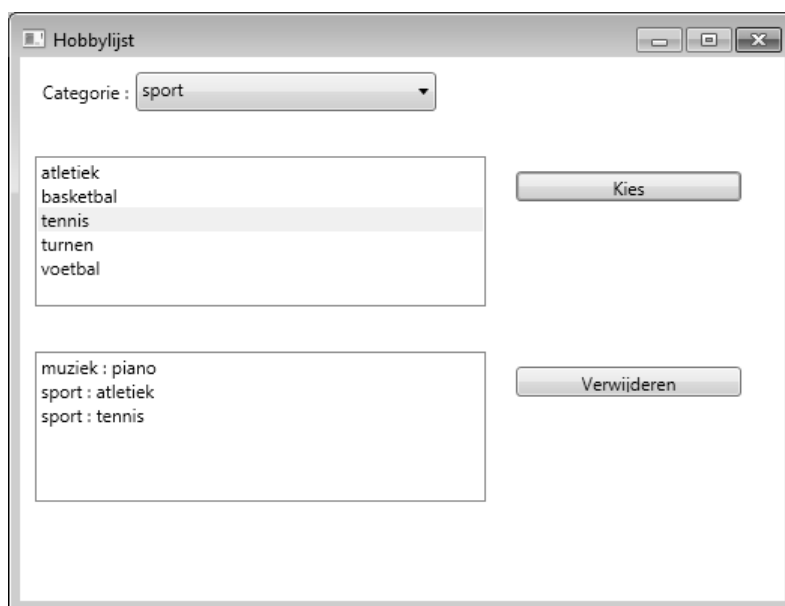
SelectionChanged	Als de gebruiker een keuze maakt, wordt dit event geactiveerd.
-------------------------	--

5.1 Simpele inhoud(tekst) in de Boxen

Om de soorten *ListBox*en en *ComboBox*en uit te leggen, ga je met een nieuw voorbeeld van start:

Je maakt een nieuwe WPF Application met de naam *HobbyLijst*. Zorg ervoor dat *MainWindow* is hernoemd naar *HobbyLijstWindow* (zowel het bestand als de interne klasse).

Het eindresultaat zou gelijkaardig aan het onderstaande moeten zijn.



Alvorens te beginnen met de inhoud van de applicatie, moet er eerst een klasse worden gemaakt van de categorie, activiteit en een symbolische weergave (die later pas gebruikt wordt).

Klik op de projectnaam met de rechtermuisknop en kies Add – Class.

Vul deze klasse zoals hieronder:

```
using System.Windows.Media.Imaging; (1)

namespace HobbyLijst
{
    public class Hobby
    {
        public string Categorie { get; set; }
        public string Activiteit { get; set; }
        public BitmapImage Symbool { get; set; }

        public Hobby(string nCategorie, string nActiviteit, (2)
            BitmapImage nSymbool)
        {
            Categorie = nCategorie;
            Activiteit = nActiviteit;
            Symbool = nSymbool;
        }
    }
}
```

- (1) De `using System.Windows.Media.Imaging;` laat toe om de klasse *BitmapImage* aan te spreken
- (2) Er wordt een geparameteriseerde constructor gedefinieerd zodat een object van deze klasse alleen kan worden gemaakt als alle onderdelen zijn ingevuld.

In de XAML wordt nu de layout en de controls gedefinieerd zodat je zicht krijgt op wat waar terecht gaat komen.

Vergeet niet alle controls een *Name* te geven die je later nodig hebt in het code-behind venster.

```
<Window x:Class="HobbyLijst.HobbyLijstWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hobbylijst" Height="400" Width="525" Loaded="Window_Loaded">
    <StackPanel>
        <StackPanel Margin="10" Orientation="Horizontal">
            <Label>Categorie : </Label>
            <ComboBox Name="ComboBoxCategorie" Width="200"
                HorizontalAlignment="Left">
            </ComboBox>
        </StackPanel>
        <StackPanel Orientation="Horizontal" Margin="0 20">
            <ListBox Name="ListBoxHobbies" Height="100" Width="300"
                Margin="10 0"></ListBox>
            <Button Name="ButtonKies" Width="150" Height="20"
                VerticalAlignment="Top" Margin="10" Content="Kies"></Button>
        </StackPanel>
        <StackPanel Orientation="Horizontal" Margin="0 10">
            <ListBox Name="ListBoxGekozen" Height="100" Width="300">
```

```

        Margin="10 0"></ListBox>
        <Button Name="ButtonVerwijderen" Height="20" Width="150"
            VerticalAlignment="Top" Margin="10" Content="Verwijderen"></Button>
    </StackPanel>
</StackPanel>
</Window>

```

Via het code-behind venster ga je in de *Window-Loaded EventHandler* de *ComboBox* en de *ListBox* opvullen:

```

public List<Hobby> hobbies = new List<Hobby>(); (1)
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    hobbies.Add(new Hobby("sport", "voetbal", (2)
        new BitmapImage(new Uri(@"images\voetbal.jpg", UriKind.Relative))));
    hobbies.Add(new Hobby("sport", "atletiek",
        new BitmapImage(new Uri(@"images\atletiek.jpg", UriKind.Relative))));
    hobbies.Add(new Hobby("sport", "basketbal",
        new BitmapImage(new Uri(@"images\basketbal.jpg", UriKind.Relative))));
    hobbies.Add(new Hobby("sport", "tennis",
        new BitmapImage(new Uri(@"images\tennis.jpg", UriKind.Relative))));
    hobbies.Add(new Hobby("sport", "turnen",
        new BitmapImage(new Uri(@"images\turnen.jpg", UriKind.Relative))));
    hobbies.Add(new Hobby("muziek", "trompet",
        new BitmapImage(new Uri(@"images\trompet.jpg", UriKind.Relative))));
    hobbies.Add(new Hobby("muziek", "drum",
        new BitmapImage(new Uri(@"images\drum.jpg", UriKind.Relative))));
    hobbies.Add(new Hobby("muziek", "gitaar",
        new BitmapImage(new Uri(@"images\gitaar.jpg", UriKind.Relative))));
    hobbies.Add(new Hobby("muziek", "piano",
        new BitmapImage(new Uri(@"images\piano.jpg", UriKind.Relative))));

    ComboBoxCategorie.Items.Add("- alle categorieën -"); (3)
    ComboBoxCategorie.Items.Add("muziek");
    ComboBoxCategorie.Items.Add("sport");
    ComboBoxCategorie.SelectedIndex = 0; (4)
}

```

- (1) Je maakt een globale variabele van een *List Of Hobby* zodat deze overall aangesproken kan worden.
- (2) Je vult deze lijst met objecten (in de bijgevoegde bestanden vind je alle .jpg bestanden die de activiteiten voorstellen. Maak een aparte map *images* in het project en kopieer de bestanden hier naar toe via de Solution Explorer: Add – Existing Item).
- (3) Je vult manueel alle categorieën in in de *ComboBox*. De eerste Categorie is alles wat in *Hobbies* staat.
- (4) Je zorgt ervoor dat het eerste element van de *ComboBox* geselecteerd staat. Hierdoor wordt automatisch de *SelectionChanged-EventHandler* van de *ListBox* uitgevoerd waardoor de *ListBox* wordt opgevuld.

Omdat je objecten in de *ListBox* laadt, moet je bij de properties van de *ListBox* nog specificeren welk onderdeel getoond gaat worden ***DisplayMemberPath = Activiteit***.

Het volgende is om alle functionaliteit te laten werken.

Eerst de *ComboBox* : als je een keuze maakt van *Categorie*, dan moet in de *ListBox* van de *Activiteiten* alleen nog de *Activiteiten* van deze *Categorie* zichtbaar zijn. Of bij de eerste keuze moet alles te zien zijn.

Hiervoor moet in het code-behind venster volgende code geïmplementeerd worden (vergeet in de XAML-code dit event ook niet te implementeren):

```
private void ComboBoxCategorie_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    ListBoxHobbies.Items.Clear();                                (1)
    foreach (Hobby hob in hobbies)
    {
        if (hob.Categorie == ComboBoxCategorie.SelectedItem.ToString() (2)
            || ComboBoxCategorie.SelectedIndex == 0)
            ListBoxHobbies.Items.Add(hob);
    }
    ListBoxHobbies.Items.SortDescriptions.Add(                  (3)
        new SortDescription("Activiteit", ListSortDirection.Ascending));
}
```

- (1) Je maakt de huidige lijst leeg
- (2) Je doorloopt alle elementen van *hobbies* en test of ze
 - voldoen aan de juiste categorie
 - als de eerste optie van de *ComboBox* gekozen is, voldoen ze altijd
- (3) Je sorteert de *ListBox* op "Activiteit". Het zal waarschijnlijk nodig zijn om `using System.ComponentModel;` bovenaan toe te voegen

Bij het klikken op de *KiesButton* moet de gekozen *Activiteit* in de andere *ListBox* verschijnen.

Hiervoor moet in het code-behind venster volgende code geïmplementeerd worden:

```
private void ButtonKies_Click(object sender, RoutedEventArgs e)
{
    if (ListBoxHobbies.SelectedItem != null)                    (1)
    {
        Hobby gekozenHobby = (Hobby)ListBoxHobbies.SelectedItem; (2)
        ListBoxGekozen.Items.Add(gekozenHobby.Categorie + " : " + (3)
            gekozenHobby.Activiteit);
    }
}
```

- (1) Er moet er getest worden of er wel een *Activiteit* aangeduid is
- (2) Mits dat er in de *ListBox* de *Items* als objecten worden aanzien, moet het gekozen *Item* gecast worden naar een *Hobbie*.
- (3) Je voegt in de tweede *ListBox* een *Item* toe met de info van de *Hobbie*.

Als er in de tweede *ListBox* een *Activiteit* wordt aangeduid, dan kan je met de *VerWijderButton* deze keuze terug verwijderen.

Hiervoor moet in het code-behind venster volgende code

geïmplementeerd worden:

```
private void ButtonVerwijderen_Click(object sender, RoutedEventArgs e)
{
    if (ListBoxGekozen.SelectedIndex >= 0)
        ListBoxGekozen.Items.RemoveAt(ListBoxGekozen.SelectedIndex);
}
```

Probeer maar uit of alles geslaagd is!

5.2 Complexe inhoud in de Boxen

Om de inhoud van de *ListBoxen* toch een aangenaamer uiterlijk te geven, ga je het *Symbol* dat in de *Hobby* klasse zit mee gebruiken om de selectie te maken.

Onderstaand voorbeeld gaat het eindresultaat zijn van een paar aanpassingen hiervoor.



De eerste *ListBox* die je gaat aanpassen is diegene waar je de keuze gaat maken. De hobby's zullen omkaderd en onder elkaar worden weergegeven met zowel hun *Symbol* als *Activiteit*.

Daarvoor ga je als volgt te werk:

Verwijder eerst de property *DisplayMemberPath = Activiteit*, want nu ga je meer weergeven dan alleen de *Activiteit*.

De *XAML* voor deze *ListBox* wordt gewijzigd in:

```
<ListBox Name="ListBoxHobbies" Height="100" Width="300" Margin="10 0">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Border BorderBrush="Black" BorderThickness="1" Width="275">
                <StackPanel Orientation="Horizontal">
                    <Image Source="{Binding Path=Symbol}" Stretch="Fill"
                        Height="40" Width="40"></Image>
                    <TextBlock VerticalAlignment="Center"
                        Text="{Binding Path=Activiteit}"></TextBlock>
                </StackPanel>
            </Border>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

```
</Border>
</DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
```

- (1) Je maakt een *Template* (= een voorbeeld in de vorm van een beschrijving) voor de *Items*.
- (2) Deze *Template* dient om de *Data* (= de inhoud van de Hobby) te layouten
- (3) De *Hobby* ga je omkaderen, dus een *Border* wordt gedefinieerd
- (4) Omdat de *Activiteit* naast het *Symbool* moet worden weergegeven, ga je een *StackPanel* definiëren met *Orientation="Horizontal"*.
- (5) Eerst wordt het *Symbool* getoond (= een *Image*) met een hoogte en breedte van 40 en die ruimte wordt opgevuld met het *Symbool*.
De *Source*-property zegt welke *Image* getoond wordt, dus via de *Binding Path* wordt de naam van het element *Symbool* meegegeven
- (6) Daarnaast komt de *Activiteit* die op gelijkaardige wijze wordt toegevoegd: Via een *TextBlock*, verticaal gecentreerd wordt in *Text* de verbinding gemaakt via de *Binding Path* naar het element *Activiteit*.

Dit kan je al uitproberen!

De tweede *ListBox* toont de gekozen hobby's naast elkaar met *Symbool* en *Activiteit*. Hiervoor moet in de XAML code worden toegevoegd.

```
<ListBox Name="ListBoxGekozen" Height="100" Width="300" Margin="10 0">
  <ListBox.ItemsPanel> (1)
    <ItemsPanelTemplate> (2)
      <StackPanel Orientation="Horizontal"></StackPanel> (3)
    </ItemsPanelTemplate>
  </ListBox.ItemsPanel>
  <ListBox.ItemTemplate> (4)
    <DataTemplate>
      <Border BorderBrush="Black" BorderThickness="1" Width="60"> (5)
        <StackPanel> (6)
          <Image Source="{Binding Path=Symbool}" Stretch="Fill" Height="40"
            Width="40"></Image>
          <TextBlock VerticalAlignment="Center" HorizontalAlignment="Center" (7)
            Text="{Binding Path=Activiteit}"></TextBlock>
        </StackPanel>
      </Border>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

- (1) De *Items* worden naast elkaar geplaatst, dus hiervoor heb je een *Panel* nodig die dat forceert.
- (2) De *PanelTemplate* geeft aan dat deze structuur van toepassing is op deze *ListBox*
- (3) Je gebruikt een *StackPanel* met *Orientation="Horizontal"* om de *Items* naast elkaar te krijgen
- (4) Voor de inhoud van de *Hobby* wordt terug een *Template* gebruikt

- (5) De *Border* wordt met een breedte van 60 gedefinieerd om de data een maximum breedte te geven
- (6) Omdat er twee elementen getoond worden, moet dit binnen een *StackPanel* gebeuren.
- (7) Om de *Activiteit* in het midden van het *Symbol* te krijgen, wordt ook een *HorizontalAlignment="Center"* ingesteld.

Omdat in de eerste versie van de applicatie in de gekozen *ListBox* de *Categorie* en de *Activiteit* getoond werden, moet nu ook de code-behind van de *KiesButton* gewijzigd worden.

```
private void ButtonKies_Click(object sender, RoutedEventArgs e)
{
    if (ListBoxHobbies.SelectedItem != null)
    {
        Hobby gekozenHobby = (Hobby)ListBoxHobbies.SelectedItem;
        //ListBoxGekozen.Items.Add(gekozenHobby.Categorie + " : " +
        //gekozenHobby.Activiteit);
        ListBoxGekozen.Items.Add(gekozenHobby);
        ListBoxGekozen.Items.SortDescriptions.Add(
            new SortDescription("Categorie", ListSortDirection.Ascending));
        ListBoxGekozen.Items.SortDescriptions.Add(
            new SortDescription("Activiteit", ListSortDirection.Ascending));
    }
}
```

- (1) In plaats van een *String* samen te stellen, wordt nu de volledige *Hobby* als *Item* toegevoegd, die in de XAML door de *Binding Path=* element per element wordt ingevuld.
- (2) Het resultaat wordt gesorteerd weergegeven (dit is in het vooruitzicht van volgende hoofdstuk).

Dit kan je uitproberen!

6 MESSAGEBOX

Tot hiertoe heb je de gebruiker nog geen vragen gesteld, maar soms is een bevestiging van de actie die je gaat ondernemen toch een geruststelling voor de gebruiker die op dat moment nog kan afzien van zijn actie.















Of gewoon een melding van wat er gebeurd is, extra info,... is handig om aan de gebruiker te laten weten.

MessageBox kan op deze 2 manieren gebruikt worden: je kan er een vraag mee stellen of gewoon een melding geven.

De manier om een *MessageBox* te tonen, is met de *Show*-method. Al naar gelang het aantal meegegeven parameters (overloading), maak je de *MessageBox* specifieker naar de gebruiker toe.

Dit zijn de meeste combinaties:

<p><code>MessageBox.Show("melding")</code> 1^{ste} parameter: geeft inhoud van de <i>MessageBox</i> Kan gesloten worden met de OK-knop of het Close-knopje</p>	
<p><code>MessageBox.Show("melding", "titel")</code> 2^{de} parameter: geeft titel aan de <i>MessageBox</i></p>	
<p><code>MessageBox.Show("melding", "titel", MessageBoxButton.YesNo)</code> 3^{de} parameter: geeft aan welke knoppen aanwezig zijn: OK, OKCancel, YesNo, YesNoCancel Het Close-knopje heeft dezelfde waarde als de ESC-toets of Cancel.</p>	

<p><code>MessageBox.Show("melding", "titel", MessageBoxButton.YesNo, MessageBoxIcon.Information)</code></p> <p>4^{de} parameter: geeft het symbool aan dat voor de melding komt:</p> <table border="1"> <tr> <td></td> <td>None</td> </tr> <tr> <td></td> <td>Hand of Stop of Error</td> </tr> <tr> <td></td> <td>Question</td> </tr> <tr> <td></td> <td>Exclamation of Warning</td> </tr> <tr> <td></td> <td>Asterisk of Information</td> </tr> </table>		None		Hand of Stop of Error		Question		Exclamation of Warning		Asterisk of Information	
	None										
	Hand of Stop of Error										
	Question										
	Exclamation of Warning										
	Asterisk of Information										
<p><code>MessageBox.Show("melding", "titel", MessageBoxButton.YesNo, MessageBoxIcon.Information, MessageBoxResult.No)</code></p> <p>5^{de} parameter: geeft aan welke knop als "Default"-knop (= ENTER) gebruikt wordt als die aanwezig is. OK, Cancel, Yes, No</p>											

Tot zover het tonen van de *MessageBox*. Om nu te weten wat de gebruiker geantwoord heeft, moet je de *MessageBox* in een vraagstelling steken zoals bv.

```
if (MessageBox.Show("melding", "titel", MessageBoxButton.YesNo,  
MessageBoxImage.Information, MessageBoxResult.No) == MessageBoxResult.Yes)
```

dan kan je al naargelang de test de code gaan uitsplitsen.

Je gaat in het *HobbyLijst* voorbeeld nog een onschuldige knop toevoegen om dit te demonstreren.

Voeg in de XAML-code aan de onderkant een *Button* toe:

```
</StackPanel>  
<Button Name="ButtonSamenvatting" Margin="10">Samenvatting</Button>  
</StackPanel>  
</Window>
```

In het code-behind venster voeg de functionaliteit van deze *Button* toe:

```
private void ButtonSamenvatting_Click(object sender, RoutedEventArgs e)
{
    if (MessageBox.Show("Wil je de gekozen hobby's op een rijtje?",  
"Samenvatting", MessageBoxButton.YesNo,  
MessageBoxImage.Question, MessageBoxResult.No) == MessageBoxResult.Yes) (1)
    {
        string mijnTekst = "Mijn hobby's zijn: "; (2)
        string cat = string.Empty; (3)

        foreach (Object item in ListBoxGekozen.Items) (4)
        {
```

```

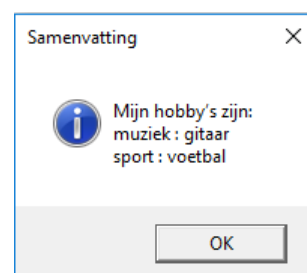
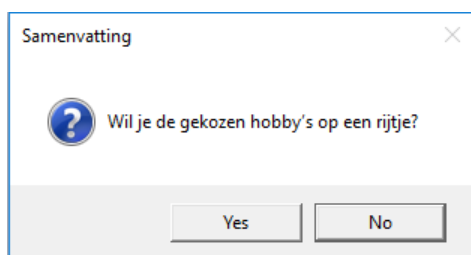
        Hobby mijnHobby = (Hobby)item;
        if (cat != mijnHobby.Categorie)
        {
            cat = mijnHobby.Categorie;
            mijnTekst += "\n" + mijnHobby.Categorie + " : " +
                mijnHobby.Activiteit;
        }
        else
        {
            mijnTekst += ", " + mijnHobby.Activiteit;
        }
    }
    if (ListBoxGekozen.Items.Count == 0)
    {
        MessageBox.Show("Ik heb geen hobby's", "Samenvatting",
            MessageBoxButton.OK, MessageBoxImage.Information);
    }
    else
    {
        MessageBox.Show(mijnTekst, "Samenvatting", MessageBoxButton.OK,
            MessageBoxImage.Information);
    }
}

```

- (1) De vraag wordt aan de gebruiker gesteld: "Wil je de gekozen hobby's op een rijtje?". Als deze **Yes** antwoordt, dan wordt de samenvatting samengesteld.
- (2) De initiële "tekst"-string wordt gedefinieerd
- (3) De "categorie"-string wordt gedefinieerd
- (4) De collection van de gekozen *Activiteiten* wordt doorlopen
- (5) Als de *Categorie* verandert, dan wordt er een nieuwe regel begonnen. Dit wordt gedaan met de karaktercombinatie : "\n".
- (6) Als er geen gekozen *Hobby's* zijn, dan wordt de "tekst"-string genegeerd en een alternatieve tekst getoond in een *MessageBox*, anders wordt de samengevoegde "tekst"-string getoond.

Dit kan je uitproberen!

Mogelijk resultaat:



Opdracht: Telefoon

7 MENU – TOOLBAR – STATUSBAR

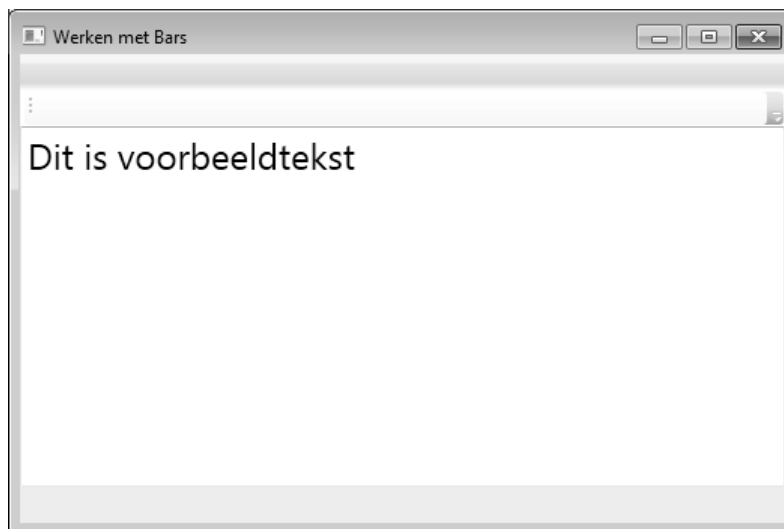
De meeste applicaties gebruiken bovenaan een *Menu* en eventueel een *ToolBar* om de gebruiker toe te laten om de functionaliteit van de applicatie op verschillende manieren toe te passen.

Je maakt een nieuwe WPF Application met de naam Bars. Zorg ervoor dat MainWindow is hernoemd naar BarWindow (zowel het bestand als de interne klasse).

Om later in het hoofdstuk niet te veel last te hebben van de positionering van de *Bars* ga je eerst een algemene structuur in de XAML-code aanbrengen:

```
<Window x:Class="Bars.BarWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Werken met Bars" Height="350" Width="525">
    <DockPanel>
        <Menu Height="24" Name="MenuBalk" DockPanel.Dock="Top"></Menu>
        <ToolBarTray Height="24" DockPanel.Dock="Top"></ToolBarTray>
        <StatusBar DockPanel.Dock="Bottom" Height="24"></StatusBar>
        <TextBox Name="TextBoxVoorbeeld" Text="Dit is voorbeeldtekst"
                VerticalAlignment="Stretch" HorizontalAlignment="Stretch"
                FontSize="24" TextWrapping="Wrap" MaxWidth="600" MaxHeight="900">
    </TextBox>
    </DockPanel>
</Window>
```

Dit geeft volgend resultaat:



Door het gebruik van een *DockPanel* kan je de verschillende *Bars* aan een bepaalde zijde laten plakken. Je mag zomaar de volgorde van de *Bars* niet veranderen, want dit gaat een andere layout te weeg brengen. De *TextBox* moet zeker als laatste gedefinieerd worden omdat die de rest van de ruimte inbeslagneemt.

7.1 Menu

7.1.1 Werkwijze

De eerste *Bar* die je gaat opvullen is de *Menu*. Je gaat ervoor zorgen dat het lettertype van de tekst in de *TextBox* veranderd kan worden in *Courier New* of *Arial*.

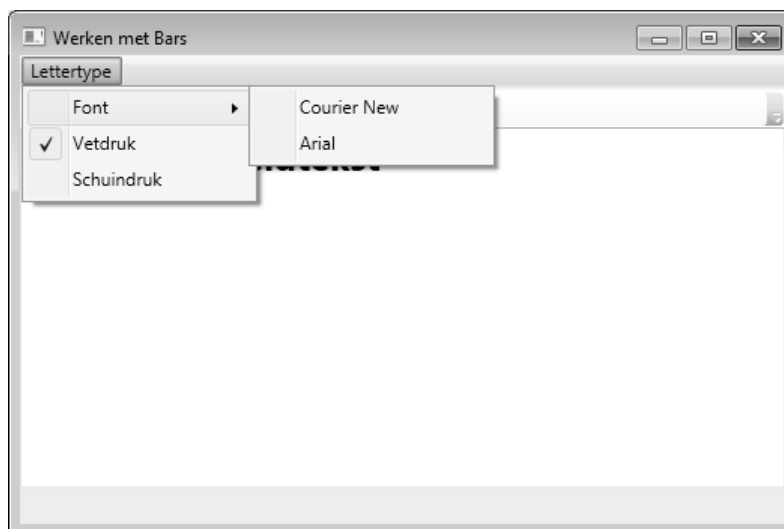
Tussen de begin- en eindtag van *Menu* voeg je in de XAML volgende code toe:

```
<Menu Height="24" Name="MenuBalk" DockPanel.Dock="Top">
<MenuItem Header="Lettertype">
    <MenuItem Name="Fontjes" Header="Font">
        <MenuItem Name="Courier" Header="Courier New" IsCheckable="True"></MenuItem>
        <MenuItem Name="Arial" Header="Arial" IsCheckable="True"></MenuItem>
    </MenuItem>
    <MenuItem Name="MenuVet" Header="Vetdruk" IsCheckable="True"></MenuItem>
    <MenuItem Name="MenuSchuin" Header="Schuindruk"
        IsCheckable="True"></MenuItem>
    </MenuItem>
</Menu>
```

Op de *Menu* staat nu één keuze nl. *Lettertype* die onderverdeeld is in *Font*, *Vet* en *Schuin*, en waarvan *Font* nog eens is onderverdeeld in *Courier New* en *Arial*.

Dus naar de tags toe: als je binnen een *MenuItem* een volgende *MenuItem* begint zonder de vorige af te sluiten, dan wordt dit een onderverdeling daarvan.

Resultaat:



Door het attribuut *IsCheckable* te gebruiken, kan je ervoor zorgen dat zoals in dit voorbeeld voor "Vetdruk" een vinkje gezet wordt.

Om nu de menukeuzes nog te laten werken (want tot hiertoe heb je alleen de layout gemaakt) moet je nog het *Click*-event per *MenuItem* coderen. Dit doe je in het code-behind venster.

Je gaat voor elke keuze een specifieke procedure schrijven zodat dit op meerdere plaatsen kan worden opgeroepen.

```
private void Vet_Aan_Uit()
{
    if (TextBoxVoorbeeld.FontWeight == FontWeights.Normal) (1)
```



```

    {
        TextBoxVoorbeeld.FontWeight = FontWeights.Bold;
        MenuVet.IsChecked = true;
    }
    else
    {
        TextBoxVoorbeeld.FontWeight = FontWeights.Normal;
        MenuVet.IsChecked = false;
    }
}

private void MenuVet_Click(object sender, RoutedEventArgs e)
{
    Vet_Aan_Uit();
}

private void Schuin_Aan_Uit()
{
    if (TextBoxVoorbeeld.FontStyle == FontStyles.Normal)
    {
        TextBoxVoorbeeld.FontStyle = FontStyles.Italic;
        MenuSchuin.IsChecked = true;
    }
    else
    {
        TextBoxVoorbeeld.FontStyle = FontStyles.Normal;
        MenuSchuin.IsChecked = false;
    }
}

private void MenuSchuin_Click(object sender, RoutedEventArgs e)
{
    Schuin_Aan_Uit();
}

private void Letterttype_Click(object sender, RoutedEventArgs e)
{
    MenuItem hetLetterttype = (MenuItem)sender;
    foreach (MenuItem huidig in Fontjes.Items)
    {
        huidig.IsChecked = false;
    }
    hetLetterttype.IsChecked = true;
    TextBoxVoorbeeld.FontFamily =
        new FontFamily(hetLetterttype.Header.ToString());
}

```

- (1) Staat de tekst in de *TextBox* op *Normal*, dan wordt de *Checked* aangezet, en tekst op *Bold* gezet, anders wordt hij terug *Normal* gezet
- (2) In de *MenuItem* wordt “Vetdruk” aan- of afgevinkt al naar gelang de situatie
- (3) In de *Click-EventHandler* zelf wordt de *Vet_Aan_Uit* procedure opgeroepen.

- (4) Bij het kiezen van een lettertype ga je via de *Sender* parameter bepalen welk lettertype er gekozen is door deze te casten naar een *MenuItem*.
- (5) Om ervoor te zorgen dat alleen het gekozen lettertype aangevinkt wordt, ga je eerst alle vinkjes afzetten door te *lopen* door de *Parent*-tag nl. *Fontjes*.
- (6) Het gekozen lettertype wordt dan aangevinkt, en daarna toegepast op de *TextBox*.

```
<MenuItem Name="Courier" Header="Courier New" IsCheckable="True"
Click="Lettertype_Click"></MenuItem>
<MenuItem Name="Arial" Header="Arial" IsCheckable="True"
Click="Lettertype_Click"></MenuItem>
```

```
<MenuItem Name="MenuVet" Header="Vetdruk" IsCheckable="True"
Click="MenuVet_Click"></MenuItem>
<MenuItem Name="MenuSchuin" Header="Schuindruk" Click="MenuSchuin_Click"
```

Probeer maar uit!

Voeg nu in de XAML-code volgende tag toe onder de andere lettertypes:

```
<MenuItem Name="Comic" Header="Comic Sans MS" IsCheckable="True"
Click="Lettertype_Click"></MenuItem>
```

Zonder iets van extra code te schrijven, komt dit probleemloos in orde.

Probeer maar uit!

7.1.2 Command-items

Omdat bepaalde handelingen binnen het Windows gebeuren als standaard beschouwd worden (bv. Cut, Copy, Paste,...), is het via een *Built-In Command Library* mogelijk om deze in een *MenuItem* aan te spreken.

Voor het *MenuItem* "Lettertype" voeg je volgende *MenuItems* toe:

```
<MenuItem Header="Editeren">
    <MenuItem Header="Knippen" Command="Cut"></MenuItem>
    <MenuItem Header="Kopiëren" Command="Copy"></MenuItem>
    <MenuItem Header="Plakken" Command="Paste"></MenuItem>
</MenuItem>
```



Door het *Command* attribuut te gebruiken, activeer je de standaard "Cut, Copy, Paste" functies die je door middel van de *Header* attribuut een Nederlandstalige benaming naar de gebruiker toe kan geven. Ook de *Shortcut*-toetsen zijn hiervoor geactiveerd en in de menu opgenomen. De functionaliteit achter deze *MenuItems* is impliciet geïmplementeerd.

