2010

**Reflection Studio**

**Technical help**

**Guillaume WASER**

****

Reflection Studio

# Background

On my last project, I went to experience that there is no software to test performance on mobiles for NET applications. The ones working on Winform or ASP.NET site have a high performance cost because they hook everything they can.

Reflection Studio is a multi-task program around all my work with reflection. It is planned to contain 2 modules:

* **Assembly Parsing and diagrams**: If you don’t get the ultimate/architect version of studio, parse any assemblies to make diagrams
* **Performance Injection**: It can be used to get performance traces back from the execution of any NET assemblies. I will mainly target the version 4 of the framework in my testing.
* Perhaps a 3 one about **code generation from templates**

Reflection Studio is hosted on <http://reflectionstudio.codeplex.com/> . It is completely written in C# under the NET/WPF platform. I recently move to Visual Studio 2010 and NET 4

# Requirements

You will need:

* Visual Studio 2010 and Blend 4
* NET 4.0
* Some external assemblies that must be provided in the <Dependencies> folder

# Documentation

The documentation may not reflect the latest stand of my code, so if you want to implement something or go more in details, please download the latest “release”. See also my publication on <http://www.codeproject.com>

Acknowledgement

I would like to thanks all the internet contributors which help me during my work and research about NET and WPF. I hope Reflection Studio is going to help as you do to me.

Solution overview

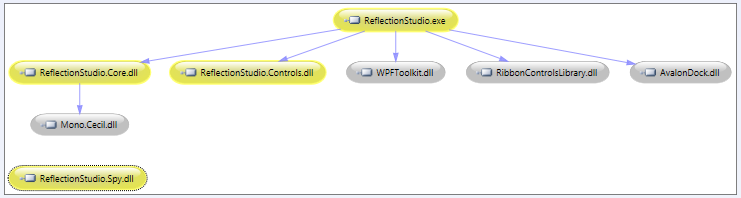
As a short introduction to the project, below is the assembly diagram. See also the Architecture chapter for more details.

In grey, the external ones:

* **Moco.cecil** from Mono project to manage assemblies
* **WPFToolkit** from Codeplex to get the datagrid, accordion controls
* **RibbonControlsLibrary** from MS as CT version - Wait for release or be replaced by "open" version
* **AvalonDock** is a nice project from codeplex to get VisualStudio like interface with docking, panels, toolbox and tabbed documents

Internally, we have 3 main assemblies and the runtime for performance traces:

* **ReflectionStudio** is the main UI with everything specific to it
* **ReflectionStudio.Controls** - all generic UI related elements
* **ReflectionStudio.Core** - all business and helpers not related to UI or controls
* **ReflectionStudio.Spy** - the runtime that make performance traces



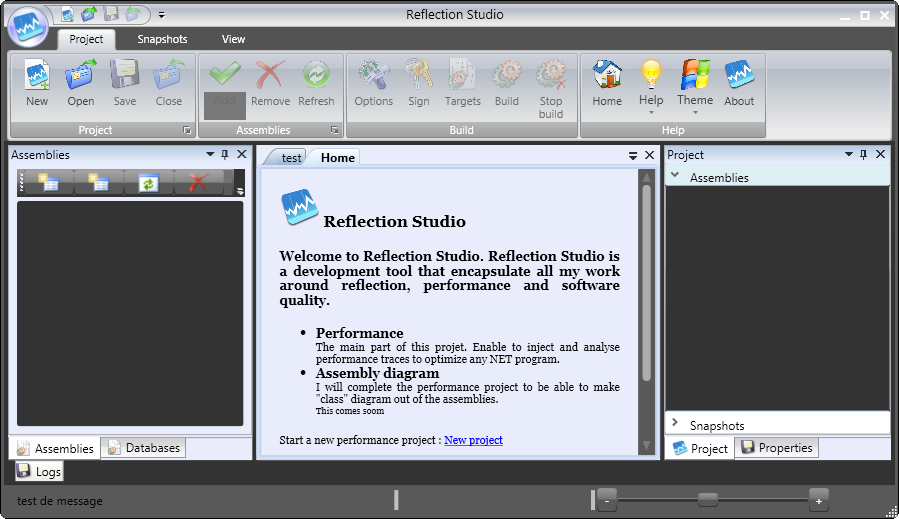
Office Style

I went in trouble for several days with the WPF ribbon and not getting the "office look" because of the rounded border, the title bar, the grip in the status bar....

After several steps forwards or backwards, I finally got a solution! The code is certainly not perfect but the style is quite good!

# How to...

The aim is to create an Office like window without border, rounded corner, gripper, a title bar, 3 standard buttons like the next one...Why do we spend our time trying to reproduce our daily "usage" and Microsoft is not giving it to us by default?



Status bar zone

The ribbon

The title zone

To achieve that, we define a customized class based on WPF window associated with a style. Here is the xaml :

<ResourceDictionary.MergedDictionaries>

<ResourceDictionary Source="/ReflectionStudio.Controls;component/Resources/Common.xaml"/>

<ResourceDictionary Source="/ReflectionStudio.Controls;component/Resources/FrameButtons.xaml"/>

</ResourceDictionary.MergedDictionaries>

<!--OFFICE STYLE WINDOW-->

<Style x:Key="{x:Type ucc:OfficeWindow}" TargetType="{x:Type ucc:OfficeWindow}">

<Setter Property="AllowsTransparency" Value="True"/>

<Setter Property="WindowStyle" Value="None"/>

<Setter Property="ResizeMode" Value="CanResizeWithGrip"/>

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="{x:Type ucc:OfficeWindow}">

<AdornerDecorator>

<Border x:Name="PART\_WindowBorder" CornerRadius="4" Focusable="False"

Background="{DynamicResource WindowBackgroundBrush}">

<Grid>

<Grid.RowDefinitions>

<RowDefinition/>

</Grid.RowDefinitions>

<Border Background="{DynamicResource WindowBackgroundBrush}"

CornerRadius="5"

VerticalAlignment="Stretch"

BorderThickness="1"

