# NFM Project structure

## \bin

Содержит *Release* версию собранного приложения, пригодную для поставки. Краткое описание некоторых файлов приложения, соответствующих отдельным проектам (обязательные имена папок, при переименовании которых приложение не сможет корректно работать с дополнениями, выделены **жирным)**:

1. bin\**AddInSideAdapters**\Nfm.AddInSideAdapter.dll – адаптер со стороны дополнения;
2. bin\**AddInViews**\Nfm.AddInView.dll – представление дополнения;
3. bin\**Contracts**\Nfm.Contract.dll – интерфейс дополнения («контракт»);
4. bin\Nfm.Core.dll – ядро приложения;
5. bin\Nfm.Core.Tests.dll – тесты для ядра (только в *Debug* конфигурации – в папке *\test*);
6. bin\**HostSideAdapters**\Nfm.HostSideAdapter.dll – адаптер со стороны приложения;
7. bin\Nfm.HostView.dll – представление приложения (**обязательно в папке с приложением**);
8. bin\Nfm.Loader.exe – основной WPF проект (само приложение);
9. bin\ **AddIns**\ Nfm.TestViewNodeAddIn\Nfm.TestViewNodeAddIn.dll – тестовое дополнение (**обязательно каждое в своей подпапке с произвольным именем**).

Содержимое пересоздается при сборке приложения в конфигурации *Release*, поэтому папка может быть удалена при необходимости.

## \doc

Содержит проектную документацию.

## \lib

Содержит дополнительные внешние библиотеки, используемые в проектах приложения, которые также будут дополнительно скопированы в папку *\bin* при сборке проекта в конфигурации *Release* или в папку *\test* при сборке проекта в конфигурации *Debug*.

## \obj

Содержит промежуточные файлы компиляций, вынесенные из папки исходных кодов приложения. Временная папка, поэтому может быть удалена при необходимости.

## \src

Содержит исходный код всех проектов приложения *NFM*.

## \test

Содержит *Debug* версию собранного приложения, пригодную для отладки. Структура аналогична *Release*конфигурации, размещаемой в папке *\bin*. Содержимое пересоздается при сборке приложения в конфигурации *Debug*, поэтому папка может быть удалена при необходимости.

## build.cmd

Используется для сборкиприложения из командной строки в конфигурации *Release* с помощью *MSBuild*.

## Nfm.MsBuild.targets

Обязательный дополнительный файл для сборки приложения с помощью *Visual Studio 2008* в конфигурациях *Debug* или *Release*. Должен включаться в конце каждого файла проекта для выноса папки \obj из папки проекта на два уровня вверх по дереву (*..\..\obj\*) и периодического удаления папки *obj\debug\TempPE*, которую пересоздает *Visual Studio* каждый раз при загрузке проекта. Для этого необходимо выгрузить проект через контекстное меню *Unload Project,* а затем выбрать в нем пункт *Edit <Project Filename>* и добавить строчку в конце файла, сразу за строкой включения стандартного файла *Microsoft.CSharp.targets*:

<Import Project="..\..\scripts\Nfm.MsBuild.targets" />

Вынос папки *\bin* из папки проекта на два уровня вверх (*..\..\bin\* - для *Release* конфигурации и *..\..\test\* - для *Debug* конфигурации) по дереву производится дополнительно через свойства проекта в *Visual Studio*.

## Nfm.sln

Основной файл решения NFM включающий все проекты.

# Main Window

## Transparency

1. *Window.AllowTransparency = true;* // заметно ухудшает производительность
2. *Window.WindowStyle = None;*
3. *Window. Background = Transparent;*

Кроме того, окна со стилем *WindowStyle.None* имеют проблему с режимом *WindowState.Maximized* – они закрывают панель задач («full screen» режим). Существует два варианта workaround-а:

1. С использованием *WinForms* типов;
2. С использованием *P/Invoke* вызовов *Win32 API* функций.

Второй вариант реализован в проекте в классе *Legacy /MultimonitorMaximizer.cs*.

## Moving

1. Window.DragMove();

# Templates and Styles

## TabControl.ItemTemplate

Шаблон типа *DataTemplate* – формирует данные в каждом заголовке закладки.

## TabControl.ContentTemplate

Шаблон типа *DataTemplate* – формирует данные в каждой панели закладки.

## TabControl.Template

Шаблон типа *ControlTemplate* – формирует внешний вид самой основной панели закладки.

## TabControl.Style

Свойство типа *Style* – устанавливает стиль (значения свойств) для всего *TabControl*-а. Среди прочих свойств в стиле устанавливается и свойство *TabControl.Template* в вышеопределенный *ControlTemplate* для данного *TabControl*-а.