Probeer maar uit! (denk eraan dat je eerst iets moet selecteren om te kunnen knippen of kopiëren.)

7.1.3 Toevoegen van ShortCut Keys en HotKeys

Om een **ShortCut Key** aan de gewone *MenuItems* toe te voegen, is er meer werk aan de winkel.

Je wil bijvoorbeeld dat *Vetdruk* ook aan- en afgezet kan worden met de toetscombinatie Ctrl+B, en dat *Schuindruk* met de toetscombinatie Ctrl+I wordt uitgevoerd.

Door in de XAML-code het *InputGestureText* attribuut te gebruiken bij de *MenuItems* kan je visueel al een *ShortCut Key* toevoegen. Dit is enkel het zicht maar niet de functionaliteit!

```
<MenuItem Name="MenuVet" Header="Vetdruk" IsCheckable="True"
Click="MenuVet_Click"
    InputGestureText="Ctrl+B"></MenuItem>
<MenuItem Name="MenuSchuin" Header="Schuindruk" IsCheckable="True"
Click="MenuSchuin_Click" InputGestureText="Ctrl+I"></MenuItem>
```

Het toevoegen van de functionaliteit:

- 1) je definieert in het code-behind venster een *RoutedCommand* (simplistisch uitgelegd: je gaat een "route"-beschrijving aan een object toevoegen met een specifieke taak (*Command*) die hij moet uitvoeren als die route bewandeld wordt).

je voegt volgende code toe juist binnen de *Window* klasse

```
public partial class BarWindow : Window
{
    public static RoutedCommand mijnRouteCtrlB = new RoutedCommand();
    public static RoutedCommand mijnRouteCtrlI = new RoutedCommand();
    . . .
```

- 2a) **code-behind versie:**

binnen de constructor van *BarWindow* ga je volgende code toevoegen:

```
public BarWindow()
{
    InitializeComponent();

    CommandBinding mijnCtrlB =
        new CommandBinding(mijnRouteCtrlB, ctrlBExecuted);           (1)
    this.CommandBindings.Add(mijnCtrlB);                             (2)
    KeyGesture toetsCtrlB =
        new KeyGesture(Key.B, ModifierKeys.Control);                (3)
    KeyBinding mijnKeyCtrlB =
        new KeyBinding(mijnRouteCtrlB, toetsCtrlB);                  (4)
    this.InputBindings.Add(mijnKeyCtrlB);                             (5)

    CommandBinding mijnCtrlI =
        new CommandBinding(mijnRouteCtrlI, ctrlIExecuted);
    this.CommandBindings.Add(mijnCtrlI);
    KeyGesture toetsCtrlI =
        new KeyGesture(Key.I, ModifierKeys.Control);
```

```

        KeyBinding mijnKeyCtrlI =
            new KeyBinding(mijnRouteCtrlI, toetsCtrlI);
        this.InputBindings.Add(mijnKeyCtrlI);
    }

    private void ctrlBExecuted(object sender, ExecutedRoutedEventArgs e)    (6)
    {
        Vet_Aan_Uit();
    }

    private void ctrlIExecuted(object sender, ExecutedRoutedEventArgs e)
    {
        Schuin_Aan_Uit();
    }

```

- (1) je definieert een *CommandBinding* die specificeert welke procedure er uitgevoerd wordt als een bepaalde *RoutedCommand* wordt geactiveerd
- (2) je verbindt de *CommandBinding* met een object (in dit geval *this* = *Window*) die die *Route* kan opvangen als ze geactiveerd wordt
- (3) *KeyGesture* definieert een toetscombinatie door middel van zijn twee parameters: 1^{ste} = gewone toets (= in dit geval de letter B), 2^{de} = een *ModifierKey* (= in dit geval de Ctrl-toets)
Een *ModifierKey* is een toets van het klavier (Alt, Shift, Control en Windows) die gecombineerd wordt met de gewone toets.
- (4) Je verbindt de gedefinieerde toetscombinatie met het *RoutedCommand*.
- (5) Je voegt deze verbinding toe aan het *Window*-object waardoor die kan reageren met de procedure die aan die *Route* verbonden is.
- (6) De procedure die moet worden uitgevoerd.

Probeer maar uit!

Voordat je de volgende versie uitprobeert, kan je best alle vorige code in commentaar zetten behalve de *CtrlBExecuted*- en *CtrlIExecuted*-procedures en de declaratie van de *RoutedCommands*.

2b) XAML versie:

volgende code moet worden bijgevoegd:

```

<Window x:Class="Bars.BarWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:custom="clr-namespace:Bars"
        Title="Werken met Bars" Height="350" Width="525">    (1)

    <Window.CommandBindings>    (2)
    <CommandBinding
        Command="{x:Static custom:BarWindow.mijnRouteCtrlB}"
        Executed="ctrlBExecuted" />    (3)
    <CommandBinding
        Command="{x:Static custom:BarWindow.mijnRouteCtrlI}"
        Executed="ctrlIExecuted" />
    </Window.CommandBindings>

    <Window.InputBindings>    (4)

```

```
<KeyBinding Key="B" Modifiers="Control" (5)
    Command="{x:Static custom:BarWindow.mijnRouteCtrlB}"/>
<KeyBinding Key="I" Modifiers="Control"
    Command="{x:Static custom:BarWindow.mijnRouteCtrlI}"/>
</Window.InputBindings>
```

...

```
</Window>
```

- (1) In de *Window*-tag wordt een lokale namespace gedefinieerd (om in de assembly de nodige procedures terug te vinden)
- (2) *Window.CommandBindings* is de verzameling van *RoutedCommands* die bij het object *Window* horen
- (3) je definieert een *CommandBinding* die specificeert welke procedure er uitgevoerd (*Executed*) wordt als een bepaalde *RoutedCommand* wordt geactiveerd
De procedure zelf heb je in de code-behind versie al gemaakt.
- (4) *Window.InputBindings* is de verzameling van *KeyBindings* die een *RoutedCommand* activeert
- (5) *KeyBinding* verbindt een toetscombinatie (*Key* + *Modifier*) aan een *RoutedCommand*

Een **HotKey** is een toetscombinatie die een *MenuItem* activeert door de *Alt*-toets in te drukken en daarna de gespecificeerde letter. Door in het *Header* attribuut een *underscore* (*_*) voor de letter te zetten, bepaal je de *HotKey-letter*.

Zet bij de *MenuItem-Header* "Lettertype" een "_" voor de "L".

Zet bij de *MenuItem-Header* "Vetdruk" een "_" voor de "V".

Zet bij de *MenuItem-Header* "Schuindruk" een "_" voor de "S".

Hierdoor bepaal je bijvoorbeeld bij "Lettertype" dat bij het indrukken van de *Alt*-toets en daarna de letter "L" de *MenuItem* "Lettertype" opengaat. Als je *Alt*-toets inhoudt en dan de letter "V" indrukt, dan wordt de *MenuItem* "Vetdruk" uitgevoerd m.a.w. je zet de tekst in vetdruk of juist niet meer.

7.2 ToolBar

Een *ToolBar* is een werkbalk waar je verschillende knoppen, comboboxen, ... in kan plaatsen. Indien je meer dan één *ToolBar* gebruikt, dan kan je deze best in een *ToolBarTray* zetten zodat je ze onderling kan manipuleren (van plaats wisselen, naast elkaar, onder elkaar, enz.).

Maak in het project een nieuwe map aan met de naam: Images

Kopieer de .png bestanden naar deze map

Stel de *ToolBars* samen zoals in onderstaande XAML-code:

```
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar Name="EditeerBalk">
        <Button Command="Cut"><Image Source="images/cut.png"></Image></Button>
        <Button Command="Copy"><Image Source="images/copy.png"></Image></Button>
```

```
<Button Command="Paste"><Image Source="images/paste.png"></Image></Button>
</ToolBar>
<ToolBar Name="LettertypeBalk">
  <ToggleButton Name="ButtonVet" Click="MenuVet_Click">
    <Image Source="images/bold.png"></Image>
  </ToggleButton>
  <ToggleButton Name="ButtonSchuin" Click="MenuSchuin_Click">
    <Image Source="images/italic.png"></Image>
  </ToggleButton>
</ToolBar>
</ToolBarTray>
```

Resultaat: werkende *ToolBars* met volgend uitzicht:



Omdat *Vet* en *Schuin ToggleButtons* zijn, moet je in de code-behind deze knoppen ook manipuleren als er in de *Menu* voor deze functies gekozen is.

Hierdoor moet je volgende code aanpassen:

```
private void Vet_Aan_Uit()
{
    if (TextBoxVoorbeeld.FontWeight == FontWeights.Normal)
    {
        TextBoxVoorbeeld.FontWeight = FontWeights.Bold;
        MenuVet.IsChecked = true;
        ButtonVet.IsChecked = true;
    }
    else
    {
        TextBoxVoorbeeld.FontWeight = FontWeights.Normal;
        MenuVet.IsChecked = false;
        ButtonVet.IsChecked = false;
    }
}

...

private void Schuin_Aan_Uit()
{
    if (TextBoxVoorbeeld.FontStyle == FontStyles.Normal)
    {
        TextBoxVoorbeeld.FontStyle = FontStyles.Italic;
        MenuSchuin.IsChecked = true;
        ButtonSchuin.IsChecked = true;
    }
    else
    {
        TextBoxVoorbeeld.FontStyle = FontStyles.Normal;
        MenuSchuin.IsChecked = false;
        ButtonSchuin.IsChecked = false;
    }
}
```

Tot hiertoe heb je alleen maar *Buttons* gebruikt, maar op een *ToolBar* kan je ook andere *Controls* gebruiken zoals een *ComboBox*.

Bijna alle mogelijkheden zoals in de *Menu* zijn aanwezig, maar DE grote afwezige momenteel is het kiezen van het lettertype.

Omdat in een *ComboBox* vele mogelijkheden kunnen gestockeerd worden, ga je hier een andere weg op namelijk je gaat alle beschikbare lettertypes laten zien.

Voeg volgende XAML-code toe binnen de *ToolBar* onder de *Buttons*:

```
<ComboBox Name="LettertypeComboBox" Width="150"
  ItemsSource="{Binding Source={x:Static Member=Fonts.SystemFontFamilies}}"> (1)
  <ComboBox.ItemTemplate> (2)
    <DataTemplate> (3)
      <TextBlock FontFamily="{Binding}" Text="{Binding}"/> (4)
    </DataTemplate>
  </ComboBox.ItemTemplate>
</ComboBox>
```

- (1) Je specificeert dat de *Source* van *Items* kan verbonden worden met de huidige *Font.SystemFontFamilies* = de verzameling van *FontFamily* objecten die in *Windows* geïnstalleerd zijn.
- (2) Om al deze objecten op een deftige manier te kunnen tonen in de *ComboBox* maak je gebruik van een *Template* (= een voorbeeld in de vorm van een beschrijving) voor de *Items*.
- (3) Deze *Template* dient om de *Data* (= de inhoud van de *FontFamily*) te layouten.
- (4) De inhoud is een *TextBlock* die per *Item* als *Text*-property de naam van de *FontFamily* toont `Text="{Binding}"`, en deze ook in zijn eigen lettertype toont `FontFamily="{Binding}"`



Om de lijst van lettertypes in de *ComboBox* gesorteerd (alfabetisch) te tonen, moet je nog een *SortDescription* (using `System.ComponentModel`; bovenaan erbij zetten) toevoegen in de *Window_Loaded EventHandler*:

```
LettertypeComboBox.Items.SortDescriptions.Add(
  new SortDescription("Source", ListSortDirection.Ascending));
```

Het tonen van het beginlettertype van de *TextBox* in de *ComboBox* doe je door ook in de *Window_Loaded EventHandler* volgende regel te plaatsen:

```
LettertypeComboBox.SelectedItem =
  new FontFamily(TextBoxVoorbeeld.FontFamily.ToString());
```

Om ervoor de zorgen dat de tekst in de *Textbox* in het gekozen lettertype wordt gezet, kan je in de XAML-code van dit element volgend deel toevoegen:

```
<TextBox Name="TextBoxVoorbeeld" Text="Dit is voorbeeldtekst"
  VerticalAlignment="Stretch" HorizontalAlignment="Stretch"
  FontSize="24" TextWrapping="Wrap" MaxWidth="600" MaxHeight="900">
```

```
FontFamily="{Binding ElementName=LettertypeComboBox, Path=SelectedValue}">
    </TextBox>
```

Probeer maar uit!

Om het geheel een beetje af te werken, kan je ervoor zorgen dat als je een lettertype kiest via de *Menu*, je ook zorgt dat deze in de *ComboBox* wordt geselecteerd.

Verander in de *Lettertype_Click* procedure volgende code van:

```
TextBoxVoorbeeld.FontFamily = new FontFamily(hetLettertype.Header.ToString());
```

naar:

```
LettertypeComboBox.SelectedItem = new
FontFamily(hetLettertype.Header.ToString());
```

Andersom hoort het ook te kloppen: als je in de *ComboBox* een lettertype kiest dat ook in de *Menu* staat, dan moet dit worden aangevinkt, of anders moet alles afgevinkt zijn.

Voeg bij de *ComboBox* een *SelectionChanged*-event toe:

```
private void LettertypeComboBox_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    foreach (MenuItem huidig in Fontjes.Items)
    {
        if (LettertypeComboBox.SelectedItem.ToString() ==
            huidig.Header.ToString())
            huidig.IsChecked = true;
        else
            huidig.IsChecked = false;
    }
}
```

7.3 StatusBar

In de *StatusBar* ga je het gekozen lettertype en de huidige instellingen van *Vet* en *Schuin* laten zien.

Resultaat moet worden:



De *StatusBar* is intern opgebouwd vanuit een *DockPanel*. Dit is belangrijk om weten om de *Items* op de juiste plaats te krijgen.

Voeg onderstaande XAML-code toe binnen de bestaande *StatusBar*-tag:


```

<StatusBar DockPanel.Dock="Bottom" Height="24">
  <StatusBarItem Name="StatusVet" DockPanel.Dock="Right" Width="50"           (1)
    HorizontalContentAlignment="Right" Content="Vet"></StatusBarItem>
  <StatusBarItem Name="StatusSchuin" DockPanel.Dock="Right" Width="50"       (2)
    HorizontalContentAlignment="Right" Content="Schuin"></StatusBarItem>
  <StatusBarItem Name="StatusLettertype"
    Content="{Binding ElementName=LettertypeComboBox, Path=SelectedValue}"> (3)
  </StatusBarItem>
</StatusBar>

```

- (1) Binnen de *StatusBar* ga je van rechts naar links werken omdat het anders onmogelijk wordt om iets aan de rechterkant te positioneren. Het attribuut `DockPanel.Dock="Right"` betekent dat je dit gegeven aan de rechterkant van de *StatusBar* gaat zetten en via `HorizontalContentAlignment="Right"` zorg je ervoor dat binnen een `Width="50"` dit aan de rechterkant van de ruimte plaatst.
- (2) Door dezelfde attributen te gebruiken voor de *Schuindruk* zorg je ervoor dat binnen de overgebleven ruimte alles aan de rechterkant gezet wordt.
- (3) Het gekozen lettertype in de *ComboBox* kan rechtstreeks gekoppeld worden als inhoud (`Content`) van de overgebleven ruimte binnen de *StatusBar*. Via `="{Binding ElementName=LettertypeComboBox, Path=SelectedValue}"` wordt het getoonde lettertype in de *ComboBox* ook hier getoond. Doordat dit een rechtstreekse koppeling is, moet er geen extra code geschreven worden om dit up to date te houden.

Om het geheel een beetje af te werken, kan je ervoor zorgen dat *Vet* en *Schuin* in de *StatusBar* steeds overeenkomen met de rest van de instellingen.

Hiervoor is het nodig om in de XAML-code nog een property toe te voegen:

```

<StatusBarItem Name="StatusVet" DockPanel.Dock="Right" Width="50"
  HorizontalContentAlignment="Right" Content="Vet"
  FontWeight="{Binding ElementName=TextBoxVoorbeeld,
  Path=FontWeight}"></StatusBarItem>
<StatusBarItem Name="StatusSchuin" DockPanel.Dock="Right" Width="50"
  HorizontalContentAlignment="Right" Content="Schuin"
  FontStyle="{Binding ElementName=TextBoxVoorbeeld, Path=FontStyle}">
</StatusBarItem>

```

Probeer maar uit!

8 BESTANDSOPERATIES

8.1 Common dialog box

Windows heeft een aantal *Dialog Boxes* geïmplementeerd die voor alle applicaties gebruikt kunnen worden zoals het opslaan, het openen en afdrukken van bestanden. Hierdoor kan je het de gebruiker gemakkelijker te maken om deze acties te laten verrichten omdat zij al gewend zijn aan het gebruik ervan. Daarom worden ze ook *Common Dialog Boxes* genoemd.

WPF heeft deze boxes opgenomen en stelt ze beschikbaar als klassen via de library `Microsoft.Win32`. (Dus bovenaan in de code-behind de code `using Microsoft.Win32` gebruiken).

Denk eraan: op zich tonen zij enkel een *Window* waar je de bestandsnaam kan kiezen of maken, maar doen ze de acties (opslaan, openen en afdrukken) NIET. Hiervoor moet steeds een koppeling naar een procedure gemaakt worden, die dan effectief de actie doet.

Je gaat het voorbeeld van vorig hoofdstuk nog verder uitbreiden met extra *MenuItems* en extra *Buttons* op de *ToolBar*.

Voeg onderstaande XAML-code toe aan de bovenkant binnen de *Menu*-tag:

```
<Menu Height="24" Name="MenuBalk" DockPanel.Dock="Top">
    <MenuItem Header="_Bestand">
        <MenuItem Header="Openen" Command="Open"></MenuItem>
        <MenuItem Header="Opslaan" Command="Save"></MenuItem>
        <MenuItem Header="Afdrukvoorbeeld" Command="PrintPreview"></MenuItem>
        <MenuItem Header="Afdrukken..." Command="Print"></MenuItem>
        <Separator></Separator>
        <MenuItem Header="Afsluiten" Command="Close"></MenuItem>
    </MenuItem>
    <MenuItem Header="_Editeren">
        . . .
```

Geeft dit als resultaat:



Voeg onderstaande XAML-code toe aan de bovenkant binnen de *ToolBarTray*-tag:

```
<ToolBarTray DockPanel.Dock="Top" Height="30">
    <ToolBar Name="BestandBalk">
        <Button Command="Close">
            <Image Source="images/afsluiten.png"></Image>
        </Button>
        <Button Command="Open">
            <Image Source="images/open.png"></Image>
        </Button>
        <Button Command="Save">
            <Image Source="images/save.png"></Image>
```

```
</Button>
<Button Command="PrintPreview">
    <Image Source="images/preview.png"></Image>
</Button>
<Button Command="Print">
    <Image Source="images/print.png"></Image>
</Button>
</ToolBar>
<ToolBar Name="EditeerBalk">
    . . .
```

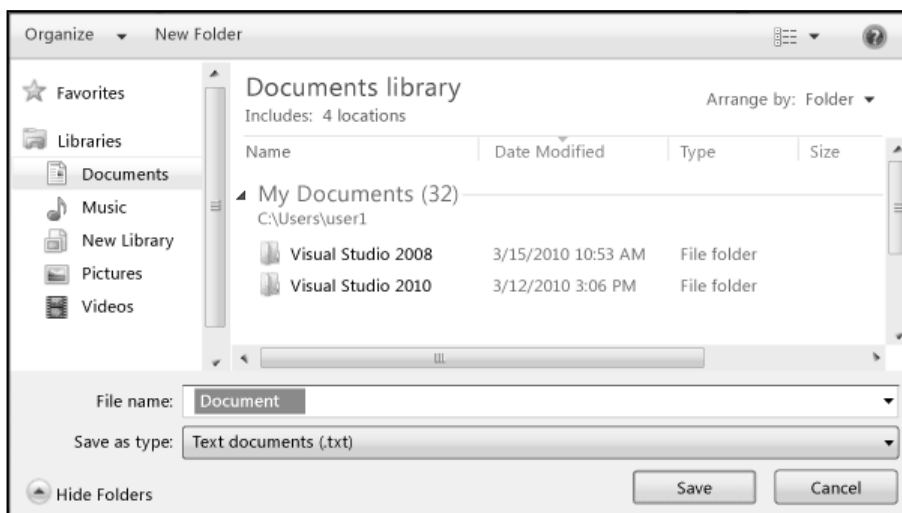
Geeft dit als resultaat:



8.1.1 SaveFileDialog

Je gaat eerst de tekst in de *TextBox* opslaan, je onthoudt ook het gebruikt lettertype, en of vet en schuin aan- of uitstaat.

Voorbeeld van het opslaan-window:



Interessante properties:

AddExtension	Default op true . Als de gebruiker zelf geen extentie aan de bestandsnaam zet, wordt er automatisch de <i>DefaultExt</i> aan de bestandsnaam toegevoegd.
CheckFileExists	Default op false . Moet er een waarschuwing worden gegeven als het bestand nog niet bestaat
CheckPathExists	Default op true . Moet er een waarschuwing worden gegeven als de mappenstructuur nog niet bestaat
CreatePrompt	Default op false . Moet er gevraagd worden om het bestand te mogen creëren
DefaultExt	De extentie die als <i>Filter</i> gebruikt wordt als de <i>Filter</i> property niet is ingevuld, of niet voldoet

FileName	Bevat het volledige pad en naam van het bestand
Filter	Bepaalt welke types van bestanden worden getoond als je het venster opent. Moet ingevuld worden met een vaste structuur zoals bv. Text doc (.txt) *.txt het zichtbare – piping teken – de gecodeerde beschrijving
OverwritePrompt	Default op true . Moet er een waarschuwing gegeven worden als je een bestaand bestand probeert te overschrijven
Title	Bepaalt de tekst in de <i>TitleBar</i> van het venster

De belangrijkste method van dit element:

ShowDialog	Toont het Save-window. Return waarde is true als er OK geklikt is, en false voor de andere mogelijkheden.
-------------------	---

Voeg in de XAML-code volgende *CommandBinding* toe:

```
<Window.CommandBindings>
    . . .
    <CommandBinding Command="Save" Executed="SaveExecuted"></CommandBinding> (1)
</Window.CommandBindings>
```

- (1) Het *Command* “Save” is één van de acties die via een *Built-In Command Library* aanspreekbaar zijn. De shortcut Ctrl+S krijg je er gratis bij, maar in tegenstelling tot *Cut*, *Copy* en *Paste* moet de code voor het uitvoeren van die handeling zelf geschreven worden. (Dit gaat gebeuren in de procedure *SaveExecuted*).

Het volgende dat dient te gebeuren is het schrijven van de code in het code-behind venster:

<pre>private void SaveExecuted(object sender, ExecutedRoutedEventArgs e)</pre>	(1)
<pre>{</pre>	
<pre> try</pre>	
<pre> {</pre>	
<pre> SaveFileDialog dlg = new SaveFileDialog();</pre>	(2)
<pre> dlg.FileName = "Document";</pre>	(3)
<pre> dlg.DefaultExt = ".txt";</pre>	
<pre> dlg.Filter = "Text documents *.txt";</pre>	
<pre> if (dlg.ShowDialog()==true)</pre>	(4)
<pre> {</pre>	
<pre> using (StreamWriter bestand = new StreamWriter(dlg.FileName))</pre>	(5)
<pre> {</pre>	
<pre> bestand.WriteLine(LettertypeComboBox.SelectedValue);</pre>	(6)
<pre> bestand.WriteLine(TextBoxVoorbeeld.FontWeight.ToString());</pre>	(7)
<pre> bestand.WriteLine(TextBoxVoorbeeld.FontStyle.ToString());</pre>	(8)
<pre> bestand.WriteLine(TextBoxVoorbeeld.Text);</pre>	(9)
<pre> }</pre>	
<pre> }</pre>	
<pre> }</pre>	

```
}  
    catch (Exception ex)  
    {  
        MessageBox.Show("opslaan mislukt : " + ex.Message);  
    }  
}
```

- (1) De procedure die je zelf maakt, moet de opgelegde structuur hebben m.a.w. je moet twee parameters definiëren: sender als `object` en `e` als `ExecutedRoutedEventArgs`.
- (2) Je definieert een nieuw *SaveFileDialog* object
- (3) Je geeft als standaardnaam "Document" mee
De default extentie voor het op te slaan document is .txt
Je laat via de *Filter* alleen bestanden zien die de extentie .txt hebben:
het eerste deel is wat je ziet in de bestandsnaambox, en een "|" en het tweede deel is de syntax van de effectieve filter.
- (4) Als er "OK" geantwoord is in de *SaveFileDialog* dan ga je effectief de gegevens opslaan. In alle andere gevallen doe je niets.
- (5) Om gewone tekst naar een bestand te schrijven, gebruik je een *StreamWriter*. Deze is via de `using` `System.IO` aanspreekbaar. Door de `using` in de code te gebruiken, wordt dit object automatisch geopend en op het einde terug gesloten zodat je zelf hier geen aandacht moet besteden.
- (6) Je schrijft de naam van het huidige lettertype als string weg.
- (7) Je schrijft als string of het lettertype *Bold* of *Normal* is.
- (8) Je schrijft als string of het lettertype *Italic* of *Normal* is.
- (9) Je schrijft de inhoud van de *TextBox* als één enkele string weg.

!!! merk op dat de gehele schrijf-operatie binnen een try-catch geschreven is, zodat alles wat misgaat tijdens het schrijven geen crash veroorzaakt, maar als foutmelding getoond wordt.

Mogelijke inhoud van een weggeschreven bestand:

```
Arial  
Normal  
Italic  
Dit is voorbeeldtekst
```

Probeer maar uit!

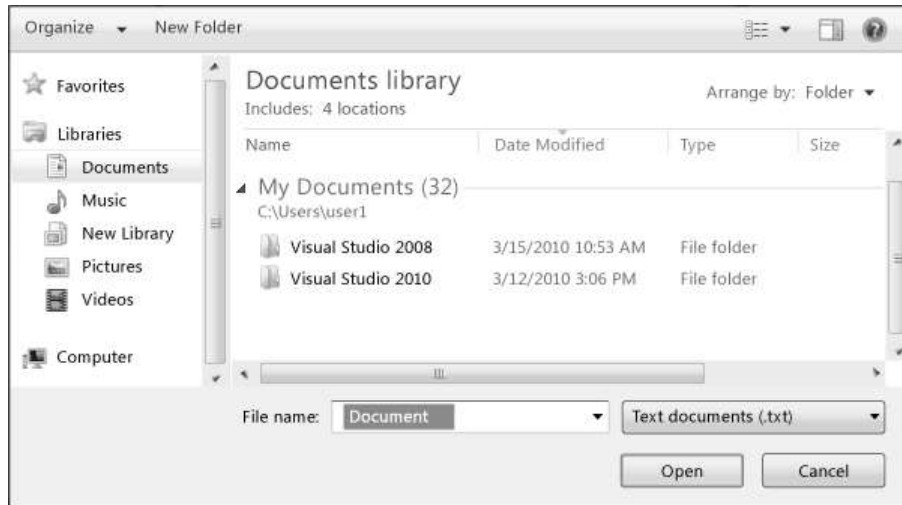
8.1.2 OpenFileDialog

Nu alles opgeslagen is, kan je ook de opgeslagen versie gaan openen en gebruiken in je applicatie.

Heel aanverwant aan de *SaveFileDialog* is zijn tegenhanger *OpenFileDialog* die een *Open* window gaat tonen, maar natuurlijk ook zelf het bestand niet gaat openen.

De meeste properties zijn dezelfde als bij de *SaveFileDialog* (behalve de *CreatePrompt* en de *OverwritePrompt*).

Voorbeeld van het openen-window:



Voeg in de XAML-code volgende *CommandBinding* toe:

```
<Window.CommandBindings>
    . . .
    <CommandBinding Command="Open" Executed="OpenExecuted"></CommandBinding> (1)
</Window.CommandBindings>
```

- (1) Het *Command* "Open" is één van de acties die via een *Built-In Command Library* aanspreekbaar zijn. De shortcut Ctrl+O krijg je er gratis bij, maar de code voor het uitvoeren van die handeling moet zelf geschreven worden. (Dit gaat gebeuren in de procedure *OpenExecuted*).

Het volgende dat dient te gebeuren is het schrijven van de code in het code-behind venster:

```
private void OpenExecuted(object sender, ExecutedRoutedEventArgs e)
{
    try
    {
        OpenFileDialog dlg = new OpenFileDialog();
        dlg.FileName = "Document";
        dlg.DefaultExt = ".txt";
        dlg.Filter = "Text documents |*.txt";

        if (dlg.ShowDialog() == true)
        {
            using (StreamReader bestand = new StreamReader(dlg.FileName)) (1)
            {
                LettertypeComboBox.SelectedValue = (2)
                    new FontFamily(bestand.ReadLine());

                TypeConverter convertBold = (3)
                    TypeDescriptor.GetConverter(typeof(FontWeight));
                TextBoxVoorbeeld.FontWeight = (4)
                    (FontWeight)convertBold.ConvertFromString(bestand.ReadLine());
                Vet_Aan_Uit(true); (5)
            }
        }
    }
}
```

```

        TypeConverter convertStyle =
            TypeDescriptor.GetConverter(typeof(FontStyle));
        TextBoxVoorbeeld.FontStyle =
            (FontStyle)convertStyle.ConvertFromString(bestand.ReadLine());
        Schuin_Aan_Uit(true);

        TextBoxVoorbeeld.Text = bestand.ReadLine();           (6)
    }
}
}
catch (Exception ex)
{
    MessageBox.Show("openen mislukt : " + ex.Message);
}
}

```

- (1) Om gewone tekst vanuit een bestand te lezen, gebruik je een *StreamReader*. Deze is via de `using` *System.IO* aanspreekbaar. Door de `using` in de code te gebruiken, wordt dit object automatisch geopend en op het einde terug gesloten zodat je zelf hier geen aandacht moet besteden.
- (2) Je leest de eerste regel (=lettertype), en via een nieuw *FontFamily* object wordt dit als selectie van de *ComboBox* doorgegeven. Doordat de *ComboBox* verbonden is met het lettertype van de *TextBox* komt die in het juiste lettertype te staan.
- (3) Om *Normal* of *Bold* om te zetten van een string naar een *FontWeight* is er meer werk nodig. Je definieert een *TypeConverter* die via een *TypeDescriptor* de nodige data krijgt van een bepaald object, hier *FontWeight* zodat mogelijke conversies gedaan kunnen worden.
- (4) De *FontWeight* van de inhoud van de *TextBox* wordt ingesteld via een casting van een geconverteerde ingelezen string.
- (5) Via de procedure *Vet_Aan_Uit(true)* worden alles gesynchroniseerd (de Menu, toolbar en statusbar). De *true* parameter krijgt in onderstaande procedure pas zijn betekenis.

Dezelfde werkwijze voor *FontStyle* (3 – 5)

- (6) De laatste ingelezen regel bevat de inhoud van de *TextBox*.

!!! merk op dat de gehele lees-operatie binnen een try-catch geschreven is, zodat alles wat misgaat tijdens het inlezen van de data geen crash veroorzaakt, maar als foutmelding getoond wordt.