BorderBrush="Black"></Border>

<!-- Drag container for the titlebar -->

<Control x:Name="PART\_Titlebar" Grid.Row="0" Background="Transparent" Height="26"

VerticalAlignment="Top" HorizontalAlignment="Stretch" Focusable="False"

Panel.ZIndex="100" Margin="180,0,65,0">

<Control.Template>

<ControlTemplate>

<Grid>

<TextBlock

Text="{Binding Title, RelativeSource={RelativeSource FindAncestor,AncestorType={x:Type Window}}}"

HorizontalAlignment="Stretch" VerticalAlignment="Center"

Foreground="{DynamicResource WindowForegroundBrush}" Focusable="False" Margin="4"

Visibility="Hidden"

/>

</Grid>

</ControlTemplate>

</Control.Template>

</Control>

<StackPanel Grid.Row="0" Orientation="Horizontal" Margin="0,2,5,0" VerticalAlignment="top"

HorizontalAlignment="Right" Panel.ZIndex="100">

<Button Width="18" Height="18" Style="{StaticResource minimizeButton}"

x:Name="PART\_Minimize" SnapsToDevicePixels="True" IsTabStop="False" Focusable="False">

</Button>

<Button Width="18" Height="18" Margin="2,0,0,0" IsTabStop="False" Focusable="False"

Style="{StaticResource maximizeButton}" x:Name="PART\_Maximize">

</Button>

<Button Width="18" Height="18" Margin="2,0,0,0" IsTabStop="False" Focusable="False"

Style="{StaticResource closeButton}" x:Name="PART\_Close">

</Button>

</StackPanel>

<!-- Window content -->

<ContentPresenter Grid.Row="0" Margin="4"/>

<!-- Resize helpers -->

<Rectangle Grid.Row="0" Grid.Column="0" Fill="Transparent" HorizontalAlignment="Stretch" Focusable="False"

VerticalAlignment="Top" Height="10" x:Name="PART\_ResizeTop" Cursor="SizeNS"/>

<Rectangle Grid.Row="1" Grid.Column="0" Fill="Transparent" HorizontalAlignment="Stretch" Focusable="False"

VerticalAlignment="Bottom" Height="10" x:Name="PART\_ResizeBottom" Cursor="SizeNS"/>

<Rectangle Grid.Row="0" Grid.Column="0" Grid.RowSpan="2" Fill="Transparent" HorizontalAlignment="Left"

VerticalAlignment="Stretch" Width="10" x:Name="PART\_ResizeLeft" Cursor="SizeWE" Focusable="False"/>

<Rectangle Grid.Row="0" Grid.Column="0" Grid.RowSpan="2" Fill="Transparent" HorizontalAlignment="Right"

VerticalAlignment="Stretch" Width="10" x:Name="PART\_ResizeRight" Cursor="SizeWE" Focusable="False"/>

<ResizeGrip Grid.Row="1" Grid.Column="0" HorizontalAlignment="Right"

VerticalAlignment="Bottom" Width="17" Height="17" Focusable="False"

x:Name="PART\_ResizeGrip" Cursor="SizeNWSE"/>

</Grid>

</Border>

</AdornerDecorator>

<ControlTemplate.Triggers>

<Trigger Property="WindowState" Value="Maximized">

<Setter Property="CornerRadius" TargetName="PART\_WindowBorder" Value="0"/>

</Trigger>

<Trigger Property="ResizeMode" Value="CanMinimize">

<Setter TargetName="PART\_Maximize" Property="IsEnabled" Value="False"/>

</Trigger>

<Trigger Property="ResizeMode" Value="CanMinimize">

<Setter TargetName="PART\_Maximize" Property="IsEnabled" Value="False"/>

</Trigger>

<Trigger Property="ResizeMode" Value="NoResize">

<Setter TargetName="PART\_Maximize" Property="Visibility" Value="Collapsed"/>

<Setter TargetName="PART\_Minimize" Property="Visibility" Value="Collapsed"/>

<Setter TargetName="PART\_ResizeLeft" Property="Visibility" Value="Collapsed"/>

<Setter TargetName="PART\_ResizeBottom" Property="Visibility" Value="Collapsed"/>

<Setter TargetName="PART\_ResizeTop" Property="Visibility" Value="Collapsed"/>

<Setter TargetName="PART\_ResizeRight" Property="Visibility" Value="Collapsed"/>

<Setter TargetName="PART\_ResizeGrip" Property="Visibility" Value="Collapsed"/>

</Trigger>

</ControlTemplate.Triggers>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

The code behind is simple. Main things to do, is to retrieve the parts of the template to handle events: click, double-click, resize...

/// <summary>

/// Applies the control template to the window

/// </summary>

public override void OnApplyTemplate()

{

base.OnApplyTemplate();

Button minimizeButton = (Button)GetTemplateChild(MinimizeButtonPart);

Button maximizeButton = (Button)GetTemplateChild(MaximizeButtonPart);

Button closeButton = (Button)GetTemplateChild(CloseButtonPart);

closeButton.Click += OnCloseButtonClick;

minimizeButton.Click += OnMinimizeClick;

maximizeButton.Click += OnMaximizeClick;

Control titlebar = (Control)GetTemplateChild(TitleBarPart);

titlebar.MouseDown += OnTitleBarMouseDown;

titlebar.MouseDoubleClick += OnTitleBarDoubleClick;

AttachResizeRegions();

}

That's with this template that I learn several things:

* Usage of the ***TemplatePart*** in class header to allow part discovering in derived type with the ***GetTemplateChild*** method
* Usage of the ***OnApplyTemplate*** override to get the parts of my xaml to attach events

There was one big drawback under WindowsXP+7 (which seems rely on WPF) - is that a maximized window with no border style does not take the system bar into account and overlap it. I fix this with **MaximizeHelper** that rely on hooking the WM\_GETMINMAXINFO. So I do that:

protected override void OnSourceInitialized(System.EventArgs e)

{

base.OnSourceInitialized(e);

MaximizeHelper.Manage(this);

}

Next one was, at the beginning, how to get this work in an external assembly, not in the main project. Take care that you have to define a theme folder resource with a generic.xaml dictionary that contains all your xaml for style, templates...

