Documentatie de laborator

Sursa: http://www.wpf-tutorial.com/

INTRODUCERE

WPF este un framework grafic pentru realizarea de aplicatii folosind .NET Framework.

Exista mai multe framework-uri pentru realizarea de GUI (Graphic User Interface), dar sub .NET Framework, cele mai importante sunt Windows Forms si WPF.

WPF a aparut sub .NET Framework 3.0 si pune la dispozitie o serie de clase si assembly-uri care permit scrierea de cod mult mai usor si mult mai flexibil. Spre exemplu, foloseste tehnologia de randare grafica Direct3D care permite redarea de grafica 3D in aplicatii in care performanta este un element cheie, asa cum e cazul jocurilor video. (Pentru detalii suplimentare vezi https://en.wikipedia.org/wiki/Direct3D) Astfel, componentele reprezentate in ferestrele aplicatiei vor avea contururi mult mai line, putand solicita astfel si resursele hardware ale calculatorului. In cazul in care se foloseste traditionalul Windows Forms Application nu vor fi solicitate performante grafice avansate, acesta fiind mult mai ineficient decat un WPF Application.

WPF introduce un concept nou, si anume acela de „template" care permite redefinirea controalelor, dar despre care vom vorbi mai tarziu.

WPF este mai nou decat Windows Form si prezinta cateva avantaje fata de acesta:

- Este mai nou si mai in ton cu noile standarde de programare
- Este mai **flexibil**, deci se pot realiza mai multe lucruri cu el fara a crea sau cumpara controale noi
- Partea de cod XAML pe care o foloseste WPF-ul face editarea interfetei grafice mult mai usoara si **permite separarea codului** pe care il scrie programatorul (C# in cazul nostru) de codul pe care il scrie designer-ul (XAML)
- Legarea datelor de controale (**data binding**) permite o separare mai buna a datelor de interfata grafica (de layout-ul aplicatiei)
- **Foloseste si performantele resurselor hardware** pentru desenarea de GUI, pentru a avea performante mai ridicate (are inclus un motor care identifica ce nivel de accelerare hardware este suportat de catre sistem si isi ajusteaza singur performantele in functie de acesta). Astfel, impactul cel mai mare il au Video RAM-ul, calcularea efectelor per pixeli, aplicarea texturilor multiple, etc
- Permite realizarea de interfete grafice atat petru **aplicatii desktop** cat si pentru **aplicatii web**

Dupa cum se poate subantelege din cele precizate anterior, WPF este o combinatie de cod XAML cu cod C# sau VB.NET sau orice alt cod al unui limbaj suportat de catre .NET.

Pentru scrierea codului vom folosi IDE-ul (Integrated Development Environment) Visual Studio, cel folosit de majoritatea programatorilor .NET. Acesta se poate descarca de pe Microsoft Imagine (in varianta Profesional, Premium sau Ultimate) sau se poate descarca gratuit de pe site-ul Microsoft varianta Express, care contine mai putine functionalitati decat celelalte variante, dar este suficienta pentru dezvoltarea de aplicatii simple pentru laborator.

Puteti descarca versiunea de Visual Studio 2015 sau o versiune mai noua (spre exemplu, versiunea Visual Studio 2019 aparuta in iunie 2018).

XAML vine de la **eXtensible Application Markup Language** si reprezinta varianta Microsoft, cu ajutorul careia se realizeaza interfete grafice, a XML-ului clasic. Acest limbaj este folosit pentru specificarea si setarea de caracteristici pentru clase. Cu alte cuvinte, se vor putea seta variabile, se pot defini proprietati pentru clase si folosi ulterior in aplicatie (in codul .cs) **Parser-ul XAML va converti codul in mod automat si va crea obiecte in momentul rularii aplicatiei.** Cu XAML se pot realiza doar aspectele vizuale ale aplicatiei, nu si partea de logica a aplicatiei. Pentru aceasta este nevoie de un limbaj de programare (cum ar fi C# sau VB.NET)

In varianta de Windows Form, GUI-ul se crea neaparat folosind limbajul de programare in care se dezvolta aplicatia (C#, VB.NET, etc); dar folosind XAML, realizarea interfetei grafice seamana mai mult cu realizarea unei interfete grafice cu HTML. Intr-o aplicatie realizata folosind WPF, pagina sau **fereastra este compusa din doua parti**: **partea de interfata**, cu toate elementele sale, in fisierul XAML si **partea de Code Behind** (fisierele cu extensia .cs sau .vb) care trateaza evenimentele si care poate manipula codul XAML.