## TabControl.ItemContainerStyle

Свойство типа *Style* – устанавливает стиль для контейнерного элемента, который генерируется для каждого item-а в коллекции (т.е. это стиль для *TabItem*, в который включается так же и установка свойства *TabItem.Template* в соответствующий *ControlTemplate* для *TabItem*-а).

## TabItem.Template

Шаблон типа *ControlTemplate* – формирует внешний вид заголовка закладки.

## TabItem.Style

Свойство типа *Style* – устанавливает стиль (значения свойств) для каждого заголовка закладки в *TabControl*-е. Среди прочих свойств в стиле устанавливается и свойство *TabItem.Template* в вышеопределенный *ControlTemplate* для данного *TabItem*-а.

## TabItem.ItemsPanel

Свойство типа *ItemsPanelTemplate* – задает тип используемого контейнера (панели) для хранения элементов (закладок).

# Strings

## ToLowerInvariant() and ToUpperInvariant()

**Security Considerations**: If you need the lowercase or uppercase version of an operating system identifier, such as a file name, named pipe, or registry key, use the **ToLowerInvariant** or **ToUpperInvariant** methods.

Also note, they do provide a small – but not insignificant – performance boost (**~15%)** compared to **ToLower**/**ToUpper** methods.

Source: <http://msdn.microsoft.com/en-us/library/system.string.tolowerinvariant(VS.85).aspx>

# Layout

## IPanel

Основная единица информации. Обычно отображает один *INode*.

## IPanelContainer

Содержит коллекцию панелей.

## StackContainer

Располагает все панели друг за дружкой – горизонтально (по умолчанию) или вертикально, равномерно распределяя между ними все доступное пространство. Пользователь не может непосредственно закрыть такой контейнер через *UI*.

## TabContainer

Располагает все панели в виде *Tab*-ов, с помощью *TabControl*-а. Такой контейнер может быть закрыт непосредственно – с помощью крестика в правой верхней части *TabControl*-а. Кроме того, каждая дочерняя панель может быть закрыта непосредственно – с помощью крестика соответствующего *Tab*-а.

# Focus

Статья в MSDN: <http://msdn.microsoft.com/en-us/aa969768.aspx>

Еще один полезный цикл статей от Mark Smith’а: <http://www.julmar.com/blog/mark/PermaLink,guid,507386bd-a72e-455e-b345-315e0dcf35e9.aspx>

## FocusVisualStyle

Класс *FocusManager* отвечает за работу с логическим фокусом, а класс *Keyboard* – за работу с фокусом клавиатуры; для того, что бы убрать точечную рамку с элемента, когда он находится в фокусе клавиатуры, необходимо для данного элемента сбросить стиль **FocusVisualStyle**, например:

<ListViewItem Content="ListViewItem" **FocusVisualStyle= "{x:Null}"** />

## Keyboard

Для возможности получения клавиатурного фокуса оба свойства должны быть установлены:

UIElement.Focusable – это свойство необходимо снимать со всех элементов, которые не должны получать фокус клавиатуры!

UIElement.IsVisible

Так же иногда используются:

IInputElement.IsKeyboardFocused

IInputElement.IsKeyboardFocusWithin

и статические:

Keyboard.FocusedElement – получить текущий элемент с клавиатурным фокусом

Keyboard.Focus(…) – установить клавиатурный фокус на определенный элемент

А так же события:

Keyboard.PreviewGotKeyboardFocus

Keyboard.GotKeyboardFocus

Keyboard.PreviewLostKeyboardFocus

Keyboard.LostKeyboardFocus

## Logical

Control.TabIndex

Control.IsTabStop

UIElement.IsFocused

и статические:

FocusManager.FocusedElement – управляет логическим фокусом в пределах *focus scope*;

FocusManager.IsFocusScope – присоединяется к элементу в *XAML*, что бы превратить его в *focus scope*;

FocusManager.SetIsFocusScope(…) – превращает элемент в *focus scope* из кода;

FocusManager.GetFocusScope(…) – возвращает ближайший *focus scope* для элемента;

По умолчанию focus scope установлен для Window, MenuItem, ToolBar и ContextMenu.

FocusManager.GetFocusedElement(…) – возвращает элемент с логическим фокусом для заданного *focus scope*;

FocusManager.SetFocusedElement(…) – устанавливает логический фокус на элемент в заданном *focus scope*;

А так же события:

UIElement.GotFocus

UIElement.LostFocus

## Keyboard Navigation

KeyboardNavigation.TabNavigation – меняет фокус по TAB-у:

1. **Continue** - each focusable element receives focus and the container is exited when the edge is reached.
2. **Cycle** - focus does not leave the container but wraps around the edges
3. **Once** - the container itself is treated as a single focusable element where only the first child receives focus
4. **Local** - uses TabIndex locally within the container - independant of any outside elements.
5. **Contained** - focus statys in the container but does not wrap (stays at edges when top/bottom are reached)
6. **None** - no keyboard navigation allowed in the container

The default is **Continue**, but you can set the attached property on any element to change it for that element and any children.