TODO - property for the left side of the title text box, manage the 3 resizing corners (waiting for Blend4)

# Usage

To use it, create a new window - after that copy the all code or insert a reference to ReflectionStudio.Controls. Change the start and closing tag of the form, include the namespace of the OfficeWindow. Do not forget the assembly parameter if in extern dll

<Controls:OfficeWindow x:Class="ReflectionStudio.MainWindow"

xmlns:Controls="clr-namespace:ReflectionStudio.Controls;assembly=ReflectionStudio.Controls"

Title="{Binding Title}" Height="600" Width="800">

</Controls:OfficeWindow>

In the code behind, change the based class of the window. There is nothing special to manage.

public partial class MainWindow : OfficeWindow

{

………….

# Binding

To simplify the binding through all objects I have in the program, I define a **BindingView** singleton class which contains shortcuts to all binded objects. The **BindingView** class is binded to the MainWindow datacontext.

internal class BindingView : BindableObject

That’s why you can see in the xaml definition of the MainWindow the binding to **Title** which is a property of the **BindingView** that change regarding the project

Title="{Binding Title}" Height="600" Width="800">

Ribbon

As it was not easy to find documentation about the Microsoft ribbon, find bellow a synthesis off all I got on internet. Thanks to all contributors.

# The different parts

We will see in details all the ribbon parts as presented below. We will have a short look at the command definition because it must disappear in the final version as it is not compliant with actual Net commands.

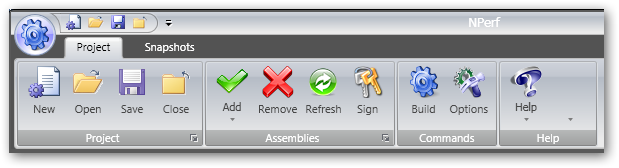
Then, how to define the application menu and the quick access toolbar followed by tabs, group, group sizing and layout, and finally the command controls.

Quick access toolbar

Customize Quick Access toolbar

Application menu

Tabs



Group dialog

Commands

Group

# Include

Include the library in the main form header

xmlns:r="clr-namespace:Microsoft.Windows.Controls.Ribbon;assembly=RibbonControlsLibrary"

In the form, define a grid with 3 rows for ribbon, controls in the middle, and status bar at bottom

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" MinHeight="142.071" />

<RowDefinition Height="\*" />

<RowDefinition Height="Auto" MinHeight="24.679" />

</Grid.RowDefinitions>

Add the xaml for the ribbon, the content, etc...

<!--RIBBON CONTROL-->

<r:Ribbon Grid.Row="0" HorizontalAlignment="Stretch" Name="ribbonMain" SelectedTabChanged="ribbonMain\_SelectedTabChanged" MouseDoubleClick="OnTitleBarDoubleClick"

MouseLeftButtonDown="OnTitleBarMouseDown"

Title="{Binding RelativeSource={RelativeSource FindAncestor,AncestorType={x:Type Window}},Path=Title}"

>

Handle the events on the ribbon to get the move because it is hiding the office window. Just plug the ribbon handler to the existing office window methods

MouseDoubleClick="OnTitleBarDoubleClick

MouseLeftButtonDown="OnTitleBarMouseDown

# Commands

As they are going to disappear because they are not compliant to the **ICommand**, here is just a short overview. Commands are defined in the Window.Resources dictionary and contain properties which are going to be mapped to the controls they bind to, so you don’t have to define the label of each controls where associate the command in a centralized manner. Here is sample:

<!-- APPLICATION COMMANDS-->

<r:RibbonCommand

x:Key="AppCommand"

LabelTitle="NPerf"

ToolTipFooterDescription="Press F1 for help"

ToolTipDescription="Click here to open, save, close or print projects"

ToolTipTitle="NPerf button"

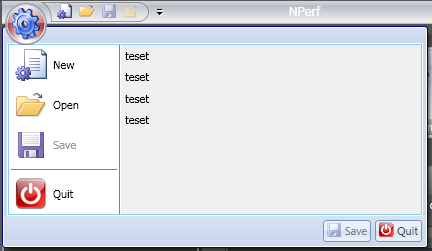
ToolTipImageSource="Resources\Images\Autres\main\_menu.png"

LargeImageSource="Resources\Images\32x32\build.png"

CanExecute="CanExecuteAppCommand"

/>

# Application menu



The application menu is the big rounded button on the left/top part of the ribbon. It can contains a tooltip, a command, a menu with commands, a recent file list

Just after the ribbon tag <r:Ribbon, define the following xaml depending your need

Define the following xaml to start the menu

<r:Ribbon.ApplicationMenu>

<r:RibbonApplicationMenu Command="{StaticResource AppCommand}">

Inside, add this for a recent item list : a binding and a selected event handler

<r:RibbonApplicationMenu.RecentItemList>

<r:RibbonHighlightingList

MostRecentFileSelected="ribbonApplicationMenu\_MostRecentFileSelected"

ItemsSource="{Binding Workspace.RecentFiles}"

DisplayMemberPath="Name" />

</r:RibbonApplicationMenu.RecentItemList>

In the code behind, define the datacontext of the window to the object that contain a “RecentFiles” property. Adjust event handler and “DisplayMemberPath”

this.DataContext = BindingView.Instance;

Then add “MenuItem” binded to command to define the main application menu

<r:RibbonApplicationMenuItem Command="{StaticResource NewProjectCommand}"/>

<r:RibbonSeparator/>

<r:RibbonApplicationMenuItem Command="{StaticResource QuitCommand}"/>

Finish the application menu by the footer definition

<r:RibbonApplicationMenu.Footer>

<DockPanel LastChildFill="False" >

<r:RibbonButton DockPanel.Dock="Right" Margin="2" Command="{StaticResource QuitCommand}" />

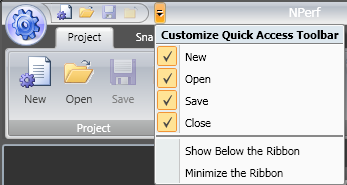
<r:RibbonButton DockPanel.Dock="Right" Margin="2" Command="{StaticResource SaveProjectCommand}" />