DISPATCHER THREAD SI THREAD AFFINITY

In momentul in care se ruleaza o aplicatie WPF se creaza in mod automat doua fire de executie unul este „**Rendering Thread**” care ii este ascuns programatorului si nu poate fi accesat sau modificat de catre acesta din aplicatie, iar celalalt este „**Dispatcher Thread**” care cuprinde toate elementele interfetei grafice. Se poate spune astfel ca dispatcher-ul este firul de executie grafic, cel care leaga toate elementele create folosind WPF. **Faptul ca WPF solicita legarea tuturor componentelor grafice in Dispatcher mai este numita si „Thread Affinity”.** Exista astfel clasa **Dispatcher** care se ocupa de aceasta afinitate catre firul de executie grafic. Aceasta contine o bucla prin care sunt canalizate toate elementele de interfata

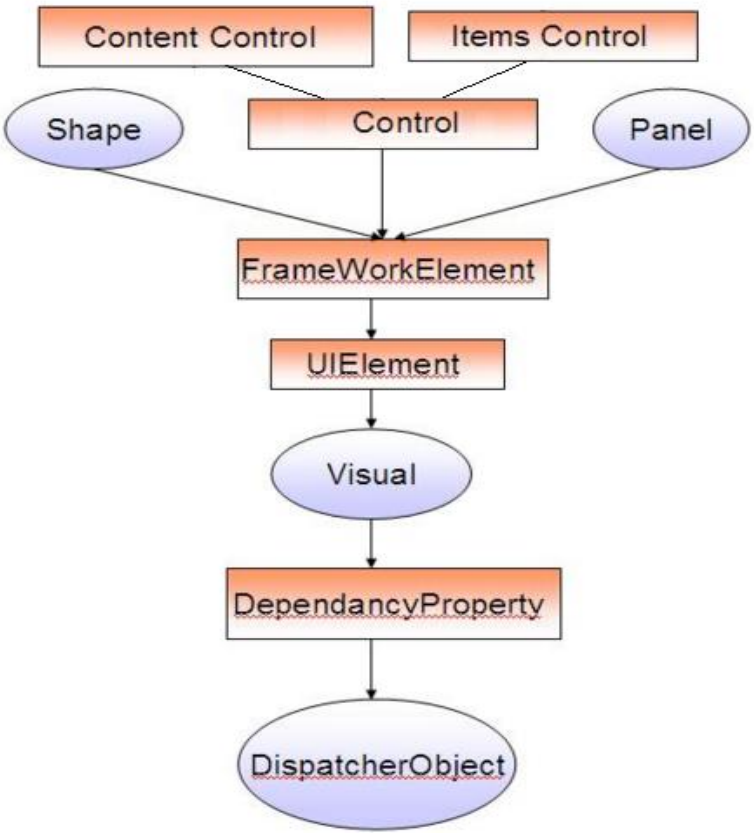
grafica. Fiecare element grafic este derivat din **DispatcherObject** care defineste o proprietate numita **Dispatcher**, care indica spre firul de executie grafic**. Astfel, daca dintr-un alt fir de executie se doreste accesarea vreunei componente grafice, se va apela firul Dispatcher. Principala atributie a lui DispatcherObject este aceea de a verifica daca firul de executie are acces la obiect.**

FOLOSIREA LUI „DependencyObject”

Orice control WPF este derivat din **DependencyObject**. Aceasta este o clasa care suporta **DependencyProperty** (o proprietate sistem oferita de catre WPF). Cum toate componentele grafice sunt derivate din clasa DependencyObject, prin urmare acestea mostenesc comportamente si proprietati ale acestei clase, cum ar fi asocierea cu un trigger de evenimente, proprietati de asociere, animatii, etc.

IERARHIA DE OBIECTE

Exista cateva obiecte care definesc un control WPF si sunt prezentate in desenul de mai jos. Clasele abstracte sunt desenate intr-un oval, iar cele concrete in dreptunghi.

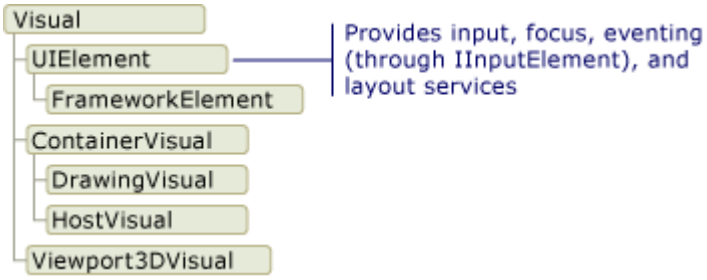


De jos in sus, avem urmatoarea ierarhie:

1. **DispatcherObject** - este parintele tuturor controalelor WPF; este clasa care are grija de firul de executie grafic

2. **DependencyObject** - este clasa care construieste "observatorul" pentru DependencyProperty

3. **Visual** - este legatura dintre DependencyObject si UIElements. Acesta clasa reprezinta abstractizarea de baza pentru toate obiectele de tip "interfata grafica". Imaginea de mai jos prezinta ierarhia obiectelor vizuale oferite de arhitectura WPF:



Pentru detalii vezi

https://msdn.microsoft.com/en-us/library/system.windows.media.visual(v=vs.110).aspx

4. **UIElement** - creaza suportul pentru comportamente WPF ca: layout-uri, evenimente, inputuri

