# Замануха

Есть две статьи на Хабре, касающиеся страничной навигации в WPF

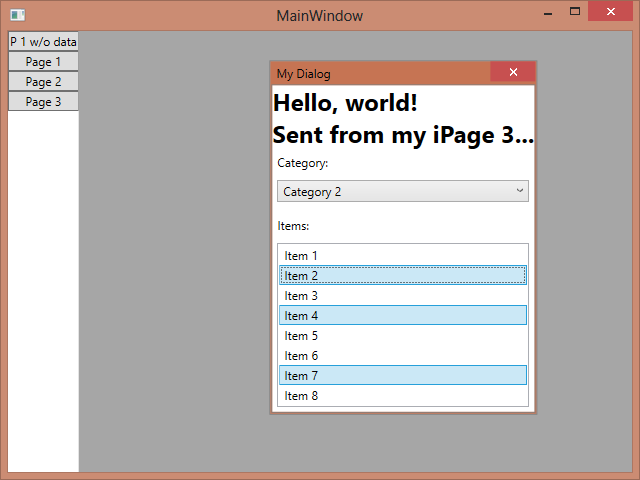
* <http://habrahabr.ru/post/63767/> - Добавление контролов в NavigationWindow
* <http://habrahabr.ru/post/140749/> - Используем связку Page + Frame

В этих статьях рассказывается в основном о создании контролов и совсем немного о реализации самой навигации. Этой статьей хочется восполнить данный пробел и представить готовое решение навигации для WPF приложения на основе паттерна MVVM.

# Введение

Те, кто уже работал с WPF, наверняка знакомы с паттерном MVVM (если нет, то в конце дал ссылки на его описание). Концепция MVVM несложна и как минимум интуитивно должен быть понятным профит от использования данного паттерна. Чтобы паттерн проявил себя во всей красе, нужно как можно меньше логики помещать в Code Behind контролов и ни в коем случае не использовать прямые ссылки на UI внутри ViewModel’ей. Тогда мы получим большой профит в виде возможности тестирования ViewModel’ей отдельно от контролов. Так же хорошей практикой будет сведение к минимуму инстанциирования ViewModel’ей напрямую в контролах. Полбеды, если контрол сам для себя создает ViewModel конкретного типа – в этом случае просто труднее будет подсунуть контролу какую-нибудь тестовую куклу. А вот когда некий родительский контрол занимается созданием ViewModel’ей для разных частных экранов, код обычно превращается в нетестируемую кучу спагетти. Лучше, если созданием ViewModel’ей будут заниматься другие ViewModel’и.

Как это может выглядеть на практике? Давайте представим себе приложение с панелью навигации и несколькими экранами и диалоговыми окнами. Нечто подобное представлено ниже.



Мы можем разглядеть несколько сущностей: главное окно, панель навигации с кнопками, текущая страница и диалог над этой страницей. В нашем приложении для страничной навигации можно было бы использовать HyperLink, подложив вместо кнопок TextBlock с HyperLink в качестве контента. У HyperLink есть свойство, указывающее имя Frame, в котором выполнять переход на новую страницу. И вроде все нормально, можно использовать. Но с использованием HyperLink представляется трудным передача странице нужной ViewModel’и.

Я видел в сети решение, когда в событии Navigated фрейма в главном окне приложения через Code Behind добирались до загруженного во фрейм содержимого и подкидывали туда созданную там же в Code Behind ViewModel. Таким образом, в одном обработчике события придется реализовать длинную портянку if…else if… либо switch. Про то, что тестирование такого Hard Coded процесса навигации крайне трудно автоматизировать, я молчу. Где-то я даже видел создание экземпляра Page и ViewModel’и под нее, подкладывание ViewModel’и в DataContext и вызов Navigate у фрейма с передачей созданного экземпляра Page. Это решение немного лучше предыдущего, но по-прежнему совсем не MVVM-way. Потому код не привожу, чтоб не дай Бог не скопипастили.