</DockPanel>

</r:RibbonApplicationMenu.Footer>

</r:RibbonApplicationMenu>

# Quick access toolbar



Then, you can define the Quick Access toolbar

<!-- Quick pick menu -->

<r:Ribbon.QuickAccessToolBar>

<r:RibbonQuickAccessToolBar CanUserCustomize="True">

<r:RibbonButton Command="{StaticResource NewProjectCommand}" r:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar" />

<r:RibbonButton Command="{StaticResource OpenProjectCommand}" r:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar" />

<r:RibbonButton Command="{StaticResource SaveProjectCommand}" r:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar" />

<r:RibbonButton Command="{StaticResource CloseProjectCommand}" r:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar" />

</r:RibbonQuickAccessToolBar>

</r:Ribbon.QuickAccessToolBar>

The property “**CanUserCustomize**” will display the little arrow next to the quick bar. The placement property allows defining where you want the command to appear.

# D:\_GWA\PROJECTS\NPerf\Documents\group_sizing.pngTabs, Group, Sizing an layout

When you define **Tabs**, like “Project” here, you have to add **Group** of commands.

<r:**RibbonTab** Name="ribbonTabProject" Label="Project"

GroupSizeReductionOrder="Help,Project,Commands,Assemblies">

<r:**RibbonGroup** Name="Project" HasDialogLauncher="True"

GroupSizeDefinitions="{StaticResource RibbonLayout}">

<r:RibbonGroup.Command>

<r:RibbonCommand LabelTitle="Project"/>

</r:RibbonGroup.Command>

<r:RibbonButton Command="{StaticResource NewProjectCommand}" />

<r:RibbonButton Command="{StaticResource OpenProjectCommand}" />

<r:RibbonButton Command="{StaticResource SaveProjectCommand}" />

</r:RibbonGroup>

The property “**HasDialogLauncher**” will display the little bottom/right arrow that allow to respond to a dialog corresponding to the “**Group**”

The “**GroupSizeReductionOrder**” define the group order to reduce or enlarge them when the window size is modified. It’s linked to the “**GroupSizeDefinition**”, which define different layouts to resize the commands in the group.

Themes management

Create a structure like the next one in the Resource folder of your solution.

<Resources>

<Themes>

<ThemeBlue>

Use here the xaml you need

<ThemeRed>

Use here the xaml you need

ThemeBlue.xml

ThemeRed.xml

Fill the ThemeBlue.xml by registering the xaml of the corresponding folder:

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

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

<ResourceDictionary.MergedDictionaries>

<ResourceDictionary Source="/RibbonControlsLibrary;component/Themes/Office2007Black.xaml" />

<ResourceDictionary Source="Black\ExpressionDark.xaml" />

<ResourceDictionary Source="Black\Windows.xaml" />

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

Be sure that xaml properties are Page + Do not copy + MSBuild:Compile, so that xaml will be integrated in the final assembly.

# ThemeHelper

The **ThemeHelper** class provided by ReflectionStudio.Controls helps you to discover and apply the themes defined in the main assembly. It has 2 static methods:

Discover the theme by returning a list of colour string:

static public List<String> DiscoverThemes()

Load the theme with a function like this, just by passing the required colour as a string:

static public void LoadTheme(string themeColor)

This works also for files. So you can choose to externalize the xaml, discover by code the files and folder in the <Theme> subdirectory of the application running path.

I am quite not sure about how to style generic controls in my ReflectionStudio.Controls assembly and regarding the usage of the WPF themes from the toolkit.

Has I don’t want to override all style in the application, I decided to use the resource from the WPF themes as DynamicResource in my styles and templates

UI Elements

# ImageButton

In several place, I had to use button with image. At first step, I was defining the content directly...then I decided to create a style....but I always loose the button frame...Then I found that I had to include a button into my template

<!--ImageButton Style-->

<Style x:Key="{x:Type local:ImageButton}" TargetType="{x:Type local:ImageButton}">

<Setter Property="Template" Value="{StaticResource ImageButtonTemplate}" />

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="{x:Type local:ImageButton}">

<**Button**

# Round Glass Button

From the net, this very nice template is used in the status bar slider at the moment. Note that to use it, I have to include the resource directly in the user control or style is not resolved.

<UserControl.Resources>

<ResourceDictionary>

<ResourceDictionary.MergedDictionaries>

<ResourceDictionary Source='/ReflectionStudio.Controls;component/Resources/RoundGlassButton.xaml'/>

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

</UserControl.Resources>

# Others:

You will also find in the control assembly the following controls:

* PropertyGrid
* Toolbox
* WaitSpin

Not really used and debugged actually, I will describe them later if needed.

Dialogs

# DialogWindow

After several dialogs and always copying the same code to get the same look and feel, I decide to investigate in classes and templates. I had a need for a simple rounded corner frame with a close button and a more complete one with a header containing icon, title and description. The idea was to define 2 similar templates associated with a DialogWindow derivate into HeaderedDialogWindow.

Create a new DictionnaryResource or use an existing one

In the style, first define the properties to get the wanted result:

<!-- DialogWindow Style -->

<Style x:Key="DialogWindow" TargetType="{x:Type ucd:DialogWindow}">

<Setter Property="SnapsToDevicePixels" Value="True"/>

<Setter Property="AllowsTransparency" Value="True"/>

<Setter Property="WindowStyle" Value="None"/>

<Setter Property="Background" Value="Transparent"/>

<Setter Property="ShowInTaskbar" Value="False"/>

It define a style named **DialogWindow** which only apply to the user defined **DialogWindow** type.

Then define the template: a simple grid with a rectangle as the dialog frame and a closing button. The **ContentPresenter** is where the xaml of the dialog will be.