5. **FrameworkElement** - implementeaza interfata UIElement

6. **Shape** - clasa de baza pentru toate formele geometrice standard

7. **Panel** – clasa de baza pentru toate tipurile de paneluri din WPF

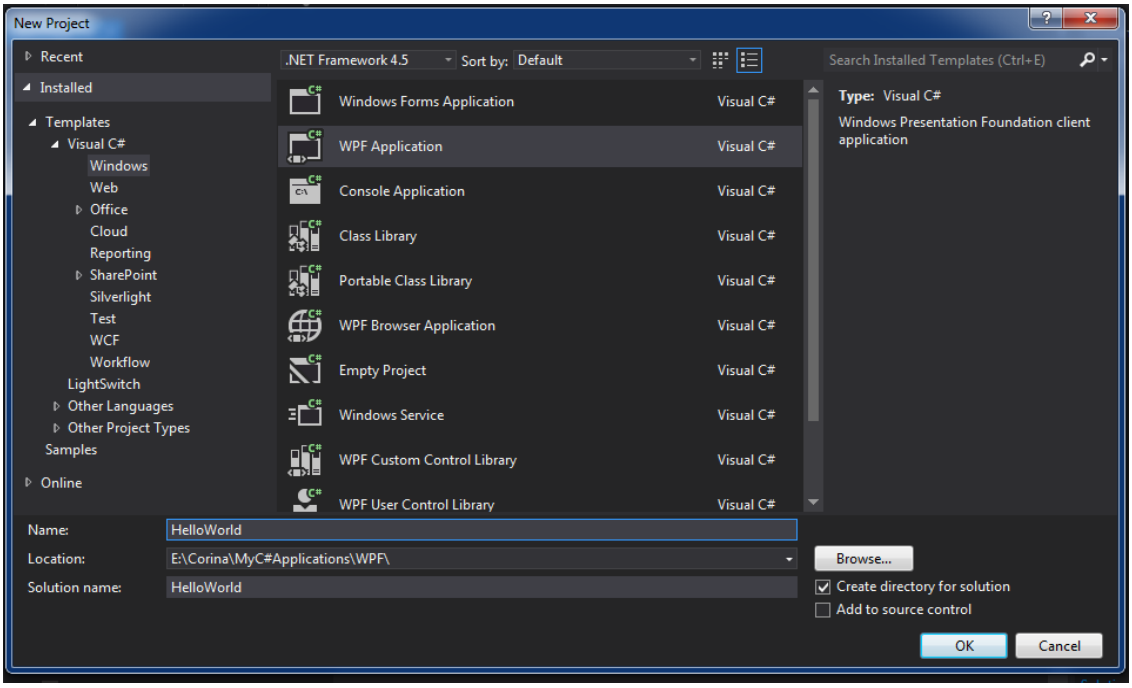
8. **Control** - elementul grafic care interactioneaza cu utilizatorul. Controalelor li se pot aplica template-uri pentru schimbarea infatisarii

9. **ContentControl** - clasa de baza pentru toate controalele care au continut singular: TextBlock, Label, etc...

10. **ItemsControl** - clasa de baza pentru toate controalele care permit afisarea unei colectii: ListView, ComboBox, etc...