Какое решение могло бы более-менее гладко вписаться в контекст MVVM? У класса Page в Silverlight есть перегружаемый метод OnNavigatedTo. В этом методе было бы удобно принимать ViewModel, переданную в NavigationService.Navigate(Uri uri, object navigationContext) вторым параметром. Однако в WPF такого метода нет (ну или я не нашел его или чего-то эквивалентного). Таким образом, нам нужен некий посредник или, если хотите, менеджер, который будет контролировать переходы по страницам и перекладывать из параметра метода в DataContext нужную ViewModel.

О реализации такого менеджера навигации и пойдет речь в данной статье. В следующем разделе я расскажу о реализации ядра решения, о менеджере навигации. В следующих разделах я расскажу о том, что нужно реализовать на UI и ViewModel слоях. Таким образом, для экономии времени можете прочитать раздел «Менеджер навигации», а остальное додумать по ходу решения своих задач будет делом техники.

[Код выложен на GITHUB](https://github.com/homoluden/MVVM-Navigation).

# Менеджер навигации

Этот менеджер реализован в виде синглтона с двойной проверкой экземляра на null (так называемый Double-Check Locking Singleton, многопоточная версия синглтона). Про реализации синглтонов можете почитать [на странице MSDN](http://msdn.microsoft.com/en-us/library/ff650316.aspx). Аксессор экземпляра синглтона сделан приватным, чтобы лишнего не торчало наружу, но можно его и открыть.

Код реализации синглтона смотрите ниже.

#region Singleton

private static volatile Navigation instance;

private static object syncRoot = new Object();

private Navigation() { }

private static Navigation Instance

{

get

{

if (instance == null)

{

lock (syncRoot)

{

if (instance == null)

instance = new Navigation();

}

}

return instance;

}

}

#endregion

Вместо приватного экземпляра наружу выставлен статический аксессор сервиса навигации (NavigationService), который перенаправляет геттер и сеттер в скрытый экземпляр синглтона. Можно было сделать наоборот, но тогда бы все вызовы снаружи приходилось делать через экземпляр синглтона, т.е.

Navigation.Instance.Service

вместо

Navigation.Service

Мне кажется последний вариант пока проще, но он требует дополнительной реализации статических свойств и методов. Поэтому с реализацией нового функционала может стать выгоднее открыть свойство экземпляра (Navigation.Instance).

Теперь перейдем к самой функциональности менеджера.

Navigation.Service

Это свойство типа NavigationService. Предполагается, что при старте приложения в это свойство будет положена ссылка на **NavigationService** главного фрейма приложения. Через этот экземпляр сервиса навигации и будут выполняться все переходы по страницам. Кстати говоря, необязательно экземпляр задавать только при старте приложения. Можно, к примеру, «натравливать» менеджер на фрейм какого-либо диалогового окна или ChildWindow из WPF Toolkit Extended. Этот функционал расширить несложно.

Пока же наш менеджер работает предельно просто. В сеттере сервиса навигации он помимо присвоения нового значения внутреннему полу делает отписку и подписку на событие **Navigated** сервиса.

public static NavigationService Service

{

get { return Instance.\_navService; }

set

{

if (Instance.\_navService != null)

{

Instance.\_navService.Navigated -= Instance.\_navService\_Navigated;

}

Instance.\_navService = value;

Instance.\_navService.Navigated += Instance.\_navService\_Navigated;

}

}

Как видно, если поле сервиса уже было выставлено ранее, то сначала делается отписка от его события, чтобы не держать экземпляр сервиса в памяти обработчиком в синглтоне. Далее идут присвоение и подписка на событие нового экземпляра. По-хорошему, в сеттере (и публичных методах менеджера тоже) не хватает использования **lock**. Но вообще, если у Вас в приложении параллельно с вызовом какого-либо метода навигации будет производиться замена **NavigationService**, то, скорее всего, что-то реализовано некорректно. Пока для простоты обойдемся без **lock**, но я Вас предупредил ;)=/

Публичные методы навигации, упомянутые выше, представлены следующими методами.