Om er nu voor te zorgen dat zowel de *Menu*, de *ToolBar* als de *StatusBar* de juiste waarden tonen, moet je een paar wijzigingen in de *Vet_Aan_Uit* en de *Schuin_Aan_Uit* procedure gaan doen:

Voeg volgende wijzigingen uit in onderstaande code:

```

private void Vet_Aan_Uit(Boolean wissel = false)           (1)
{
    if ((wissel == true                                     (2)
        && TextBoxVoorbeeld.FontWeight == FontWeights.Bold)
        || (wissel == false

```

```

        && TextBoxVoorbeeld.FontWeight == FontWeights.Normal))
    {
        TextBoxVoorbeeld.FontWeight = FontWeights.Bold;
        MenuVet.IsChecked = true;
        StatusVet.FontWeight = FontWeights.Bold;
        ButtonVet.IsChecked = true;
    }
    else
    {
        TextBoxVoorbeeld.FontWeight = FontWeights.Normal;
        MenuVet.IsChecked = false;
        StatusVet.FontWeight = FontWeights.Normal;
        ButtonVet.IsChecked = false;
    }
}

private void Schuin_Aan_Uit(Boolean wissel = false)
{
    if ((wissel == true
        && TextBoxVoorbeeld.FontStyle == FontStyles.Italic)
        || (wissel == false
        && TextBoxVoorbeeld.FontStyle == FontStyles.Normal))
    {
        TextBoxVoorbeeld.FontStyle = FontStyles.Italic;
        MenuSchuin.IsChecked = true;
        ButtonSchuin.IsChecked = true;
        StatusSchuin.FontStyle = FontStyles.Italic;
    }
    else
    {
        TextBoxVoorbeeld.FontStyle = FontStyles.Normal;
        MenuSchuin.IsChecked = false;
        ButtonSchuin.IsChecked = false;
        StatusSchuin.FontStyle = FontStyles.Normal;
    }
}

```

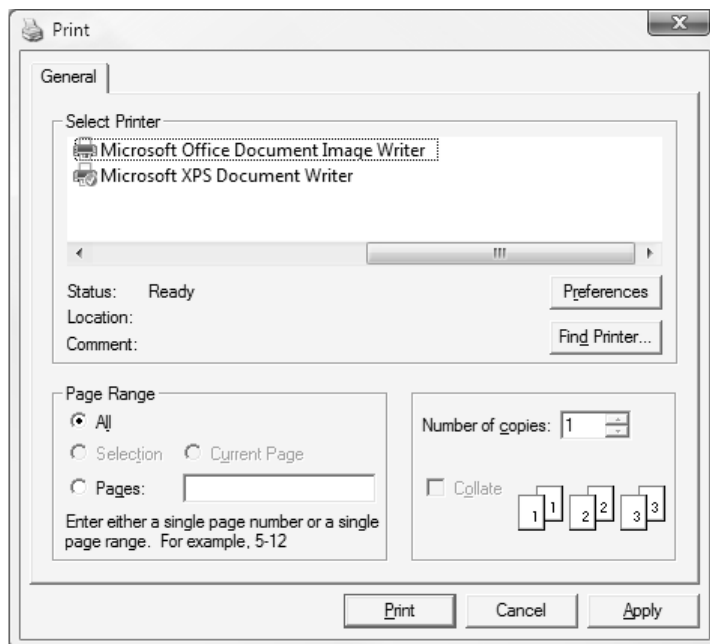
- (1) Tot hiertoe werd er vanuit gegaan dat alles in de *TextBox* juist stond ingesteld in synchronisatie met alle *Bars*, zodat als de procedure werd aangeroepen alles moest veranderd worden. Indien er echter gegevens vanuit bestand worden ingelezen en worden toegepast op de *TextBox* dan zijn de *Bars* niet meer ingesteld. Dus moet er rekening gehouden met het feit of alles al juist staat of juist niet. Dit wordt gedaan door een optionele parameter toe te voegen **Boolean wissel = false**. De normale werking (dus zonder bestand inlezen) blijft behouden doordat de parameter optioneel is en standaard op **false** staat ingesteld.
- (2) Als je de procedure aanroept vanuit het ingelezen bestand, dan moet de situatie in de *TextBox* blijven zoals ze is : **(wissel == true && TextBoxVoorbeeld.FontWeight == FontWeights.Bold)** → dan blijft ook alles in't Bold.
In het andere geval (dus een instelling via een *Bar*) moeten de instellingen veranderen: **(wissel == false && TextBoxVoorbeeld.FontWeight == FontWeights.Normal)** → dan wordt alles in't Bold gezet.

Dezelfde werkwijze geldt voor de *FontStyle*-instellingen.

Probeer alle variaties maar eens uit!

8.1.3 PrintDialog

De *PrintDialog* is een window dat toelaat om een printer te kiezen en eventuele instellingen te doen. Dus alleen printer en de printinstellingen worden hier gekozen, maar er wordt geen printopdracht gegeven. Print = OK-knop, Apply betekent keuzes instellen, maar er niets mee doen, bij alle andere mogelijkheden wordt er aan de printerinstellingen niets veranderd.



Voeg in de XAML-code volgende *CommandBinding* toe:

```
<Window.CommandBindings>
    . . .
    <CommandBinding Command="Print" Executed="PrintExecuted"></CommandBinding> (1)
</Window.CommandBindings>
```

- (1) Het *Command* “Print” is één van de acties die via een *Built-In Command Library* aanspreekbaar zijn. De shortcut Ctrl+P krijg je er gratis bij, maar de code voor het uitvoeren van die handeling moet zelf geschreven worden. (Dit gaat gebeuren in de procedure *PrintExecuted*).

Het uiteindelijke resultaat van een *Print*-opdracht zou onderstaande tekst op papier moeten opleveren:

gebruikt lettertype : Arial
gewicht van het lettertype : Normal
stijl van het lettertype : Italic

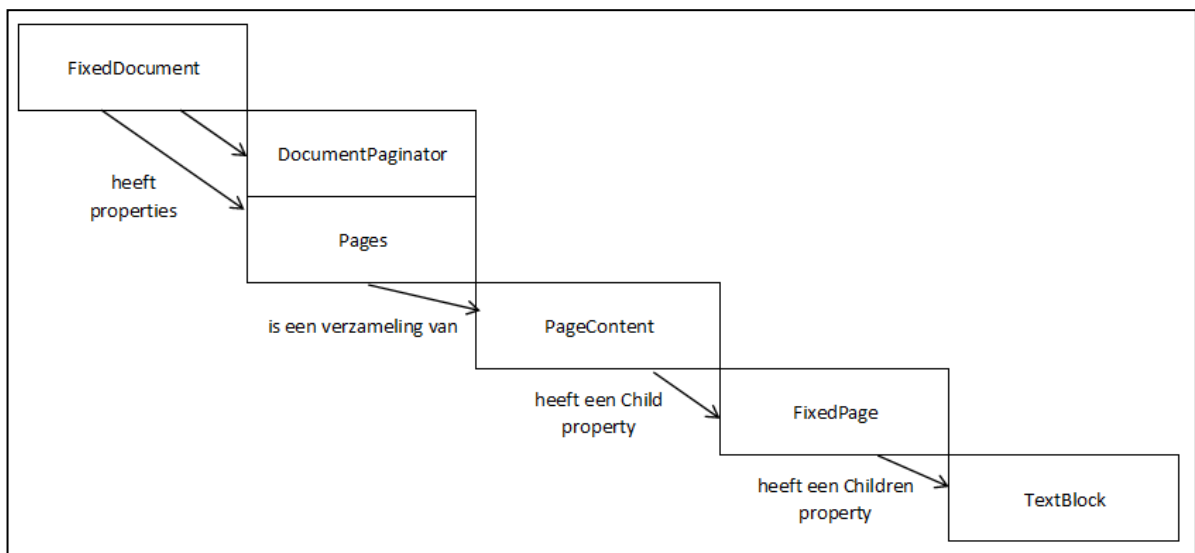
inhoud van de tekstbox :
Dit is voorbeeldtekst

De afdruk samenstellen vormt een heel ander verhaal. Doordat er verschillende info op de afdruk staat, kan je geen gewone schermafdruck nemen. Dus moet je het afdrukdocument zelf gaan samenstellen.

De *PrintDialog* heeft een *PrintDocument* method waarmee je een “document” naar de printer kan sturen. Dit “document” moet van het type *DocumentPaginator* zijn (wat op zich al een onderdeel is van een *FixedDocument*).

FixedDocument opbouw

Om de opbouw van het afdrukdocument te kunnen volgen, is het handig om even de schematische voorstelling van de structuur te bekijken:



Je gaat dus eerst een *FixedDocument* samenstellen waarin de inhoud van de afdruk in zijn geheel moet worden samengesteld. Hiervoor dien je volgende code te schrijven in het code-behind venster:

```

private double A4breedte = 21 / 2.54 * 96; (1)
private double A4hoogte = 29.7 / 2.54 * 96;
private double vertPositie;

private FixedDocument StelAfdrukSamen() (2)
{
    FixedDocument document = new FixedDocument();
    document.DocumentPaginator.PageSize = (3)
        new System.Windows.Size(A4breedte, A4hoogte);

    PageContent inhoud = new PageContent(); (4)
    document.Pages.Add(inhoud);
  
```

```

FixedPage page = new FixedPage();
inhoud.Child = page; (5)

page.Width = A4breedte; (6)
page.Height = A4hoogte;
vertPositie = 96;

page.Children.Add(Regel("gebruikt lettertype : " +
    TextBoxVoorbeeld.FontFamily.ToString())); (7)
page.Children.Add(Regel("gewicht van het lettertype : " +
    TextBoxVoorbeeld.FontWeight.ToString()));
page.Children.Add(Regel("stijl van het lettertype : " +
    TextBoxVoorbeeld.FontStyle.ToString()));
page.Children.Add(Regel(""));
page.Children.Add(Regel("inhoud van de tekstbox : "));
for (int i = 0; i < TextBoxVoorbeeld.LineCount; i++) (8)
{
    page.Children.Add(Regel(TextBoxVoorbeeld.GetLineText(i)));
}
return document; (9)
}

private TextBlock Regel(string tekst) (10)
{
    TextBlock deRegel = new TextBlock();
    deRegel.Text = tekst;
    deRegel.FontSize = TextBoxVoorbeeld.FontSize;
    deRegel.FontFamily = TextBoxVoorbeeld.FontFamily;
    deRegel.FontWeight = TextBoxVoorbeeld.FontWeight;
    deRegel.FontStyle = TextBoxVoorbeeld.FontStyle;
    deRegel.Margin = new Thickness(96, vertPositie, 96, 96);
    vertPositie += 30;
    return deRegel;
}

```

- (1) Om het papierformaat (A4) te kunnen specificeren ga je eerst 2 variabelen definiëren met *A4breedte* als breedte (21cm) en *A4hoogte* als hoogte (29,7cm). De normale eenheid is in pixels, dus is er een omrekening nodig. De 3^{de} variabele gaat de verticale positie op het papier bijhouden.
- (2) Je maakt een procedure *StelAfdrukSamen* die een *FixedDocument* als returnwaarde heeft.
- (3) Je stelt bij een nieuw *FixedDocument.DocumentPaginator* object een *PageSize* in met de afmetingen van een A4-papier.
- (4) Je voegt een nieuw *PageContent* object toe als *Page* van het *FixedDocument*.
- (5) Je voegt een nieuw *FixedPage* object toe als *Child* van de *PageContent*.
- (6) Je definieert de *FixedPage* als een A4-blad met een begin-bovenmarge van 96 pixels (=2,54cm).
- (7) Je voegt de inhoud van de pagina *Regel* per *Regel* toe
- (8) De inhoud van de *TextBox* kan uit verschillende lijnen bestaan, dus wordt er met een *for* gewerkt om lijn per lijn als *Regel* toe te voegen

- (9) Je geeft het samengestelde *FixedDocument* als returnwaarde terug
- (10) Je maakt per regel een nieuw *TextBlock* met
als karakteristieken de instellingen van de *TextBox*,
als inhoud de tekst,
als marge steeds weer een regel lager via de *vertPositie*.

Als de pagina is samengesteld, dan kan je deze gaan afdrukken. Je gaat hiervoor de *PrintExecuted* procedure gebruiken die al gekoppeld is aan het *Command Print*.

Voeg hiervoor in het code-behind venster volgende code toe:

```
private void PrintExecuted(object sender, ExecutedRoutedEventArgs e) (1)
{
    PrintDialog afdrukken = new PrintDialog(); (2)
    if (afdrukken.ShowDialog() == true) (3)
    {
        afdrukken.PrintDocument(StelAfdrukSamen().DocumentPaginator,
            "tekstbox");
    }
}
```

- (1) De procedure heeft weer de opgelegde structuur.
- (2) Je definieert een nieuw *PrintDialog* object
- (3) Als er op de *Print*-knop gedrukt is, dan wordt via de *PrintDocument* method een *FixedDocument* (= *StelAfdrukSamen()*) met zijn *DocumentPaginator* doorgegeven met een *string* "tekstbox" als *description* voor de afdrukopdracht.

Probeer maar eens uit!

8.2 PrintPreview - DocumentViewer

Het is natuurlijk altijd leuk om eerst een idee te hebben wat je gaat afdrukken, alvorens de afdruk zelf te maken (goed voor het milieu = papierbesparing).

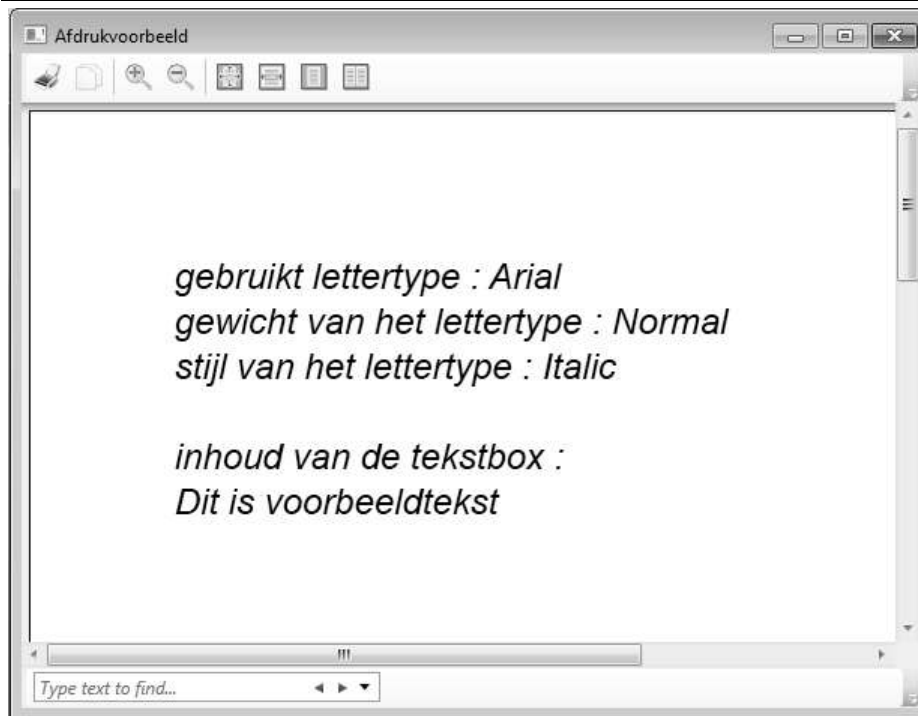
Nu zit het *Command "PrintPreview"* wel in de *Built-In Command Library*, maar daar houdt het voorgekauwde stuk op (geen windows die openen,...).

Je gaat een nieuw Window creëren en gebruik maken van een *DocumentViewer*. Dit is een control die een document laat zien van het type *FixedDocument*. Daar dit soort document al als een pagina is opgebouwd, wordt het als een soort *PrintPreview* getoond.

De belangrijkste property van een *DocumentViewer* :

Document	Dit is het document dat als preview getoond gaat worden. Het moet via <i>IDocumentPaginatorSource</i> aanspreekbaar zijn. (zoals een <i>FixedDocument</i> , die deze interface ook implementeert).
-----------------	---

Voorbeeld van een resultaat:



Je maakt eerst een 2^{de} Window pagina aan:

- In de Solution Explorer klik je met de rechtermuisknop op de projectnaam, en je kiest Add - Window
- Geef het de naam Afdrukvoorbeeld.xaml
- Vervang de *Grid*-tag door een *DocumentViewer*-tag met als attribuut *Name="printpreview"*

```
<Window x:Class="Bars.Afdrukvoorbeeld"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Afdrukvoorbeeld" Height="300" Width="300">
    <DocumentViewer Name="printpreview"></DocumentViewer>
</Window>
```

In het code-behind venster voeg je volgende code toe:

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

    public IDocumentPaginatorSource AfdrukDocument (1)
    {
        get { return printpreview.Document; }
        set { printpreview.Document = value; }
    }
}
```

- (1) Omdat dit *Window* vanuit een ander *Window* wordt aangesproken, is het noodzakelijk om een publieke property te hebben die info van het ene *Window* in het andere *Window* gebruikt.

Hiervoor maak je een public property *AfdrukDocument* van het type *IDocumentPaginatorSource*.

Bij de *set* wordt de property *Document* van de *DocumentViewer* ingevuld, en bij de *get* wordt deze property gereturned.

Je kan de solution "Builden", dit zou geen fouten meer mogen geven.

Om dit *Window* aan te spreken vanuit het *Command* "PrintPreview" is er nog wat werk te verrichten:

Voeg in de XAML-code volgende *CommandBinding* toe:

```
<Window.CommandBindings>
    . . .
    <CommandBinding Command="PrintPreview" Executed="PreviewExecuted">
    </CommandBinding>
</Window.CommandBindings>
```

Voeg in het code-behind venster volgende code toe:

```
private void PreviewExecuted(object sender, ExecutedRoutedEventArgs e)
{
    Afdrukvoorbeeld preview = new Afdrukvoorbeeld();           (1)
    preview.Owner = this;                                       (2)
    preview.AfdrukDocument = StelAfdrukSamen();                 (3)
    preview.ShowDialog();                                       (4)
}
```

- (1) Je maakt eerst een nieuw object aan van een *Afdrukvoorbeeld*.
- (2) Door de *Owner* op het huidige *Window* te zetten, bekom je een relatie tussen beide *Windows*.
- (3) Je vult de publieke property van *Afdrukvoorbeeld* nl. *AfdrukDocument* met een *FixedDocument* (dat je door de procedure *StelAfdrukSamen()* gemaakt hebt).
- (4) Je toont het *Afdrukvoorbeeld* als een *Dialog*-venster wat wil zeggen dat dit eerst terug gesloten moet worden om te kunnen verdergaan met het hoofdvenster.

Probeer maar eens uit!

8.3 Afsluiten

Het programma beëindigen is ook één van de *Commands* die in de *Built-In Command Library* aanwezig is. Je hebt er een *MenuItem* en een *ToolBar-Button* voor voorzien, maar je kan ook het venster sluiten met bv. de *Close-button* rechtsboven in het venster of de toetscombinatie *Alt+F4*. Kortom er zijn veel manieren op het programma af te sluiten.

Om er nu zeker van te zijn dat je steeds op dezelfde manier afsluit, ga je gebruikmaken van een *Event* van het *Window* zelf: *Window_Closing*. Dit treedt op juist VOORDAT het venster gesloten wordt.

Het is altijd netjes om aan de gebruiker te vragen of hij inderdaad het programma wil beëindigen. Vandaar dat je een *MessageBox* gaat voorzien:



Ga naar de *Events* van het *Window* object, en dubbelklik op het *Closing*-event.

Hier ga je de vraag stellen om al dan niet te stoppen met het programma: Voeg hiervoor volgende code toe:

```
private void Window_Closing(object sender, CancelEventArgs e)
{
    if (MessageBox.Show( "Programma afsluiten ?",           (1)
                        "Afsluiten",
                        MessageBoxButton.YesNo,
                        MessageBoxImage.Question,
                        MessageBoxResult.No) == MessageBoxResult.No)
    {
        e.Cancel = true;                                     (2)
    }
}
```

- (1) Je toont een *MessageBox* met
 - de vraag : "Programma afsluiten ?"
 - in de *TitleBar*: "Afsluiten"
 - de *Buttons* Yes en No
 - als figuur links van de vraag: een vraagteken
 - als *DefaultButton* de NO-Button
- (2) Als er No geantwoord wordt, dan wordt de *Close*-event geannuleerd en gebeurt er dus niets, ander gaat het sluiten gewoon verder.

Dit kan je al uitproberen met de *Close-button* rechtsboven in het venster of de toetscombinatie *Alt+F4*.

Nu nog zorgen dat de keuzes op de *Menu* en de *ToolBar* ook werken door het *Command* en koppeling naar de procedure *CloseExecuted* te geven, en deze ook in code te schrijven.

Voeg in de XAML-code volgende *CommandBinding* toe:

```
<Window.CommandBindings>
    . . .
    <CommandBinding Command="Close" Executed="CloseExecuted">
    </CommandBinding>
</Window.CommandBindings>
```

Voeg in het code-behind venster volgende code toe:

```
private void CloseExecuted(object sender, ExecutedRoutedEventArgs e)
{
    this.Close();
}
```



Opdracht: ParkingBon

9 LAYOUT MOGELIJKHEDEN

9.1 StaticResources

Om het duidelijk te maken hoe je met layout te werk gaat, is het gemakkelijk om een instelling in een voorbeeld te zien.

Hiervoor ga je een nieuw project *KleurenKiezer* maken

Noem de pagina *BrushesWindow.xaml* en pas alle benamingen en verwijzingen aan.

Vervang de *Grid*-tag door een *StackPanel*-tag.

Indien bepaalde instellingen (zoals bv. een bepaalde *Color* of *Brush*) op meerdere plaatsen gebruikt gaan worden in de applicatie is het soms handiger om een benaming aan de vaste waarde (*Static*) te geven en ze met hun naam aan te spreken dan telkens opnieuw alle attributen te specificeren.

Je kan hiervoor gebruikmaken van *Resources*. Al naar gelang de plaats waar je ze definieert zijn ze gekend bv. in het *Window* (= *Window.Resources*) : dan zijn ze alleen gekend in dat *Window*.

Voorbeeld:

```
<Window x:Class="KleurenKiezer.BrushesWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="BrushesWindow" Height="160" Width="210">
    <Window.Resources>
        <Color x:Key="paarsColor" A="255" R="170" G="40" B="165"></Color>
        <SolidColorBrush x:Key="paarsBrush"
            Color="{StaticResource paarsColor}">
        </SolidColorBrush>
    </Window.Resources>
    <StackPanel Background="{StaticResource paarsBrush}"></StackPanel>
</Window>
```

Je kan ze ook definiëren op applicatie-niveau. , dan worden ze gedefinieerd in een apart bestand.

Klik met de rechtermuisknop op de projectnaam in de *Solution Explorer* en kies dan voor *Add - New Folder*. Geef deze de naam *Resources*.

Klik met de rechtermuisknop op die folder en kies dan voor *Add - Resource Dictionary...*

Tik als bestandsnaam *turquoise.xaml*.

Binnen de root-tag ga je dan bv. *Colors*, *Brushes* enz. definiëren die als *Resource* bekend gaan zijn.

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Color x:Key="turquoiseColor" A="255" R="50" G="220" B="200"></Color>
    <SolidColorBrush x:Key="turquoiseBrush"
        Color="{StaticResource turquoiseColor}">
    </SolidColorBrush>
```



```
</ResourceDictionary>
```

Het verplichte attribuut **x:Key="..."** is de benaming die gaat aangeroepen worden.

Om deze *Dictionary* voor de ganse applicatie beschikbaar te maken, moet je ze nog in het *App.xaml* bestand als *Application.Resource* definiëren.

```
<Application x:Class="KleurenKiezer.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="BrushesWindow.xaml">
    <Application.Resources>
        <ResourceDictionary Source="Resources/turquoise.xaml">
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

In de applicatie zelf verander je het *Background*-attribuut van het *StackPanel*.

```
<StackPanel Background="{StaticResource turquoiseBrush}"></StackPanel>
```

9.2 Colors en Brushes

9.2.1 Color

Color is de definitie van een kleur. Meestal wordt een kleur voorgesteld met zijn naam (dit zijn voorgedefinieerde kleuren bv. Red, Blue,...) of als hexadecimale waarde (of ARGB-code).

Een hexadecimale waarde voorstellen doe je altijd beginnend met een #. Hierna volgen nog 4 delen van 2 digits:

- A = alpha channel = transparantie = doorzichtigheid :
de waarde varieert van 00 tot FF waar 0 doorzichtig is, een FF ondoorzichtig indien de A wordt weggelaten, dan wordt een *Solid*(=*ondoorzichtig*) kleur bedoeld m.a.w. de waarde FF.
- R = hoeveelheid rood in de kleur: 0 = niks tot FF = alle rood dat er is
- G = hoeveelheid groen in de kleur: 0 = niks tot FF = alle groen dat er is
- B = hoeveelheid blauw in de kleur: 0 = niks tot FF = alle blauw dat er is

Voorbeelden hiervan zijn:

#FFFF0000 = Red = rood

#FF0000FF = Blue = blauw

#FFFFFFFF = White = wit

#FF000000 = Black = zwart

Als R, G en B allemaal dezelfde waarden hebben, dan krijg je een tint van grijs bv. #FF666666.

Opgepast: #00FF0000 = rood dat je niet ziet omdat het transparant is enz.

9.2.2 Brush

Een *Brush* vult een gebied op. Dit kan zijn met een kleur, kleuren, een patroon, een tekening of een afbeelding.

De properties (attributen) van objecten die door een *Brush* worden opgevuld zijn o.a. *Background*, *Fill*, *BorderBrush* en *Stroke*.

Om de mogelijkheden van een *Brush* duidelijk te maken, ga je spelen met de *Background* van dit *StackPanel* die een *Brush* als opvulling verwacht.

Kopieer vanuit de oefenbestanden "regenboog.xaml" naar de *Resources*-map van het project.

Om meerdere bestanden met *Resources* naast elkaar te kunnen gebruiken, wordt er gebruikgemaakt van *ResourceDictionary.MergedDictionaries*.

Verander in het *App.xaml* bestand.

```
<Application x:Class="KleurenKiezer.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="BrushesWindow.xaml">
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Resources/turquoise.xaml">
      </ResourceDictionary>
        <ResourceDictionary Source="Resources/regenboog.xaml">
      </ResourceDictionary>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Het is de bedoeling dat je elk voorbeeld uitprobeert in de XAML-code van de applicatie en het resultaat bekijkt.

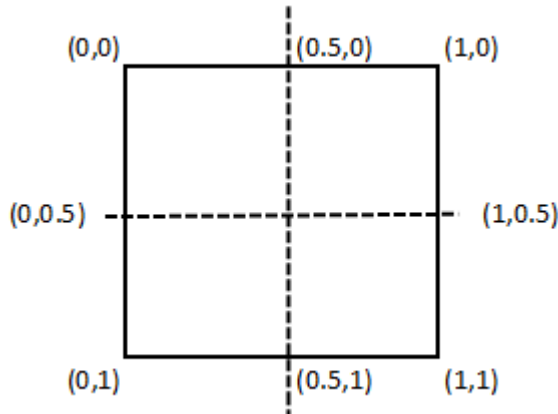
Soorten *Brushes* die als klasse zijn afgeleid:

1) SolidColorBrush : vult een gebied met een ondoorzichtige kleur.

De voorgedefinieerde kleuren zijn beschikbaar als *Color*, maar ook als *SolidColorBrush*. Hierdoor is het mogelijk om de code `Background="Red"` te gebruiken. Je kan ook de hexadecimale waarde van een kleur gebruiken: `Background="#FFAA28A5"` (zou een paarse achtergrond moeten geven)

2) LinearGradientBrush: vult een gebied met lineair gegradeerde kleuren. De gradatie loopt over een lijn die gedefinieerd wordt via een *StartPoint* en een *EndPoint*. Je kan tussenstoppen op die lijn definiëren (= *GradientStop*) waarbij je een waarde geeft van een positie op die lijn (= *Offset*) en de kleur die je op dat moment daar wil.

De lijn wordt met een XY-as bepaald: linksboven = 0,0 en rechtsonder = 1,1



Een horizontale gradatie bv. gaat van `StartPoint="0,0.5"` `EndPoint="1,0.5"`

In de *ResourceDictionary* is er een "regenboog"-kleurcombinatie gedefinieerd. De *LinearGradientBrush* hiervan is benoemd als *rainbowGradient*. Via de code `Background="{StaticResource rainbowLinear}"` kan je deze *Brush* aanroepen. Speel maar in het *regenboog.xaml* bestand met het *Startpoint* en *Endpoint* om de verschillen te zien.

Als de oppervlakte van *StartPoint* tot *EndPoint* niet het volledig gebied omvat, dan hangt het van de *SpreadMethod* af hoe het overschot wordt opgevuld. Standaard staat deze op "Pad" wat inhoudt dat het overschot met de eindkleur wordt opgevuld. Andere mogelijkheden: "Reflect" de opvulling wordt gespiegeld verdergezet of "Repeat" de opvulling wordt herhaald.

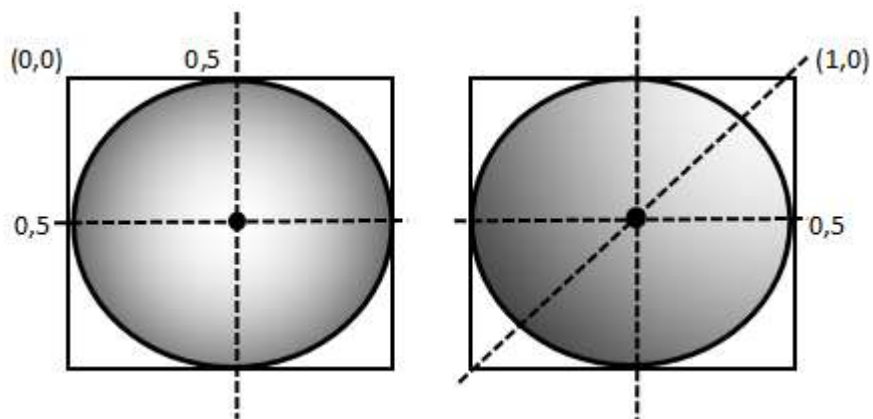
3) RadialGradientBrush: vult een gebied met radiaal gegradeerde kleuren, dus de gradatie wordt vanuit een cirkel gevuld. Het startpunt van de kleurverspreiding wordt bepaald door de *GradientOrigin* (bv. bij `GradientOrigin="0.5,0.5"` begint de kleuring vanuit het midden). Het middelpunt van de cirkel wordt bepaald door de *Center*-property (= is standaard `Center="0.5,0.5"`). Met de instellingen *RadiusX* en *RadiusY* kan je de waarde bepalen van de radiaal van de cirkel (standaard= `RadiusX="0.5"` `RadiusY="0.5"`). De opvullingsmethode wordt bepaald door de *SpreadMethod* (zie *LinearGradientBrush*).

Links:

```
GradientOrigin="0.5,0.5"
Center="0.5,0.5"
RadiusX="0.5"
RadiusY="0.5"
```

Rechts:

```
GradientOrigin="1,0"
Center="0.5,0.5"
RadiusX="0.5"
RadiusY="0.5"
```



4) ImageBrush: vult een gebied met een afbeelding (*Image*)

Kopieer vanuit de oefenbestanden "vdab.bmp" naar het project.

De afbeelding is een *BitmapImage* die via de *ImageSource* wordt opgeladen. Mogelijke formaten zijn .BMP, .GIF, .JPEG, .JPG, .PNG en .TIFF

Doordat je in het *Background*-attribuut van het *StackPanel* niet rechtstreeks een *ImageBrush* kan aanspreken, moet dit gedefinieerd worden via *property element syntax*, of op een andere manier gezegd met *Child elements*.

```
<StackPanel>
  <StackPanel.Background>
    <ImageBrush ImageSource="vdab.bmp"></ImageBrush>
  </StackPanel.Background>
</StackPanel>
```

De *ImageBrush* tag heeft een *ImageSource* attribuut waarin het eventuele *Path* en *Image* naam wordt gezet zoals bv. "vdab.bmp" of "images/vdab.bmp".