KeyboardNavigation.ControlTabNavigation – меняет фокус по Ctrl+TAB-у

KeyboardNavigation.DirectionalNavigation – меняет фокус по нажатию стрелок управления курсором

## Programming Navigation

UIElement.MoveFocus(…) – устанавливает фокус

UIElement.PredictFocus(…) – предсказывает следующий элемент, который получит фокус

# How to Debug Data Binding

## Listeners in app.config

Добавить в app.config из проекта Nfm.Loader секцию:

<system.diagnostics>

<sources>

<!--

<source name="System.Windows.Data" switchName="SourceSwitch" >

<listeners>

<add name="textListener" />

</listeners>

</source>

-->

<!--

<source name="System.Windows.DependencyProperty" switchName="SourceSwitch" >

<listeners>

<add name="textListener" />

</listeners>

</source>

-->

<!--

<source name="System.Windows.Freezable" switchName="SourceSwitch" >

<listeners>

<add name="textListener" />

</listeners>

</source>

-->

<!--

<source name="System.Windows.RoutedEvent" switchName="SourceSwitch" >

<listeners>

<add name="textListener" />

</listeners>

</source>

-->

<!--

<source name="System.Windows.Media.Animation" switchName="SourceSwitch" >

<listeners>

<add name="textListener" />

</listeners>

</source>

-->

<!--

<source name="System.Windows.NameScope" switchName="SourceSwitch" >

<listeners>

<add name="textListener" />

</listeners>

</source>

-->

<!--

<source name="System.Windows.ResourceDictionary" switchName="SourceSwitch" >

<listeners>

<add name="textListener" />

</listeners>

</source>

-->

<!--

<source name="System.Windows.Markup" switchName="SourceSwitch" >

<listeners>

<add name="textListener" />

</listeners>

</source>

-->

<!--

<source name="System.Windows.Documents" switchName="SourceSwitch" >

<listeners>

<add name="textListener" />

</listeners>

</source>

-->

</sources>

<switches>

<!--

<add name="SourceSwitch" value="All" />

-->

<!--add name="SourceSwitch" value="Off" -->

<!--add name="SourceSwitch" value="Verbose" -->

<!--add name="SourceSwitch" value="Warning" -->

<!--add name="SourceSwitch" value="Activity" -->

</switches>

<sharedListeners>

<!-- This listener sends output to the console -->

<!--

<add name="console"

type="System.Diagnostics.ConsoleTraceListener"

initializeData="false"/>

-->

<!-- This listener sends output to an Xml file named TraceLog.xml -->

<!--

<add name="xmlListener"

type="System.Diagnostics.XmlWriterTraceListener"

traceOutputOptions="None"

initializeData="TraceLog.xml" />

-->

<!-- This listener sends output to a file named TraceLog.txt -->

<!--

<add name="textListener"

type="System.Diagnostics.TextWriterTraceListener"

initializeData="TraceLog.txt" />

-->

</sharedListeners>

<trace autoflush="true" indentsize="4"></trace>

</system.diagnostics>

Затем раз комментировать необходимые секции.

## DebugConvertor and DebugBindingExtension

Использовать вспомогательный класс-конвертор, в котором поставить точки останова на операциях *Conver(…)* и *ConvertBack(…)*.

## PresentationTraceSources.TraceLevel

В *xaml* файле у нужного *binding* использовать конструкцию:

{Binding NotifyOnTargetUpdated=True, NotifyOnSourceUpdated=True, Converter={converters:DebugBinding}, diagnostics:PresentationTraceSources.TraceLevel=High}

Так же потребуется добавить в заголовок *xaml* файла диагностический *namespace*:

xmlns:diagnostics="clr-namespace:System.Diagnostics;assembly=WindowsBase"

# Object cloning (ICloneable)

*NET Framework Design* Guidelines *state that you* ***shouldn't use it***. Аргументируется это тем, что единственный метод этого интерфейса – *Clone()* – никак не уточняет, какой тип копирования используется – *deep* или *shallow* и таким образом клиенту необходимо знать, как именно работает реализуемое им клонирование посредством этого интерфейса, что не всегда является возможным.