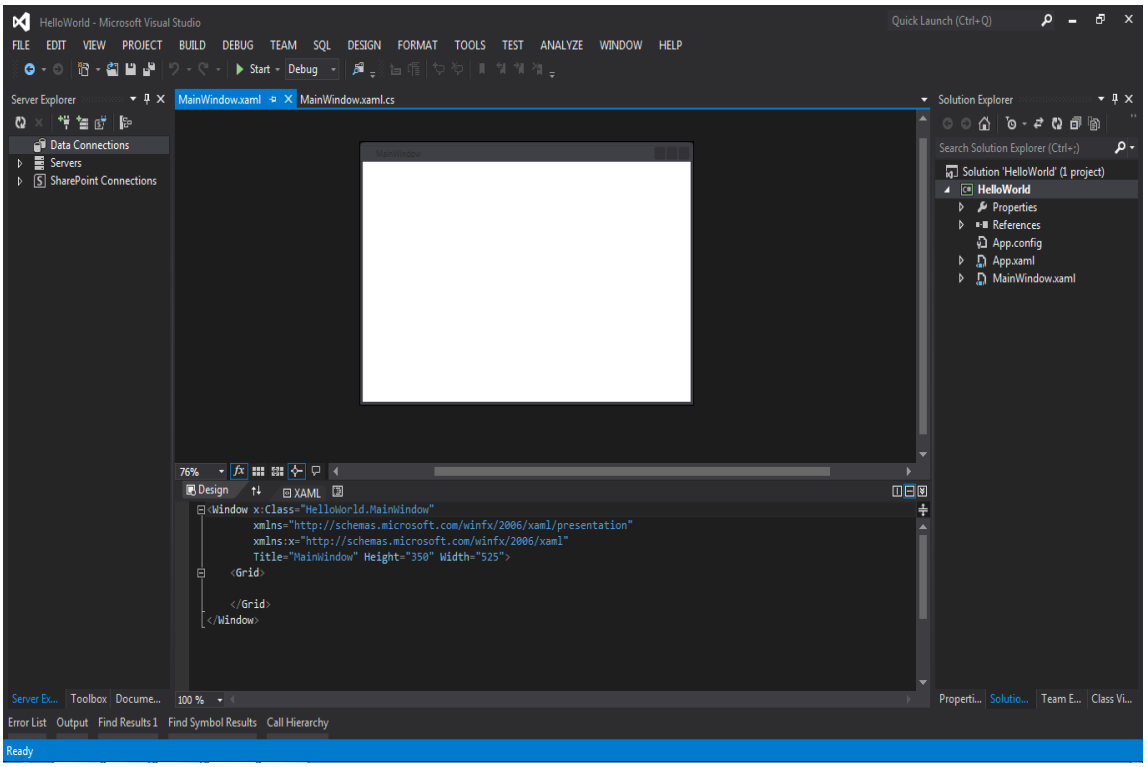
PRIMA APLICATIE WPF

In fereastra de Visual Studio selectam din meniul de sus **File -> New Project**. Va aparea o fereastra ca cea din imaginea de mai jos.



Alegem din partea stanga **Templates -> Visual C# -> Windows**

Selectam din partea centrala a ferestrei tipul proiectului – **WPF Application** – si apoi alegem calea unde vom salva proiectul si ii dam un nume (**Ex: HelloWorld**) in casetele de text din partea de jos a ferestrei, unde se cer aceste informatii.



Va aparea o fereastra ca in imaginea de mai sus, unde putem observa fisierul **MainWindow.xaml**. Aceasta este fereastra principala care se va incarca la lansarea in

executie a aplicatiei (Acest lucru se poate modifica prin schimbarea valorii atributului **StartupUri** din fisierul **App.xaml**, pe care il putem observa in fereastra **Solution Explorer**).

In continuare vom afisa un mesaj in fereastra prin adaugarea unei componente **TextBlock**, careia ii vom seta 3 atribute: **VerticalAlignment**, **HorizontalAlignment** si **FontSize**. Aceasta componenta se va introduce in interiorul grid-ului din fereastra **MainWindow.xaml**, iar codul va arata asa:

```
<Window x:Class="HelloWorld.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">
    <Grid>
        <TextBlock VerticalAlignment="Center" HorizontalAlignment="Center" FontSize="50">
            Hello world!!!
        </TextBlock>
    </Grid>
</Window>
```

Dupa executia programului (butonul F5) va aparea o fereastra pe care va scrie „Hello world!!!”

INTRODUCERE IN XAML

Dupa cum am vazut in exemplul anterior, **crearea unui control in XAML se face prin scrierea numelui acestuia in limba engleza intre paranteze ascutite**. De exemplu, pentru a crea un buton vom scrie <Button>. Acesta este un marcator (tag) XAML. Marcatorii trebuiesc intotdeauna inchisi. Inchiderea unui marcator se face fie prin adaugarea unui slash(/) la finalul marcatorului de inceput, inaintea parantezei ascutite (**Exemplu:** <Button />), fie prin adaugarea unui marcator final, diferit de marcatorul de inceput (**Exemplu:** <Button> </Button>).

Unele controale permit scrierea de text intre marcatorul de inceput si cel final (**Exemplu:** <Button> Click me</Button>)

Controalelor li se pot seta si atribute. **Acestea corespund proprietatilor controlului** (spre exemplu ale obiectului de tip Button)

Pentru a seta textul de pe un buton si grosimea scrisului vom scrie:

<Button FontWeight=”Bold” Content=”Click me” />

Proprietatea **FontWeight** indica grosimea scrisului, iar proprietatea **Content** indica textul scris pe buton.

Atributele unui control mai pot fi scrise si sub forma de marcatori „fiu” pentru marcatorul cu numele controlului. Diferenta este doar de sintaxa, nu apar si functionalitati diferite in cele doua cazuri de scriere.

```
<Button>
    <Button.FontWeight>Bold</Button.FontWeight>
    <Button.Content>Click me</Button.Content>
</Button>
```

Daca dorim sa avem text de diferite culori pe un buton, atunci vom scrie urmatorul cod:

```
<Button>
    <Button.FontWeight>Bold</Button.FontWeight>
    <Button.Content>
        <WrapPanel>
            <TextBlock Foreground=”Red”>R</TextBlock>
            <TextBlock Foreground=”Green”>G</TextBlock>
            <TextBlock Foreground=”Blue”>B</TextBlock>
        </WrapPanel>
    </Button.Content>
</Button>
```

In exemplul de mai sus am folosit un element de tip <WrapPanel> deoarece <Button.Content> nu permite decat un singur marcator fiu, si noi aveam nevoie de 3 TextBlock-uri, fiecare cu cate o culoare. Asa ca le-am inclus pe toate 3 pe un singur astfel de panel. Prescurtat, am fi putut scrie astfel:

```
<Button FontWeight =”Bold”>
    <WrapPanel>
        <TextBlock Foreground=”Red”>R</TextBlock>
        <TextBlock Foreground=”Green”>G</TextBlock>
        <TextBlock Foreground=”Blue”>B</TextBlock>
    </WrapPanel>
</Button>
```