#region Public Methods

public static void Navigate(Page page, object context)

{

if (Instance.\_navService == null || page == null)

{

return;

}

Instance.\_navService.Navigate(page, context);

}

public static void Navigate(Page page)

{

Navigate(page, null);

}

public static void Navigate(Uri uri, object context)

{

if (Instance.\_navService == null || uri == null)

{

return;

}

Instance.\_navService.Navigate(uri, context);

}

public static void Navigate(Uri uri)

{

Navigate(uri, null);

}

#endregion

Легко заметить, что это просто отражение подмножества навигационных методов **NavigationService**. Этого вполне достаточно для выполнения переходов с передачей контекста данных. Остается только подложить этот контекст в свойство DataContext целевой страницы.

#region Private Methods

void \_navService\_Navigated(object sender, NavigationEventArgs e)

{

var page = e.Content as Page;

if (page == null)

{

return;

}

page.DataContext = e.ExtraData;

}

#endregion

Как видно из кода выше, в обработчике события **Navigated** делается попытка приведения контента фрейма к типу Page. *ЕЩЕ РАЗ ПРОШУ КАПСОМ: не используйте оператор «****as»*** *без последующей проверки на null (почему так разъясню в комментариях, если нужно).* Таким образом, обработаны будут только переходы на Page. Все остальные отфильтруются. Если хотите, то можете убрать этот «железный занавес».

Если приведение типа прошло успешно, то нам остается закинуть переданный контекст данных в целевую страницу. В свойстве **ExtraData** аргументов события будет лежать объект, переданный в метод **Navigate** вторым параметром.

С менеджером, вроде бы, все. Если есть силы и время, переходите к следующим разделам с описанием реализации UI и ViewModel’ей. Если нет, то ниже кратко изложу, что нужно еще сделать.

Осталось создать сборку с реализацией страниц и сборку ViewModel’ей. Также я реализовал сборку Helpers, в которой я разместил код реализации RelayCommand для ViewModel’ей. За основу взята [реализация с сайта snipplr.com](http://snipplr.com/view/13642/). Эта реализация упрощает реализацию биндинга команд между UI и ViewModel.

Для каждой страницы я создал отдельную ViewModel. Эти, «частные», ViewModel’и инстанциируются в их родительской ViewModel, я назвал ее MainViewModel. Эта главная ViewModel хранится в DataContext главного окна, но с таким же успехом ее можно было бы определить в виде статического ресурса XAML в ресурсах главного окна или даже на уровне всего приложения. В таком случае в биндингах нужно будет указывать что-то типа Source={StaticResource MainViewModelDataSourceKey}. Но зато не придется париться о том, наследуется ли в нужном месте DataContext логического родителя.

В MainViewModel я создал несколько команд. Одну для перехода по указанному в CommandParameter строковому пути (переход без передачи контекста данных). Другие команды содержат в их делегате Execute захардкоженный адрес целевой страницы, а через CommandParameter они принимают контекст данных. Вроде бы есть еще CommandTarget свойство у кнопок, поэтому может получиться обойтись и без хард-кода адреса. Поисследуйте, если есть время и желание.

Вот и все вкратце. За деталями можете перейти [на github](https://github.com/homoluden/MVVM-Navigation).

# Сборка ViewModels

В этой сборке представлена базовая ViewModel, которая реализует **INotifyPropertyChanged**.

public class BaseViewModel : INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged;

protected void RaisePropertyChanged(string propertyName)

{

if (string.IsNullOrWhiteSpace(propertyName))

{

return;

}

if (PropertyChanged != null)

{

PropertyChanged(this, new PropertyChangedEventArgs(propertyName));

}

}

}

Остальные ViewModel’и наследуются от нее. Частные ViewModel предельно просты. Они содержат одно строковое поле с уникальным именем.

public class Page1ViewModel : BaseViewModel

{

public string Page1Text

{

get { return "Hello, world!\nSent from my iPage 1..."; }

}

}

Обратите внимание, здесь свойство только для чтения и без вызова RaisePropertyChanged(…) где бы то ни было. На практике такие свойства ViewModel’ей встречаются нечасто, т.к. биндинг на таких свойствах сработает лишь раз. Даже если я добавлю сеттер без RaisePropertyChanged(…), то биндинг все равно будет «одноразовым».