Zonder expliciet andere attributen te zetten, wordt de ganse *Background* opgevuld met de *Image* (eventueel verkleind, vergroot, verbreed, enz.)

Wil je dit op een andere manier opvullen, dan zijn daar 2 mogelijkheden voor:

- het *Stretch*-attribuut met zijn mogelijkheden:



- het *Viewport*-attribuut in combinatie met het *TileMode*-attribuut:

Viewport bepaalt de beginpositie in de *Image* en de grootte van de *Image* ten opzichte van het opvulvlak. Bv.

Viewport="0,0,0.2,0.1"

0,0 is de beginpositie (=gewoon de linkerbovenhoek)
0.2 is 1/5de van de breedte van het opvulvlak
0.1 is 1/10de van de lengte van het opvulvlak

Resultaat hangt af van de instelling van *TileMode*:



Dit hoeft zich niet te beperken tot achtergronden, dit soort *Brush* kan je bv. ook gebruiken als opvulling van tekst :

Pas de inhoud van het *StackPanel* aan naar onderstaande code:

```
<StackPanel>
    <TextBlock Text="M" FontSize="100" FontWeight="ExtraBold">
        <TextBlock.Foreground>
            <ImageBrush ImageSource="vdab.bmp"
                Viewport="0,0,0.2,0.1"
                TileMode="Tile" >/ImageBrush>
        </TextBlock.Foreground>
    </TextBlock>
</StackPanel>
```

Resultaat: tekst ingekleurd met VDAB-logootjes.



5) **DrawingBrush**: vult een gebied met een tekening (*Drawing*)

Je kan een gebied ook opvullen met een *Drawing*. Hiermee wordt een vorm, tekst, tekening of video bedoeld.

Je maakt in je voorbeeld zelf een tekening aan: een groepering van ellipsen die een soort oog als resultaat geven.

Verander de XAML-code van het *StackPanel* naar het volgende toe:

```
<StackPanel>
    <StackPanel.Background>
        <DrawingBrush>
            <DrawingBrush.Drawing>
                <GeometryDrawing Brush="Red">
```

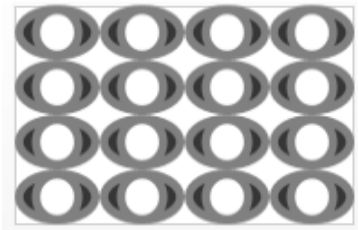
```
<GeometryDrawing.Geometry>
  <GeometryGroup>
    <EllipseGeometry RadiusX="25" RadiusY="25" Center="50,50" />
    <EllipseGeometry RadiusX="45" RadiusY="25" Center="50,50" />
  </GeometryGroup>
</GeometryDrawing.Geometry>
<GeometryDrawing.Pen>
  <Pen Thickness="10" Brush="Gray">
</Pen>
</GeometryDrawing.Pen>
</GeometryDrawing>
</DrawingBrush.Drawing>
</DrawingBrush>
</StackPanel.Background>
</StackPanel>
```

Dit zou hetvolgende als resultaat moeten geven:



Via de *Viewport* en *TileMode* kan je verschillende visualisaties totstandbrengen:

```
<DrawingBrush TileMode="Tile" Viewport="0,0,0.25,0.25">
```



6) *VisualBrush*: vult een gebied met een visueel object

De meest veelzijdige en krachtige manier om een gebied te vullen is met een *VisualBrush*. Het gebied wordt opgevuld met een *Visual*, wat één van de basisklassen binnen de grafische *Types* is. Je kan *Controls* als *Visual* gebruiken voor het gebied met andere woorden, bijna alle grafische objecten komen in aanmerking als opvulling.

Als voorbeeld ga je een *ToggleButton* en een *TextBox* ernaast vergroot weergeven als je de *Button* aanklikt, als je hem uitdrukt verdwijnt de vergroting. Hiermee krijg je ook een idee dat alles wat je tothiertoe in XAML-code gedaan hebt, evengoed in het code-behind venster kan instellen.

Vervang in de XAML-code de *StackPanel*-tag en al zijn onderliggende code door onderstaande code:

```
<DockPanel>
  <StackPanel Name="PanelMetKnop" Margin="5,0">
    <ToggleButton Name="VergrootButton" Background="Orange" Height="60"
      Width="125" Content="Vergroot"></ToggleButton>
    <TextBox Height="50" Width="125" Margin="0,5">
      hier kan je iets tikken</TextBox>
  </StackPanel>
```

```
<Rectangle Name="Vergroting" Height="400" Width="500"  
           HorizontalAlignment="Left" VerticalAlignment="Top">  
    </Rectangle>  
</DockPanel>
```

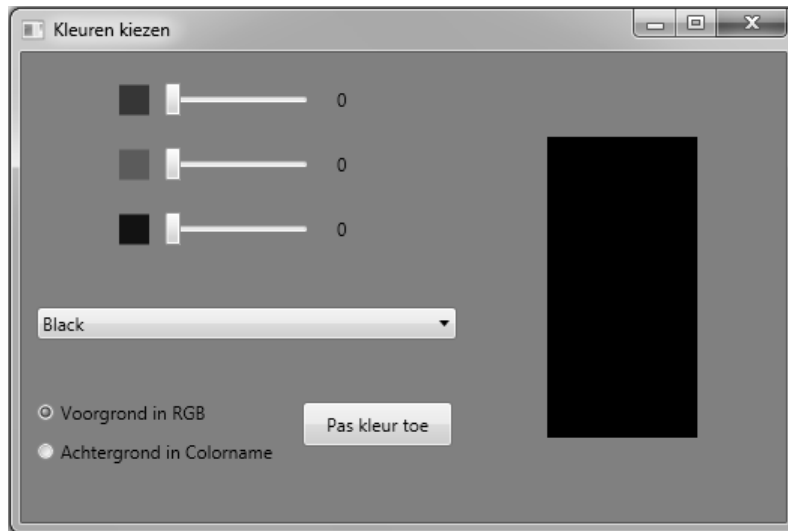
Ga naar het *Click*-event van de *Button* om in het code-behind venster de functionaliteit van de knop te schrijven:

```
private void VergrootButton_Click(object sender, RoutedEventArgs e)  
{  
    if (VergrootButton.IsChecked==true) (1)  
    {  
        VergrootButton.Content = "Zonder vergroting"; (2)  
        VisualBrush zicht = new VisualBrush(); (3)  
        zicht.TileMode = System.Windows.Media.TileMode.FlipY; (4)  
        zicht.Viewport= new Rect(0,0,1,0.5); (5)  
        zicht.Visual = PanelMetKnop; (6)  
        Vergroting.Fill =zicht; (7)  
    }  
    else  
    {  
        VergrootButton.Content = "Vergroot"; (8)  
        Vergroting.Fill = null; (9)  
    }  
}
```

- (1) Als de *ToggleButton* is ingedrukt, dan
- (2) Wijzig je de tekst op de *ToggleButton* naar "Zonder vergroting"
- (3) Je creëert een nieuwe *VisualBrush*
- (4) Je zet de *TileMode* op *FlipY* (= eronder ondersteboven herhalen)
- (5) De *ViewPort* bepaalt waar de opvulling begin (= 0,0) dus linksboven en bepaalt de grootte van de opvulling (1,0.5) = over heel de breedte, en de helft van de hoogte
- (6) Als opvulling zelf (*Visual*) geef je de naam van het *StackPanel* door waardoor de inhoud van dit *Panel* het uiterlijk van de opvulling wordt.
- (7) Je geeft aan de *Rectangle* deze *Visual* door
- (8) Je wijzigt de tekst op de *ToggleButton* naar "Vergroot"
- (9) Je zet de opvulling van de *Rectangle* op niets (*null*).

Probeer maar uit (en tik iets in de *TextBox*)

9.3 Voorbeeldapplicatie



9.3.1 De functionaliteit van het voorbeeld

Je kan op 2 manieren kleuren veranderen:

- 1) Je kan via de 3 *Sliders* in combinatie met de *RadioButton* "Voorground in RGB" de kleur van de *Rectangle* aanpassen naar de nieuwe instellingen. Hierbij moet gechecked worden of de instelling overeenkomt met een *gekende kleur*, en indien dit zo is, dan wordt in de *ComboBox* deze kleur getoond.
- 2) Je kan via een keuze in de *ComboBox* in combinatie met de *RadioButton* "Achtergrond in Colorname" de achtergrond van de het rechthoek-gedeelte aanpassen. De *Sliders* worden op de juiste RGB waardes gezet.

Door te klikken op de *Button* bevestig je de nieuwe instellingen.

9.3.2 De XAML-code

Om niet te veel tijd te verliezen, ga je in het huidige project, de XAML-pagina `KleurenKiezenWindow.xaml` en zijn code-behind-pagina `KleurenKiezenWindow.xaml.cs` kopiëren.

Verander in het `App.xaml` bestand de `StartupUri` naar dit bestand (zodat dit wordt gebruikt als de applicatie wordt opgestart).

Een beetje uitleg over de XAML-code (het meeste zou gekend moeten zijn :-):

- **Slider** : behalve zijn afmetingen zijn er belangrijke attributen ingesteld:
 - `Minimum="0"` – `Maximum="255"` : een byte heeft een range van 0 tot 255
 - `SmallChange="1"` : bij het verplaatsen is dit minstens 1 volledige eenheid
 - `IsSnapToTickEnabled="True"` : hij springt automatisch naar de –vorige– volgende geldige waarde (dus in dit geval een geheel getal)
- **Label** : de *Content* van de *Label* is gekoppeld aan de *Value* van de *Slider*
- **ComboBox** : de inhoud hiervan ga je in het code-behind venster opvullen. De *Items* van de *ComboBox* zijn van de klasse *Kleur* (zie hieronder). Hierdoor moet je specificeren welke property getoond wordt, en welke als onthouden waarde dient: `DisplayMemberPath="Naam"` `SelectedValuePath="Naam"`

- **Button** : heeft een *Click* – event die je nog in code gaat schrijven.

9.3.3 De klasse *Kleur*

Om aan verschillende gegevens over een *Kleur* te komen, is het handig om hiervoor een eigen klasse te gebruiken:

Klik met de rechtermuisknop op de projectnaam in de *Solution Explorer* en kies dan voor *Add - Class*.

Tik als bestandsnaam *Kleur.cs*

De inhoud van deze klasse:

```
using System.Windows.Media;

public class Kleur
{
    public SolidColorBrush Borstel { get; set; }
    public string Naam { get; set; }
    public string Hex { get; set; }
    public byte Rood { get; set; }
    public byte Groen { get; set; }
    public byte Blauw { get; set; }
}
```

9.3.4 De code-behind

1) Het opvullen van de *comboBoxKleuren* kan gebeuren in de constructor van het *Window*:

Voeg volgende code toe:

```
using System.Reflection; (1)
...
public KleurenKiezenWindow()
{
    InitializeComponent();
    foreach (PropertyInfo info in typeof(Colors).GetProperties()) (2)
    {
        BrushConverter bc = new BrushConverter();
        SolidColorBrush deKleur =
            (SolidColorBrush)bc.ConvertFromString(info.Name); (3)
        Kleur kleurke = new Kleur();
        kleurke.Borstel = deKleur;
        kleurke.Naam = info.Name;
        kleurke.Hex = deKleur.ToString();
        kleurke.Rood = deKleur.Color.R;
        kleurke.Groen = deKleur.Color.G;
        kleurke.Blauw = deKleur.Color.B;
        comboBoxKleuren.Items.Add(kleurke);
        if (kleurke.Naam == "Black") (4)
            comboBoxKleuren.SelectedItem = kleurke;
    }
}
```

- (1) gebruik bovenaan de `using System.Reflection`; bibliotheek om de *PropertyInfo* te kunnen aanspreken van een bepaalde klasse

- (2) via de method *GetProperties* van een klassetype nl. `typeof(Colors)` krijg je toegang tot de publieke *Properties* van alle objecten van die klasse. Door te *lopen* door alle objecten, kan je de inhoud van de *Property* "Name" van elk object toevoegen aan de *ComboBox*.
- (3) Via de *BrushConverter* kan je van de naam een *SolidColorBrush* maken en alle properties ervan opvragen.
- (4) Als de startkleur van de *Rectangle* initieel op *Black* staat, selecteer je ook deze waarde in de *ComboBox*.

2) Door te klikken op de Button activeer je de `buttonKleur_Click` *EventHandler*.

```
private void buttonKleur_Click(object sender, RoutedEventArgs e)
{
    if (radioVoorgrond.IsChecked==true) (1)
    {
        rechthoek.Fill = new SolidColorBrush(Color.FromRgb( (2)
            Convert.ToByte(labelRed.Content.ToString()),
            Convert.ToByte(labelGreen.Content.ToString()),
            Convert.ToByte(labelBlue.Content.ToString())));
        comboBoxKleuren.SelectedIndex = -1; (3)
        foreach (Kleur kleurnaam in comboBoxKleuren.Items) (4)
        {
            if (rechthoek.Fill.ToString() == kleurnaam.Hex) (5)
                comboBoxKleuren.SelectedItem = kleurnaam; (6)
        }
    }

    if ((radioAchtergrond.IsChecked == true) && (7)
        (comboBoxKleuren.SelectedIndex>=0))
    {
        kleur gekozenKleur = (Kleur)comboBoxKleuren.SelectedItem; (8)
        panelVoorbeeld.Background = gekozenKleur.Borstel; (9)
        sliderRed.Value = gekozenKleur.Rood; (10)
        sliderGreen.Value = gekozenKleur.Groen;
        sliderBlue.Value = gekozenKleur.Blauw;
    }
}
```

- (1) Als de *RadioButton* om de voorgrond te veranderen aangeboden is dan...
- (2) De *Rectangle* wordt opgevuld met een *SolidColorBrush* De *Color* zelf moet samengesteld worden vanuit de waardes in de *Labels* die de *RGB* waardes voorstellen.
De *Content* van een *Label* is een object die met de *ToString()* wordt omgezet naar een *String*. Deze *String* moet op zich naar een *Byte* worden omgezet. Via de *FromRGB* method van het *Color* object wordt de kleur samengesteld.
- (3) Om een eventuele *ColorName* ermee te laten overkomen, wordt de *ComboBox* op zijn initiële waarde gezet (en wordt er dus geen kleur getoond).
- (4) Je gaat door alle kleuren van de *ComboBox* *lopen*.

- (5) Je gaat de *Hexadecimale string* van de kleur van de opvulling vergelijken met de *Hex* property.
- (6) Als de *waardes* gelijk zijn, dan zet je het object *kleurnaam* als geselecteerd *Item* in de *ComboBox*.
- (7) Als de *RadioButton* om de achtergrond te veranderen aangeboden is EN er staat een kleur geselecteerd in de *ComboBox* dan...
- (8) Je zet het gekozen *Item* van object om naar een *Kleur*.
- (9) Je gebruikt de *Borstel* property om de achtergrondkleur in te stellen
- (10) De 3 *Sliders* moeten nu nog de juiste *RGB-waarden* tonen. Dit doe je door de *Rood*, *Groen* en *Blauw* properties in te vullen in de *Slider.Value*.

Dit zou een werkende applicatie moeten zijn, dus experimenteren maar !

9.4 Styles

Tot hiertoe heb je alle opmaak bij de inhoud geplaatst, en dit is eigenlijk tegen de regels in: inhoud en opmaak gescheiden houden.

De bedoeling van *Styles* is om voor algemene attributen (properties) een standaardwaarde te definiëren, zodat je ze niet bij elk element hoeft te herhalen. Hierdoor wordt vooral de opmaak afgezonderd van de inhoud, en hoeft een designer niets van de code te weten.

Je kan het in grote mate vergelijken met CSS.

Styles kan je toevoegen op verschillende niveau's in de applicatie: in de XAML-code van het *Window* als *Window.Resources*, of algemener als een *Resource Dictionary*, wat een duidelijker overzicht geeft. De *Styles* gedefinieerd in de *Window*-XAML-code heeft wel voorrang op de algemene aanpak. Dus liefst beide niet mengen.

Kopieer het bestand *standaardStijlen.xaml* naar de *Resources*-map.

Voeg de *ResourceDictionary* toe in het *App.xaml* bestand.

```
...
<ResourceDictionary>
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="Resources/turquoise.xaml">
    </ResourceDictionary>
    <ResourceDictionary Source="Resources/regenboog.xaml">
    </ResourceDictionary>
    <ResourceDictionary Source="Resources/standaardStijlen.xaml">
    </ResourceDictionary>
  </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
...
```

Let op : de volgorde van de verschillende *ResourceDirectories* zijn van belang, want alles wordt doorlopen van boven naar onder toe.

Bekijk de inhoud van *standaardStijlen.xaml*:

Een *Style* bestaat uit de hoofd-tag *Style*. Noodzakelijke attribuut is het *TargetType* die specificeert waarop de *Style* van toepassing is. Bv. op een *Button*:

```
<Style TargetType="{x:Type Button}"></Style>
```

`x:Type` wil zeggen dat deze *Style* voor alle elementen van dit *Type* geldt (in dit geval alle *Buttons*).

Binnen deze tag wordt via een *Setter*-tag een *Property* een bepaalde *Value* gegeven. Bv. de voorgrondkleur van de *Button* heeft de *Brush* "Blue".

```
<Style TargetType="{x:Type Button}">
  <Setter Property="Foreground" Value="Blue"></Setter>
</Style>
```

Je kan ook via een *Trigger*-attribuut zorgen dat er in bepaalde omstandigheden een andere opmaak is. Bv. Als je over de *Button* gaat, verandert de voorgrondkleur naar *Red*.

```
<Style TargetType="{x:Type Button}">
  <Setter Property="Foreground" Value="Blue"></Setter>
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Foreground" Value="Red"></Setter>
    </Trigger>
  </Style.Triggers>
</Style>
```

Als je nu voor verschillende elementen eenzelfde opmaak wenst zoals bv. een bepaald lettertype en grootte, dan kan je gebruik maken van een *Keyed Style*. Dit is een gewone *Style* met een *x:Key* attribuut, waardoor je deze kan aanroepen. Bv.

```
<Style x:Key="baseControlStijl" TargetType="{x:Type Control}">
  <Setter Property="FontSize" Value="14"></Setter>
  <Setter Property="FontFamily" Value="Comic Sans MS"></Setter>
</Style>
```

Bovenstaande *Style* is van toepassing op alle elementen die zijn afgeleid van het basiselement *Control* en zorgt ervoor dat het lettertype in *Comic Sans MS 14 punt* wordt gezet.

Afgeleide elementen kunnen dit als basis gebruiken en toch verder van hun eigen opmaak voorzien door dit als attribuut op te nemen: Bv. een *Label* heeft een *Style* gebaseerd op *baseControlStijl* maar uitgebreid met *ExtraBold* in het lettertype:

```
<Style TargetType="{x:Type Label}" BasedOn="{StaticResource baseControlStijl}">
  <Setter Property="FontWeight" Value="ExtraBold"></Setter>
</Style>
```

Je kan de applicatie uitvoeren en naar de resultaten kijken.

9.5 Templates

Templates gaan verder dan alleen maar een bepaalde *stijl* aan een element geven.

Met een *DataTemplate* bepaal je via een beschrijving eventueel het uiterlijk en de inhoud van het *Data*-element. Met een *ControlTemplate* bepaal je het ganse uiterlijk van het element.

9.5.1 Data Templates

Tijdens de cursus heb je al een paar keer gebruikgemaakt van een *DataTemplate*: bij het maken van een *ComboBox* op de *ToolBar* met de lettertypes, en bij de implementatie van de *ListBox*.

Bij het kleurenvoorbeeld ga je de *ComboBox* een verfijnder uiterlijk geven:

Voor:



Na:



Om ervoor te zorgen dat steeds de kleur in de *Rectangle* overeenkomt met de naam van de kleur erachter, ga je gebruikmaken van een *DataTemplate*.

Vervang in de XAML-code

```
<ComboBox Name="comboBoxKleuren" Margin="10,30" DisplayMemberPath="Naam"
SelectedValuePath="Naam"></ComboBox>
```

Door

```
<ComboBox Name="comboBoxKleuren" Margin="10,30">
  <ComboBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <Rectangle Fill="{Binding Path=Naam}" Stroke="Black"
          StrokeThickness="2" Height="25" Width="50">
        </Rectangle>
        <Label Content="{Binding Path=Naam}" Foreground="Black"></Label>
      </StackPanel>
    </DataTemplate>
  </ComboBox.ItemTemplate>
</ComboBox>
```

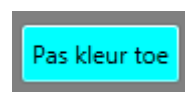
- (1) De *Items* worden beschreven in een *Template*
- (2) De *Data* (= de iteminhoud) wordt beschreven in een *Template*
- (3) De inhoud omvat een *StackPanel* (= je gaat alles naast elkaar zetten)
- (4) Je tekent een *Rectangle* met een *Fill* (=opvulkleur) die verbonden is met de *Naam*-property van het huidige element. Het *Stroke* (= omranding van de rechthoek) heeft een kleur en een dikte.
- (5) de *Content* van het *Label* is verbonden met de *Naam*-property van het huidige element, m.a.w. de naamkleur wordt erachter gezet met een zwarte kleur (*Foreground="Black"*).

9.5.2 Control Templates

Met een *Control Template* is het de bedoeling dat je het zicht en de structuur van een *Control* verandert zonder dat het gedrag van de *Control* verandert.

Een eerste simpel voorbeeld: de *Button* buttonKleur heeft momenteel een standaard uiterlijk. Je gaat alleen in dat *Window* die knop wijzigen:

Voor



Na



Hiervoor ga je eerst in het XAML-bestand `regenboog.xaml` een *Brush* bijmaken (die je ook nog op andere plaatsen gaat gebruiken):

```
<RadialGradientBrush x:Key="zwartwit" GradientOrigin="0.5,0.5">
    <GradientStop Color="White" Offset="0" />
    <GradientStop Color="#FF333333" Offset="1" />
</RadialGradientBrush>
```

Vervolgens voeg je bovenaan in de XAML-code van het *KleurenKiezenWindow*-bestand een *Window.Resource* toe :

```
<Window.Resources>
    <ControlTemplate x:Key="kleurknop" TargetType="{x:Type Button}">           (1)
        <Grid Height="30" Width="80">                                         (2)
            <Ellipse Width="80" Height="30" Fill="{StaticResource zwartwit}"   (3)
                Stroke="Black"></Ellipse>
            <ContentPresenter HorizontalAlignment="Center"                     (4)
                VerticalAlignment="Center"></ContentPresenter>
        </Grid>
    </ControlTemplate>
</Window.Resources>
```

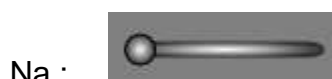
- (1) Eerst geef je de *Template* een *Key* (`x:Key="kleurknop"`) en moet je vertellen voor welke soort *Control* dit een *Template* is (`TargetType="{x:Type Button}"`)
- (2) De visualisatie van de *Button* is een *Grid*-oppervlakte
- (3) Hierin teken je een *Ellipse* die een *zwartwit* opvulling heeft
- (4) Met de *ContentPresenter* bepaal je waar de *Content* van deze *Control* wordt gepositioneerd (= in het midden)
Let op: vergeet je de *ContentPresenter* te specificëren, dan wordt er GEEN *Content* op de *Button* getoond !

Als laatste stap zorg je ervoor dat de *Button* zelf deze *Template* gebuikt:

```
<Button Name="buttonKleur" Margin="10" Height="30" Width="80" Content="Pas kleur toe" Click="buttonKleur_Click" Template="{StaticResource kleurknop}"></Button>
```

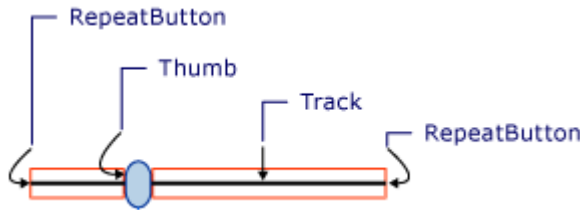
Bekijk het resultaat !

Wil je het algemener aanpakken en bepalen dat bijvoorbeeld alle *Sliders* binnen de applicatie een bepaald uiterlijk moeten hebben, dan ga je via een *Style* de *Template* definiëren voor alle *Sliders*.



Om een correcte *Template* op te bouwen is het belangrijk om te weten uit welke onderdelen een bepaalde *Control* bestaat.

Voor de *Slider* zijn dit de onderdelen waar je rekening mee gaat houden (dit zijn niet alle onderdelen !):



Elk onderdeel moet in de *Template* beschreven staan, of het verliest zijn eventuele zicht en functionaliteit.

In het bestand *standaardStijlen.xaml* ga je een *Style* bij definiëren.

```
<Style TargetType="{x:Type Slider}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type Slider}">
        <Grid>
          <Border Height="8" CornerRadius="8"
            Background="{StaticResource zwartwit}"></Border>
        </Grid>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Hiermee krijg je als resultaat een *Track* zonder *Thumb* te zien, dus je kan de waarde van de *Slider* niet rechtstreeks wijzigen.

Dus ga je een *Track* moeten definiëren die dan ook nog een vaste naam heeft (= PART_Track). Via de *TemplatePartAttribute* klasse kan je de *named parts* voor het *Template* van de *Control* te weten komen.

De *Thumb* van deze *Track* is ook een vereiste tag bij de opbouw.

Voeg volgende regels toe:

```
<ControlTemplate TargetType="{x:Type Slider}">
<Grid>
  <Border Height="8" CornerRadius="8"
    Background="{StaticResource zwartwit}"></Border>
  <Track Name="PART_Track">
    <Track.Thumb>
      <Thumb Height="20" Width="10">
    </Thumb>
    </Track.Thumb>
  </Track>
</Grid>
</ControlTemplate>
```

Hierdoor krijg je volgende resultaat te zien:



Nu nog een aangepaste *Thumb* maken om het geheel af te werken. Dit is een volwaardige *Control* dus kan je hier een *Style* voor maken.

Voeg onder de *Slider Style* volgende code toe:

```
<Style TargetType="{x:Type Thumb}">
  <Setter Property="Template">
```

```

        <Setter.Value>
            <ControlTemplate TargetType="{x:Type Thumb}">
                <Canvas Width="16" Height="16">
                    <Ellipse Width="16" Height="16"
                        Fill="{StaticResource zwartwit}" Stroke="Black"></Ellipse>
                </Canvas>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>

```

Nu zou het eindresultaat bereikt moeten zijn. Probeer maar uit.

Let er ook op de de *Repeat* functionaliteit niet meer aanwezig is (mits deze niet in de *Template* gedefinieerd is).

9.6 Themes

Bij het maken van een *WPF-applicatie* wordt er qua layout automatisch gebruik gemaakt van het actieve *Windows Theme*. Dit houdt in dat kleuren, pictogrammen, geluiden, bureaublad achtergrond, schermbeveiliging enz. gebruikt worden die door *Windows* zijn ingesteld. Dit houdt echter ook in dat een gebruiker die een ander *Theme* gebruikt dan de ontwerper een totaal ander zicht kan krijgen op het uiterlijk van je *WPF-applicatie*.

Zo'n *Theme* houdt dus in dat alle *Styles*, *Templates* enz. beschreven worden. Het is dus eigenlijk een verzameling van alle instellingen die in één bestand beschreven is. Vandaar ook dat het een XAML-bestand is.

Kopieer van de oefenbestanden-map *ShinyRed.xaml* naar de *Resources*-map van het *Project*:

Voeg onderstaande code toe in het bestand *App.xaml*.

```

...
<ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Resources/turquoise.xaml">
        </ResourceDictionary>
        <ResourceDictionary Source="Resources/regenboog.xaml">
        </ResourceDictionary>
        <ResourceDictionary Source="Resources/standaardStijlen.xaml">
        </ResourceDictionary>
        <ResourceDictionary Source="Resources/ShinyRed.xaml">
        </ResourceDictionary>
    </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
...

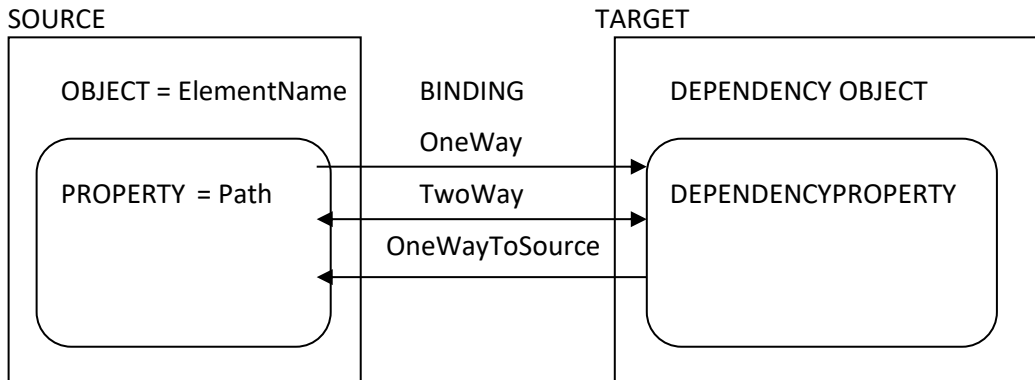
```

Alle instellingen die boven het *Theme* gebeurd zijn, worden overschreven door instellingen van het *Theme*. Wat in de *Window.Resources* geschreven is op *Window* niveau overschrijft daarna weer de *Theme*-instelling.

Voer de applicatie opnieuw uit, en zie het verschil in visualisatie!

10 DATABINDING

De bedoeling bij Databinding is dat je een Property van een Object verbindt met een Dependency Property van een Dependency Object zodat zij in runtime gesynchroniseerd worden.



Object en *Property* zouden vertrouwde termen moeten zijn en deze worden binnen de *Binding* respectievelijk *ElementName* en *Path* genoemd., terwijl het andere deel spreekt over *Dependency*. Het belangrijke voor deze cursus is dat een *DependencyObject* één van de basisklassen is binnen de *System.Windows* namespace en dat een *FrameworkElement* (zie begin cursus) een afgeleide klasse is van deze klasse. De meeste *FrameworkElement* properties zijn *Dependency*-properties die databinding ondersteunen. Je hebt je dus geen zorgen te maken.

De *Binding* kan op verschillende manieren gebeuren: van object naar dependency object (= *OneWay*), van dependency object naar object (= *OneWayToSource*) en heen en weer tussen beide (= *TwoWay*) wat meestal ook de *Defaultwaarde* is (als je niet zeker bij een bepaald object bent, test het soort *Binding* dan of zoek het op). Een variatie op *OneWay* is *OneTime* wat een éénmalige verbinding van object naar dependency object inhoudt.

Tijdens het doornemen van de cursus heb je deze techniek al verschillende keren toegepast zonder dieper in te gaan op de techniek zelf. Nu kan je *DataBinding* op verschillende niveau's toepassen:

10.1 Binding to ... een Control

Voorbeelden die je al hebt gemaakt:

- In vorig hoofdstuk door de waarde van de *Slider* te koppelen aan de waarde die in een *Label* getoond wordt :

```
<Label Name="labelBlue" VerticalAlignment="Center" Width="50"
Content="{Binding ElementName=sliderBlue, Path=Value}" ></Label>
```

- Bij het maken van de *ComboBox* om de verschillende *FontFamilies* te tonen in hun eigen lettertype :

```
<ComboBox Name="LettertypeComboBox" Width="150"
ItemsSource="{Binding Source={x:Static Member=Fonts.SystemFontFamilies}}">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <TextBlock FontFamily="{Binding}" Text="{Binding}" />
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

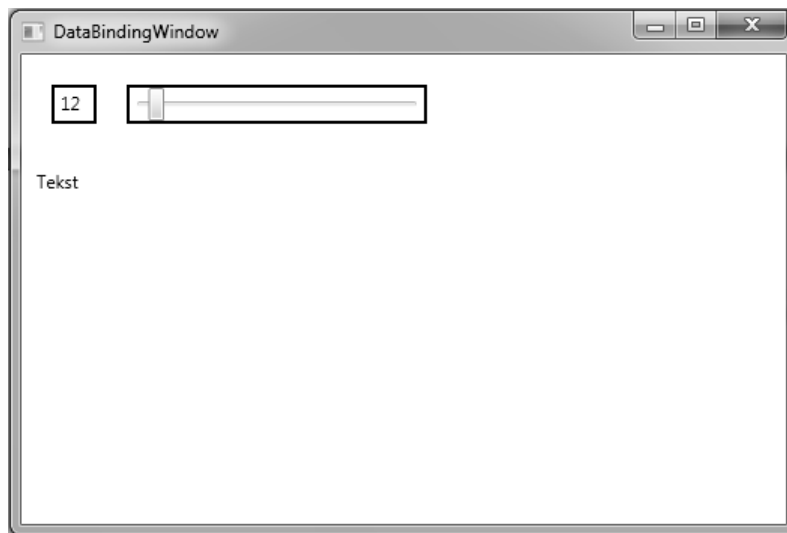
Om alles systematisch te overlopen, ga je van een propere lei beginnen:

Hiervoor ga je een nieuw project maken: `DataBinding`.