Источник: <http://blogs.msdn.com/brada/archive/2003/04/09/49935.aspx>

Как рекомендуется клонировать объекты:

<http://www.agiledeveloper.com/articles/cloning072002.htm>

<http://kristofverbiest.blogspot.com/2008/10/type-safe-pattern-to-implement.html>

<http://kristofverbiest.blogspot.com/2008/10/type-safe-pattern-to-implement.html>

Так же *shallow* копирование объектов в .NET реализовано на базовом уровне с помощью:

protected Object.MemberwiseClone();

# Drag & Drop

Реализуется с помощью *attached properties*, что бы избежать необходимости использовать *code behind* файлы и иметь возможность объявлять его декларативно прямо в *xaml* файлах.

## IDragSourceAdvisor

Описывает то, что можно будет «потянуть».

## IDropTargetAdvisor

Описывает то, куда можно будет «бросить» то, что «потянули».

## DragDropManager

Работает через два вышеописанных интерфейса с помощью *attached properties*.

Порядок вызовов ключевых методов:

IDragSourceAdvisor.**GetDataObject**();

IDropTargetAdvisor.**OnDropAccepted**();

IDragSourceAdvisor.**OnDropConfirmed**();

## PanelDragSourceAdvisor

Реализация интерфейса *IDragSourceAdvisor*, которая использует *FrameworkElement.DataContext* для получения хранящегося там объекта *IPanel* и передачи его как данные через *System.Windows.IDataObject*.

## PanelContainerDropTargetAdvisor

Реализация интерфейса *IDropTargetAdvisor*, которая использует *FrameworkElement*.*DataContext* для получения хранящегося там объекта *IPanelContainer*, что бы добавить в его коллекцию *Childs* полученный от брошенного элемента объект *IPanel*, извлеченный из *System.Windows.IDataObject*.

# ControlTemplate & SnapsToDevicePixels

Корнем любого *ControlTemplate-*а рекомендуется использовать *Border* или *Chrome.* У этого корневого элемента также необходимо устанавливать свойство *SnapsToDevicePixels* в значение *True* для удаления искажений границ этого элемента после его рендеринга.

# Themes and Skins

Ядро проекта включает две стандартных темы – *Dark* и *Light*. Тема *Dark* является темой «по умолчанию». Дополнительные темы должны размещаться в отдельных dll-сборках и подкладываться в папку к проекту (возможно в отдельную подпапку themes или skins – TBD). Каждая такая дополнительная сборка с темой(ами) будет вынуждена ссылаться на сборку ядра проекта, поскольку в ней будут переопределять «системные» стили – см. ниже.

В исходном проекте ядра должна быть папка *Themes*, в которой обязательно наличие файла *Generic.xaml*, который представляет собой «системные ресурсы» сборки, содержащие стили с шаблонами элементов управления и шаблоны данных «по умолчанию». Сам файл *Generic.xaml* содержит только ссылки на отдельные файлы ресурсов со стилями и шаблонами, объединяемые в нем с помощью *MergedDictionaries*. Кроме того файл *Generic.xaml* должен обязательно включать и тему «по умолчанию» (*Dark*). Все ресурсы, входящие в «тему по умолчанию» и объявленные в сборке ядра должны использовать ключ *ComponentResourceKey*, вместо стандартного строкового ключа. Это необходимо для того, что бы в главном приложении объекты шаблонов «по умолчанию» (которые тоже описаны в сборке ядра) смогли получить к ним доступ, поскольку *DynamicResource* не осуществляет поиск ресурсов по строковому ключу в системных ресурсах (т.е. файле *Generic.xaml*). Так же эти компонентные ключи будут использоваться для переопределения в сборках с дополнительными темами.

Смена темы заключается в добавлении в коллекцию *Application.Current.Resources.MergedDictionaries* необходимого словаря с произвольной темой:

var newSkin = Application.LoadComponent(new Uri("/Nfm.RedThemeLibrary;component/Themes/Red.xaml", UriKind.Relative)) as ResourceDictionary;

if (newSkin != null)

{

Collection<ResourceDictionary> dictionaries = Application.Current.Resources.MergedDictionaries;

if (dictionaries.Count > 0)

{

dictionaries.Clear();

}

dictionaries.Add(newSkin);

}

Очистка коллекции *Application.Current.Resources.MergedDictionaries* приведет к установке темы «по умолчанию» (*Dark*).

Основы: <http://wangmo.wordpress.com/2007/09/28/make-a-theme-changable-application/>

# Resources

Для того, что бы избегать многократной загрузки словаря ресурсов (из сборкок) каждый раз при его слиянии со словарем ресурсов некоторого элемента управления, следует использовать специальный кэш загруженных словарей, который подключать к нужному элементу через *attached property* (см. класс *ResourceCache.cs*). Более подробная информация:

<http://www.drwpf.com/blog/Home/tabid/36/EntryID/10/Default.aspx>

<http://www.codeplex.com/CompositeExtensions/Thread/View.aspx?ThreadId=42919>