MainViewModel уже значительно сложнее. Как я написал кратко в предыдущем разделе, она будет хранить частные ViewModel’и и реализовывать команды навигации. В моем случае частные ViewModel’и создаются лишь раз, при инициализации MainViewModel. Пожтому я реализовал только геттеры этих ViewModel’ей.

public Page1ViewModel Page1ViewModel

{

get { return \_p1ViewModel; }

}

public Page2ViewModel Page2ViewModel

{

get { return \_p2ViewModel; }

}

public Page3ViewModel Page3ViewModel

{

get { return \_p3ViewModel; }

}

Поля инициализируются в местах их объявления:

private Page1ViewModel \_p1ViewModel = new Page1ViewModel();

private Page2ViewModel \_p2ViewModel = new Page2ViewModel();

private Page3ViewModel \_p3ViewModel = new Page3ViewModel();

Вам может понадобиться сделать полную реализацию полей и свойств для частных ViewModel’ей. Это уже делается элементарно.

Команды реализованы уже «по полной программе»: свойства с геттерами и сеттерами, инициализация экземпляров в отдельной функции.

public RelayCommand<string> GoToPathCommand

{

get { return \_goToPathCommand; }

set

{

\_goToPathCommand = value;

RaisePropertyChanged("GoToPathCommand");

}

}

public RelayCommand<Page1ViewModel> GoToPage1Command

{

get

{

return \_goToPage1Command;

}

set

{

\_goToPage1Command = value;

RaisePropertyChanged("GoToPage1Command");

}

}

private void InitializeCommands()

{

GoToPathCommand = new RelayCommand<string>(GoToPathCommandExecute);

GoToPage1Command = new RelayCommand<Page1ViewModel>(GoToPage1CommandExecute);

GoToPage2Command = new RelayCommand<Page2ViewModel>(GoToPage2CommandExecute);

GoToPage3Command = new RelayCommand<Page3ViewModel>(GoToPage3CommandExecute);

}

private void GoToPathCommandExecute(string path)

{

if (string.IsNullOrWhiteSpace(path))

{

return;

}

var uri = new Uri(path);

Navigation.Navigate(uri);

}

private void GoToPage1CommandExecute(Page1ViewModel viewModel)

{

var uri = new Uri("pack://application:,,,/Pages;component/Page1.xaml");

Navigation.Navigate(uri, Page1ViewModel);

}

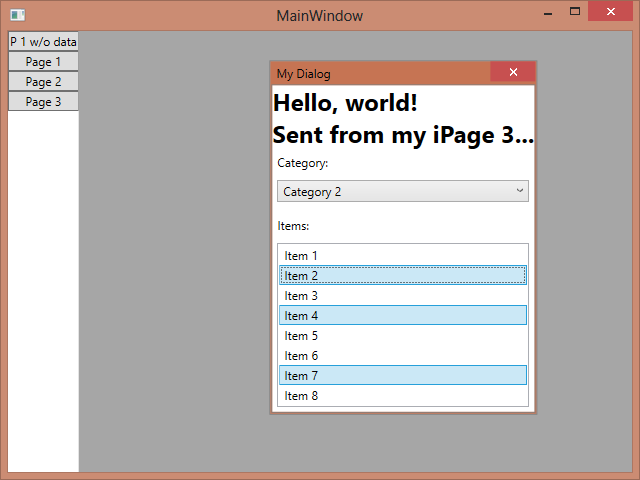
Обратите внимание на путь в URI. Это так называемые «pack URI». В данном случае в этом пути содержится имя сборки со страницами («Pages»). Эта сборка должна быть подключена к текущему проекту (к ViewModels в данном случае). После слова «component» идет локальный путь внутри целевой сборки.

После выполнения данной команды будет осуществлен переход на страницу по указанному адресу, а в DataContext страницы будет положена ссылка на указанную частную ViewModel. Таким образом, нам не надо реализовывать дополнительную логику по получению данных обратно из целевой страницы. Она будет работать с хранилищем внутри нашей MainViewModel, поэтому все изменения мы будем получать автоматически еще до перехода обратно.