Verwijder de standaard gegeven XAML-pagina.

Kopieer van de oefenbestanden-map het bestand `DataBindingWindow.xaml` en `DataBindingWindow.xaml.cs` naar het Project en verander in `App.xaml` de `StartupUri`.

Resultaat als je de applicatie uitvoert:



Je gaat ervoor zorgen dat een verschuiving van de *Thumb* van de *Slider* de lettergrootte van de *Tekst* verandert en dat de waarde in de *TextBox* getoond wordt.

Verander volgende regels in de XAML-code:

```
...
<TextBox Name="grootteTextBox" Style="{StaticResource baseControlStijl}"
Width="30" HorizontalAlignment="Left"
Text="{Binding ElementName=grootteSlider, Path=Value}"></TextBox>
...
```

(1)

```
...
<TextBlock Margin="10"
FontSize="{Binding ElementName=grootteSlider, Path=Value}"
Text="Tekst"></TextBlock>
...
```

(2)

- (1) Databinding :
Object;ElementName = grootteSlider, Property;Path = Value
Dependency Object = TextBox, Dependency Property = Text
Dus je de Value-Property van de grootteSlider met verbindt de Text-property van de TextBox.
- (2) Databinding :
Object; ElementName= grootteSlider, Property;Path = Value
Dependency Object = TextBlock, Dependency Property = Fontsize
Dus je verbindt de Value-Property van de grootteSlider met de Fontsize-property van het TextBlock.

Probeer maar uit!

Mits dat de *Defaultwaarde* voor een *TextBox- Binding TwoWay* is, wil dit zeggen dat als je het getal in de *TextBox* wijzigt en dan de TAB-toets gebruikt (=focus verplaatsen), de *Slider* zich automatisch zal aanpassen. Grotere waarden dan het maximum worden herleid naar het maximum, en kleinere waarden dan het minimum worden herleid naar het minimum;

Bij wijze van test kan je volgende regel in de XAML-code aanpassen:

```
...
<TextBox Name="grootteTextBox" Style="{StaticResource baseControlStijl}"
Width="30" HorizontalAlignment="Left"
Text="{Binding ElementName=grootteSlider, Path=Value,
Mode=OneWay}"></TextBox>
...
```

(1)

(1) Door de **Mode=OneWay** toe te voegen in de *Binding*-specificatie wordt alleen de *TextBox* aangepast als je de *Slider* wijzigt, maar niet meer omgekeerd.

Probeer maar uit!

Een andere schrijfwijze voor de *TextBox*:

```
<TextBox Name="grootteTextBox" Style="{StaticResource baseControlStijl}"
Width="30" HorizontalAlignment="Left">
    <TextBox.Text>
        <Binding ElementName="grootteSlider" Path="Value">
        </Binding>
    </TextBox.Text>
</TextBox>
```

(Binding wordt niet gesuggereerd, maar moet je zelf helemaal tikken).

10.1.1 UpdateSourceTrigger

Wis de laatste wijziging (het toevoegen van **Mode=OneWay**) maar terug uit de XAML-code.

De *UpdateSourceTrigger* is de property van de *Binding* die bepaalt wanneer de wijzigingen van de *Source* moeten worden doorgevoerd. Deze property heeft dus alleen nut bij een *TwoWay* of *OneWayToSource* mode van de *Binding*.

Voor de meeste elementen staat de *default* waarde van de *UpdateSourceTrigger* op *PropertyChanged* (wanneer de waarde wijzigt). Voor de *TextBox* is de waarde *LostFocus* (wanneer de *TextBox* zijn focus verliest). Dit is om te voorkomen dat de gebruiker niet alles kan aanpassen vooraleer de wijzigingen worden doorgevoerd. In dit voorbeeld is de minimumwaarde van de *Slider* op 6 ingesteld. Als je in de *TextBox* de waarde wijzigt, en je tikt bv. het getal 1, dan zou dit automatisch worden gecorrigeerd naar 6, terwijl het de bedoeling was om bv. 18 in te vullen.

Je kan manueel, dat is in de XAML-code, de *default*-waarde overschrijven door zelf aan de *UpdateTriggerSource* een waarde te geven.

Verander de code van de *TextBox* :

```
<TextBox Name="grootteTextBox" Style="{StaticResource baseControlStijl}"
Width="30" HorizontalAlignment="Left"
Text="{Binding ElementName=grootteSlider, Path=Value,
UpdateSourceTrigger=PropertyChanged }"></TextBox>
```

Hierdoor zeg je dat de wijzigingen in de *Binding* bij het veranderen van de property mogen worden doorgevoerd.

Probeer maar uit ! Je kan de waarde niet in bv. 18 veranderen omdat de 1 altijd een 6 wordt.

Dus tijd voor een tweede aanpassing: verander de minimumwaarde van de *Slider* naar 0.

Nu krijg je de kans om 1 in te tikken, wat dan rechtstreeks resulteert in tekst van *FontSize* 1 in de *TextBlock*. Als je de 8 tikt wordt dit weer aangepast.

Mogelijke waarden voor de *UpdateSourceTrigger* :

- *PropertyChanged* : wijzigingen worden doorgevoerd als de waarde van de property wijzigt
- *LostFocus* : wijzigingen worden doorgevoerd als de property zijn focus verliest
- *Explicit* : wijzigingen worden pas doorgevoerd als de method *UpdateSource* wordt aangeroepen
- *Default* : *UpdateSourceTrigger* wordt terug op zijn defaultwaarde gezet

10.2 Binding to ... een Collection

Als je gebruikmaakt van een *Control* waar meerdere gegevens (*Items*) in gestockeerd worden zoals een *ListBox* of een *ComboBox*, dan kan je deze gaan verbinden met een *Collection*.

Naar het voorbeeld toe, ga je een *ComboBox* toevoegen die de *FontFamilies* gaat tonen. Bij het kiezen van een *Font* verandert de inhoud van de *TextBlock* naar dat lettertype.

Voeg juist onder de *Slider*-tag volgende code toe:

```
<ComboBox Name="lettertypeComboBox" Width="180"
    Style="{StaticResource baseControlStijl}"
    ItemsSource="{Binding Source={x:Static Member=Fonts.SystemFontFamilies}}">
</ComboBox>
```

De *ItemsSource* is een verbinding naar de *Source* van de *Items* namelijk de huidige collection *Font.SystemFontFamilies* = de verzameling van *FontFamily* objecten die in *Windows* geïnstalleerd zijn.

Je kan dit stukje al eens uittesten : resultaat is *ComboBox* met lettertypes in.

Volgende stap is de *TextBlock* laten reageren op een gekozen lettertype.

Hiervoor voeg je code aan de *TextBlock*-tag:

```
<TextBlock Name="Tekstblok" Margin="10" Text="Tekst"
    FontSize="{Binding ElementName=grootteSlider,Path=Value}"
    FontFamily="{Binding ElementName=lettertypeComboBox,
    Path=SelectedValue}">
</TextBlock>
```

Doordat de *Items* in de *ComboBox* van het type *FontFamily* zijn, kan je het geselecteerd lettertype (*SelectedValue*) rechtstreeks koppelen aan het *FontFamily*-attribuut van de *ComboBox*.

Om het geheel wat afgewerkter te maken, ga je de lettertypes in de *ComboBox* nog sorteren, en bij het opstarten van de applicatie wordt het lettertype "Arial" als gekozen lettertype ingesteld.

Dit kan echter niet in de XAML-code gebeuren, dus zul je in het code-behind venster bij de constructor van het *Window* wat code moeten toevoegen:

```
using System.ComponentModel;

. . .

public DataBindingWindow()
{
    InitializeComponent();
    SortDescription sd = new SortDescription("Source",
        ListSortDirection.Ascending);
    lettertypeComboBox.Items.SortDescriptions.Add(sd);
    lettertypeComboBox.SelectedItem = new FontFamily("Arial");
}
```

10.2.1 DataContext

Als je het nu uitprobeert, dan krijg je een gesorteerde lijst te zien van alle mogelijke lettertypes, maar die staan nog allemaal in hetzelfde lettertype. Om ze in hun eigen lettertype te tonen, ga je gebruik moeten maken van een *DataTemplate* (omdat voor elk *Item* het lettertype moet wijzigen).

Verander in de XAML-code de *ComboBox*-tag:

```
<ComboBox Name="lettertypeComboBox" Width="180"
    Style="{StaticResource baseControlStijl}"
    DataContext="{x:Static Member=Fonts.SystemFontFamilies}"
    ItemsSource="{Binding}">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <TextBlock FontFamily="{Binding}" Text="{Binding}"/>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

DataContext is een property waarvan de inhoud de *Source* van een *Binding* definieert. Alle *Child*-tags van de tag waar dit gespecificeerd is, erven automatisch deze *Binding* instelling.

Zo hoef je de *Binding Source* maar één maal de specificeren, en kan je in alle onderliggende tags met het simpele `"{Binding}"` er naar refereren.

Probeer maar uit!

10.3 Binding to ... een klasse

Ook dit heb je reeds in de cursus gedaan: bij het maken van een *ListBox* van *Hobbies*, heb je de *ListBox* gevuld met objecten van de klasse *Hobby*.

Om een bepaalde property van de klasse aan te spreken, kan je via `{Binding Path=property}"` deze aanspreken.

Een deel van de XAML-code voor het voorbeeld dat je toen gemaakt hebt, staat hierna:

```
<ListBox Name="ListBoxHobbies" Height="100" Width="300" Margin="10 0">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Border BorderBrush="Black" BorderThickness="1" Width="275">
                <StackPanel Orientation="Horizontal">
                    <Image Source="{Binding Path=Symbol}" Stretch="Fill"
                        Height="40" Width="40"></Image>
                    <TextBlock VerticalAlignment="Center"
                        Text="{Binding Path=Activiteit}"></TextBlock>
                </StackPanel>
            </Border>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

10.3.1 INotifyPropertyChanged

Zolang je werkt via de voorgedefinieerde *Binding* mogelijkheden en *Objecten* gebruikt die deze functionaliteit ondersteunen, is er geen probleem. Wat als er zelfgemaakte klassen en code gebruikt wordt? Een voorbeeld :

Je maakt zelf de klasse *Persoon*

```
public class Persoon
{
    public string Naam { get; set; }
    public Persoon(string nNaam)
    {
        Naam = nNaam;
    }
}
```

In de XAML-code voeg je een *TextBox* en twee *Buttons* toe juist boven de onderste afsluiting van het *StackPanel* (zodat dit onder de tekst komt te staan).

```
<StackPanel Name="veranderPanel" Orientation="Horizontal" Margin="10">
    <TextBox Name="veranderTextBox" Width="100" Height="30" Margin="10"></TextBox>
    <Button Name="veranderButton" Content="verander" Margin="10"></Button>
    <Button Name="toonNaamButton" Margin="10" Content="toon naam"></Button>
</StackPanel>
```

In het code-behind venster maak je een *Persoon*-object en je koppelt dat als *DataContext* aan de *StackPanel* die de *TextBox* en de *Buttons* omvat.

```
public Persoon persoon = new Persoon("jan");

public DataBindingWindow()
{
    InitializeComponent();
    SortDescription sd = new SortDescription("Source",
ListSortDirection.Ascending);
    lettertypeComboBox.Items.SortDescriptions.Add(sd);
    lettertypeComboBox.SelectedItem = new FontFamily("Arial");
    veranderPanel.DataContext = persoon;
}
```

Je verbindt de *Text*-property van de *TextBox* met de *Naam*-property van de *Persoon*:

```
<TextBox Name="veranderTextBox" Width="100" Height="30"
  Text="{Binding Path=Naam}" Margin="10"></TextBox>
```

Je kan dit uitproberen : in de *TextBox* verschijnt "jan" en als je de "jan" verandert, verandert ook de inhoud van de property *Naam* van het *Persoon-object*.

Met de *Button* *toonNaamButton* ga je in een *MessageBox* de inhoud van de *Naam* van het *Persoon-object* tonen:

```
private void toonNaamButton_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(persoon.Naam);
}
```

De *Button* zal altijd dezelfde inhoud tonen als de *TextBox*.

Met de tweede *Button* ga je de inhoud van het *Persoon-object* wijzigen:

```
private void veranderButton_Click(object sender, RoutedEventArgs e)
{
    persoon.Naam = "piet";
}
```

Nu zie je in de *TextBox* geen "piet" staan (ook al is deze gekoppeld met het *Persoon-object*), maar als je de *Naam* toont met de *Button* geeft deze wel "piet" als inhoud.

Om ervoor te zorgen dat ook de wijzigingen via code van het object worden doorgegeven aan de *TextBox* ga je de interface *INotifyPropertyChanged* in de klasse *Persoon* implementeren. In de set van de property wordt een method opgeroepen die op zijn beurt de *PropertyChanged* event gaat oproepen voor de property met de naam die als parameter wordt doorgegeven.

```
using System.ComponentModel;
...
public class Persoon : INotifyPropertyChanged
{
    private string naamValue;
    public string Naam {
        get
        {
            return naamValue;
        }
        set
        {
            naamValue = value;
            RaisePropertyChanged("Naam");
        }
    }
    public Persoon(string nNaam)
    {

```

```

    Naam = nNaam;
}

private void RaisePropertyChanged(String info)
{
    if (PropertyChanged != null)
    {
        PropertyChanged (this, new PropertyChangedEventArgs(info));
    }
}

public event PropertyChangedEventHandler PropertyChanged;
}

```

Als je nu via de Button de Naam van de Persoon wijzigt, zal de inhoud van de *TextBox* mee wijzigen.

10.4 Binding parameters

Bij sommige gegevens is het handig om deze een bepaalde layout te geven zoals bijvoorbeeld een geldelijke waarde (valuta) met het juiste symbool weer te geven, of een datum op een bepaalde manier te tonen.

Voeg ter voorbereiding in de Persoon klasse 2 properties toe:

```

private decimal weddeValue;

public decimal Wedde
{
    get { return weddeValue; }
    set { weddeValue = value;
        RaisePropertyChanged("Wedde");
    }
}

private DateTime inDienstValue;

public DateTime InDienst
{
    get { return inDienstValue; }
    set { inDienstValue = value;
        RaisePropertyChanged("InDienst");
    }
}

```

Pas ook de constructor aan :

```

public Persoon(string nNaam, decimal nWedde, DateTime nInDienst)
{
    Naam = nNaam;
    Wedde = nWedde;
    InDienst = nInDienst;
}

```

De XAML-code breid je uit met twee *TextBoxen* die onder de *Buttons* komen te staan:

```

...
<Button Name="toonNaamButton" Margin="10" Content="toon naam"

```



```
Click="toonNaamButton_Click"></Button>
<TextBox Name="weddeTextBox" Margin="10" Width="75"
    Text="{Binding Path=Wedde}"/>
<TextBox Name="datumTextBox" Margin="10" Width="130"
    Text="{Binding Path=InDienst}"/>
</StackPanel>
```

Nu de code-behind nog aanpassen:

```
public Persoon persoon = new Persoon("jan", 3500m, new DateTime(2011, 2, 13));
. . .

private void veranderButton_Click(object sender, RoutedEventArgs e)
{
    persoon.Naam = "piet";
    persoon.Wedde = 4125.5m;
    persoon.InDienst = new DateTime(2010, 12, 21);
}
```

Als je nu het programma uitvoert, krijg je volgend resultaat te zien:

3500	2/13/2011 12:00:00 AM
------	-----------------------

De wedde is zonder duizendtalscheidingsteken en valuta-symbool.

De datum wordt getoond in de vorm mm/dd/yyyy met de *Time* erbij.

Er zijn 2 parameters (attributes) in *Binding* die ervoor zorgen dat de waarden op een andere manier getoond worden:

10.4.1 StringFormat

Je kan gebruikmaken van vooral gedefinieerde formaten waarvan je een overzicht op <https://msdn.microsoft.com/en-us/library/dwhawy9k.aspx> vindt.

Om bijvoorbeeld de wedde als een valuta-getal te tonen kan je *StringFormat=C* toevoegen in de *Binding* :

```
Text="{Binding Path=Wedde, StringFormat=C}"
```

Om de datum alleen als datum te tonen zonder het *Time*-gedeelte erbij, gebruik je *StringFormat=d* :

```
Text="{Binding Path=InDienst, StringFormat=d}"
```

Resultaat :

\$3,500.00	2/13/2011
------------	-----------

Dit is al iets beter van opmaak, maar standaard gebruikt XAML de oorspronkelijke instelling van de geïnstalleerde Windows, dus in dit geval de Amerikaanse instellingen (het dollarteken, en m/d/y voor de datum).

Je kan ook zelf je formaat samenstellen, wat in het geval van de datum een goed idee zou zijn. Door gebruik te maken van symbolen (zie lijst) kan je zelf je formaat samenstellen. Door dd-MM-yyyy te gebruiken bedoel je:

- dd = dag in 2 cijfers voorstellen
- MM = maand in 2 cijfers voorstellen (de kleine "m" staat voor minuut en niet voor maand, dus vergis je je daar niet in)
- yyyy = jaartal wordt in 4 cijfers getoond

Verander het *StringFormat* van *InDienst* naar:

```
StringFormat=dd-MM-yyyy
```

en kijk naar het resultaat :

\$3,500.00	13-02-2011
------------	------------

Hiermee staat de datum in het gewenste formaat.

10.4.2 ConverterCulture

XAML gebruikt standaard de language instelling (en-US). Hierdoor komt er een \$-teken voor het bedrag van de wedde te staan. Om gebruik te maken van de waarden die jij hebt ingesteld in Windows (Control Panel – Language-Region – Additional settings) ga je gebruikmaken van de *CultureInfo*.

Deze klasse is terug te vinden in de namespace *System.Globalization*, waardoor je bovenaan in de *Window*-tag onderstaande regel moet toevoegen:

```
xmlns:global="clr-namespace:System.Globalization;assembly=mscorlib"
```

Om gebruik te maken van *CurrentCulture* moet je ervoor zorgen dat je project met **minimaal .Net Framework 4.6** werkt (dit is terug te vinden in de properties van het project).

Om de wedde te tonen met de huidige instellingen (bv. nl-BE) ga je in de *Binding* volgend attribuut toevoegen:

```
Text="{Binding Path=Wedde, StringFormat=C,  
ConverterCulture={x:Static global:CultureInfo.CurrentCulture}}"
```

Als je dit uitprobeert, krijg je :

€ 3.500,00	13-02-2011
------------	------------

Je kan dit natuurlijk ook toepassen op de *InDienst* datum: verander

```
StringFormat=dd-MM-yyyy in  
StringFormat=d, ConverterCulture={x:Static global:CultureInfo.CurrentCulture}
```

Hierdoor krijg je volgend resultaat:

€ 3.500,00	13/02/2011
------------	------------

Je ziet dat de dag-maand-jaar gescheiden zijn door een slash zoals ingesteld in de Region-instellingen.

Een andere mogelijkheid is dat je bijvoorbeeld bij de *InDienst* niet het euro-teken wil zien, maar voluit het woordje 'euro'. Dan kan je volgende combinatie toepassen:

```
StringFormat='euro #,##0.00',  
ConverterCulture={x:Static global:CultureInfo.CurrentCulture}
```

waardoor je volgend resultaat krijgt:

euro 3.500,00	13-02-2011
---------------	------------

De mogelijkheden zijn legio.



Opdracht: BloemSamenstelling

11 DRAG AND DROP

In een applicatie wordt er veel met de muis gewerkt, en daarom is het soms ook handig om bepaalde handelingen met de muis uit te voeren zoals bv. het verplaatsen of invullen van gegevens.

De actie bestaat uit 2 handelingen nl. DRAG (het slepen) en DROP (het neerzetten). Wat je sleept en neerzet is een object. Soms is dit object heel eenvoudig zoals bv. een *Rectangle*. Maar soms is het meer complex dan dit bv. een *ListBox* met *ListItems* die *Objecten* bevatten. Dus weet wat je sleept!

11.1 Een simpel object

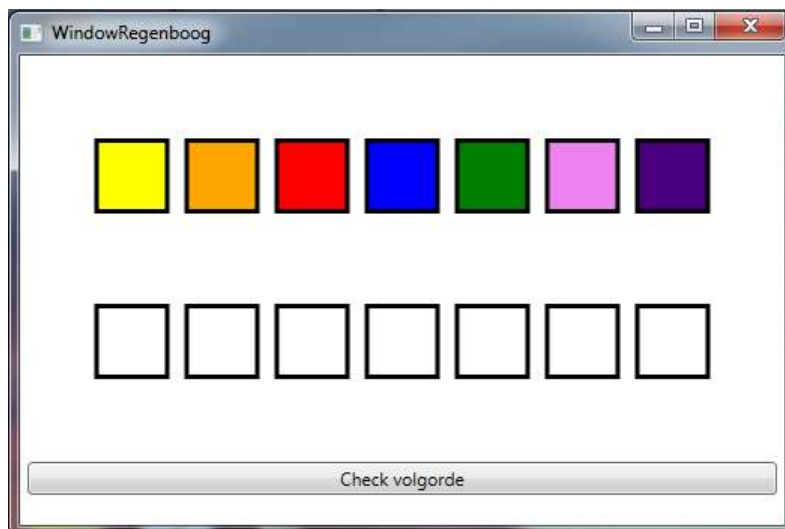
Eerst ga je een simpel voorbeeld uitwerken.

Hiervoor ga je een nieuw project maken: *RegenboogDragDrop*.

Verwijder de standaard gegeven XAML-pagina.

Kopieer van de oefenbestanden-map de bestanden *RegenboogWindow.xaml* en *RegenboogWindow.xaml.cs* naar het *Project* en verander in *App.xaml* de *StartupUri*.

Resultaat:



In de XAML-code zie je dat de bovenste reeks de *Fill*-property de 7 kleuren bevat, en dat in de *Name* van de onderste reeks deze kleuren gebruikt worden. De *Fill*-property van deze reeks staat op *White*. (je moet de rechthoek opvullen met een witte achtergrond omdat deze anders transparant is, en niet als *Drop-zone* kan worden gebruikt).

De bedoeling van deze applicatie is om de kleuren bovenaan te verslepen naar onder, en met de *Button* te checken of de volgorde van de kleuren dezelfde is als deze van een regenboog. Aan de hand van de *Name* en *Fill* kan je dit controleren.

11.1.1 Drag-fase

De kleur (*Rectangle*) die je wil gaan slepen, moet in de ***MouseMove-EventHandler*** testen of de linkermuisknop ingedrukt is, het *sleepobject* definiëren en de *Drag*-actie starten. Hiervoor moet je in de code-behind volgende code toevoegen onder de constructor van dit *Window*:

```

private Rectangle sleeprechthoek = new Rectangle(); (1)

private void Rectangle_MouseMove(object sender, MouseEventArgs e)
{
    sleeprechthoek = (Rectangle)sender; (2)
    if ((e.LeftButton == MouseButtonState.Pressed) && (3)
        (sleeprechthoek.Fill != Brushes.White)) (4)
    {
        DataObject sleepKleur = new DataObject("deKleur", (5)
            sleeprechthoek.Fill);
        DragDrop.DoDragDrop(sleeprechthoek, sleepKleur, (6)
            DragDropEffects.Move);
    }
}

```

- (1) je definieert een private variabele om te kunnen onthouden welke *Rectangle* je gaat slepen (en die juiste versleping wordt "uitgewit").
- (2) zet het object dat je sleept (=sender) in de private variabele
- (3) test of de linkermuisknop is ingedrukt
- (4) en test of de kleur al dan niet al naar onder is versleept (want dan zou je wit verslepen en dat is niet de bedoeling)
- (5) maak het *DataObject* aan dat bestaat uit een formaat dat je als string beschrijft = ("deKleur") en de gegevens zelf (=sleeprechthoek.Fill)
- (6) Start de DragDrop actie met als parameters het *DependencyObject* (=sleeprechthoek), het *DataObject* (=sleepKleur) en de muisvisualisatie (=DragDropEffects.Move) omdat je zagezegd de kleur gaat verplaatsen. *DependencyObject* is één van de WPF-objecten die als basis dient van vele andere *Objecten*.

Selecteer bij het *MouseMove*-event van elke *Rectangle* in de bovenste reeks de gemaakte procedure!

11.1.2 Drop-fase

Als je het *sleepobject* wil droppen, moet je er voor zorgen dat bij het *dropobject* de property **AllowDrop** op *True* staat (dit wil zeggen dat het toegestaan is om hier iets te droppen).

In het code-behind venster kan je tevens 3 events gaan uitschrijven die ervoor zorgen en dat bv. de kaderdikte van de *Rectangle* verandert als je over een *Object* sleept waar je kan droppen (**DragEnter**-event en **DragLeave**-event), en dat de *sleepkleur* wordt toegepast in het *dropobject* (**Drop**-event).

Voeg volgende procedures toe in het code-behind venster:

```

private void Rectangle_DragEnter(object sender, DragEventArgs e)
{
    Rectangle kader = (Rectangle)sender;
    kader.StrokeThickness = 5; (1)
}

private void Rectangle_DragLeave(object sender, DragEventArgs e)
{

```

```

    Rectangle kader = (Rectangle)sender;
    kader.StrokeThickness = 3;
  }
  
```

(2)

- (1) Als je over een kader komt, wordt de kaderrand dikker
- (2) Als je weg van de kader gaat, wordt de kaderrand terug normaal

```

private void Rectangle_Drop(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent("deKleur"))
    {
        Brush gesleepteKleur = (Brush)e.Data.GetData("deKleur");
        Rectangle rechthoek = (Rectangle)sender;
        if (rechthoek.Fill == Brushes.White)
        {
            rechthoek.Fill = gesleepteKleur;
            sleeprechthoek.Fill = Brushes.White;
        }
        rechthoek.StrokeThickness = 3;
    }
}
  
```

(1)
(2)
(3)
(4)
(5)
(6)
(7)

- (1) eerst test je of je het juiste *sleepobject* wil droppen
- (2) het *DataObject* dat je versleept hebt, moet terug omgezet worden naar een *Brush* om zijn kenmerken terug te krijgen, en wordt in een lokale variabele bijgehouden (=gesleepteKleur)
- (3) De *sender* is het object waar je het *sleepobject* wil droppen. Dit moet terug naar het specifiek object gecast worden om zijn kenmerken te kunnen gebruiken
- (4) Alleen als de droprechthoek nog een witte vulling heeft, mag de kleur verplaatst worden, anders gebeurt er niets.
- (5) De kleur wordt in de juiste rechthoek ingevuld
- (6) De oorspronkelijke rechthoek (van waarin je de sleep-actie begon) wordt "uitgewit".
- (7) De kaderrand waar eventueel gedropt wordt, wordt terug op de originele dikte gezet (ongeacht of je echt een kleur verplaatst hebt)

Selecteer bij het *DragEnter*-event, het *DragLeave*-event en het *Drop*-event van elke *Rectangle* in de onderste reeks de respectievelijk gemaakte procedures!

Probeer maar uit!

Wat momenteel nog niet gebeurt, is het terugzetten van kleuren die je verkeerd hebt verplaatst, of ze opschuiven als ze fout staan. Beperking is wel dat je ze niet kan plaatsen in een rechthoek waar al een kleur staat.

Hiervoor moet je geen extra code schrijven, maar wel de juiste properties en EventHandlers aanpassen:

- in de onderste reeks moet je de *MouseMove-EventHandler* koppelen aan de bestaande code van de *MouseMove*. Hierdoor mag je beginnen slepen vanuit de rechthoeken in de onderste reeks.

- in de bovenste reeks moet je de property *AllowDrop* aanzetten, en bij de *Events* de *DragEnter*-, *DragLeave* en de *Drop*-event koppelen aan de bestaande procedures. Hierdoor laat je toe de sleepkleur te droppen in de bovenste reeks.

Probeer maar uit!

11.1.3 Testen op kleurvolgorde

Om je applicatie te vervolledigen, moet je, bij het drukken op de *Button*, nog gaan testen of je een juist resultaat hebt.

Dit ga je doen door alle rechthoeken van de onderste reeks (verzameld in een *StackPanel*) te testen op een deel van hun naam (omgezet naar een *Brush*) en deze te vergelijken met de kleur van de vulling. Is deze hetzelfde dan wordt deze rechthoek groen omkaderd, anders rood.

```
private void ButtonCheck_Click(object sender, RoutedEventArgs e)
{
    foreach (Rectangle rechthoek in DropZone.Children)           (1)
    {
        string naam = rechthoek.Name.Substring(4);               (2)
        Brush naamkleur = (Brush)new BrushConverter().ConvertFromString(naam); (3)
        Brush kleur = rechthoek.Fill;                             (4)
        if (naamkleur == kleur)                                   (5)
        {
            rechthoek.Stroke = Brushes.Green;
        }
        else
        {
            rechthoek.Stroke = Brushes.Red;
        }
    }
}
```

- (1) De onderste rechthoeken staan allemaal binnen een *StackPanel* met de naam *DropZone*. Met de property *Children* kan je alle *Controls* binnen dit element aanspreken. Mits er alleen *Rectangles* in staan, kan je via de *foreach* elke rechthoek bevragen.
- (2) Je haalt de kleurnaam uit de benaming van de rechthoek
- (3) Je zet deze kleurnaam om naar een *Brush* met de *BrushConverter.ConvertFromString* method.
- (4) Je haalt de *Brush* uit de *Fill*-property van de rechthoek
- (5) Bij vergelijking van beide *Brushes* ga je de rechthoek een rode of groene rand geven.

Probeer maar uit!

11.2 Een complex object

Hiervoor ga je een nieuw project maken met de naam `OpleidingenDragDrop`.

Verwijder de standaard gegeven XAML-pagina.

Kopieer van de oefenbestanden-map:

- het bestand `OpleidingenWindow.xaml`, `OpleidingenWindow.xaml.cs` en `OfficeProgramma.cs` naar het Project en verander in `App.xaml` de `StartupUri`.
- alle afbeeldingen uit de `Office`-submap naar een (nieuw gemaakte) `images`-map in het project.