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="{x:Type ucd:DialogWindow}">

<Grid Margin="10">

<Rectangle x:Name="windowRect" RadiusX="16" RadiusY="16" Opacity="0.8" StrokeThickness="0"

Fill="{DynamicResource WindowBackgroundBrush}">

<Rectangle.BitmapEffect>

<DropShadowBitmapEffect></DropShadowBitmapEffect>

</Rectangle.BitmapEffect>

</Rectangle>

<Button Style="{DynamicResource closeButton}" x:Name="PART\_Close" Height="11"

HorizontalAlignment="Right" Margin="0,9,11,0" VerticalAlignment="Top" Width="11"

ToolTip="Close" IsCancel="True"/>

<!-- PART\_ContentPresenter -->

<ContentPresenter x:Name="PART\_ContentPresenter"

HorizontalAlignment="Stretch"

VerticalAlignment="Stretch"

/>

</Grid>

Create a class to be associated with it. It will :

* Set the owner to the main window
* Set the StartupLocation to CenterOwner
* Handle the MouseMoveEvent and the closing when clicked on the button

public class DialogWindow : Window

{

static DialogWindow()

{

FrameworkElement.DefaultStyleKeyProperty.OverrideMetadata(

typeof(DialogWindow), new FrameworkPropertyMetadata(typeof(DialogWindow)));

}

protected override void OnInitialized(System.EventArgs e)

{

base.OnInitialized(e);

if (!DesignerProperties.GetIsInDesignMode(this))

this.WindowStartupLocation = WindowStartupLocation.CenterOwner;

}

public override void OnApplyTemplate()

{

base.OnApplyTemplate();

if (!DesignerProperties.GetIsInDesignMode(this))

{

Button close = this.Template.FindName("PART\_Close", this) as Button;

if (close != null)

close.Click += new RoutedEventHandler(close\_Click);

}

}

protected override void OnMouseLeftButtonDown(System.Windows.Input.MouseButtonEventArgs e)

{

base.OnMouseLeftButtonDown(e);

this.DragMove();

}



The trick to get the click event of a control into a template is to define is name like PART\_Close. Then it’s easy to get it in the code by using the OnApplyTemplate WPF function

<Button Style="{DynamicResource closeButton}" x:Name="**PART\_Close**"

I faced a designer problem, because each time i wanted a dialog to be opened it crashed. I discover this was due to the next instructions...sure that there is no MainWindow in design mode...

if (!DesignerProperties.GetIsInDesignMode(this))

this.WindowStartupLocation = WindowStartupLocation.CenterOwner;

To define the dialog, create classic new Window, and then import the namespace of the **DialogWindow**. Change the <Window> tag to the user defined <DialogWindow>.

<**ucd**:**DialogWindow** x:Class="NPerf.UI.Components.Dialogs.Startup.StartupDlg"

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

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

xmlns:**ucd**="clr-namespace:NPerf.UI.Components.Dialogs.Common"

Title="StartupDlg" Height="600" Width="400"

**Style="{DynamicResource DialogWindow}**" Loaded="StartupDlg\_Loaded">

See StartupDialog for an example

# HeaderedDialogWindow

The next step is to create the “headered” dialog. Like the previous one it is based on a class and a template. For the class, **HeaderedDialogWindow** will inherit from DialogWindow so we do not have to rewrite the same logic for move, closing and positioning.

The template is the same completed with the header control we develop in the previous chapter.

Define the template. First trick: we use the BasedOn property for the style. This allows us to inherit from DialogWindow style and not redefined the properties

<!-- HeaderedDialogWindow Style -->

<Style x:Key="HeaderedDialogWindow" BasedOn="{StaticResource DialogWindow}" TargetType="{x:Type ucd:HeaderedDialogWindow}">

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="{x:Type ucd:HeaderedDialogWindow}">

<Grid Margin="10">

<Grid.RowDefinitions>

<RowDefinition Height="80" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<!--Windows Frame rectangle-->

<Rectangle x:Name="windowRect" Style="{DynamicResource RectangleFrame}">

<Rectangle.BitmapEffect>

<DropShadowBitmapEffect></DropShadowBitmapEffect>

</Rectangle.BitmapEffect>

</Rectangle>

Add the header control in the template. Bind his properties so it will take the parent Title, DialogDescription and DialogImage properties

<!--Header-->

<ucc:HeaderUC Grid.Row="0" x:Name="PART\_Header" VerticalAlignment="Top" Height="70"

TitleHeader="{TemplateBinding Property=Title}"

ImageHeader="{TemplateBinding Property=DialogImage}"

DescriptionHeader="{TemplateBinding Property=DialogDescription}" />

The title property is an existing Window property. We must add the **DialogDescription** and **DialogImage** as **DependencyProperty** to the class so that we can define their values in the xaml and make binding on them. That’s the next step in the class definition

Add a new class named HeaderedDialogWindow that you will inherit from DialogWindow class like below :

public class HeaderedDialogWindow : **DialogWindow**

{

static HeaderedDialogWindow()

{

FrameworkElement.DefaultStyleKeyProperty.OverrideMetadata(

typeof(HeaderedDialogWindow), new FrameworkPropertyMetadata(typeof(HeaderedDialogWindow)));

}

public HeaderedDialogWindow() : base()

{

}

Add the dependency properties:

public static readonly DependencyProperty DialogDescriptionProperty =

DependencyProperty.Register("DialogDescription", typeof(string), typeof(HeaderedDialogWindow));

public string DialogDescription

{

get { return (string)GetValue(DialogDescriptionProperty); }

set { SetValue(DialogDescriptionProperty, value); }

}

public static readonly DependencyProperty DialogImageProperty =

DependencyProperty.Register("DialogImage", typeof(ImageSource), typeof(HeaderedDialogWindow));

public ImageSource DialogImage

{

get { return (ImageSource)GetValue(DialogImageProperty); }

set { SetValue(DialogImageProperty, value); }

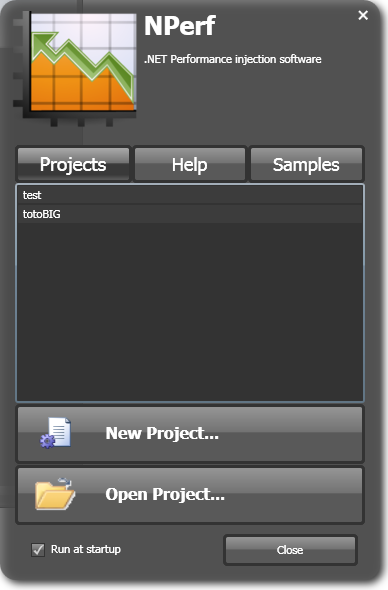
}

See Message Box for an example

Dialog samples

# StartUp

Base on blend 3, it shows at the first start of the programme. I do not manage a splashing mode...mean it does not display before the main window...it’s too much code...