Вроде бы все с ViewModel’ями. Переходим к UI.

# Главное окно и сборка Pages

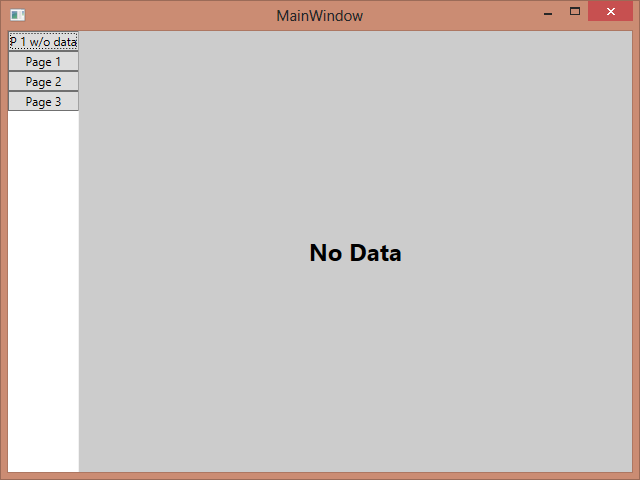
Приведу снова для удобства вид тестового приложения.



Слева представлены четыре кнопки. Первая кнопка привязана к команде GoToPathCommand и выполняет переход на Page1 без контекста данных. К чему это приводит увидим смотрите ниже. В биндинге свойства Text текстового блока я указал параметр FallBackValue.

<TextBlock Text="{Binding Page1Text, FallbackValue='No Data'}"

Если по биндингу к контролу «прилетит» null, то вместо null будет использовано значение из этого параметра.



Остальные кнопки привязаны к «частным» командом с захардкоженным путем до соответствующей страницы. Разметка главного окна представлена ниже.

<Window x:Class="Navigator.MainWindow"

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

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

Title="MainWindow" Height="480" Width="640">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition />

</Grid.ColumnDefinitions>

<ScrollViewer HorizontalScrollBarVisibility="Disabled" VerticalScrollBarVisibility="Hidden">

<StackPanel>

<Button Content="P 1 w/o data" Command="{Binding GoToPathCommand}" CommandParameter="pack://application:,,,/Pages;component/Page1.xaml"/>

<Button Content="Page 1" Command="{Binding GoToPage1Command}" CommandParameter="{Binding Page1ViewModel}"/>

<Button Content="Page 2" Command="{Binding GoToPage2Command}" CommandParameter="{Binding Page2ViewModel}"/>

<Button Content="Page 3" Command="{Binding GoToPage3Command}" CommandParameter="{Binding Page3ViewModel}"/>

</StackPanel>

</ScrollViewer>

<Frame x:Name="MainFrame" Grid.Column="1" Background="#CCCCCC"/>

</Grid>

</Window>

В Code Behind окна я реализовал присвоение менеджеру навигации актуального значения NavigationService и создание MainViewModel.

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

Loaded += MainWindow\_Loaded;

}

void MainWindow\_Loaded(object sender, RoutedEventArgs e)

{

Navigation.Service = MainFrame.NavigationService;

DataContext = new MainViewModel();

}

}

Обратите внимание присвоение NavigationService я делаю в обработчике события Loaded. До момента полной загрузки UI свойство NavigationService главного фрейма возвращает null.

Сборка Pages содержит три страницы: Page1, Page2, Page3. Первые две просто содержат текстовые блоки, привязанные к свойству соответствующей частной ViewModel. Третью я немного усложнил, чтобы заострить внимание на соблюдении рекомендаций паттерна MVVM.