Resultaat:



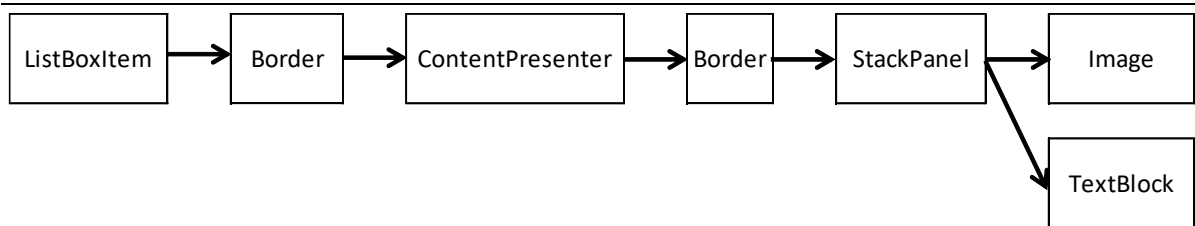
In dit geval ga je met *ListBoxen* te werk. Een element in de *Listbox* is niet zomaar een *String* die je toont, maar bestaat uit een *Image* en een *TextBlock* met hierrond een *Border*. Dus elk *Item* dat je wil verslepen, vertrekt vanuit de *Image*, *TextBlock* of *Border*. Om dit element volwaardig in een andere *ListBox* te zetten, moet je dus uitzoeken van welk *ListItem* het een onderdeel is.

De XAML-code van de structuur van een *ListBoxItem* in de uit te kiezen lijst:

```
<ListBox.ItemTemplate>
  <DataTemplate>
    <Border BorderBrush="Black" BorderThickness="1" Width="70">
      <StackPanel>
        <Image Source="{Binding Path=symbool}"
              Stretch="Fill" Height="50" Width="50"></Image>
        <TextBlock VerticalAlignment="Center"
                  HorizontalAlignment="Center"
                  Text="{Binding Path=naam}"></TextBlock>
      </StackPanel>
    </Border>
  </DataTemplate>
</ListBox.ItemTemplate>
```

De *VisualTree* (= de structuur van de visuele nesting van controls) van je *ListBoxItem* kan je grotendeels afleiden van de structuur in je XAML-code (er zijn nog een paar elementen aanwezig die je niet echt ziet):

Hij ziet er als volgt uit:



Om te weten van welke *ListBoxItem* je een onderdeel versleept, moet je in het code-behind venster hiervoor een *Functie* schrijven binnen de *Window* klasse:

```

private ListBoxItem VindListBoxItem(Object sleepitem) (1)
{
    DependencyObject keuze = (DependencyObject)sleepitem; (2)
    while (keuze != null) (3)
    {
        if (keuze is ListBoxItem) (4)
            return (ListBoxItem)keuze;
        keuze = VisualTreeHelper.GetParent(keuze); (5)
    }
    return null; (6)
}
  
```

- (1) Je maakt een functie die wat je sleept binnenkrijgt, en als returnwaarde de ganse *ListBoxItem* teruggeeft waartoe het behoort.
- (2) Je cast het sleepitem naar een *DependencyObject* (dit is een basisklasse binnen de WPF-klassestructuur) omdat de *VisualTreeHelper* dit als parameter nodig heeft.
- (3) Zolang er een hogerliggend object is, kan je zijn *Parent* vragen
- (4) Als de keuze van het type *ListBoxItem* is, geef je deze *ListBoxItem* als returnwaarde terug
- (5) Als de keuze een foutief type is, vraag je zijn *Parent* met de *VisualTreeHelper.GetParent* functie.
- (6) Als je geen *ListBoxItem* bent tegengekomen, geef je uiteindelijk een *null* als returnwaarde terug.

Nu al het voorbereidend werk is gebeurd, ga je terug de normale manier van drag-and-drop uitwerken:

DRAG:

In de *MouseMove-EventHandler* van alle *ListBoxen* van waaruit je wil verslepen (*ListBoxProgrammas*, *ListBoxGekend* en *ListBoxTeVolgen*) moet hetzelfde gebeuren. Je gaat daarvoor een algemene procedure maken, zodat de code maar éénmaal aanwezig is:

```

ListBox draglijst; (1)

private void DragListBox_MouseMove(object sender, MouseEventArgs e)
{
    if (e.LeftButton == MouseButtonState.Pressed) (2)
    {
  
```



```

    draglijst = (ListBox)sender;           (3)
    ListBoxItem programmaitem = VindListBoxItem(e.OriginalSource); (4)
    if (draglijst.SelectedIndex >= 0 && programmaitem != null) (5)
    {
        DataObject sleepdata =
            new DataObject("mijnprogramma", programmaitem.Content); (6)
        DragDrop.DoDragDrop(programmaitem, sleepdata,
            DragDropEffects.Move); (7)
    }
  }
}

```

- (1) je definieert een globale variabele om bij te kunnen houden vanuit welke *ListBox* je begint te slepen
- (2) de linkermuisknop moet ingehouden zijn
- (3) je cast de sender (die als object binnenkomt) naar de *draglist*.
- (4) je gebruikt de *e.OriginalSource* (die het object bevat dat je sleept) als parameter om de bijhorende *ListBoxItem* te vinden
- (5) als je vanuit een geselecteerd *ListBoxItem* begint te slepen (dus bv. niet vanuit een leeg gedeelte binnen de *ListBox*) dan ga je de *Drag*-operatie uitvoeren
- (6) je maakt een *DataObject* aan met als naam "mijnprogramma" en als inhoud de inhoud van de *ListBoxItem* (wat eigenlijk een *Object* is van het type *OfficeProgramma*).
- (7) Start de *DragDrop*-actie.

Selecteer bij het *MouseMove*-event van elke *ListBox* de gemaakte procedure!

DROP :

Zet bij alle *ListBoxen* die als drop-zone kunnen gebruikt worden, de property ***AllowDrop*** op *True*.

Je gaat nu niets bijzonders doen als je met de muis over een *ListBox* sleept, dus hoeven de *DragEnter*- en de *DragLeave*-events niet gedefinieerd te worden. Alleen de *Drop*-event gaat de effectieve drop doen. Omdat je in meerdere *ListBoxen* het dragobject kan droppen, ga je een algemene procedure voorzien om de code maar éénmaal te moeten schrijven:

```

private void DropListBox_Drop(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent("mijnprogramma"))
    {
        OfficeProgramma sleepprogramma =
            (OfficeProgramma)e.Data.GetData("mijnprogramma"); (1)
        ListBox droplijst = (ListBox)sender; (2)
        if (draglijst != droplijst) (3)
        {
            droplijst.Items.Add(sleepprogramma); (4)
            draglijst.Items.Remove(draglijst.SelectedItem); (5)
        }
    }
}

```

- (1) het *DataObject* dat je versleept hebt, moet terug omgezet worden naar een *OfficeProgramma* om zijn kenmerken terug te krijgen, en wordt in een lokale variabele bijgehouden (=sleepprogramma)
- (2) De *sender* is het object waar je het *sleepobject* wil droppen. Dit moet terug naar het specifiek object gecast worden om zijn kenmerken te kunnen gebruiken
- (3) Alleen als de *dropListBox* verschillend is van de *dragListBox* is het nodig om het *OfficeProgramma* te verplaatsen (=in de *dropListBox* toe te voegen en te wissen in de *dragListBox*).
- (4) Voeg het *OfficeProgramma* toe aan de *dropListBox*
- (5) Verwijder het *OfficeProgramma* in de *dragListBox*

Selecteer bij het *Drop*-event van elke *ListBox* de respectievelijke gemaakte procedure!

Probeer maar uit!

Om het programma af te werken, ga je de *Button* zijn ding nog laten doen: je gaat in een *MessageBox* een overzicht geven van alle *ListBoxen* met hun *OfficeProgramma's*.

```
private void ButtonDoorgeven_Click(object sender, RoutedEventArgs e)
{
    String tekst = "Volgende programma's zijn:\n";
    if (ListBoxProgrammas.Items.Count > 0)
    {
        tekst += "\nNiet toegewezen: ";
        foreach (OfficeProgramma prog in ListBoxProgrammas.Items)
        {
            tekst += prog.naam + ", ";
        }
        tekst = tekst.Substring(0, tekst.Length - 2);
    }

    if (ListBoxGekend.Items.Count > 0)
    {
        tekst += "\nGekend: ";
        foreach (OfficeProgramma prog in ListBoxGekend.Items)
        {
            tekst += prog.naam + ", ";
        }
        tekst = tekst.Substring(0, tekst.Length - 2);
    }

    if (ListBoxTeVolgen.Items.Count > 0)
    {
        tekst += "\nTe volgen: ";
        foreach (OfficeProgramma prog in ListBoxTeVolgen.Items)
        {
            tekst += prog.naam + ", ";
        }
        tekst = tekst.Substring(0, tekst.Length - 2);
    }
}
```

```
}  
  
    MessageBox.Show(tekst, "Overzicht");  
}
```

 (5)

- (1) Je definieert een *String* met als begininhoud: "Volgende programma's zijn:\n" \n = nieuwe regel beginnen
- (2) Als er een *Item* in de *ListBox* aanwezig is, kan je beginnen met de opsomming
- (3) Elk *Item* in de *ListBox* ga je omzetten naar zijn objecttype nl. *OfficeProgramma* zodat je de properties hiervan kan gebruiken zoals zijn naam. Hierachter wordt nog een komma met spatie toegevoegd
- (4) Na de opsomming staat in de *tekst-string* als laatste 2 karakters een komma en een spatie, dus om het proper af te werken, worden deze verwijderd.
- (5) Nadat de 3 *ListBoxen* zijn overlopen, wordt de inhoud van de *tekst-string* via een *MessageBox* aan de gebruiker getoond.

Probeer maar uit!



Opdracht: Schuifspel

12 RIBBON

Vanaf .NET Framework 4.5 is er een nieuwe control ter beschikking namelijk de *Ribbon*. De *Ribbon* is de opvolger van alle Menu-Toolbar-Strips enz. die je tot hiertoe hebt gebruikt. Alles zit in één *Control* vervat.

Omdat dit zo'n uitgebreide *Control* is, ga je via een nieuw project dit systematisch uitbouwen met vooral de mogelijkheden van deze *Ribbon* in gedachte en niet zozeer de functionaliteit van de applicatie zelf. (Dus sommige knoppen worden niet tot in de detail uitgewerkt.)

Hiervoor ga je een nieuw project maken:
`WindowMetRibbonControl`.

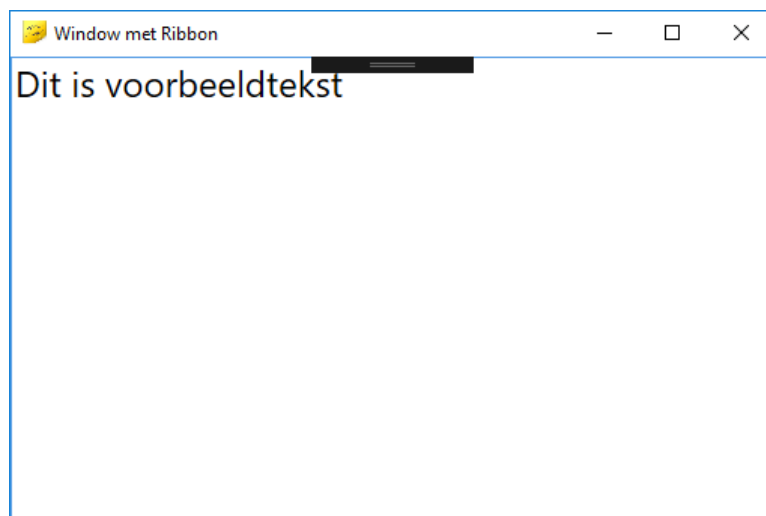
Verwijder de standaard gegeven XAML-pagina.

Kopieer van de oefenbestanden-map het bestand `WindowMetRibbon.xaml` en `WindowMetRibbon.xaml.cs` naar het Project en verander in `App.xaml` de `StartupUri`.

Voeg de Reference `System.Windows.Controls.Ribbon` toe.

Maak een map "images" en kopieer alle afbeeldingen voor de project naar deze map.

Resultaat als je de applicatie uitvoert:



Als je het `WindowMetRibbon.xaml` bestand opent, ga je merken dat

- er al *CommandBindings* voor dit `WindowMetRibbon` gedefinieerd zijn (die vroeger in de cursus aan bod zijn gekomen)
- er een *DockPanel* gedefinieerd is met momenteel als enige onderdeel een *TextBox* en zijn attributen.

In de code-behind zijn de *Command* op een simplistische manier uitgewerkt.

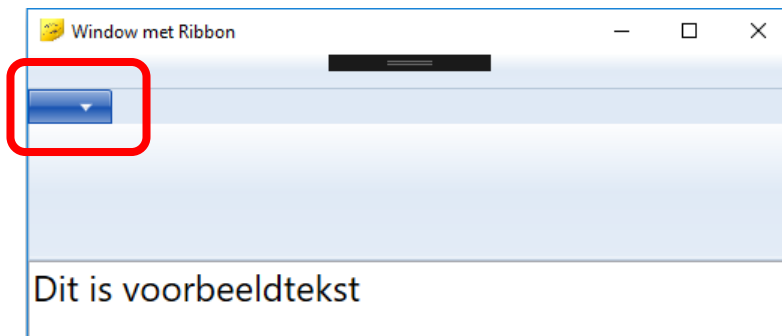
Voeg een *Ribbon* control toe boven de *TextBox* in het *DockPanel* en plaats hem bovenaan het *DockPanel*.

```
<DockPanel LastChildFill="True">
  <Ribbon DockPanel.Dock="Top">
```

```
</Ribbon>
<TextBox . . .
```

Nu zie je het uiterlijk van de *Ribbon* al verschijnen. De opvulling van zijn mogelijkheden worden per onderdeel toegevoegd.

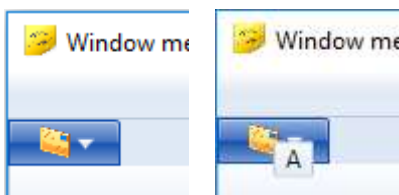
12.1 ApplicationMenu



De *ApplicationMenu* is de meest linkse *Tab* van de *Ribbon*. De functionaliteit is bedoeld voor de *Applicatie* en niet voor iets specifiek. Dit is een soort afspraak, maar in principe kan je daar om het even wat in kwijt.

Voeg twee tags toe nl. `Ribbon.ApplicationMenu` (zeggen dat dit onderdeel volgt) en `RibbonApplicationMenu` (het effectieve onderdeel) .

```
<Ribbon DockPanel.Dock="Top">
  <Ribbon.ApplicationMenu>
    <RibbonApplicationMenu SmallImageSource="images\bestand64.png" KeyTip="A">
    </RibbonApplicationMenu>
  </Ribbon.ApplicationMenu>
</Ribbon>
```



Gebruikte attributen:

- *SmallImageSource* : zorgt ervoor dat je een icon naast de pijl te zien krijgt
- *KeyTip* : om een *Shortcut* te maken : als je nu de *Alt* toets gebruikt, kan je met de letter *A* de *ApplicationMenu* activeren.

Het invullen van de *Menu* zelf kan op verschillende manieren gebeuren. Volgende *tags* kunnen als *Item* gebruikt worden:

12.1.1 RibbonApplicationMenuItem

Dit *Item* geeft een *MenuItem* weer in de *ApplicationMenu*.

Mogelijke attributen:

- *Header* : geeft de tekst weer naast de *ImageSource*
- *ImageSource* : zorgt ervoor dat je een icon te zien krijgt
- *Command* : door dit attribuut te gebruiken, wordt de *CommandBinding* actief met zijn bijhorende code (die al in het project aanwezig is).

12.1.2 RibbonSeparator:

Dit *Item* heeft een horizontale streep tussen de *MenuItems*.

12.1.3 RibbonApplicationSplitMenuItem

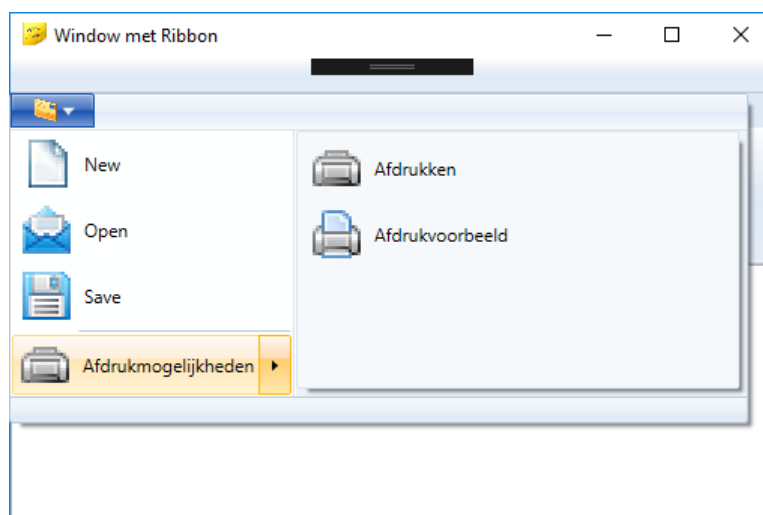
Dit *Item* geeft aan dat er nog een onderverdeling volgt met zijn onderliggende *MenuItems*.

Mogelijke attributen:

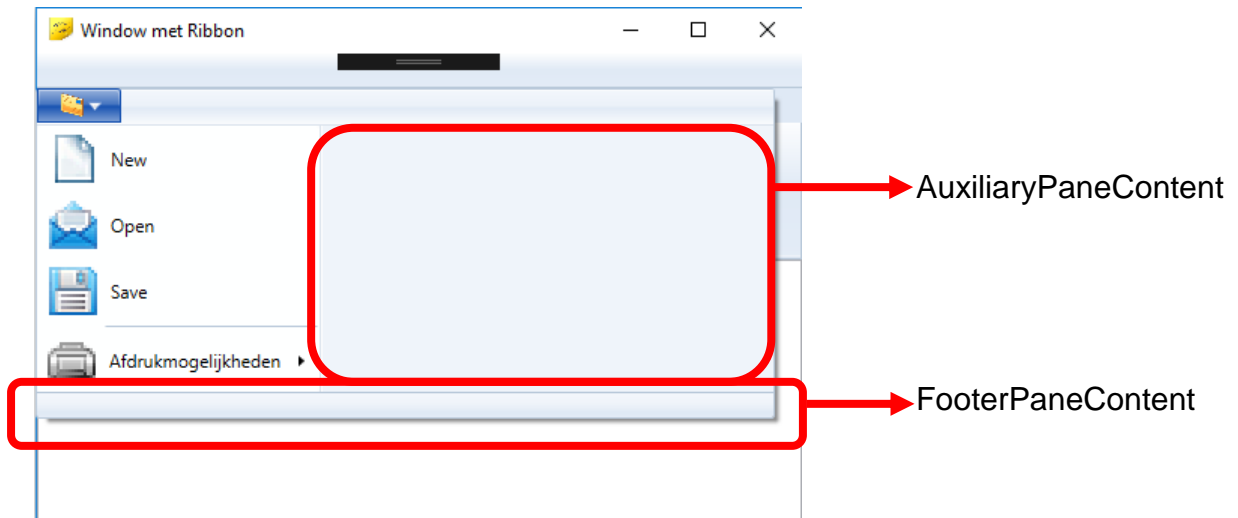
- *Header* : geeft de tekst weer naast de *ImageSource*
- *ImageSource* : zorgt ervoor dat je een icon te zien krijgt

Voeg volgende tags toe binnen de *RibbonApplicationMenu* en probeer de applicatie maar uit:

```
<RibbonApplicationMenu SmallImageSource="images\bestand64.png" KeyTip="A">
  <RibbonApplicationMenuItem Header="New" Command="New"
    ImageSource="images\new16.png"></RibbonApplicationMenuItem>
  <RibbonApplicationMenuItem Header="Open" Command="Open"
    ImageSource="images\open16.png"></RibbonApplicationMenuItem>
  <RibbonApplicationMenuItem Header="Save" Command="Save"
    ImageSource="images\save16.png"></RibbonApplicationMenuItem>
  <RibbonSeparator></RibbonSeparator>
  <RibbonApplicationSplitMenuItem Header="Afdrukmogelijkheden"
    ImageSource="images\printer64.png">
    <RibbonApplicationMenuItem Header="Afdrukken" Command="Print"
      ImageSource="images\printer64.png"></RibbonApplicationMenuItem>
    <RibbonApplicationMenuItem Header="Afdrukvoorbeeld" Command="PrintPreview"
      ImageSource="images\preview64.png"></RibbonApplicationMenuItem>
  </RibbonApplicationSplitMenuItem>
</RibbonApplicationMenu>
```

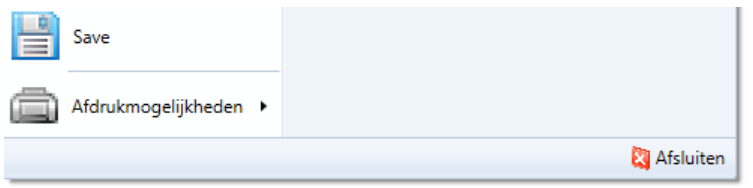


Er zijn binnen de *RibbonApplicationMenu* nog twee gebieden die je apart kan aanspreken namelijk *FooterPaneContent* en *AuxiliaryPaneContent*.



12.1.4 FooterPaneContent

Dit is een balk aan de onderkant van de *RibbonApplicationMenu*, vaak gebruikt om "Opties" of "Sluiten" als *Button* te tonen.



Voeg volgende code toe aan de XAML-code:

```

. . .
</RibbonApplicationSplitMenuItem>
<RibbonApplicationMenu.FooterPaneContent>
    <RibbonButton Command="Close" SmallImageSource="images\close64.png"
        HorizontalAlignment="Right" Label="Afsluiten"></RibbonButton>
</RibbonApplicationMenu.FooterPaneContent>
</RibbonApplicationMenu>
. . .

```

Mogelijke attributen:

- *Command* : door dit attribuut te gebruiken, wordt de *CommandBinding* actief met zijn bijhorende code (die al in het project aanwezig is).
- *SmallImageSource*: zorgt ervoor dat je een icon te zien krijgt
- *Label* : geeft de tekst weer naast de *SmallImageSource*
- *HorizontalAlignment*: *Right* = *Button* wordt aan de rechterkant gezet

12.1.5 AuxiliaryPaneContent

Bij het openklappen van de *RibbonApplicationMenu* krijg je aan de rechterkant een heel gebied dat als *RibbonGallery* gebruikt kan worden. Deze *Gallery* kan je, indien nodig, verder onderverdelen in *RibbonGalleryCategory* 's. Vaak wordt dit gebied

opgevuld met de meest recente gebruikte (MRU=most recently used) bestanden, maar in principe kan je dit ook voor andere dingen gebruiken.

Voeg volgende code toe aan de XAML-code achter de FooterPaneContent-tag:

```

. . .
<RibbonApplicationMenu.AuxiliaryPaneContent>
    <RibbonGallery Name="MRUGallery" CanUserFilter="False"
        ScrollViewer.VerticalScrollBarVisibility="Auto"
        SelectionChanged="MRUGallery_SelectionChanged">
        <RibbonGalleryCategory Name="MRUGalleryCat" Header="Recent Documents">
            </RibbonGalleryCategory>
        </RibbonGallery>
    </RibbonApplicationMenu.AuxiliaryPaneContent>
. . .

```

Mogelijke attributen RibbonGallery:

- *Name* : is nodig als je in code naar deze tag wil refereren.
- *CanUserFilter* : bepaalt of de gebruiker kan filteren of niet
- *ScrollViewer.VerticalScrollBarVisibility* : als alles vertikaal gezien niet getoond kan worden, verschijnen er *Automatisch scrollbars*.

Gebruikt event:

- *SelectionChanged* : als de gebruiker een *Item* aanduidt, kan er tot actie worden overgegaan (via code in de bijhorende procedure).

Voeg volgende code toe in het code-behind venster:

```

private void MRUGallery_SelectionChanged(object sender,
RoutedPropertyChangedEventArgs<object> e)
{
    LeesBestand(MRUGallery.SelectedValue.ToString());
}

```

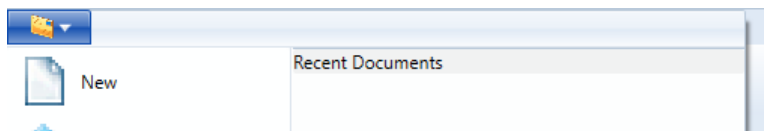
(1)

- (1) Open het bestand dat de gebruiker gekozen heeft nl. de *SelecteValue* van de *MRUGallery* (die op zich een *Object* is), dus met een *ToString()* erachter.

Mogelijke attributen RibbonGalleryCategory:

- *Name* : is nodig als je in code naar deze tag wil refereren.
- *Header* : toont een titel boven de opgesomde items

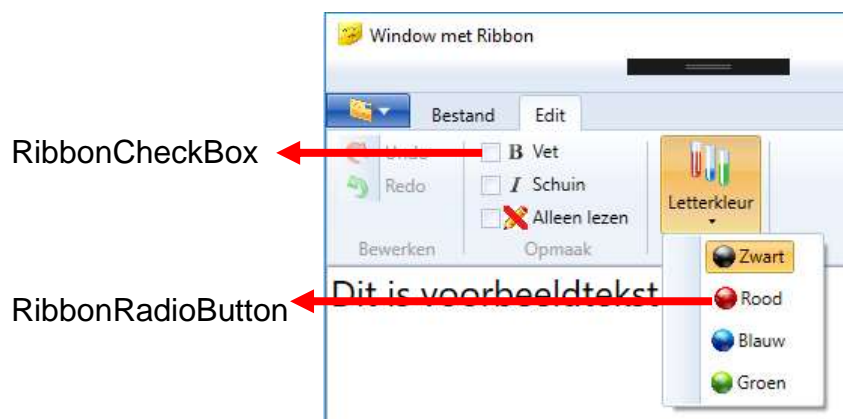
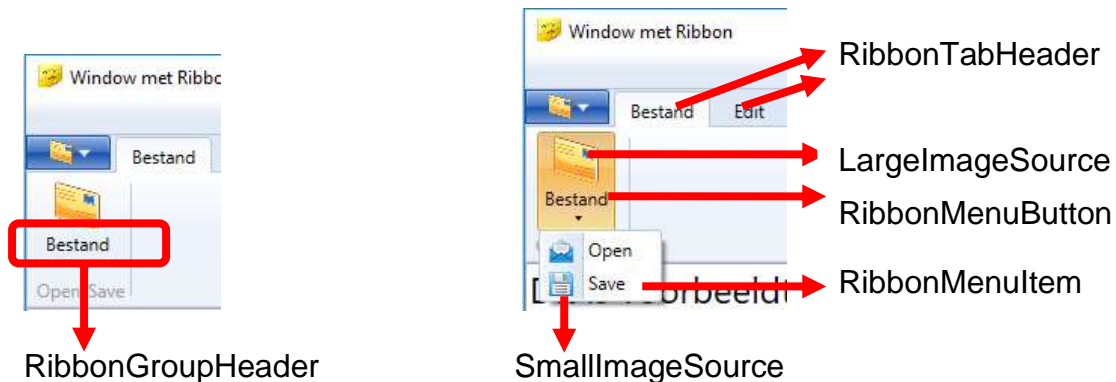
Je kan dit al uitproberen, maar je zal natuurlijk geen bestanden zien staan onder de titel.



Om dit uit te werken komen er nog verschillende nieuwe items aan bod, dus we laten dit even zoals het is omdat de uitwerking eigenlijk niets met de *Ribbon* zelf te maken heeft. Als laatste punt in dit hoofdstuk wordt hier verder op ingegaan.

12.2 RibbonTab

De *RibbonTab* is visueel het grootste onderdeel van de *Ribbon*. Je kan dit onderdeel best vergelijken met de klassieke werkbalk. Elke *RibbonTab* bestaat uit één of meerdere *RibbonGroups*. Elke *RibbonGroup* wordt afgebakend door een verticale streep.



Het voorbeeld hierboven ga je implementeren. Er zijn 2 tabs met hun verschillende onderdelen en mogelijkheden.

Voeg onderstaande code toe onder de eindtag van `Ribbon.ApplicationMenu`

```
<RibbonTab Header="Bestand" KeyTip="B">
  <RibbonGroup Header="Open/Save">
    <RibbonMenuItem LargeImageSource="images\bestand64.png" Label="Bestand"
      KeyTip="B">
      <RibbonMenuItem Command="Open" ImageSource="images\open16.png"
        Header="Open" KeyTip="O"></RibbonMenuItem>
      <RibbonMenuItem Command="Save" ImageSource="images\save16.png"
        Header="Save" KeyTip="S"></RibbonMenuItem>
    </RibbonMenuItem>
  </RibbonGroup>
</RibbonTab>

<RibbonTab Header="Edit" KeyTip="E">
  <RibbonGroup Header="Bewerken">
```

```
<RibbonMenuItem Command="Undo" ImageSource="images\undo16.png"
    Header="Undo"></RibbonMenuItem>
<RibbonMenuItem Command="Redo" ImageSource="images\redo16.png"
    Header="Redo"></RibbonMenuItem>
</RibbonGroup>

<RibbonGroup Header="Opmaak">
    <RibbonCheckBox SmallImageSource="images\bold.png" Label="Vet">
    </RibbonCheckBox>
    <RibbonCheckBox SmallImageSource="images\italic.png" Label="Schuin">
    </RibbonCheckBox>
    <RibbonCheckBox Name="CheckBoxAlleenLezen"
        SmallImageSource="images\ReadOnly16.png" Label="Alleen lezen">
    </RibbonCheckBox>
</RibbonGroup>

<RibbonGroup Header="Kleur">
    <RibbonMenuButton Name="MenuKleur" LargeImageSource="images\kleur64.png"
        Label="Letterkleur">
        <RibbonRadioButton SmallImageSource="images\black48.png" Label="Zwart"
            IsChecked="True" Tag="#FF000000"></RibbonRadioButton>
        <RibbonRadioButton SmallImageSource="images\red48.png" Label="Rood"
            Tag="#FFFFFF0000"></RibbonRadioButton>
        <RibbonRadioButton SmallImageSource="images\blue48.png" Label="Blauw"
            Tag="#FF0000FF"></RibbonRadioButton>
        <RibbonRadioButton SmallImageSource="images\green48.png" Label="Groen"
            Tag="#FF00FF00"></RibbonRadioButton>
    </RibbonMenuButton>
</RibbonGroup>
</RibbonTab>
```

Mogelijke attributen van de verschillende Ribbon onderdelen:

- *Name* : is nodig als je in code naar deze tag wil refereren.
- *Header* - *Label* : toont de bijhorende tekst
- *SmallImageSource* - *LargeImageSource* - *ImageSource* : toont de image
- *Command* : door dit attribuut te gebruiken, wordt de *CommandBinding* actief met zijn bijhorende code (die al in het project aanwezig is).
- *KeyTip* : door een letter te speciëren wordt dit onderdeel van de *Ribbon* in combinatie met de *ALT* toets geactiveerd.
- *Tag* : bevat informatie die handig is om via code te gebruiken. In dit geval is het de hexadecimale weergave van de desbetreffende kleur.

Bestand

Deze *RibbonTab* is helemaal geïmplementeerd via de *CommandBindings* en heeft geen extra attributen of code nodig.

Edit

Deze *RibbonTab* bestaat uit 3 *RibbonGroups* met verschillende implementatie mogelijkheden:

- Bewerken : wordt volledig via *CommandBindings* geïmplementeerd
- Opmaak :
De *Bold*- en *Italic-CheckBox* hebben een converter nodig om de *IsChecked* om te

zetten naar hun specifieke waarde. Je kan deze via *Binding* met elkaar koppelen, maar dit ga je later uitwerken in het onderwerp “*Converters*”.