Create a new standard WPF window. In the XAML:

* Change the Window tag to DialogWindow
* Include the namespace for dialogs
* Set the style

<ucd:DialogWindow x:Class="NPerf.UI.Components.Dialogs.Startup.StartupDlg"

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

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

**xmlns:ucd="clr-namespace:NPerf.UI.Components.Dialogs.Common"**

Title="StartupDlg" Height="600" Width="400"

**Style="{DynamicResource DialogWindow}"** Loaded="StartupDlg\_Loaded">

Change the Window base class to DialogWindow, include your code and your XAML

In this case the interesting point is “how to send commands” to the main window like open, recent projects...

Create a new class named StartupCommand like below. It create a new RoutedUICommand

public static class StartupCommand

{

private static RoutedUICommand \_StartupCmd =

new RoutedUICommand("StartupCommand", "StartupCommand", typeof(StartupCommand));

public static RoutedUICommand StartupCmd

{

get { return StartupCommand.\_StartupCmd; }

set { StartupCommand.\_StartupCmd = value; }

}

}

And define an enum for the startup command you want to process

public enum eStartupCommand

{

New, Open, QuickOpen, Help, HelpTechnical, Sample

}

Followed by a class representing the startup parameters you need. Here the file name will be necessary

public class StartupParameters

{

public StartupParameters(eStartupCommand cmd)

{

Command = cmd;

File = string.Empty;

}

public StartupParameters(eStartupCommand cmd, string file)

{

Command = cmd;

File = file;

}

public eStartupCommand Command

{

get;

set;

}

public string File

{

get;

set;

}

}

Define delegates like on the recent projects list. Close the dialog and fire the command by using the Execute function

private void RecentProjectList\_SelectionChanged(object sender, System.Windows.Controls.SelectionChangedEventArgs e)

{

this.Close();

StartupCommand.StartupCmd.Execute(

new StartupParameters(eStartupCommand.QuickOpen, ((RecentFileItem)RecentProjectList.SelectedItem).FilePath),

Application.Current.MainWindow);

}

In the Main Window, register the new StartupCommand with the code below

//register the startup dialog commands

Application.Current.MainWindow.CommandBindings.Add(

new CommandBinding(StartupCommand.StartupCmd, OnStartupCommand));

Then, define the callback to answer the command and just forward it to the existing command handlers

private void OnStartupCommand(object sender, ExecutedRoutedEventArgs e)

{

e.Handled = true;

StartupParameters param = (StartupParameters)e.Parameter;

switch (param.Command)

{

case eStartupCommand.New:

OnNewProject(sender, e);

break;

case eStartupCommand.Open:

OnOpenProject(sender, e);

break;

case eStartupCommand.QuickOpen:

if (ProjectManager.Instance.Open(param.File))

UpdateUI();

break;

case eStartupCommand.Help:

OnHelp(sender, e);

break;

case eStartupCommand.HelpTechnical:

OnTechnicalHelp(sender, e);

break;

case eStartupCommand.Sample:

if (ProjectManager.Instance.Open(param.File))

UpdateUI();

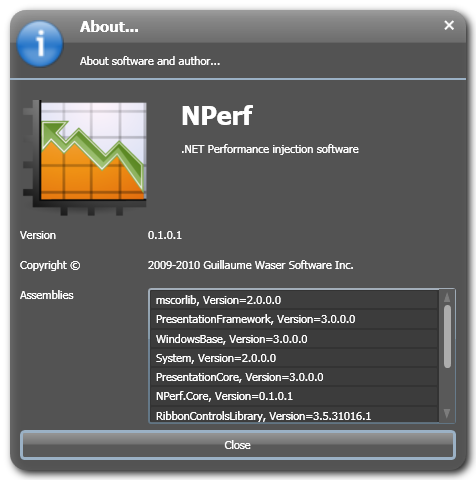
break;

}

}

# About Box

Classic “About box”. Interesting part is the binding to all assemblies and the executable version.



In the loaded event, :bind the label to the assembly version

private void Window\_Loaded(object sender, RoutedEventArgs e)

{

Assembly EntryAssembly = Assembly.GetEntryAssembly();

// fill the executable version

this.labelApplicationVersion.Content = EntryAssembly.GetName().Version.ToString();

And fill the ListBox with text as a concatenation of assembly name and version

//fill with the dependencies

this.listBoxAssembliesList.Items.Clear();

foreach (AssemblyName assembly in EntryAssembly.GetReferencedAssemblies())

{

this.listBoxAssembliesList.Items.Add(assembly.Name + ", Version=" + assembly.Version);

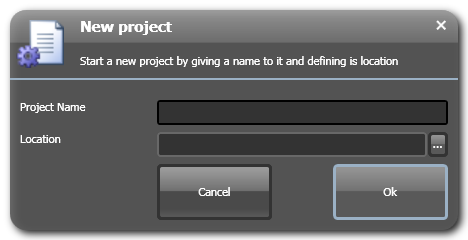
}



Use a shared AssemblyVersionInfo.cs in all your project properties, so that you will get the same version for all assemblies

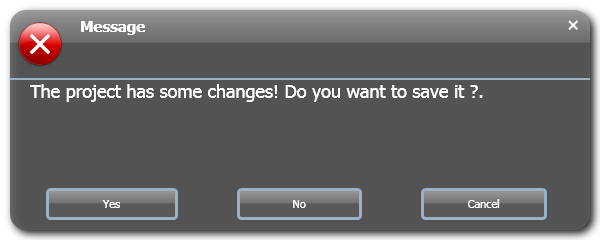
# New Project

Very simple **HeaderedDialog**



# Message box

Based on the standard NET message box, it is a **HeaderedDialog**, that contain static method like Show



public static MessageBoxResult Show(string message, string title)

{

return MessageBoxDlg.Show(message, title, MessageBoxButton.OKCancel, MessageBoxImage.None);

}

The parameters you give to the function will configure the dialog as needed :

//message

MessageBoxResult answer = MessageBoxDlg.Show(

ReflectionStudio.Properties.Resources.MSG\_PRJ\_ASK\_SAVE,

ReflectionStudio.Properties.Resources.MSG\_TITLE,

MessageBoxButton.YesNoCancel,

MessageBoxImage.Error);

Assembly parsing

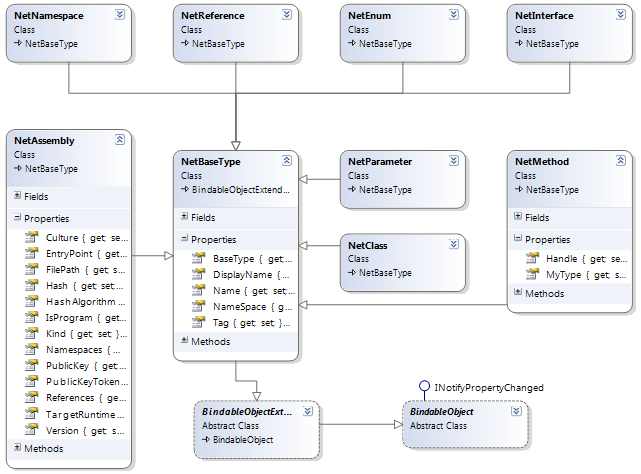
The aim was to create an engine for the performance injection trace, but i had planned to also include a "class designer". So I create  a factory which provide either a NET or CECIL based assembly parser. Why both? Because NET parsing is easy (for class designer) and CECIL is capable of injection (for performance injection).

# The model

I create a object model to be able to to fit the need of:

* Fill a treeview to display assembly content for my assembly explorer like in studio - namespace, references, all types...
* Fill a treeview for injection selection - check boxes, no namespace view
* Injection - only interested in class and methods

Here is the model:



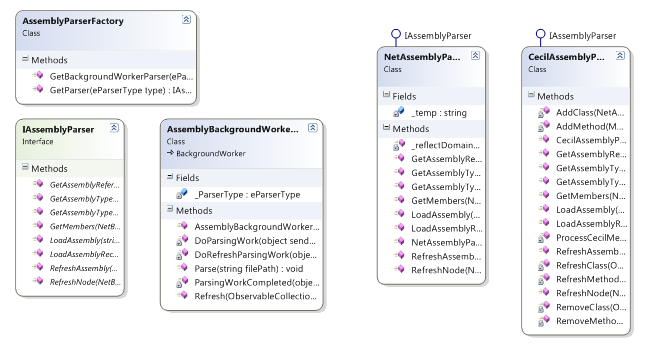
In a short way: Everything is based on **NetBaseType**. It is based on two of my core object: **BindableObject** that implement the **INotifyPropertyChanged** interface for convenience with WPF binding. **BindableObjectExtended** has got the parent+children+selection pattern pluged to it.

**NetBaseType** store in the **Tag** property the underlying cecil or net type. **Name** property contains type name and **DisplayName** is used for more human readable and shorter storage. **BaseType** contain the type it inherits from.

Everything can be serialized through my **SerializeHelper** that can process binary or XML. Note that the **Tag** property is not saved on purpose. This means you have to call refresh functions of the parser after a reload to synchronize your structure with the assembly content. This allow also to manage the changes.

## The Factory

The **AssemblyParserFactory** provides you a **GetParser**(eParserType) method where the eParserType enum is NET or CECIL and return back a **IAssemblyParser** interface. The parser is able to make full recursive or on demand node parsing through his methods. For more interactive UI, I allso provide a background worker parsing usefull in case of full recursive parsing.



## Background worker usage

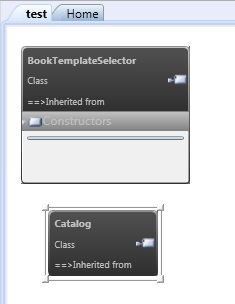
To use it, ask the factory for the type of worker you need, and then plug your ending function to the event ParsingWorkCompleted to be able to finalize the step. Here is the assembly explorer example:

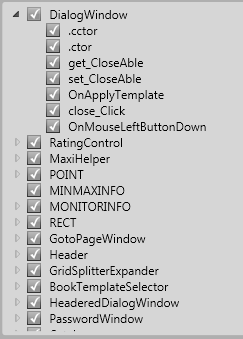
# Related controls

Related to the assembly parsing, Reflection Studio contains UI controls like an assembly explorer with treeview, a class diagram document, a check box treeview for selecting injection target...

## Assembly explorer

## Class diagram document



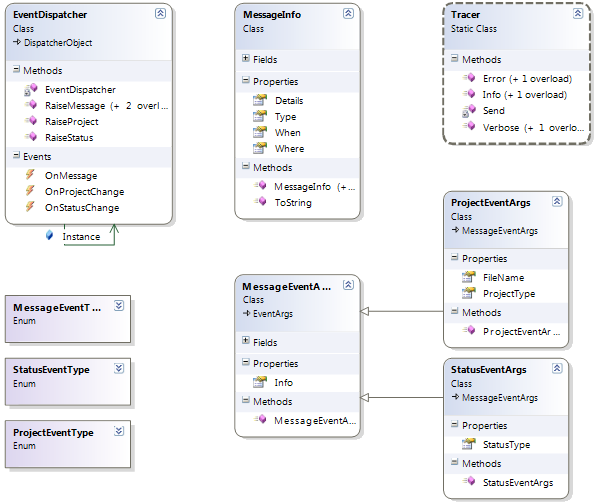


## Injection target tree

Event System / Tracing

The idea was to develop a tool that suit to the user which will use Reflection Studio and me during development but also a good way to update the interface and share events between all components. I don't like to write trace in my code, because it's un-useful since a problem not easy occurs...so I decided my traces to be visible in the UI.

# Model: EventDispatcher and Tracer

****

## EventDispatcher class

It serves in underground to the **Tracer** class with the basic messaging system. It's available to race 3 types of event (all of them send a message as the base information)

1. Message: pure text information, completed with Type, Where and When, ....
2. Status : with start or stop, allow to update the progress in the status bar
3. Project: more customized and business oriented, allow to signal events like opened, closed, saved...

Here is an example of usage to fire project and status event by using the **EventDispatcher** class, but also used by the **Tracer** one in the try/catch management.

private bool LoadProject()

{

try

{

EventDispatcher.Instance.RaiseStatus(Resources.CORE\_LOADING, StatusEventType.StartProgress);

EventDispatcher.Instance.RaiseProject(Current, ProjectEventType.Opening);

Current = new ProjectDAC().Load(Current.ProjectFilePath);

if (Current != null)

{

//refresh by assembly parsing

Refresh();

EventDispatcher.Instance.RaiseProject(Current, ProjectEventType.Opened);

return true;

}

else

{

EventDispatcher.Instance.RaiseStatus(Resources.CORE\_LOADING\_ERROR, StatusEventType.StopProgress);

return false;

}

}

catch (Exception err)

{

Tracer.Error("ProjectService.LoadProject", err);

return false;

}

finally

{

EventDispatcher.Instance.RaiseStatus(Resources.CORE\_PROJECT\_LOADED, StatusEventType.StopProgress);

}

}

## Tracer class

Allow me to trace all development steps through classical function like in the MS trace or debug system: Error, Info, and Verbose. I generally use a template like the following one and it's very useful when writing code and testing as you can see trace directly in the UI:

public bool Open( string fileName )

{

Tracer.Verbose("ProjectService:Open", "START");

try

{

}

catch (Exception err)

{

Tracer.Error("ProjectService.Open", err);

return false;

}

finally

{

Tracer.Verbose("ProjectService:Open", "END");

}

}

The main function of the **Tracer** class creates message info, send it to the MS Trace system (if configured) and then send it as a message (if it matches the workspace settings)

private static void Send(MessageEventType typ, string from, string message)

{

MessageInfo info = new MessageInfo(typ, from, message);

//trace is allways used for all levels and can be de-activated in config

Trace.TraceInformation(info.ToString());

//send message only if trace level is under from level toolbox and settings

if (info.Type <= WorkspaceService.Instance.Entity.LogLevel)

EventDispatcher.Instance.RaiseMessage(info);

}

## Trace configuration

I use a function like the next one to configure or not MS trace system. This allow me to delete the log on each start and to activate it depending on the software settings - call it in AppLoad

private void TraceConfiguration()

{

try

{

//delete the old log file

string logPath = Path.Combine(PathHelper.ApplicationPath, "ReflectionStudio.exe.log");

if (File.Exists(logPath))

File.Delete(logPath);

if (ReflectionStudio.Properties.Settings.Default.UseTraceListener)

{

//configure the trace

System.Diagnostics.Trace.AutoFlush = true;

System.Diagnostics.Trace.IndentSize = 2;

//configure the text listenner

System.Diagnostics.TraceListenerCollection listeners = System.Diagnostics.Trace.Listeners;

listeners.Add(new System.Diagnostics.TextWriterTraceListener(logPath, "LOG"));

}

}

catch (Exception error)

{

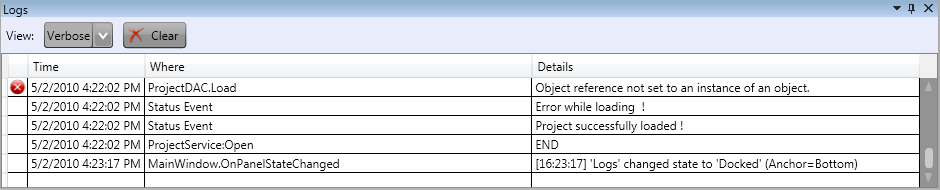
Tracer.Error("Reflection Studio.TraceConfiguration", error);

}

}

## Toolbox

To complete all this, I had to write a control to display this in the UI. I do not invent the wheel and take Visual Studio as a good sample. So I create a LogToolBox user control plugged in Avalon content. Nice was to had a converter to change the message type into an error icon!



And for all this to work, you just have to plug the events handlers all together like I do in the main window creation:

//catch events

EventDispatcher.Instance.OnProjectChange += new EventHandler<ProjectEventArgs>(OnProjectChange);

EventDispatcher.Instance.OnProjectChange += new EventHandler<ProjectEventArgs>(SnapshotService.Instance.OnProjectChange);

EventDispatcher.Instance.OnStatusChange += new EventHandler<StatusEventArgs>(this.MainStatusBar.OnStatusChange);

//forward also all changes to the log toolbox

EventDispatcher.Instance.OnMessage += new EventHandler<MessageEventArgs>(this.LogToolbox.OnMessage);

Architecture

The architecture, as presented below, is divided into 3 main blocks: The Tracer included with the injected application, the core modules, user interface. In more details, it is like

1. The user interface in WPF will be used to manage the project, build the injected assembly, view and analyse snapshot results, etc...
2. The injected assemblies will be linked to the NPerf.Tracer.Factory and the dependant modules
3. The UI through the listener modules will be able to make snapshots of the execution
4. The core module of NPerf contains the injection profiler and all
5. Some additional modules like analysis, reports, exports will also be created

ANNEXE: solution details

The solution is actually composed of 3 assemblies, one main programme and solution folders like described below:

* Solution
  + Dependencies
    - All externs assemblies referenced by other projects
  + ReflectionStudio (1, 2, 3, 4, 5)
    - This is the main programme that contains all the graphical UI and also about performance project
  + ReflectionStudio.Controls
    - Contains all the necessary graphical components i need
  + ReflectionStudio.Core
    - All the stuff needed to parse an assembly
    - All that can be factorized as generic and not specific to this application
  + ReflectionStudio.Spy (6)
    - The core stuff that will be used at runtime to get some performance traces