<Page x:Class="Pages.Page3"

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

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

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:tkx="http://schemas.xceed.com/wpf/xaml/toolkit"

mc:Ignorable="d"

d:DesignHeight="400" d:DesignWidth="400">

<Grid>

<tkx:ChildWindow WindowState="Open" Caption="My Dialog" IsModal="True" WindowStartupLocation="Center">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition />

</Grid.RowDefinitions>

<TextBlock Text="{Binding Page3Text, FallbackValue='No Data'}" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="24" FontWeight="Bold"/>

<StackPanel Grid.Row="1">

<TextBlock Text="Category:" Margin="5"/>

<ComboBox Text="Select..." Margin="5">

<ComboBoxItem Content="Category 1"/>

<ComboBoxItem Content="Category 2"/>

<ComboBoxItem Content="Category 3"/>

</ComboBox>

<TextBlock Text="Items:" Margin="5 10 5 5"/>

<ListBox Margin="5" SelectionMode="Multiple">

<ListBoxItem Content="Item 1"/>

<ListBoxItem Content="Item 2"/>

<ListBoxItem Content="Item 3"/>

<ListBoxItem Content="Item 4"/>

<ListBoxItem Content="Item 5"/>

<ListBoxItem Content="Item 6"/>

<ListBoxItem Content="Item 7"/>

<ListBoxItem Content="Item 8"/>

</ListBox>

</StackPanel>

</Grid>

</tkx:ChildWindow>

</Grid>

</Page>

На этой странице я разместил простенький диалог с выбором элементов из определенной категории. Это пример, приближенный к реальности. Вид диалога был показан на снимке главного окна выше. Обратите внимание, текстовый блок, кажется, лежит в диалоговом окне, но биндится к DataContext’у страницы напрямую, без всяких ухищрений. Это заслуга ChildWindow из WPF Toolkit Extended. Этот контрол на самом деле лишь имитирует поведение диалогового окна и является прямым потомком своего родителя в разметке XAML. Таким образом, DataContext наследуется в ChildWindow от Grid, в который я его поместил.

Теперь о соблюдении паттерна MVVM. Была задача в кратчайшие сроки реализовать такой диалог, вернее переделать реальный Window на ChildWindow. Перед реализацией всего этого в стиле MVVM было препятствие. Список элементов для ListBox’а в диалоге содержал не ViewModel или какие-нибудь ObservableObject, а десериализованные сущности (DTO). Поэтому с наскока реализовать как требует того MVVM не получилось. Чтобы вернуть во ViewModel список выбранных элементов ListBox’a я не могу использовать биндинг, т.к. свойство ListBox.**SelectedItems** не поддерживает биндинг ни в каком режиме. Пришлось подписываться на SelectionChanged и уже в Code Behind жонглировать выбранными элементами. Надо ли говорить, что протестировать такой код будет сложновато?

Как это нужно реализовывать? Нужно в списке элементов для ListBox’а хранить ViewModel’и со свойством IsSelected. Далее следует переопределить ListBox.**ItemContainerStyle**.

<ListBox.ItemContainerStyle>

<Style TargetType="ListBoxItem">

<Setter Property="IsSelected" Value="{Binding IsSelected, Mode=TwoWay}"/>

</Style>

</ListBox.ItemContainerStyle>

Это своего рода «ход конем». Мы не привязываем некое свойство ViewModel к контейнеру выбранных элементов в ListBox, а прокидываем через биндинг свойство IsSelected во ViewModel каждого элемента в списке. Нам останется потом выполнить LINQ запрос с фильтрацией по свойству IsSelected. С таким подходом намного проще тестировать, но требует дисциплины в реализации оберточных ViewModel’ей.

Некоторые разработчики ленятся и не делают создание оберток. А ведь помимо сложностей с тестированием, такой подход (биндинг к списку сущностей без INotifyPropertyChanged) с большой вероятностью породит утечки памяти.

# Дополнительная литература

[О синглтоне на Википедии](http://ru.wikipedia.org/wiki/%D0%9E%D0%B4%D0%B8%D0%BD%D0%BE%D1%87%D0%BA%D0%B0_%28%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29)

[О синглтоне на хабре](http://habrahabr.ru/post/147373/)

[О паттерне MVVM на Википедии](http://ru.wikipedia.org/wiki/Model-View-ViewModel)

[О паттерне MVVM на Хабре](http://habrahabr.ru/post/137541/)