Exact acelasi lucru il putem realiza si folosind cod C#, dar scriind mai mult cod. Acest lucru se poate face si in cazul unei aplicatii WPF, dar mai cu seama s-ar fi facut daca doream o aplicatie de tip WindowsForm.

```
Button btn = new Button();
btn.FontWeight = FontWeight.Bold;
```

```
WrapPanel pnl = new WrapPanel();
TextBlock txt = new TextBlock();
txt.Text = ”R”;
txt.Foreground = Brushes.Red;
pnl.Children.Add(txt);
txt = new TextBlock();
txt.Text = ”G”;
txt.Foreground = Brushes.Green;
pnl.Children.Add(txt);
txt = new TextBlock();
txt.Text = ”B”;
txt.Foreground = Brushes.Blue;
pnl.Children.Add(txt);
btn.Content = pnl;
mainPanel.Children.Add(btn);
```

EVENIMENTE IN XAML

Ca majoritatea framework-urilor pentru GUI, si WPF-ul permite tratarea de evenimente. Aceasta inseamna ca **toate controalele, inclusiv Window (care este derivat din clasa Control) dispun de o serie de evenimente la care pot subscrie. Subscrierea la evenimente inseamna ca aplicatia va fi notificata atunci cand se petrec aceste evenimente, iar controlul asupra caruia se produc, trebuie sa raspunda.**

Sunt mai multe tipuri de evenimente care se pot produce, dar cele mai frecvent intalnite sunt acelea care raspund la interactiunea utilizatorului cu mouse-ul sau cu tastatura. La majoritatea controalelor vom intalni evenimente ca: **KeyDown** (apasarea unei taste), **KeyUp** (eliberarea unei taste), **MouseDown** (apasarea butonului stang al mouse-ului), **MouseUp** (eliberarea butonului stang al mouse-ului), **MouseEnter** (cursorul mouse-ului intra peste un control), **MouseLeave** (cursorul mouse-ului paraseste un control), **Click**, etc.

Pentru a lega evenimentul unui control de o metoda din Code Behind (codul din spatele ferestrei) va trebui sa setam marcatorului cu numele controlului un atribut cu numele evenimentului, atribut a carei valoare va fi data de numele metodei care se va executa.

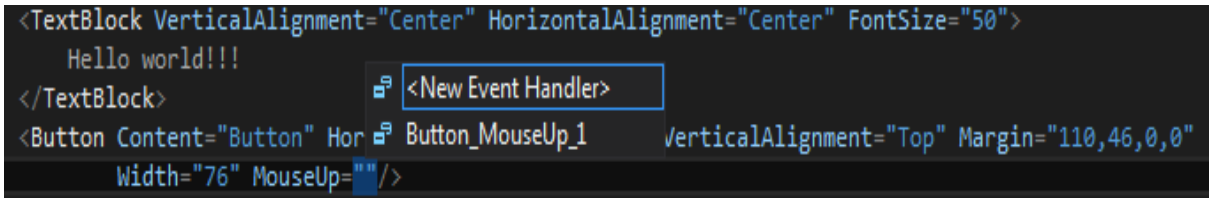
Spre exemplu, pentru a face sa apara o fereastra cu un mesaj la click pe un buton vom scrie urmatorul cod in XAML: <Button Content=”Click me” Click=”btnClick” />, unde btnClick este numele metodei din clasa C# din spatele ferestrei curente.

Atentie! Metoda trebuie sa respecte o anumita semnatura.

```
private void btnClick(object sender, RoutedEventArgs e)
{
    MessageBox.Show(”Hello World!!!”);
}
```

Evenimentul **Click** foloseste un delegat numit **RoutedEventHandler** la care va subscrie butonul. Acesta are 2 parametri: „**sender**”, care este de tip **object** si reprezinta controlul asupra caruia s-a produs evenimentul si „**e**” care este de tip **RoutedEventArgs** si care contine informatii utile cu privire la evenimentul produs.

Atunci cand scriem o metoda care trateaza un eveniment, trebuie sa avem grija ce delegat vom folosi. Spre exemplu evenimentul **Click** foloseste delegatul **RoutedEventHandler**, evenimentele **MouseUp** si **MouseDown** folosesc delegatul **MouseButtonEventHandler**, iar evenimentul **MouseMove** foloseste delegatul **MouseEventHandler**. VisualStudio ofera sprijin in acest sens, intrucat daca scriem numele evenimentului asociat controlului in XAML, IntelliSense-ul pus la dispozitie va face sa apara o fereastra de unde putem alege optiunea <New Event Handler>, iar VisualStudio va genera automat metoda cu parametrii potriviti in fisierul cu codul C#. Metoda generata automat va avea un nume implicit de forma **NumeControl_NumeEveniment(lista parametri)**. Acest lucru poate fi vizualizat in imaginea de mai jos.



Daca dorim sa facem subscrierea la un eveniment folosind doar codul C#, atunci va trebui sa **folosim operatorul „+=” pentru a adauga *handlerul de evenimente* direct obiectului**.