De *ReadOnly-CheckBox* kan wel rechtstreeks gekoppeld worden via *Binding* aan de *IsReadOnly-attribuut* van de *TextBox*:

je voegt in de *TextBox-tag* volgend attribuut toe :

```
IsReadOnly="{Binding ElementName=CheckBoxAlleenLezen,  
Path=IsChecked}"
```

- Kleur :

Bij het kiezen van de tekstkleur zijn er voorzorgsmaatregelen genomen: in de *Tag-attribuut* staat de hexadecimale waarde die de overeenkomstige kleur voorstelt.

Als je een kleur kiest (via het *Click-event*) dan kan voor elke *RadioButton* dezelfde code gebruikt worden.

je voegt in de *RadioButtons* van de kleuren een *Click-event* toe :

```
Click="Radio_Click"
```

en in de code-behind :

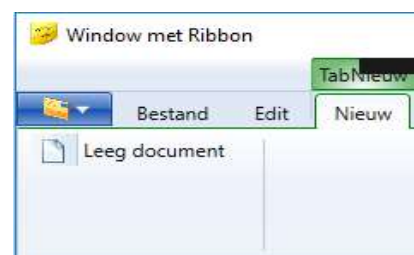
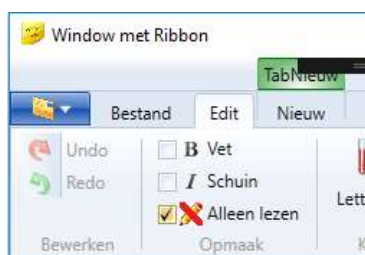
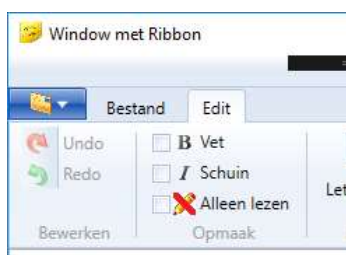
```
private void Radio_Click(object sender, RoutedEventArgs e)
{
    RibbonRadioButton keuze = (RibbonRadioButton)sender;           (1)
    BrushConverter bc = new BrushConverter();                         (2)
    SolidColorBrush kleur =                                           (3)
        (SolidColorBrush)bc.ConvertFromString(keuze.Tag.ToString());
    TextBoxVoorbeeld.Foreground = kleur;                             (4)
}
```

- (1) Converteer de geklikte *Button (sender)* om naar zijn specifiek object: *RibbonRadioButton*.
- (2) Definieer een *BrushConverter*.
- (3) Zet de hexadecimale waarde van de kleur in het *Tag-attribuut* om naar een *SolidColorBrush*.
- (4) Verzet de tekstkleur naar deze gemaakte *Brush*.

12.3 ContextualRibbonTab

Om een *RibbonTab* alleen maar te visualiseren in een bepaalde *context* heb je een gewone *RibbonTab* nodig en een *RibbonContextualTabGroup* waarin gespecificeerd wordt wanneer de extra *RibbonTab* zichtbaar wordt.

Als voorbeeld ga je er voor zorgen dat bij het aanvinken van de “*Alleen-lezen*” mogelijkheid, de extra *RibbonTab* “*TabNieuw*” getoond wordt. Bij het uitvinken van deze mogelijkheid, wordt hij terug onzichtbaar. *TabNieuw* is een *RibbonTab* met als inhoud een *RibbonMenuItem* die een lege *TextBox* toont.



Omdat je het *Visibility-attribuut* van de *ContextualTabGroup* manipuleert om te tonen of te verbergen, kan je een *converter* gebruiken die bij VS aanwezig is, en via een *Resource* aanspreekbaar gemaakt wordt. Deze *converter* zet de logische waarde *True* en *False* om naar respectievelijk *Visible* en *Collapsed*.

Voeg bovenaan juist na de begintag van de *Window* volgende *Resource* toe:

```
<Window.Resources>
    <BooleanToVisibilityConverter x:Key="LogischNaarVisueel" />
</Window.Resources>
```

(1)

(1) De *Key* is de aanspreeksleutel binnen een attribuut.

Voeg onderaan na de laatste *RibbonTab* maar binnen de *Ribbon* zelf volgende tags toe:

```
<RibbonTab Header="Nieuw" ContextualTabGroupHeader="TabNieuw">
    <RibbonGroup>
```

(1)

```
        <RibbonMenuItem Command="New" ImageSource="images\New16.png"
            Header="Leeg document"></RibbonMenuItem>
    </RibbonGroup>
</RibbonTab>
<Ribbon.ContextualTabGroups>
```

(2)

```
    <RibbonContextualTabGroup Header="TabNieuw" Visibility="{Binding
        ElementName=TextBoxVoorbeeld, Path=IsReadOnly,
        Converter={StaticResource LogischNaarVisueel}}" Background="Green"/>
</Ribbon.ContextualTabGroups>
```

(3)

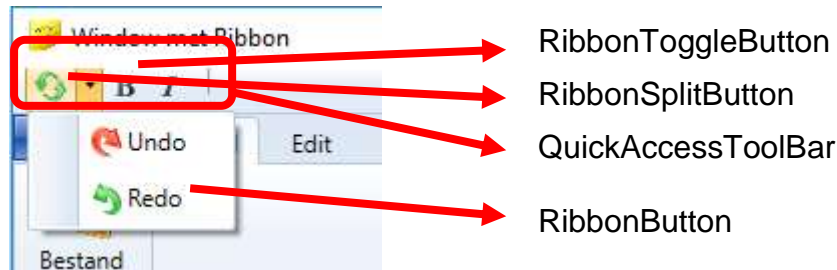
- (1) in de *RibbonTab* wordt in het attribuut *ContextualTabGroupHeader* de inhoud van het *Header* attribuut van de *RibbonContextualTabGroup* gezet
- (2) Definieer na alle *RibbonTabs* een onderdeel *ContextualTabGroups* van de *Ribbon* zelf
- (3) Definieer een *RibbonContextualTabGroup* met noodzakelijke attributen:
 - *Header* : is de verwijzing in de gewone *RibbonTab*
 - *Visibility* : heeft 2 waarden : *Visible* (zichtbaar) en *Collapsed* (onzichtbaar)
 Dit attribuut is via *Binding* gekoppeld aan de *IsReadOnly* –attribuut van de *TextBox*. Via de *converter* worden de waarden omgezet.
 - *Background* : is niet noodzakelijk, maar zo valt het een beetje op.

Je kan dit uitproberen

12.4 QuickAccessToolBar (QAT)

De *QuickAccessToolBar* komt juist onder de. De bedoelling van deze *bar* is een snelle toegang tot de meest gebruikte items binnen de applicatie geven.

Je kan er voor zorgen dat de gebruiker zelf knoppen kan bijvoegen of niet (zie verder).



Voeg volgende tags toe juist onder de begintag van de *Ribbon*:
Let op dat je de QAT al een "Name" geeft zodat deze later in de code-behind kan gemanipuleerd worden.

```
<Ribbon.QuickAccessToolBar>  
  <RibbonQuickAccessToolBar Height="24" Name="Qat">  
    <RibbonSplitButton SmallImageSource="Images\refresh16.png">  
      <RibbonButton SmallImageSource="Images\undo16.png" Command="Undo"  
        Label="Undo"></RibbonButton>  
      <RibbonButton SmallImageSource="Images\redo16.png" Command="Redo"  
        Label="Redo"></RibbonButton>  
    </RibbonSplitButton>  
  
    <RibbonToggleButton Name="ButtonVet" SmallImageSource="Images\bold.png">  
  </RibbonToggleButton>  
    <RibbonToggleButton Name="ButtonSchuin" SmallImageSource="Images\italic.png">  
  </RibbonToggleButton>  
  </RibbonQuickAccessToolBar>  
</Ribbon.QuickAccessToolBar>
```

De *Undo* en *Redo* zijn gewone *RibbonButtons* en worden via *CommandBindings* automatisch uitgevoerd; Zij zijn ondergegroepeerd via een *RibbonSplitButton*.

De *Vet* en *Schuin* knoppen zijn *RibbonToggleButton*s (zij blijven aan- of uitstaan) en worden later via *Binding* van hun functionaliteit voorzien.

12.5 QAT-items bijvoegen

Vanuit de *ApplicationMenu* en de *RibbonTabs* is het mogelijk om via de rechtermuisknop te klikken op een *Item* en te kiezen voor "Add to Quick Access Toolbar". Deze mogelijkheid gaat alleen actief staan als de functionaliteit is uitgewerkt via een *Command* en niet via een *Event* (zoals bv. de *Click*). Bij het clonen van het *Item* naar de QAT wordt de *Command* wel mee gecloned, maar de *Events* niet.

Om ervoor te zorgen dat je in de QAT ook een image te zien krijgt, moet bij het originele *Item* het attribuut *QuickAccessToolBarImageSource* ingevuld worden met de gewenste image.

Voeg het extra attribuut toe aan onderstaande tag:

```
<RibbonApplicationMenuItem Header="Afdrukvoorbeeld" Command="PrintPreview"
ImageSource="images\preview64.png"
QuickAccessToolBarImageSource="images\preview64.png">
</RibbonApplicationMenuItem>
```

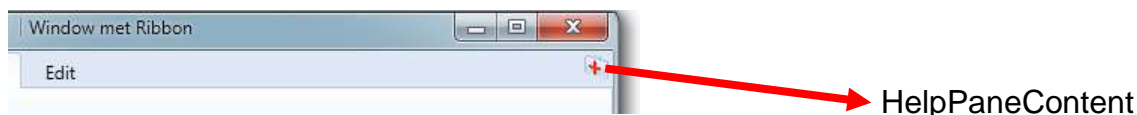
Voer de applicatie uit en kies met een klik met de rechtermuisknop op het afdrukvoorbeeld om dit toe te voegen aan de QAT.

Klik op de toegevoegde button, en je zal merken dat de functionaliteit ook aanwezig is.

Sluit de applicatie en start het opnieuw: de button is terug verdwenen. Deze wijziging wordt dus NIET automatisch ergens opgeslagen !

12.6 HelpPaneContent

Aan de rechterkant van de *RibbonTabs* kan je een *Help* gebied voorzien. Meestal wordt dit gedeelte opgevuld door een knop die dan de nodige hulp biedt.



Voeg volgende tags toe juist onder de begintag van de *Ribbon*:

```
<Ribbon.HelpPaneContent>
  <RibbonButton SmallImageSource="Images\help64.png"
    Command="Help"></RibbonButton>
</Ribbon.HelpPaneContent>
```

Door deze knop via *CommandBinding* aan de *Help-command* te koppelen, is de functionaliteit van de knop al aanwezig (zit in de bijgeleverde code).

Je kan dit uitproberen.

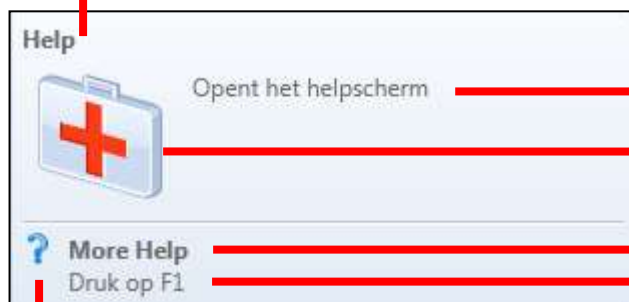
12.7 SuperToolTip

De *RibbonButton* heeft de mogelijkheid om een *SuperToolTip* weer te geven. Deze heeft meer mogelijkheden dan de standaard *ToolTip*.

Voeg bij de *RibbonButton* van de *HelpPaneContent* volgende attributen toe:

```
<RibbonButton SmallImageSource="Images\help64.png" Command="Help"
  ToolTipTitle="Help"
  ToolTipDescription="Opent het helpscherm"
  ToolTipFooterTitle="More Help"
  ToolTipFooterDescription="Druk op F1"
  ToolTipFooterImageSource="images\question16.png"
  ToolTipImageSource="images\help64.png">
</RibbonButton>
```

ToolTipTitle



ToolTipDescription

ToolTipImageSource

ToolTipFooterTitle

ToolTipFooterDescription

ToolTipFooterImageSource

13 CONVERTERS

Soms is het nodig om een bepaalde waarde om te zetten naar een andere waarde, of om het met andere woorden te zeggen het ene te converteren naar het andere.

In Visual Studio zijn er een paar *Converters* aanwezig zoals de reeds gebruikte *BooleanToVisibilityConverter* die je hebt gebruikt bij de *ContextualTabGroup*.

Indien de omzetting niet intern aanwezig is, wordt er via de *Interface IValueConverter* de mogelijkheid geboden om zelf een *Converter* te maken.

Je hebt in de applicatie een *Vet* en *Schuindruk* mogelijkheid ingebouwd waarvan de functionaliteit nog niet is uitgewerkt.

Je kan via een *RibbonToggleButton* of een *RibbonCheckBox* deze functie aan- en uitzetten via het attribuut *IsChecked (boolean)*, maar voor een *TextBox* is dit niet aan en uit, maar voor *Vet* is dit *Normal* en *Bold*, en voor *Schuin* is dit *Normal* en *Italic*.

Je gaat dus zelf een *Converter* klasse moeten coderen om van een *Boolean*-waarde, de resultaatwaarde te maken.

Ga naar het code-behind venster, ga juist voor de laatste accolade staan en tik volgende regel:

```
public class BooleanToFontWeight : IValueConverter
```

en kies bij *IValueConverter* (bij het uitklappen van het streepje onder de *I*) om de *Interface* te implementeren. Volgende code is het resultaat :

```
public class BooleanToFontWeight : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
        CultureInfo culture)
    {
        throw new NotImplementedException();
    }

    public object ConvertBack(object value, Type targetType, object parameter,
        CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

Je krijgt een *Convert* en een *ConvertBack* method als implementatie die je naar believen kan aanpassen. Het enige dat van belang is, is de *value* dus op de andere parameters ga je niet verder ingaan.

Verander de code in:

```
public class BooleanToFontWeight : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
        CultureInfo culture)
    {
        if ((Boolean)value)                                     (1)
            return "Bold";                                     (2)
    }
}
```



```

        else
            return "Normal";
    }

    public object ConvertBack(object value, Type targetType, object parameter,
CultureInfo culture)
    {
        return null;
    }
}

```

(3)

- (1) de *value* krijg je als *object* binnen, dus die moet je eerst converteren naar een *Boolean* waarde
- (2) als de waarde *true* is, dan wordt er als resultaat de string "Bold" teruggegeven, en anders "Normal"
- (3) de method *ConvertBack* wordt in de applicatie niet gebruikt, maar null als returnwaarde geven, is minder drastisch dan een *Exception* gooien.

Op dezelfde manier ga je een *Converter* voorzien om een *Boolean* om te zetten naar een *FontStyle* :

```

public class BooleanToFontStyle : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
CultureInfo culture)
    {
        if ((Boolean)value)
            return "Italic";
        else
            return "Normal";
    }

    public object ConvertBack(object value, Type targetType, object parameter,
CultureInfo culture)
    {
        return null;
    }
}

```

De rest moet in de XAML-code gebeuren:

Je definieert een lokale namespace in de begintag van de *Window*:

```
xmlns:local="clr-namespace:WindowMetRibbonControl"
```

Je definieert de *Converters* als *Resources* in de *Window.Resources*:

```

<Window.Resources>
    <BooleanToVisibilityConverter x:Key="LogischNaarVisueel" />
    <local:BooleanToFontWeight x:Key="LogischNaarFontDikte" />
    <local:BooleanToFontStyle x:Key="LogischNaarFontStijl" />
</Window.Resources>

```

Implementeren van *Vet*:

De *RibbonToggleButton* heeft de *Name* "ButtonVet" gekregen, en daar kan dus naar gerefereerd worden als het *IsChecked* attribuut verandert.

De andere plaats die altijd mee verandert, is in de *RibbonTab* “Edit” in de *RibbonGroup* “Opmaak”.

Je koppelt via *DataBinding* het *IsChecked*-attribuut van de *RibbonCheckBox* aan de bovenvermelde button:

```
<RibbonCheckBox SmallImageSource="images\bold.png" Label="Vet"
IsChecked="{Binding ElementName=ButtonVet, Path=IsChecked}"></RibbonCheckBox>
```

Je koppelt via *DataBinding* de *FontWeight* van de *TextBox* aan het *IsChecked*-attribuut van de *RibbonCheckBox* gebruikmakend van de zelfgecreëerde *Converter*:

```
<TextBox Name="TextBoxVoorbeeld"

. . .
    FontWeight="{Binding ElementName=ButtonVet, Path=IsChecked,
    Converter={StaticResource LogischNaarFontDikte}}}"
</TextBox>
```

Implementeren van *Schuin*:

```
<RibbonCheckBox SmallImageSource="images\italic.png" Label="Schuin"
IsChecked="{Binding ElementName=ButtonSchuin, Path=IsChecked}"></RibbonCheckBox>
```

```
<TextBox Name="TextBoxVoorbeeld"

. . .
    FontStyle="{Binding ElementName=ButtonSchuin, Path=IsChecked,
    Converter={StaticResource LogischNaarFontStijl}}}"
</TextBox>
```

Je kan dit uitproberen.

14 USER EN APPLICATION SETTINGS

Soms is het handig om instellingen of data van de applicatie op te slaan zodat je bijvoorbeeld bij het starten van de applicatie deze items kan reproduceren.

Je kan dit op 2 niveaus bijhouden namelijk per gebruiker (user setting) of voor de applicatie zelf (application setting).

Je kan gegevens bijhouden van verschillende types gaande van een getal, een string, een *color* tot een collection van strings.

Je definieert deze settings in de *Properties* van het *Project*.

Klik in de *Solution Explorer* met de rechtermuisknop op de projectnaam en kies *Properties*.

Kies daarna eventueel aan de linkerkant nog voor *Settings*.

In je voorbeeldapplicatie zijn er momenteel nog twee dingen aanwezig die je nog niet hebt uitgewerkt:

- het bijhouden van toegevoegde QAT-items

Maak een *Setting*:

- Name : qat
- Type : System.Collections.Specialized.StringCollection
- Scope : User

- het bijhouden van de MRU-lijst in de AuxiliaryPaneContent

Maak een *Setting*:

- Name : mru
- Type : System.Collections.Specialized.StringCollection
- Scope : User

Beide lijken het beste per gebruiker bij te houden zodat de QAT en de laatste geopende bestanden van die gebruiker zelf zijn.

Eens ze zijn gedefinieerd, kan je de settings in code aanspreken met *NameSpace.Properties.Settings.Default.naamSetting* en wordt dit in het *Project* in de map *Properties* met de naam *Settings.settings* opgeslagen.

14.1.1 Opslaan van toegevoegde QAT-items

Je zal gemerkt hebben dat je gemakkelijk (via de rechtermuisknop) items aan de QAT kan toevoegen. Minder leuk is het feit dat als je de applicatie sluit en daarna terug opent, dat deze toegevoegde *Buttons* terug verdwenen zijn.

Om ze te bewaren en bij het opstarten terug aan te maken, heb je van deze *Buttons* volgende attributen nodig: de image en het command.

Dit zijn dus twee strings die je moet onthouden en bijgevolg gaat stockeren in een collection van strings die door de settings gekend is namelijk in een *System.Collections.Specialized.StringCollection*.

Je gaat op 2 plaatsen code moeten voorzien:

- bij de *Closing* : om de toegevoegde items op te slaan
- bij de constructor : om de opgeslagen items te reproduceren.

Opslaan

Definieer voor de *Window* een *Closing*-event met volgende code:

```
private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
    System.Collections.Specialized.StringCollection qatlijst =
        new System.Collections.Specialized.StringCollection();           (1)
    if (WindowMetRibbonControl.Properties.Settings.Default.qat != null)    (2)
        WindowMetRibbonControl.Properties.Settings.Default.qat.Clear();  (3)
    foreach (Object li in Qat.Items)                                       (4)
    {
        if (li is RibbonButton)                                           (5)
        {
            RibbonButton knop = (RibbonButton)li;                       (6)
            RoutedUICommand commando = (RoutedUICommand)knop.Command;    (7)
            qatlijst.Add(commando.Text);                                   (8)
            qatlijst.Add(knop.SmallImageSource.ToString());               (9)
        }
    }
    if (qatlijst.Count > 0)                                               (10)
    {
        WindowMetRibbonControl.Properties.Settings.Default.qat = qatlijst; (11)
    }
    WindowMetRibbonControl.Properties.Settings.Default.Save();           (12)
}
```

- (1) definieer een nieuw lokale variable "qatlijst"
- (2) test of de setting "qat" al bestaat
- (3) maak ze leeg
- (4) overloop alle items van de QAT
- (5) als het type item gelijk is aan een *RibbonButton* dan is het een bijgevoegde item (de reeds bestaande zijn een *RibbonSplitButton* en *RibbonToggleButton*.)
- (6) Converteer het item naar een *RibbonButton*
- (7) Converteer de *Command* van het item naar een *RoutedUICommand*
- (8) Voeg het *Command* in tekstvorm toe aan de "qatlijst"
- (9) Voeg de image in tekstvorm toe aan de "qatlijst"
- (10) test of er uiteindelijk iets bij de houden is
- (11) zet de aangemaakte collection "qatlijst" in de setting "qat"
- (12) bewaar de user- en applicatiesettings. (dit vooral NIET VERGETEN)

Inlezen

Voeg in de constructor na de "InitializeComponent();" volgende code toe:

```
public WindowMetRibbon()
{
    InitializeComponent();

    if (WindowMetRibbonControl.Properties.Settings.Default.qat != null)    (1)
    {
        System.Collections.Specialized.StringCollection qatlijst =
            WindowMetRibbonControl.Properties.Settings.Default.qat;        (2)
        int lijnrr = 0;
        while (lijnrr < qatlijst.Count)                                     (3)
        {
            String commando = qatlijst[lijnrr];                             (4)
        }
    }
}
```

```

String png = qatlijst[lijnnr + 1]; (5)
RibbonButton nieuweKnop = new RibbonButton(); (6)
BitmapImage icon = new BitmapImage(); (7)
icon.BeginInit();
icon.UriSource = new Uri(png);
icon.EndInit();
nieuweKnop.SmallImageSource = icon;

CommandBindingCollection ccol = this.CommandBindings; (8)
foreach (CommandBinding cb in ccol)
{
    RoutedUICommand rcb = (RoutedUICommand)cb.Command;
    if (rcb.Text == commando) (9)
        nieuweKnop.Command = rcb;
}
Qat.Items.Add(nieuweKnop); (10)
lijnnr += 2;
}
}
}

```

- (1) test of de setting bestaat
- (2) lees de collection van de setting "qat" in de variable "qatlijst" in
- (3) zolang er nog een string te lezen is
- (4) lees de regel van de setting in de string "commando" in
- (5) lees de regel van de setting in de string "png" in
- (6) definieer een nieuwe *RibbonButton*
- (7) definieer een nieuwe *BitmapImage* en koppel deze aan de image
- (8) defineer een collection van de bestaande *CommandBindings* en doorloop ze om het juiste *command* te vinden
- (9) voeg de *command* aan de *RibbonButton* toe
- (10) voeg de *RibbonButton* aan de QAT toe

Het is tijd om de implementatie uit te testen !

14.1.2 Opslaan MRU-bestanden

Het bijhouden van de MRU-bestanden is het andere item dat nog niet geïmplementeerd is. Dit is een veel voorkomend item, maar niet in standaard code aanwezig in Visual Studio. Dus dat ga je zelf moeten beheren. Het aantal dat bijgehouden wordt, is beperkt, maar zeker meer dan één (in jouw geval ga je de laatste 6 bijhouden), dus je zal weer een collection van strings nodig hebben die door de settings gekend is namelijk in een *System.Collections.Specialized.StringCollection*. Je hebt hiervoor reeds een user setting aangemaakt namelijk "mru".

Je gaat op verschillende plaatsen code moeten voorzien:

- bij het starten van de applicatie : uitlezen van mrulijst
- bij de opslaan : *SaveExecuted* : via het Save-menu : veranderen van mrulijst (het opgeslagen bestand komt bovenaan als meest recent te staan)
- bij het openen : *LeesBestand* : via selectie in de *AuxiliaryPaneContent* of via het Open-menu: veranderen van mrulijst (geopend bestand komt bovenaan als meest recent te staan)

Je gaat dit oplossen in 2 verschillende procedures:

- het tonen van de mrulijst
- het aanpassen van de mrulijst

Tonen van mrulijst

De *AuxiliaryPaneContent* bestaat uit een *RibbonGalleryCategory* (*MRUGalleryCat*) die je nu moet opvullen met de laatst geopende bestanden.

Maak in de code-behind volgende procedure:

```
private void LeesMRU()
{
    MRUGalleryCat.Items.Clear();                                     (1)
    if (WindowMetRibbonControl.Properties.Settings.Default.mru != null) (2)
    {
        System.Collections.Specialized.StringCollection mrulijst =
            WindowMetRibbonControl.Properties.Settings.Default.mru; (3)
        for (int lijnnr = 0; lijnnr < mrulijst.Count; lijnnr++) (4)
        {
            MRUGalleryCat.Items.Add(mrulijst[lijnnr]);
        }
    }
}
```

- (1) maak de bestaande *MRUGalleryCat* leeg, zodat je ze kan opvullen
- (2) test of de setting bestaat
- (3) lees de collection van de setting "qat" in de variable "mrulijst" in
- (4) zet elke item dat je leest als een *Item* in de *MRUGalleryCat*

Bij het opstarten van de applicatie moet deze lijst in de *Gallery* gezet worden, dus:

Voeg in de constructor na de "InitializeComponent();" de regel "LeesMRU();" toe:

```
public WindowMetRibbon()
{
    InitializeComponent();
    LeesMRU();
    . . .
}
```

Veranderen van mrulijst

Als er een bestand wordt geopend of wordt opgeslagen, dan verandert de volgorde van de mrulijst, en moet dus ook de *Setting* en de *Gallery* veranderd worden.

Maak in de code-behind volgende procedure:

```
private void BijwerkenMRU(string bestandsnaam) (1)
{
    System.Collections.Specialized.StringCollection mrulijst = new
    System.Collections.Specialized.StringCollection(); (2)

    if (WindowMetRibbonControl.Properties.Settings.Default.mru != null) (3)
    {
```

```

    mrulijst = WindowMetRibbonControl.Properties.Settings.Default.mru; (4)
    int positie = mrulijst.IndexOf(bestandsnaam); (5)
    if (positie >= 0) (6)
    {
        mrulijst.RemoveAt(positie);
    }
    else (7)
    {
        if (mrulijst.Count >= 6) mrulijst.RemoveAt(5);
    }
}
mrulijst.Insert(0, bestandsnaam); (8)
WindowMetRibbonControl.Properties.Settings.Default.mru = mrulijst; (9)
WindowMetRibbonControl.Properties.Settings.Default.Save(); (10)
LeesMRU(); (11)
}

```

- (1) definieer de procedure met als parameter *bestandsnaam*
- (2) definieer een nieuw lokale variable "mrulijst"
- (3) als er al iets in de *Setting* staat
- (4) zet de inhoud van de *Setting* in de lokale variabele
- (5) zoek of het bestand al in de lijst aanwezig is
- (6) als dit bestand al bestaat, dan verwijder je dit
- (7) anders als de lijst al 6 items bevat, dan verwijder je het onderste
- (8) je voegt het huidige bestand op de bovenste plaats toe (index=0)
- (9) zet de nieuwe lijst in de *Setting*
- (10) BEWAAR deze *Setting*
- (11) pas de *Gallery* aan

Je moet nu nog op 2 plaatsen ervoor zorgen dat deze procedure wordt opgeroepen:

1) bij het openen van een bestand. Dit kan zowel via de menu *Open* als via een keuze in de *Gallery*. Beide echter roepen dezelfde procedure aan namelijk *LeesBestand*. Je kan de mrulijst best aanpassen nadat je zeker bent dat het betreffende bestand succesvol is geopend.

Voeg de instructie "*BijwerkenMRU(bestandsnaam);*" toe als laatste regel binnen de Try van de procedure "*LeesBestand*".

2) bij het opslaan van een bestand.

Je kan de mrulijst best aanpassen nadat je zeker bent dat het betreffende bestand succesvol is opgeslagen.

Voeg de instructie "*BijwerkenMRU(dlg.FileName);*" toe als laatste regel binnen de Try van de procedure "*SaveExecuted*".

Het is nu tijd om de implementatie uit te testen !

15 WPF PROJECT MET MVVM-PATTERN

Het MVVM-pattern is één van de designpatterns waarmee een project gemaakt kan worden. MVVM staat voor Model – View – ViewModel. In praktijk wil dit zeggen dat de data, de logica (business layer) en de visuele presentatie uit elkaar gehaald worden zodat bijvoorbeeld de data (*Model*) en de logica (*ViewModel*) kan geschreven worden door een ontwikkelaar, en de presentatie (*View*) door een designer. De communicatie tussen beide gebeurt via het *ViewModel*. Ook naar testing toe, en het hergebruiken van componenten heeft dit zijn voordelen. Doordat het gebruik van *Commands* (het doorgeven van user-input) en *DataBinding* - *DataContext* (het verbinden van data aan een *FrameworkElement*) binnen WPF doorgedreven geïmplementeerd zijn, wordt het MVVM-pattern aanzien als **HET** WPF-pattern.



15.1 Toevoegen van MVVM Framework aan VS

Doordat dit een specifiek pattern is, wordt bij een standaard installatie van VS2015 geen template voor dit soort project geïnstalleerd. Om fouten te voorkomen kan je best werken met het .net Framework 4.6 of later.

Er zijn verschillende bedrijven die voor dit pattern een *Framework of een template* gemaakt hebben, en dit gratis voor iedereen op het internet ter beschikking stellen.

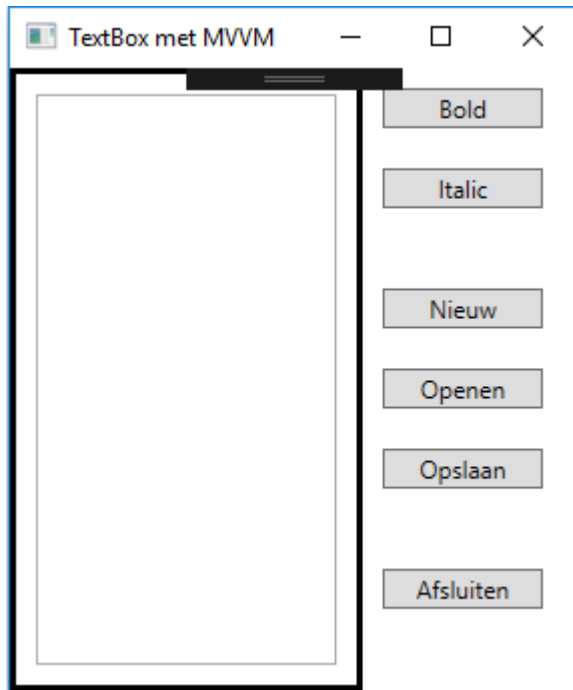
Momenteel zijn de meest populaire *Frameworks/templates* :

- MVVM Light Toolkit (dit ga je gebruiken in de cursus)
- WPF Model-View-ViewModel Toolkit
- Prism
- Caliburn
- Cinch

15.2 Simpel voorbeeld

Dit voorbeeld heeft niets te maken met echte gegevensverwerking (al dan niet uit een database) maar gewoon met de scherm inhoud.

Zo zou het resultaat er moeten uitzien :



Het is een *Window* met een *TextBox* op, 2 *ToggleButton*s om de tekst van de *TextBox* al dan niet in het vet en/of schuindruk te zetten, en 3 *Buttons* om van een propere lei te beginnen (Nieuw), de inhoud met eventueel vet en/of schuin op te slaan (Opslaan) en een opgeslagen bestand in te lezen (Openen). 1 *Button* om de applicatie af te sluiten (Afsluiten).

Project klaarmaken voor gebruik

- maak een nieuwe WPF application met de naam MVVMVoorbeeld.
- Installatie MVVM Light Toolkit:
 - kies in het menu *Tools* voor de item *NuGet Package Manager* en hieronder voor *Manage NuGet Packages for Solution...*
 - tik linksbovenaan bij de *Search Online* "mvvm light" in
 - kies voor "MVVM Light" en installeer dit framework in dit nieuwe project MVVMVoorbeeld.
- maak een Map "View" in de root van het project
- kopieer vanuit de oefenbestanden in de map MVVMVoorbeeld "TextBoxView.xaml" en "TextBoxView.xaml.cs" naar de View-map van het project.
- verwijder in de map van *ViewModel* volgende 2 bestanden:
 - MainViewModel.cs
 - ViewModelLocator.cs
- kopieer vanuit de oefenbestanden in de map MVVMVoorbeeld "Converters.cs" naar de *ViewModel*-map van het project.
- maak een map "Model" in de root van het project
- wis in de hoofdmap het "MainWindow.xaml" bestand
- in het bestand "App.xaml":
 - verander je de "StartURI" naar "View\TextBoxView.xaml".
 - wis je de *Application.Resources*-tag en de ganse inhoud.

Probeer de applicatie maar uit, het zicht zou ok moeten zijn, maar natuurlijk totaal zonder functionaliteit.

Bij het bekijken van de XAML-code van *TextBoxView.xaml* merk je dat er gebruikgemaakt is van 2 converters: *BooleanToFontWeight* en *BooleanToFontStyle* (zie vroeger in de cursus) zodat de Boolean properties *Vet* en *Schuin* overeenkomen met de juiste waarde.

15.2.1 Model

De bedoeling is nu een klasse te definiëren die alle gegevens bevat die nodig zijn om het *Window* te kunnen voorzien van al zijn data, en een bestand te kunnen opslaan en openen, met al zijn instellingen zijnde:

- de **inhoud** van de *TextBox*
- staat de tekst in het **vet** of niet
- staat de tekst in het **schuin** of niet

Je maakt de klasse *TekstMetOpmaak.cs* in de map *Model*.