Dupa ce am scris metoda btnClick vom scrie, tot in clasa C#,

btn.Click += new RoutedEventHandler(btnClick);

APLICATII IN WPF

Atunci cand creem o aplicatie folosind WPF, primul lucru pe care il putem constata pe pagina este **fereastra (Window)**, insotita in codul XAML de un marcator cu acest nume, <Window>, care are un border standard, o bara de titlu si butoane pentru minimizare, maximizare si inchidere. Acest marcator este marcatorul radacina pentru toti marcatorii pe care ii vom crea in XAML.

Codul XAML arata astfel:

```
<Window x:Class="HelloWorld.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">
    <Grid>

    </Grid>
</Window>
```

Atributul **x:Class** al marcatorului Window indica numele clasei folosite de catre aceasta fereastra. În cazul nostru, clasa este MainWindow. Atributul **xmlns** reprezinta namespace-ul de unde se pot prelua numele elementelor XAML. Daca nu am defini acest atribut, cu valoarea de mai sus, nume ca *Window, Button, TextBox, TextBlock, etc...* nu ar mai fi recunoscute. Atributul **xmlns:x** defineste namespace-uri care nu sunt implicite. Daca dorim definirea unui element sau a unui atribut in acest namespace, atunci il vom califica cu prefixul **x:**. Astfel se poate extinde vocabularul XAML.

Spre exemplu, daca vom considera definitia <StackPanel x:Name=”panelName”>, va aparea urmatoarea situatie:

1. Deoarece **StackPanel** nu este calificat cu x:, va fi rezolvat de catre spatiul de nume standard dat ca valoare atributului **xmlns**
2. Atributul **x:Name** este calificat cu x:, ceea ce inseamna ca va fi rezolvat de catre spatiul de nume dat ca valoare atributului **xmlns:x**.

Deasemenea, se poate folosi atributul **xmlns** si pentru a face trimitere din XAML la **propriile spatii de nume**. Pentru aceasta va trebui sa facem trimitere la propriul spatiu de nume definind atributul **xmlns:local=”clr-namespace:NumeNamespace”** pentru marcatorul **<Window>** (se putea pune orice nume dupa „**:**” in loc de *local*, dar acesta este un nume sugestiv pentru spatiul local de nume)

Pentru a folosi clase din spatiul de nume local vom putea scrie, spre exemplu, astfel:

<local:NumeClasa x:Key=”identificator” />

Codul C# pentru clasa **Main Window** arata ca in imaginea de mai jos:

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

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

Se poate observa faptul ca aceasta clasa este declarata ca partiala (are marcatorul *partial*) deoarece este combinata cu fisierul XAML la runtime pentru a obtine fereastra completa. Acest lucru il face apelul metodei **InitializeComponent** din constructorul clasei **MainWindow**.

Revenind la fisierul XAML, vom putea observa ca a fost creat si un marcator <Grid> in interiorul lui **<Window>**. Acesta este un panel WPF si este singurul fiu al lui Window (care oricum nu poate avea decat un singur control ca si fiu). Acest **Grid** insa poate avea mai multi fii.

Putem observa ca Window mai are si alte atribute in afara de **x:Class**. Acestea ar fi:

Title – care semnifica titul scris pe bara de titlu a ferestrei

Width – reprezinta latimea ferestrei

Height – reprezinta inaltimea ferestrei

Fereastra mai poate avea si alte atribute, cum ar fi:

Icon – Permite definirea unei iconite pentru coltul stanga sus al ferestrei, si care va aparea deasemenea si in bara de task-uri, atunci cand ruleaza aplicatia si fereastra este deschisa.

ResizeMode – defineste modul in care utilizatorul poate redimensiona fereastra. Valoarea implicita a acestui atribut este *CanResize*, care permite redimensionarea ferestrei in mod obisnuit (tragand cu mouse-ul de margini sau folosind butoanele de minimizare si maximizare). Valoarea *CanMinimize* a acestui atribut permite doar minimizarea ferestrei (fara maximizare si fara a putea trage cu mouse-ul de marginile ei pentru redimensionare). Valoarea *NoResize* a atributului **ResizeMode** face sa dispara de pe bara de titlu a ferestrei butoanele de maximizare si minimizare, iar fereastra nu poate fi redimensionata nici folosind mouse-ul.

ShowInTaskbar – are valoarea implicita *True*. Daca ii setam valoarea *False*, fereastra nu va mai aparea in bara de task-uri. Aceasta proprietate este util sa fie setata pentru ferestrele care nu sunt principale sau atunci cand dorim ascunderea in System Tray a ferestrei la minimizarea acesteia.

SizeToContent – decide daca fereastra se va auto-redimensiona in functie de continutul ei. Valoarea implicita este *Manual*, care inseamna ca fereastra nu se redimensioneaza automat. Alte valori pe care le poate lua sunt: *Width*, *Height* sau *WidthAndHeight*, care vor redimensiona in mod automat fereastra pe latime, pe inaltime sau pe ambele dimensiuni.

Topmost – valoarea implicita a acestui atribut este *False*, dar daca este *True*, atunci fereastra va fi tot timpul in fata oricarei ferestre, mai putin in cazul in care este minimizata.

WindowStartupLocation – controleaza pozitia de start a ferestrei. Valoarea implicita este *Manual*, ceea ce inseamna ca fereastra va fi pozitionata implicit in functie de coltul stanga sus al ferestrei. Alte valori pe care le poate lua acest atribut sunt: *CenterOwner* sau *CenterScrren*. *CenterOwner* permite pozitionarea implicita in centrul ferestrei detinatoare, iar *CenterScreen* va pozitiona fereastra in centul ecranului.

WindowState – controleaza starea initiala a ferestrei. Aceasta poate fi *Normal*, *Maximized* sau *Minimized*. Valoarea implicita este *Normal*, care inseamna ca fereastra este deschisa in mod normal (nici maximizata, nici minimizata).

Mai sunt si alte atribute, dar acestea sunt cele mai importante dintre ele.

FISIERUL App.xaml

Atunci cand realizam o aplicatie folosind WPF se va crea in mod automat si un fisier cu acest nume, putandu-l vedea in fereastra **SolutionExplorer** din VisualStudio. **App.xaml** este punctul de start la lansarea in executie a aplicatiei, si este compus din doua fisiere (**App.xaml** si **App.xaml.cs –** unul va folosi marcatori, iar celalalt va folosi cod C#) Clasa **App** extinde clasa **Application**, care este o clasa centrala pentru aplicatiile create cu WPF. **Aceasta este prima clasa care se executa la rularea aplicatiei, si de aici se va indica ce fereastra se va deschide prima data.**

O alta trasatura importanta a acestei clase este aceea ca **aici se pot defini resursele globale ale aplicatiei**, care pot fi accesate de oriunde din aplicatie.

Structura initiala a fisierului App.xaml este urmatoarea:

```
<Application x:Class="HelloWorld.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

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

Se poate observa atributul **StartupUri** care indica fereastra principala a aplicatiei.

Daca dorim sa specificam din cod C# fereastra care sa se deschida la lansarea in executie a aplicatiei, atunci vom inlocui atributul **StartapUri** cu atributul **Startup** care va reprezenta evenimentul de Startup si va avea ca si valoare numele unei metode ce va trata acest eveniment.

Metoda se va scrie in clasa App si arata astfel:

```
public partial class App : Application
{
    private void ApplicationStart(object sender, StartupEventArgs e)
    {
        // Crearea ferestrei de start
        MainWindow window = new MainWindow();
        // Show the window
        window.Show();
    }
}
```

Numele acestei metode va reprezenta valoarea atributului **Startup** al marcatorului Application din **App.xaml**.

UTILIZAREA RESURSELOR IN APLICATIILE WPF

In WPF, datele se pot stoca sub forma de resurse. Resursele pot fi locale, pentru un anumit control, locale pentru intreaga fereastra sau globale pentru intreaga aplicatie.

Astfel, datele se vor plasa intr-un anumit loc, din care vor fi preluate pentru a fi utilizate in unul sau mai multe locuri.

In cele mai multe situatii, resursele se folosesc pentru stilizarea paginii sau pentru crearea de template-uri.

Exemple de utilizare a resurselor:

```
<Window x:Class="HelloWorld.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <sys:String x:Key="strHello">Hello, world!</sys:String>
    </Window.Resources>
    <Grid>
        <StackPanel Margin="10">
            <TextBlock Text="{StaticResource strHello}" FontSize="30" />
            <TextBlock>
                Just another "<TextBlock Text="{StaticResource strHello}" />" example, but with resources!
            </TextBlock>
        </StackPanel>
    </Grid>
</Window>
```

In exemplul de mai sus am declarat String-ul „Hello, World!” ca resursa pentru fereastra curenta. Ca sa putem folosi resursa de tip <sys:String>, va trebui sa adaugam la Window urmatorul atribut – **xmlns:sys=”clr-namespace:System;assembly=mscorlib**”. Resursele vor avea un atribut numit **x:Key** care ne va ajuta sa ne referim ulterior la acea resursa. (Il putem vedea pe acest atribut ca pe un nume dat resursei, pe care il folosim pentru a o identifica). Stringul retinut sub forma de resursa a fost folosit ulterior pe fereastra in doua locuri, sub forma de resursa statica – s-a folosit sintagma **StaticResource**.

Exista alternativa folosirii unei resurse dinamice, folosind sintagma **DynamicResource**.

Diferenta principala dintre aceste tipuri de resurse este aceea ca **resursa statica este incarcata o singura data, la incarcarea fisierului XAML** in care este definita. Daca resursa

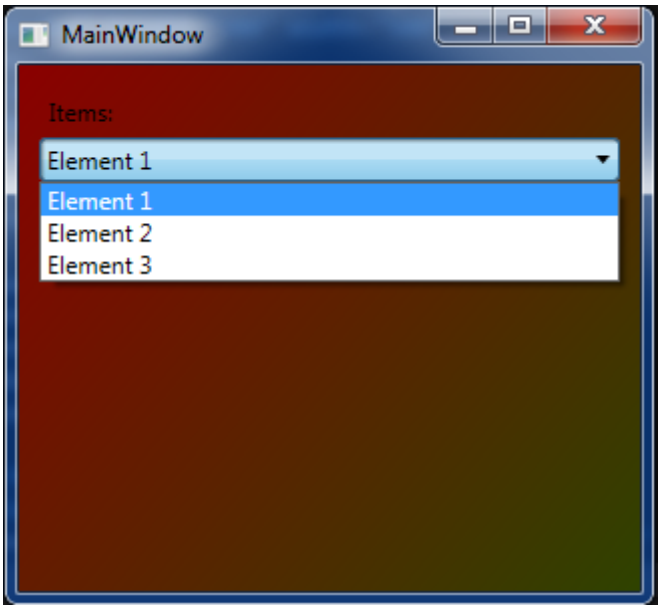
este modificata ulterior in program, aceasta modificare nu se va reflecta asupra locurilor in care apare apelata resursa folosind **StaticResource**.

Resursa apelata cu **DynamicResource este apelata doar atunci cand este nevoie de ea, si de asemenea se reapeleaza de fiecare data cand se modifica continutul ei**.

Exemplu de aplicatie care foloseste atat resursa dinamica cat si resursa statica:

```
<Window x:Class="HelloWorld.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Background="{DynamicResource WindowBackgroundBrush}"
        Title="MainWindow" Height="300" Width="500">
    <Window.Resources>
        <sys:String x:Key="ComboBoxTitle">Items:</sys:String>
        <x:Array x:Key="ComboBoxItems" Type="sys:String">
            <sys:String>Element 1</sys:String>
            <sys:String>Element 2</sys:String>
            <sys:String>Element 3</sys:String>
        </x:Array>
        <LinearGradientBrush x:Key="WindowBackgroundBrush">
            <GradientStop Offset="0" Color="DarkRed"/>
            <GradientStop Offset="1.5" Color="DarkGreen"/>
        </LinearGradientBrush>
    </Window.Resources>
    <Grid>
        <StackPanel Margin="10">
            <Label Content="{StaticResource ComboBoxTitle}" />
            <ComboBox ItemsSource="{StaticResource ComboBoxItems}" />
        </StackPanel>
    </Grid>
</Window>
```

Iar rezultatul este:



Textul de deasupra Combobox-ului este o resursa statica, identificata prin cheia **ComboBoxTitle**. Elementele din Combobox sunt reprezentate de o resursa de tip Array, identificata prin cheia **ComboBoxItems**, si apelata tot static. Fundalul ferestrei s-a realizat prin setatrea proprietatii **Background** a lui Window. Acesta a primit ca valoare o resursa apelata dinamic, identificata prin cheia **WindowBackgroundBrush**. Aceasta resursa este de tipul LinearGradientBrush.

Resursele declarate la <Window.Resources> pot fi utilizate oriunde in acea fereastra.

Putem defini resurse si numai pentru un singur control. Spre exemplu am putea muta string-ul care defineste textul de deasupra Combobox-ului ca resursa locala a StackPanel-ului, astfel:

```
<StackPanel Margin="10">
    <StackPanel.Resources>
        <sys:String x:Key="ComboBoxTitle">Items:</sys:String>
    </StackPanel.Resources>
    <Label Content="{StaticResource ComboBoxTitle}" />
    <ComboBox ItemsSource="{StaticResource ComboBoxItems}" />
</StackPanel>
```

Acum vom putea folosi aceasta resursa doar pentru fiii acestui StackPanel.

Daca dorim sa folosim o anumita resursa pe mai multe ferestre ale aplicatiei, o vom putea defini global in **App.xaml**. Astfel vom crea un marcator <Application.Resources> in care vom defini resursele in acelasi mod in care le-am definit si pentru fereastra sau pentru un control.

Am putea sa adaugam resursele si din cod C#. Spre exemplu am putea adauga un buton pe pagina, iar la click pe acel buton am putea incarca Label-ul de deasupra Combobox-ului cu textul din resursa. Vom adauga astfel metoda:

```
private void btnClickMe_Click(object sender, RoutedEventArgs e)
{
    String labelText = panel.FindResource("ComboBoxTitle").ToString();
    lbl.Content = labelText;
}
```

Aceasta metoda o vom apela la evenimentul Click al butonului. La Label, in XAML, in loc de Content=”{StaticResource ComboBoxTitle}” vom adauga proprietatea Name=”lbl”. Astfel vom putea identifica Label-ul in codul C#. Pentru a identifica StackPanel-ul ii vom pune si acestuia proprietatea **Name** cu valoarea *panel*. Din obiectul de tip StackPanel am apelat metoda **FindResource(*nume_resursa*)** care va returna resursa sub forma de obiect (**object**), in cazul in care o gaseste.

Daca apelam metoda **FindResource** dintr-un control al ferestrei, iar resursa cautata nu se gaseste, atunci aceasta este cautata mai sus in ierarhie. Mai exact, este cautata printre resursele controlului parinte, printre resursele lui Window si in ultima instanta printre resursele globale din Application.