```
public class TekstMetOpmaak
{
    public string Inhoud { get; set; }
    public Boolean Vet { get; set; }
    public Boolean Schuin { get; set; }
}
```

15.2.2 ViewModel

Alles van de *ViewModel* wordt gecommuniceerd naar de *View* via *Binding* en *Commands*. Bij dit gedeelte wordt het gebruik van de "MVVM Toolkit" pas duidelijk: alles i.v.m. *Binding* zoals het implementeren van de *INotifyPropertyChanged*-interface en de werking van *RoutedCommands* en de *ICommand*-interface is onder andere via een eigen (dus van GalaSoft) klasse *ViewModelBase* en een *RelayCommand* uitgewerkt.

15.2.2.1 Klassen voorzien van de *INotifyPropertyChanged*-interface

Het enige wat jij moet doen, is zorgen dat de klassen van het *Model* worden aangepast zodat deze geschikt is om via *Binding* in de *View* aan te spreken.

Dit doe je door een klasse te maken die een *Model*-klasse binnenkrijgt als parameter, en die gebaseerd is op de *ViewModelBase* klasse (van het MVVM Light framework) Door het *RaisePropertyChanged*-event te implementeren wordt er gezorgd dat bij veranderingen in de property de "verbonden" *Control* wordt bijgewerkt.

Je maakt de klasse *TekstMetOpmaakVM.cs* in de map *ViewModel*.

```
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Command;

namespace MVVMVoorbeeld.ViewModel
{
    public class TekstMetOpmaakVM : ViewModelBase
    {
```

```
private Model.TekstMetOpmaak opgemaakteTekst;

public TekstMetOpmaakVM (Model.TekstMetOpmaak deTekst)
{
    opgemaakteTekst = deTekst;
}

public string Inhoud
{
    get { return opgemaakteTekst.Inhoud; }
    set { opgemaakteTekst.Inhoud = value;
        RaisePropertyChanged("Inhoud");
    }
}

public Boolean Vet
{
    get { return opgemaakteTekst.Vet; }
    set { opgemaakteTekst.Vet = value;
        RaisePropertyChanged("Vet");
    }
}

public Boolean Schuin
{
    get { return opgemaakteTekst.Schuin; }
    set { opgemaakteTekst.Schuin = value;
        RaisePropertyChanged("Schuin");
    }
}
}
```

15.2.2.2 Het uitvoeren van Commands

Voor de functionaliteit van de Nieuw, Openen en Opslaan *Buttons* ga je een *RelayCommand* gebruiken (die de *ICommand* interface geïmplementeerd heeft), zodat in de *View* dit *Command* kan worden aangesproken.

Om de "Nieuw" *Button* te definiëren en te implementeren voeg je volgende code toe aan de *TekstMetOpmaakVM* klasse:

```
public RelayCommand NieuwCommand
{
    get { return new RelayCommand(NieuwTextBox); }
}

private void NieuwTextBox()                                     (1)
{
    Inhoud=string.Empty;
    Vet = false;
    Schuin = false;
}
```

(1) *NieuwTextBox* zet de 3 properties op initiële waarde

Gelijkaardig kan ook de functionaliteit voor de "Openen" en "Opslaan" *Buttons* gemaakt worden:

Voeg volgende code toe aan de *TekstMetOpmaakVM* klasse (het gebruik van *CommonDialogBox* is eerder uitgelegd in de cursus):

```

. . .
using Microsoft.Win32;
using System.IO;
using System.Windows;
. . .

public RelayCommand OpslaanCommand
{
    get { return new RelayCommand(OpslaanBestand); }
}

private void OpslaanBestand()
{
    try
    {
        SaveFileDialog dlg = new SaveFileDialog();
        dlg.FileName = "tekstbox";
        dlg.DefaultExt = ".box";
        dlg.Filter = "Textbox documents |*.box";

        if (dlg.ShowDialog() == true)
        {
            using (StreamWriter bestand = new StreamWriter(dlg.FileName))    (1)
            {
                bestand.WriteLine(Inhoud);
                bestand.WriteLine(Vet.ToString());
                bestand.WriteLine(Schuin.ToString());
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("opslaan mislukt : " + ex.Message);
    }
}

```

```

public RelayCommand OpenenCommand
{
    get { return new RelayCommand(OpenenBestand); }
}

private void OpenenBestand()
{
    try
    {
        OpenFileDialog dlg = new OpenFileDialog();
        dlg.FileName = "";
        dlg.DefaultExt = ".box";
        dlg.Filter = "Textbox documents |*.box";

        if (dlg.ShowDialog() == true)
        {
            using (StreamReader bestand = new StreamReader(dlg.FileName))    (2)
            {
                Inhoud = bestand.ReadLine();
                Vet = Convert.ToBoolean(bestand.ReadLine());
                Schuin = Convert.ToBoolean(bestand.ReadLine());
            }
        }
    }
}

```

```

    }
}
}
catch (Exception ex)
{
    MessageBox.Show("openen mislukt : " + ex.Message);
}
}

```

- (1) je schrijft de 3 properties weg naar een bestand : door Vet en Schuin van het type Boolean zijn, wordt de ToString() gebruikt om dit als strings weg te kunnen schrijven.
- (2) je lees de properties regel per regel in met terug een conversie van string naar Boolean.

Als laatste voeg je de code voor de Afsluiten-*Button* toe:

```

public RelayCommand AfsluitenCommand
{
    get { return new RelayCommand(AfsluitenApp); }
}

private void AfsluitenApp()
{
    Application.Current.MainWindow.Close();
}

```

15.2.2.3 Het uitvoeren van Events

Om ervoor te zorgen dat bij het sluiten van het programma de vraag “Wilt u het programma sluiten ?” gesteld wordt, moet er voor gezorgd worden deze in het *Closing*-event van de *View* wordt uitgevoerd. Het MVVM Light framework heeft hiervoor een *EventToCommand* gecreëerd die een *Event* omzet naar een *Command*. Je moet zelf wel bepalen wanneer dit event in actie mag schieten (triggeren) zodat in dit geval bij het sluiten van de *Window* dit *Event* wordt uitgevoerd.

In de *Window*-tag ga je volgende namespaces toevoegen, zodat de nieuwe tags herkend worden :

```
xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.Interactivity"
```

is nodig om *Interaction.Triggers* te herkennen

```
xmlns:gal="http://www.galasoft.ch/mvvmlight"
```

is nodig om *EventToCommand* te herkennen. Deze verwijzing is nodig vanaf MVVM Light framework v5.0 (oktober 2014).

Vorige versies van dit framework (v4.x) gebruiken :

```
xmlns:gal="clr-namespace:GalaSoft.MvvmLight.Command;assembly=GalaSoft.MvvmLight.Extras.WPF45"
```

Juist onder de *Window*-tag ga je volgende code invoegen.

```

<i:Interaction.Triggers>
  <i:EventTrigger EventName="Closing" >
    <gal:EventToCommand Command="{Binding ClosingCommand}"

```

```
PassEventArgsToCommand="True" />
</i:EventTrigger>
</i:Interaction.Triggers>
```

Een woordje uitleg :

de *Interaction.Triggers* worden toegepast op de tag waar je inzit, dus in dit geval de *Window*-tag. De *EventTrigger* is het *Event* waar iets mee moet gebeuren als het zich voordoet.

De *EventToCommand* (hoort bij het MVVM Light framework) specificeert welk *Command* moet worden uitgevoerd en of er eventueel parameters moeten worden doorgegeven (*PassEventArgsToCommand*).

In het geval van de *Closing* heb je daar nood aan omdat je de *Close* eventueel moet *Cancelen*, en dit kan via de *CancelEventArgs* parameter.

In het *ViewModel* zelf ga je de *Command ClosingCommand* uitwerken zoals een gewoon *Command* met als extra de doorgegeven parameter:

```
public RelayCommand<CancelEventArgs> ClosingCommand
{
    get { return new RelayCommand<CancelEventArgs>(OnWindowClosing); }
}

public void OnWindowClosing(CancelEventArgs e)
{
    if (MessageBox.Show("Afsluiten", "Wilt u het programma sluiten ?",
        MessageBoxButton.YesNo, MessageBoxImage.Question, MessageBoxResult.No) ==
        MessageBoxResult.No)
        e.Cancel = true;
}
```

15.2.3 View

Nu alles is voorbereid in het *ViewModel*, kan je de verbindingen gaan leggen in de *View*.

In de XAML-code (*TextBoxView.xaml*) voeg je volgende attributen toe aan de tags:

- *TekstTextBox* : *Text="{Binding Inhoud}"*
- *ButtonBold* : *IsChecked="{Binding Vet}"*
- *ButtonItalic* : *IsChecked="{Binding Schuin}"*

Op deze manier zijn alle properties van de klasse *TekstMetOpmaakVM* verbonden met de bijbehorende *Controls*.

Naar de *Buttons* toe, moet je nu nog de *Commands* koppelen:

Voeg het attribuut *Command="{Binding NieuwCommand}"*, *Command="{Binding OpslaanCommand}"*, *Command="{Binding OpenenCommand}"* en *Command="{Binding AfsluitenCommand}"* aan de overeenkomstige *Buttons* toe.

Om nu alles met elkaar te verbinden, ga je bij het starten van de applicatie niet rechtstreeks een *View* xaml-bestand te tonen, maar moet je ervoor zorgen dat de verbindingen tussen Model, ViewModel en View tot stand gebracht zijn.

In het hoofdstuk over *DataBinding* is er gesproken over *DataContext*. Dit attribuut kan je instellen om de *Source* van je data te speciëren. Alle onderliggende tags gebruiken automatisch ook deze instelling (wordt dus geërfd).

Hiervoor ga je eerst het *App.xaml* bestand moeten aanpassen zodat alles effectief wordt gecreëerd en verbonden:

- verwijder `StartupUri="View/TextBoxView.xaml"`
- voeg in het *App.xaml.cs* bestand volgende code toe:

```
protected override void OnStartup(StartupEventArgs e)
{
    base.OnStartup(e);
    Model.TekstMetOpmaak mijnTekst = new Model.TekstMetOpmaak();
    ViewModel.TekstMetOpmaakVM vm = new ViewModel.TekstMetOpmaakVM(mijnTekst);
    View.TextBoxView mijnTekstboxView = new View.TextBoxView();

    mijnTekstboxView.DataContext = vm;
    mijnTekstboxView.Show();
}
```

- (1) mits je de *OnStartup* overschrijft, ga je eerst de basis *OnStartup* uitvoeren
- (2) maak een object "mijnTekst" van de Model-klasse *TekstMetOpmaak*
- (3) maak een object "vm" van de ViewModel-klasse *TekstMetOpmaakVM*
- (4) maak een object "mijnTekstboxView" om de *View* te instantiëren
- (5) koppel de "vm" aan het *DataContext* attribuut van de *View*.
- (6) Toon de *View*

Probeer maar uit (als inhoud geen <enter> gebruiken, maar gewoon doortikken) !

Om het programma een beetje af te werken, kan je ervoor zorgen dat de "Opslaan"-*Button* alleen *Enabled* is als er iets in de *TextBox* staat (dus de *Text.Length* moet groter zijn dan 0). Hiervoor is geen code nodig, maar wel een extra *Binding* :

- voeg in de *Window.Resources* tag volgende regel toe:

```
<local:IntToBoolean x:Key="IntegerNaarLogisch" />
```

De converter *IntToBoolean* is één van de 3 converters in het bijgeleverde bestand en zet de *Integer 0* om in *false*, en alle andere *Integers* in *true*.

- voeg bij de "Opslaan"-*Button* volgend attribuut toe:

```
IsEnabled="{Binding ElementName=TekstTextBox, Path=Text.Length, Converter={StaticResource IntegerNaarLogisch}}"
```

Probeer maar uit !

15.3 Voorbeeld met data uit een database

In vele gevallen wordt er in de applicatie data vanuit een database (of namaak-database in ons voorbeeld) gebruikt. Om deze implementatie uit te werken ga je van start met een nieuwe applicatie.

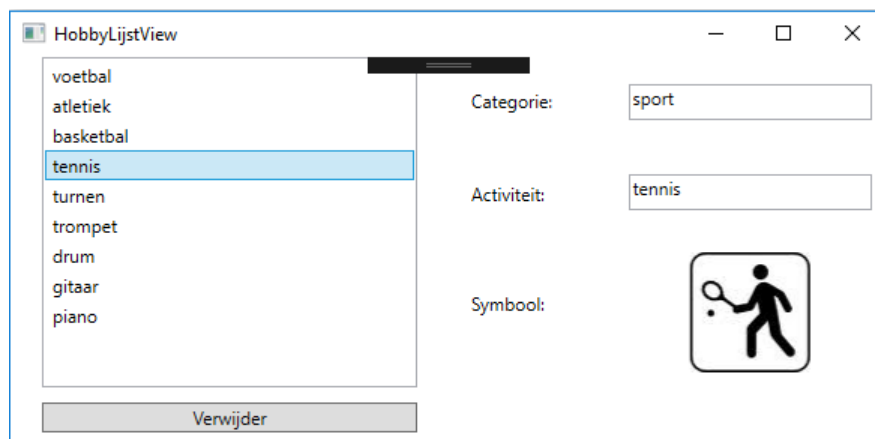
Project klaarmaken voor gebruik

- Maak een nieuwe WPF application met de naam MVVMHobby.
- Installeer MVVM Light:

Indien je in dezelfde solution blijft werken, is de toolkit al geïnstalleerd voor de andere WPF application.

- **kies in het menu *Tools* voor de item *NuGet Package Manager* en hieronder voor *Manage NuGet Packages for Solution...***
 - **kies links bovenaan "*Installed packages*"**
 - **kies voor "*MVVM Light*" en "*Manage*" en vink de nieuwe applicatie aan.**
-
- maak een Map "*View*"
 - Kopieer vanuit de oefenbestanden in de map MVVM "*HobbyLijstView.xaml*" en "*HobbyLijstView.xaml.cs*" naar de View-map van het project.
 - maak een map "*Images*"
 - Kopieer vanuit de oefenbestanden in de map Hobby alles naar deze *Images*-map.
 - maak een map "*Model*"
-
- Verwijder in de map van *ViewModel* volgende 2 bestanden:
 - *MainViewModel.cs*
 - *ViewModelLocator.cs*
 - wis in de hoofdmap het "*MainWindow.xaml*" bestand
 - in het bestand "*App.xaml*":
 - verander je de "*StartURI*" naar "*View\HobbyLijstView.xaml*"
 - wis je de *Application.Resources*-tag en de ganse inhoud.

Je kan de applicatie nu nog niet uitproberen omdat er geen bestanden staan in de map *ViewModel*, maar het eindresultaat gaat er zoals hieronder uitzien:



15.3.1 Model

In dit voorbeeld heb je een klasse nodig die alles van één hobby bevat, dus eigenlijk gelijk is aan de structuur van de data uit de database.

Je maakt de klasse *Hobby.cs* in de map *Model*:

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

namespace MVVMHobby.Model
{
    public class Hobby
    {
        public string Categorie { get; set; }
        public string Activiteit { get; set; }
        public BitmapImage Symbool { get; set; }

        public Hobby(string nCategorie, string nActiviteit,
            BitmapImage nSymbool)
        {
            Categorie = nCategorie;
            Activiteit = nActiviteit;
            Symbool = nSymbool;
        }
    }
}
```

De gegevens van de hobby's ga je rechtstreeks in de lijst invullen (is dus eigenlijk een database-tabel namaken) in de constructor van de klasse *HobbyLijstVM*.

15.3.2 ViewModel

Het ViewModel is het verbindingsstuk naar een View toe. Dit wil zeggen dat de klasse van het *Model* moet worden aangepast naar *Binding* toe. Met andere woorden er wordt een aanverwante klasse gemaakt die de *Model*-klasse als parameter binnenkrijgt, en die via *ViewModelBase* en *RaisePropertyChanged* worden klaargezet om uiteindelijk in de View verbonden te worden.

Je maakt een klasse *HobbyVM* in de map *ViewModel* van je project:

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

namespace MVVMHobby.ViewModel
{
    public class HobbyVM : ViewModelBase (1)
    {
        private Model.Hobby hobby;
        public HobbyVM(Model.Hobby nhobby) (2)
        {
            this.hobby = nhobby;
        }

        public string Categorie (3)
    }
}
```

```
{
    get
    {
        return hobby.Categorie;
    }
    set
    {
        hobby.Categorie = value;
        RaisePropertyChanged("Categorie");
    }
}

public string Activiteit
{
    get
    {
        return hobby.Activiteit;
    }
    set
    {
        hobby.Activiteit = value;
        RaisePropertyChanged("Activiteit");
    }
}

public BitmapImage Symbool
{
    get
    {
        return hobby.Symbool;
    }
    set
    {
        hobby.Symbool = value;
        RaisePropertyChanged("Symbool");
    }
}
}
```

- (1) Je hermaakt de klasse *Hobby* maar dan via de base-klasse *ViewModelBase* waardoor de *INotifyPropertyChanged* geïmplementeerd is
- (2) In de constructor wordt de gewone *Hobby* (uit het model) als parameter meegegeven
- (3) Bij elke property van een *Hobby* wordt de *RaisePropertyChanged*-methode toegevoegd, zodat de *Binding* zijn ding kan doen

Om een “collection” bruikbaar te maken voor *Binding* moet je in plaats een *List* gebruiken van een *ObservableCollection*.

Aan de rechterkant ga je alle properties van de gekozen *Hobby* tonen. De lijst moet opgebouwd zijn met *Bindable* objecten zodat ook in één element alle *Binding*-mogelijkheden aanwezig zijn. De lijst moet dus bestaan uit *HobbyVM* elementen.

Je maakt een klasse *HobbyLijstVM* in de map *ViewModel* van je project:

...

```

using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Command;
using System.Collections.ObjectModel;

namespace MVVMHobby.ViewModel
{
    public class HobbyLijstVM : ViewModelBase
    {
        public HobbyLijstVM()
        {
            HobbyLijst.Add(new HobbyVM(new Model.Hobby("sport", "voetbal",
new BitmapImage(new Uri("pack://application:,,,/Images/voetbal.jpg",
UriKind.Absolute))));
            HobbyLijst.Add(new HobbyVM(new Model.Hobby("sport", "atletiek",
new BitmapImage(new Uri("pack://application:,,,/Images/atletiek.jpg",
UriKind.Absolute))));
            HobbyLijst.Add(new HobbyVM(new Model.Hobby("sport", "basketbal",
new BitmapImage(new Uri("pack://application:,,,/Images/basketbal.jpg",
UriKind.Absolute))));
            HobbyLijst.Add(new HobbyVM(new Model.Hobby("sport", "tennis",
new BitmapImage(new Uri("pack://application:,,,/Images/tennis.jpg",
UriKind.Absolute))));
            HobbyLijst.Add(new HobbyVM(new Model.Hobby("sport", "turnen",
new BitmapImage(new Uri("pack://application:,,,/Images/turnen.jpg",
UriKind.Absolute))));
            HobbyLijst.Add(new HobbyVM(new Model.Hobby("muziek", "trompet",
new BitmapImage(new Uri("pack://application:,,,/Images/trompet.jpg",
UriKind.Absolute))));
            HobbyLijst.Add(new HobbyVM(new Model.Hobby("muziek", "drum",
new BitmapImage(new Uri("pack://application:,,,/Images/drum.jpg",
UriKind.Absolute))));
            HobbyLijst.Add(new HobbyVM(new Model.Hobby("muziek", "gitaar",
new BitmapImage(new Uri("pack://application:,,,/Images/gitaar.jpg",
UriKind.Absolute))));
            HobbyLijst.Add(new HobbyVM(new Model.Hobby("muziek", "piano",
new BitmapImage(new Uri("pack://application:,,,/Images/piano.jpg",
UriKind.Absolute))));
        }

        private ObservableCollection<HobbyVM> hobbyLijst =
            new ObservableCollection<HobbyVM>();
        public ObservableCollection<HobbyVM> HobbyLijst
        {
            get
            {
                return hobbyLijst;
            }
            set
            {
                hobbyLijst = value;
                RaisePropertyChanged("HobbyLijst");
            }
        }

        private HobbyVM selectedHobby;
        public HobbyVM SelectedHobby
        {
            get

```

(1)

(2)

(3)

```

        {
            return selectedHobby;
        }
        set
        {
            selectedHobby = value;
            RaisePropertyChanged("SelectedHobby");
        }
    }
}

```

- (1) In de constructor wordt een *ObservableCollection* van *HobbyVM* objecten aangemaakt om een tabel uit een database na te bootsen
- (2) Je voorziet een property *HobbyLijst* die als vulling van de *ListBox* gaat dienen
- (3) Je voorziet een property *SelectedHobby* die het geselecteerde item van de lijst voorstelt.

Hiermee zijn de properties klaar om in de *View* gebruikt te worden.

Nu moet alleen nog de *Command* van de *Button* gedefinieerd worden zodat er in de *View* geen code-behind ontstaat.

Je voegt in de klasse *HobbyLijstVM* volgende definities toe:

```

public RelayCommand<RoutedEventArgs> VerwijderCommand (1)
{
    get { return new RelayCommand<RoutedEventArgs>(Verwijder); }
}

private void Verwijder(RoutedEventArgs e) (2)
{
    HobbyLijst.Remove(SelectedHobby);
}

```

- (1) Voorzie een publiek *RelayCommand* dat door de *View* kan worden aangesproken en die als returnwaarde een *RelayCommand* teruggeeft die de method *Verwijder* uitvoert.
- (2) De method *Verwijder* verwijdert de geselecteerde *Hobby* uit de lijst

15.3.3 View

In de *View* ga je nu al de nodige *Bindings* definiëren.

Je voegt in de *Button*-tag volgend attribuut toe:

Command="{Binding VerwijderCommand}"

In de *ListBox* wordt de lijst van *Hobbies* weergegeven door hun *Activiteit* te laten zien. Door een *Hobby* aan te duiden, wordt de *SelectedHobby* ingevuld

Je voegt in de *ListBox*-tag volgende attributen toe:

ItemsSource="{Binding HobbyLijst}"
DisplayMemberPath="Activiteit"

```
SelectedItem="{Binding SelectedHobby}"
```

Je voegt in de eerste *TextBox*-tag volgend attribuut toe:

```
Text="{Binding SelectedHobby.Categorie}"
```

Je voegt in de tweede *TextBox*-tag volgend attribuut toe:

```
Text="{Binding SelectedHobby.Activiteit}"
```

Om er voor te zorgen dat een wijziging in de *TextBox* dadelijk zichtbaar is in de *ListBox*, moet je dit uitbreiden tot

```
Text="{Binding SelectedHobby.Activiteit,  
UpdateSourceTrigger=PropertyChanged}"
```

Je voegt in de *Image*-tag volgend attribuut toe:

```
Source="{Binding SelectedHobby.Symbol}"
```

Om ten slotte nog een voorbeeld van *Events* in dit project te verwerken, ga je bij het inhouden van de muisknop op de *Image* een extra venster laten zien met als inhoud een uitvergroting van deze *Image*. Bij het loslaten van de muisknop verdwijnt dit venster terug.

Dit moet gebeuren met de *Events MouseDown* en *MouseUp* die in de *MouseEventArgs* parameter de *Image* kunnen doorgeven.

Je voegt in de *Windows*-tag volgende regels toe

```
xmlns:i="clr-namespace:System.Windows.Interactivity;  
assembly=System.Windows.Interactivity"  
xmlns:gala="http://www.galasoft.ch/mvmlight"
```

Je voegt tussen de *Image*-tags volgende tags toe:

```
<i:Interaction.Triggers>  
  <i:EventTrigger EventName="MouseDown">  
    <gala:EventToCommand Command="{Binding MouseDownCommand}"  
PassEventArgsToCommand="True" />  
  </i:EventTrigger>  
  <i:EventTrigger EventName="MouseUp">  
    <gala:EventToCommand Command="{Binding MouseUpCommand}"  
PassEventArgsToCommand="True" />  
  </i:EventTrigger>  
</i:Interaction.Triggers>
```

In de View-map maak je een nieuw Window aan met de naam *ImageView*.

Specifieer een attribuut *WindowStyle* in de Window-tag als "*ToolWindow*".

Vervang de Grid-tag door een Image-tag en geef deze de naam "*GroteImage*".

```
<Window x:Class="MVVMHobby.View.ImageView"  
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
  Title="ImageView" Height="300" Width="300"  
WindowStyle="ToolWindow">  
  <Image Name="GroteImage"></Image>
```

</Window>

In de code van *HobbyLijstVM* wordt de volgende code toegevoegd:

```
View.ImageView groteView; (1)

public RelayCommand<MouseEventArgs> MouseDownCommand
{
    get { return new RelayCommand<MouseEventArgs>(MuisIn); }
}

private void MuisIn(MouseEventArgs e)
{
    Image tg = (Image)e.OriginalSource; (2)
    groteView = new View.ImageView(); (3)
    groteView.GroteImage.Source = tg.Source; (4)
    groteView.Show(); (5)
}

public RelayCommand<MouseEventArgs> MouseUpCommand
{
    get { return new RelayCommand<MouseEventArgs>(MuisUit); }
}

private void MuisUit(MouseEventArgs e)
{
    if (groteView != null) (6)
        groteView.Close();
    groteView = null; (7)
}
```

- (1) definieer een *ImageView*
- (2) In de *MouseEventArgs* parameter *e* kan je via de *OriginalSource* achterhalen van welk object dit *Event* komt. Je weet dat dit de *Image* is, dus cast je dit
- (3) maak een instance van de *ImageView*
- (4) zet dezelfde *Source* in de *ImageView*
- (5) toon de *Window*
- (6) Als er een instance van de *ImageView* bestaat, dan sluit je deze
- (7) zet de *ImageView* op *null* zodat dit object wordt vrijgegeven en door de Garbage Collector kan worden opgekuist.

Om bij het opstarten van de applicatie alles te definiëren en te verbinden, wordt `StartupUri="MainWindow.xaml"` verwijderd in *App.xaml*:

Je voegt in het *App.xaml.cs* bestand in de *App* klasse volgende method toe:

```
protected override void OnStartup(StartupEventArgs e)
{
    base.OnStartup(e);
    ViewModel.HobbyLijstVM vm = new ViewModel.HobbyLijstVM();
    View.HobbyLijstView hobbyView = new View.HobbyLijstView();
    hobbyView.DataContext = vm;
    hobbyView.Show();
}
```

}

Je kan de applicatie uitproberen :

- een *Hobby* selecteren
- de tekst van een *Activiteit* veranderen
- een *Hobby* verwijderen
-



Opdracht: ParkingbonMVVM

16 DEPLOYMENT

Vanaf Visual Studio 2012 wordt het maken van een msi-bestand niet meer gesupporteerd. Er is open source software beschikbaar (zoals WIX) om toch zo'n bestand aan te maken, maar dit valt buiten de functionaliteit van deze cursus.


De enige ingebouwde manier om een afgewerkte installeerbare applicatie aan te bieden is de *ClickOnce*-methode en staat hieronder beschreven.

We gebruiken het project MVVMVoorbeeld om deze procedure uit te proberen.

Om het "juist echt" te maken, ga je de *DataBinding* applicatie een bijhorend *icon* geven:

Kopieer van de oefenbestanden-map het bestand *font.ico* naar het *Project*.

Voeg achteraan in de *Window*-tag een attribuut toe: **Icon="font.ico"**

Als je nu de applicatie start, dan zal er in de linkerbovenhoek het  icoon te zien zijn.

Het is de bedoeling dat je dit icoon gebruikt als applicatie-icoon, maar ook voor het maken van een *ShortCut* op de *Desktop* en in de *Programs Menu* na de installatie van de applicatie.

Klik met de rechtermuisknop op de *Project*naam en kies onderaan voor *Properties*.

Op de *Application*-tab: *Icon and manifest*:

- Icon : font.ico
- Manifest : Create application without a manifest

Op de *Build*-tab: Configuration = Release

Na het kiezen van deze instellingen *Build* je het *Project*.

De meest recente manier om bv. een installatie-CD te maken, is gebruikmaken van de *ClickOnce* methode. Door een *Publish* te activeren, wordt een bijna automatische aanmaakprocedure gestart.

Klik in de *Solution Explorer* met de rechtermuisknop op de projectnaam en kies *Publish...* of kies in het menu *Build* voor *Publish DataBinding*.

Door overal de standaardinstellingen te laten staan:

- de map \publish
- from a CD-ROM or a DVD-ROM
- The application will not check for updates

verkrijg je een submap *publish* binnen de *Project*map met de inhoud voor bv. een CD. Het is de bedoeling om de ganse inhoud van de map bv. te branden. Er staat een *autorun.inf* bestand op dat ervoor zorgt dat al naar gelang de instellingen van het toestel waarop je het wil installeren, automatisch de *setup* start. Indien dat niet het geval is, dan kan je de *setup.exe* handmatig starten.

Je kan dit rechtstreeks vanuit de map eens uitproberen.

Omdat de *publisher* niet gekend is op het toestel waarop je het installeert, zal er altijd de vraag gesteld worden of je de installatie wel wil uitvoeren.

De *uninstall* kan gebeuren via langs de normale weg van een geïnstalleerde applicatie: *Control Panel – Programs and Features*.

17 COLOFON

Sectorverantwoordelijke:

Cursusverantwoordelijke:

Medewerkers: Chris Van Loon

Versie: September 2017

Nummer dotatielijst: