MvvmBindingPack

Copyright © 2013-2018, Alexander Paskhin, All Rights Reserved.

Table of contents

Introduction

The decision to develop this binding package had been made in response of minimizing a cost of the quality WPF UI development. The package provides full “Dependency Inversion” between Views and ViewModels.

New in the latest release

Version 2.5.0

Adopted for using .Net Core light weight DI package “`Microsoft.Extensions.DependencyInjection`” and removed dependency on `ServiceLocation`.
`Example of initialization:`

```
ServiceCollection services = new ServiceCollection();
services.AddSingleton<AutoBindingViewModel>();
services.AddSingleton<IocBindingViewModel>();
AutoWireVmDataContext.ServiceProvider = services.BuildServiceProvider();
```

Version 1.8.5

- [**AppendViewModel] -** the mapping attribute that appends(extends) the binding list now is supported with:
  - `BindEventHandler`
  - `BindCommand`

Version 1.8.2.3

- Improved the way resolving IoC the View Model Type, Singleton with using with Unity.

  The order of resolving via the **IoC,** hosted via adapter that implements ServiceLocation interface:

Version 1.8.1

- [**AutoWireVmDataContext**] has been improved:
  - Added a feature that allowed to use interfaces as expected wiring types.
  - `New property IncludeInterfaces -` Include interfaces from the loaded assemblies in the list of candidate types. It allows to use via ViewModelNameOverwrite the interfaces that can be resolved via IoC container.

- Improved the way resolving the View Model Type.
- Order of resolving via the Resource locater:

Version 1.8.0.1

- [**AppendViewModel] -** the new mapping attribute that appends(extends) the binding list candidates from an aggregated object. Value type, boxed value type and types started with "**System**" .. "**MicroSoft**" will be ignored. The aggregated object members are appended to a list. They have a low priority in the lookup. Recursive view model appending is not supported.
- Added Windows 10 universal application support into the NuGet package.

Welcome to MvvmBindingPack

MvvmBindingPack is the robust **MVVM** framework development platform for high-quality **WPF UX** solutions, based on using of IoC/DI containers. **MVVM** pattern is widely used in developing of XAML-based GUI applications. It is impossible to provide the quality UX design implementations without using this pattern. Quality of UX design directly depends on the techniques or features that used for implementing the **MVVM** pattern. Clear separation of concerns between View (XAML code coupled with its code-behind) and View Model characterizes a profession level and quality of the product. The package has the compatible functional features for **XAML WPF, UWP 10**
MVVM pattern provides a principal of "**Dependency Inversion**" between **View** and **View Model**.

MvvmBindingPack Binding elements

- AutoWireVmDataContext - XAML MVVM extension enhancer, it automatically locates and sets the Dependency property (default is "**DataContext**") to a **View Model** reference.
    - [ViewModelClassAlias] - The attribute maps a View Model class to a View by giving an alias of a candidate type name.
- AutoWireViewConrols - XAML MVVM extension enhancer, it automatically locates and binds the **View** controls to **View Model** class members.
    - [ViewTarget] - The mapping attribute that marks a method or property name (or x:Name candidate) with set "**targets**" for a **View** XAML x:Name element.
    - [ViewXNameAlias] - The mapping attribute that marks a filed, method or property name (or x:Name candidate) with set "**names**" + "**targets**" for **View** XAML x:Name element.
    - [ViewXNameSourceTargetMapping] - The mapping attribute that marks a field reference to `ViewXNameSourceTarget` type for a View XAML x:Name element. This class will be used to access to properties or events of the View XAML element.
    - [ViewXNameSourceObjectMapping] - The mapping attribute that marks the field of the any type where the reference to XAML x:Named element will be set to.
    - [AppendViewModel] - it supports an aggregation for the **View Model**.

- BindEventHandler - XAML mark-up, BindXAML.AddEvents and BindXAML.AddPropertyChangeEvents extensions; it binds a control event to a method with a compatible signature of the object which is located in **DataContext** referenced object.
- BindEventHandlerIoc - XAML mark-up, BindXAML.AddEvents and BindXAML.AddPropertyChangeEvents extensions; it binds a control event to a method with a compatible signature of the object which is located in a type resolved via the **IoC** container.
- BindEventHandlerResource - XAML mark-up, BindXAML.AddEvents and BindXAML.AddPropertyChangeEvents extensions; it binds control events to a method with a compatible signature of the object which is located in **Resources**.
- BindCommand - XAML mark-up and BindXAML.BindToCommand extensions; it binds binds a control command property to methods with using **ICommand** interface compatible signature methods.
- BindCommandIoc - XAML mark-up and BindXAML.BindToCommand extensions; it binds binds a control command property to methods with using **ICommand** interface compatible signature methods.
- BindCommandResource - XAML mark-up and BindXAML.BindToCommand extensions; it binds binds a control command property to methods with using **ICommand** interface compatible signature methods.
- IocBinding - XAML mark-up and BindXAML.AssignProperties extensions; it binds to **IoC** container elements.
- LocateDataContext - XAML mark-up and BindXAML.AssignProperties extensions; it finds in the chain of **DataContext** objects, the first, which contains the exact method or property. It comes through the parent elements of logical and visual trees.
- BindXAML.AddEvents - XAML attached property, a fake collection that used for processing extensions: BindEventHandler, BindEventHandlerIoc, and BindEventHandlerResource.
- BindXAML.AssignProperties - XAML attached property, a fake collection that used for processing IocBinding and LocateDataContext extensions.
- BindXAML.BindToCommand - XAML attached property, a fake collection that used for processing extensions: BindCommand, BindEventHandlerIoc, and BindEventHandlerResource.
- BindXAML.AddPropertyChangeEvents - XAML attached property, fake collection, is used for processing extensions: BindEventHandler, BindEventHandlerIoc, BindEventHandlerResource. It binds a **View dependency property change event handler** to the event handler in the View Model. It is supported only for WPF.
- BindXAML.ProcessMvvmExtensions - XAML attached property, a fake collection that used for processing extensions: AutoWireVmDataContext, AutoWireViewConrols.

AutoBinding View to View Model

AutoWireVmDataContext

XAML MVVM extension enhancer, it automatically locates and sets(binds) the View dependency property (default is "**DataContext**") to a **View Model** reference.

- ViewModelNamespaceOverwrite - Overwrites the x:Class namespace; it will be used for exact defining of the view model expected type namespace. Original, the x:Class namespace will be ignored.
- ViewModelNameOverwrite - Overwrites the x:Class name; it will be used for exact type name defining of view model expected type name candidates. Original, the x:Class name will be ignored.
- TargetPropertyName - The target dependency property name. Default value is “DataContext”. It will be set to a resolved reference to a **View Model**.
- UseTheFirstOne - If it is set to 'true' (default), it limits the types of x:Class and x:Name to the first found control in the logical tree.
- ResolveIocContainer - If it is set to 'true', the IoC container will be used to resolve a **View Model** type or instance. It has the first priority. Default value is true.
- ResolveResources - If it is set to 'true', the static Resources will be used to resolve a **View Model** instance. It has the second priority. Default value is true.
- ResolveCreateInstance - If it is set to 'true', the static CLR Activator will be used to create a **View** Model instance. It has the third priority. Default value is true.
- UseMaxNameSubMatch - Defines the additional sub matching ("start with") rule when a View Model expected name compared to a View Model candidate name. If it is set to 'true', the View Model expected name is considered as a match to a name if starts with 'View Model expected name'. Example: The **View Model** expected name "FrameCapturePrice" will match to the **View Model** candidate name "FrameCapturePrice_Var1".
- ViewsNamespaceSuffixSection - Defines the namespace section suffix (default "**Views**"). It will be replaced (if it is exist) on the "ViewModelsNamespaceSuffixSection" property value. It is ignored when the "ViewModelNamespaceOverwrite" is set.Example: the namespace 'Trade.GUI.Application.Views' will be transfered into 'Trade.GUI.Application.ViewModels'; the namespace 'Trade.GUI.Application' will be transferred into 'Trade.GUI.Application.ViewModels'.
- ViewModelsNamespaceSuffixSection - Defines the namespace section suffix (default "**ViewModels**"). It will be used as a replacement. Example: the namespace 'Trade.GUI.Application.Views' will be transferred into 'Trade.GUI.Application.ViewModels'; the namespace 'Trade.GUI.Application' will be transferred into 'Trade.GUI.Application.ViewModels'. It is ignored when the "ViewModelNamespaceOverwrite" is set.
- OldViewNamePart - Defines the part of the class type name (default "**View**"). If it is exist, it will be replaced on the value of the property "NewViewModelNamePart". It is ignored when the "ViewModelNameOverwrite" is set. Example: the name "MainPageView" will be transferred into "MainPageViewModel; the name "MainPageViewFrame_1" will be transferred into "MainPageViewModelFrame_1"; the name "MainPage" will be the same "MainPage".
- NewViewModelNamePart - Defines the part of the class type name (default "**ViewModel**").It is ignored when the "ViewModelNameOverwrite" is set.
- IncludeInterfaces - If it is set to 'true', there will be included interfaces from the loaded assemblies into the list of type candidates. Default value is **true**. It allows to use the interfaces in ViewModelNameOverwrite and resolve them via **IoC** container.
- IocXName - Default value is **false.** f it is set to 'true', the IoC type will be attempted to be resolved with using type and x:Name and x:Name was defined.

Attached property **BindXAML.AutoWiredViewModel** will be set to the reference to the **View Model**.

View to View Model mapping rules.

AutoWireVmDataContext setups a View dependency property with a reference to a **View Model** class instance. By default it is "**DataContext**". The name of the target dependency property can be changed via property "TargetPropertyName". The AutoWireVmDataContext logic of wiring to a View Model is based on using information from the x:Name and x:Class XAML directives:

- x:Name directive uniquely identifies XAML-defined elements in a XAML namescope.
- x:Class directive configures XAML markup compilation to join partial classes between markup and code-behind and it has the type namespace. The namespace will be used to construct expected types.

The **View** (XAML) logical tree elements will be scanned, in root direction, in order to detect non-“System.”, non-“Microsoft.”, other non - WPF class types. For each “`DependencyObject`” based class will be obtained the “Name” property value. In the result, it will be formed the list of types (namespace + name) (x:Class) and names (x:Name if it was set). For each element in the list will be applied transformation rules in order to construct the **View Model** expected types. There will be formed the new list of **View Model** expected types. The candidate list of types for matching will be obtained from loaded assemblies.

General rules for forming View Model expected type names:

1. If the View type namespace suffix section contains a "**Views**"(default see prop. ViewsNameSpaceSuffixSection), this section will be replaced on "**ViewModels**"(default see prop.ViewModelsNameSpaceSuffixSection). It forms “**expected namespace**”.
   Example namespace transformation into “**expected namespace**”:
   Trade.SuperUI.Views => Trade.SuperUI.ViewModels, but (!)
   Trade.SuperUI.Views.Views => Trade.SuperUI.Views.ViewModels
   Trade.SuperUI.RViews => Trade.SuperUI.RViews

2. If the View type namespace suffix section doesn't contains a "**Views**" suffix section and the namespace has only **one or two sections**, in this case the suffix section "**ViewModels**"(default see prop.ViewModelsNameSpaceSuffixSection) will be added. It forms “**expected namespace**”.
   Example namespace transformation into “**expected namespace**”:
   Trade.TicketPanel => Trade.TicketPanel.ViewModel, or (!)
   Trade => Trade.ViewModel

3. If a type name (i.e. x:Class name) or x:Name contains "**View**" substring (default see prop ."OldViewNamePart"), it will be replaced all occurrence on "**ViewModel**" substring (default see prop. "NewViewModelNamePart"). They form a pair of "**expected type names**”.

   Example name transformation into "**expected type name**”:
   TicketViewPanel => TicketViewModelPanel ,but (!)
   TradeViewTicketViewPanel => TradeViewModelTicketViewModelPanel

4. The “**expected fully qualified type names**” will be formed from the parts "**expected namespace**” and “**expected type names**” from x:Class name and x:Name.
5. Formed from x:Name the "expected fully qualified type name" will have a priority over the x:Class formed type name.
6. The list of candidate types and interfaces (see IncludeInterfaces) will be obtained from all loaded assemblies by filtering with "**expected namespace**". Each candidate type name will be examined on best matching to “**expected name**”.
7. Each possible candidate name will be split into a cased parts and matched against "desired name candidate“ parts.
8. The first candidate type with the full parts match will be selected.
9. If you set “`UseMaxNameSubMatch`” flag true, the first candidate with a sub-match type name will selected.

Obtain instance of the View Model type.

Type will be resolved in the sequence: IoC container, Resources and Activator. CreateInstance(). For controlling see properties “`ResolveIocContainer`”,” `ResolveResources`” and “`ResolveCreateInstance`”. In success the resolved type will set as value to "**DataContext**" dependency property (set by default "TargetPropertyName") and attached property **BindXAML.AutoWiredViewModel.**

The order of resolving via the **IoC,** hosted via adapter that implements ServiceLocation interface:

- `GetInstance(locatedItem_WiringType);`
  `!or type will be created by Activator.`

Order of resolving via the Resource Locator:

- `LocateResource(XName)`; or
- `LocateResource(locatedItem_WiringType.Name)`; or
- `LocateResource(locatedItem_WiringTType.FullName)`; or
- `LocateResource(locatedItem_WiringType).`

Examples of XAML and View Model code fragments.

Example

XAML "View" fragment example:

```
<Window x:Class="WpfDemoAutoWire.Views.WindowAutoBind"
        xmlns:mark="MvvmBindingPack"
…...
        x:Name="WindowView"
        Title="AutoWireVmDataContext AutoWireViewConrols" Height="350" Width="300">

    <mark:BindXAML.ProcessMvvmExtensions>
        <mark:AutoWireVmDataContext/>
        <mark:AutoWireViewConrols/>
    </mark:BindXAML.ProcessMvvmExtensions>
```

C# "View Model" fragment example:

```
namespace WpfDemoAutoWire.ViewModels
{

    public class WindowAutoBind : NotifyChangesBase
    {
    }
}
```

Order of resolving via the IoC:

- `GetInstance(typeof(WpfDemoAutoWire.ViewModels.WindowAutoBind)).`

Order of resolving via the Resource Locator:

- `LocateResource("WindowView");` or
- `LocateResource("WindowAutoBind");` or
- `LocateResource("WpfDemoAutoWire.ViewModels.WindowAutoBind");` or
- `LocateResource(typeof(WpfDemoAutoWire.ViewModels.WindowAutoBind)).`

Example of using overwrite properties.

XAML "View" fragment example:

```
<Window x:Class="WpfDemoAutoWire.Views.WindowAutoBind"
        xmlns:mark="MvvmBindingPack"
…...
        x:Name="WindowTrade"
        Title="AutoWireVmDataContext AutoWireViewConrols" Height="350" Width="300">

    <mark:BindXAML.ProcessMvvmExtensions>
        <mark:AutoWireVmDataContext  ViewModelNamespaceOverwrite="WpfDemo.AAA.FFF"
ViewModelNameOverwrite="ICustomTrade"/>
        <mark:AutoWireViewConrols/>
    </mark:BindXAML.ProcessMvvmExtensions>
```

C# "View Model" fragment example:

```
namespace WpfDemo.AAA.FFF
{

    public class AsdfgBertbind : NotifyChangesBase, ICustomTrade
    {
    }
}
```

Order of resolving via the IoC:

- `GetInstance(typeof(WpfDemo.AAA.FFF.ICustomTrade)).`

Order of resolving via the Resource Locator:

- `LocateResource("WindowTrade");` or
- `LocateResource("ICustomTrade");` or
- `LocateResource("WpfDemo.AAA.FFF.ICustomTrade");` or
- `LocateResource(typeof(WpfDemo.AAA.FFF.ICustomTrade)).`

Example

XAML "View" fragment example:

```
<Window x:Class="WpfDemoAutoWire.Views.WindowBind"
        xmlns:mark="MvvmBindingPack"
…...
        x:Name="WindowAutoBindView"
        Title="AutoWireVmDataContext AutoWireViewConrols" Height="350" Width="300">

    <mark:BindXAML.ProcessMvvmExtensions>
        <mark:AutoWireVmDataContext/>
        <mark:AutoWireViewConrols/>
    </mark:BindXAML.ProcessMvvmExtensions>
```

C# "View Model" fragment example:

```
namespace WpfDemoAutoWire.ViewModels
{

    public class WindowAutoBindViewModel: NotifyChangesBase
    {
    }
}
```

Example

XAML "View" fragment example:

```
<Window x:Class="WpfDemoAutoWire.Views.WindowBind"
        xmlns:mark="MvvmBindingPack"
…...
        x:Name="WindowA"
        Title="AutoWireVmDataContext AutoWireViewConrols" Height="350" Width="300">

    <mark:BindXAML.ProcessMvvmExtensions>
        <mark:AutoWireVmDataContext/>
        <mark:AutoWireViewConrols/>
    </mark:BindXAML.ProcessMvvmExtensions>
```

C# "View Model" fragment example:

```
namespace WpfDemoAutoWire.ViewModels
{

    [ViewModelClassAlias("WindowA")]
    public class WindowAbracadabra: NotifyChangesBase
    {
    }
}
```

Example

XAML "View" fragment example:

```
<Window x:Class="WpfDemoAutoWire.WindowBind"
        xmlns:mark="MvvmBindingPack"
```

```
…...
        x:Name="WindowA"
        Title="AutoWireVmDataContext AutoWireViewConrols" Height="350" Width="300">

    <mark:BindXAML.ProcessMvvmExtensions>
        <mark:AutoWireVmDataContext/>
        <mark:AutoWireViewConrols/>
    </mark:BindXAML.ProcessMvvmExtensions>
```

C# "View Model" fragment example:

```
namespace WpfDemoAutoWire.ViewModels
{

    [ViewModelClassAlias("WindowBind ")]
    public class WindowAbracadabra: NotifyChangesBase
    {
    }
}
```

[ViewModelClassAlias]

The mapping attribute that adds to a class the extra alias "candidate type names".It is used to map a **View** onto a **View Model**.

C# "View Model" fragment example:

```
    [ViewModelClassAlias("WindowAutoBindView")]
    [ViewModelClassAlias("WindowAutoBindViewModel")]
    [ViewModelClassAlias("WindowAutoBindViewModelSubMath")]
    [ViewModelClassAlias("WindowListboxSubMath")]
    [ViewModelClassAlias("WindowBindView")]
    [ViewModelClassAlias("WindowMainView")]
    public class WindowAutoBind : NotifyChangesBase
    {

    }
```

AutoWireViewConrols

XAML MVVM extension enhancer, it automatically locates and binds/wires the View controls to View Model class members.

- KnownExcludeMethodPrefixes - The default `static` string collection contains the prefixes of the internal, auxiliary class methods that should be ignored when they are reflected from the **View Model** class type. Default set is {`"get_", "set_", "add_", "remove_", "GetFieldInfo", "FieldGetter", "FieldSetter", "MemberwiseClone", "Finalize", "GetType", "GetHashCode", "ReferenceEquals", "Equals", "ToString"`}.
- Source - Gets or sets the object to use as the wiring source i.e. **View Model** instance. It has priority over 'SourcePropertyName'. It is a "back-door" feature which allows to setup the source object. If it is not set on, by default, the markup extension will use the defined **DataContext** property value or other property redefined by 'SourcePropertyName'. There may be used {IocBinding ...} or other "agnostic" mark up extension(not {Binding ...}) which provides by the independent way to a source object.
- SourcePropertyName - Source dependency property name. The property value will be used as a reference to the **View Model** object. Default dependency property name is "**DataContext**".

- UseMaxNameSubMatch - Defines the additional sub matching rule when a expected view name (x:Name without targets) compared to a view model candidate name. If it is true, the view model candidate name is considered as a match to a view expected name which starts with the 'view expected name'. Example: view name "WindowAutoBindViewModel" match to view modelName "WindowAutoBindViewModelSubMath".
- IncludeVisualTreeNames - Include visual tree x:Named elements onto wiring. Default value is false.

Name “parts” split and matching rules

For comparing to names is used case sensitive name part matching algorithm. It allows to add more flexibility in forming and using View Model naming conventions.

General rules to form name “parts”

1. The name is split into parts by capital letter or '_'. The character '_' is not included into parts.
2. The name parts are considered as a case sensitive.
3. The names are considered as matched if they have the same consequential set of parts.
4. The name is considered as sub-match if it has been started at least one or more the same consequential parts.

Examples of splitting:

The View name "**_Example_Name_**" will split into parts {"**Example**","**Name**"}.
The View name "**ExampleName**" will split into parts {"**Example**","**Name**"}.
The View name "**exampleName_Ver1**" split into parts {"**example**","**Name**","**Ver1**"}.

Examples of matching:

"**Example_Name_**" and "**ExampleName**" and "**Example___Name_**" are match because they have the same set of parts.

Examples of sub-matching:
"**Example_Name_**" and "**ExampleName_Ver**" has a sub-match with rank 2 of the same set of parts.

View to View Model controls binding/wiring.

The AutoWireViewConrols logic is based on using of the x:Name directive. x:Name directive uniquely identifies XAML elements in a XAML namescope. AutoWireViewConrols wires and binds x:Named XAML elements or View XAML UI elements to View Model properties, methods and fields. The View (XAML) element targets are dependency properties or routing events. They are subject of binding to properties, fields and methods in a View Model class.

View Control General Wiring and Binding rules:

1. One x:Named **View** (XAML) element can be bind one to many distinguish properties, fields or methods, in a **View Model**.
2. The **View Model** properties has a priority to bind over the methods with the same binding name.
3. The first found match will be bind fist. The order of the declaration is not applicable in ambiguous cases.
4. It is used always the full name match of “parts”, a part-sub match can be used as an option, see the 'UseMaxNameSubMatch' property.
5. **One to One**: The **View Model** property or event can be bind only once for one x:Named View XAML element.
6. **View** element targets (dependency properties, routing events) will be bind to **View Model** element targets.
7. The element target names should be defined in the **View Model**.
8. The **View Model** element names without targets will be ignored.
9. The x:Name is ignored if it starts with "_".
10. The attached property or event name should be set in format "**TypeOwner.Name**" example "**Grid.Row**", "**Mouse.MouseMove**" with using attributes: `[ViewXNameAlias], [ViewTarget]`.

Examples of wiring the View Model method to the View event.

Wiring goal is to wire the View element event like:

`<Button x:Name="Example_Name_" …>` and event "`Click`" to a method handler in the View Model.

The View Model wiring C# definition variants:

Without any attributes

```
void  Example_Name_Click(object sender, RoutedEventArgs e){}  or;

void  ExampleName_Click(object sender, RoutedEventArgs e){}
```

With attribute [`ViewTarget` (...)]

```
[ViewTarget("Click")]
void  ExampleName_Clk(object sender, RoutedEventArgs e){}  or;

[ViewTarget("Click")]
void  ExampleName_Other(object sender, RoutedEventArgs e){}
```

With attribute [`ViewXNameAlias` (...)]

```
[ViewXNameAlias("ExampleName","Click")]
void  AbracadbraName(object sender, RoutedEventArgs e){}  or;

[ViewXNameAlias("Example_Name","Click")]
void  _AbracadbraName(object sender, RoutedEventArgs e){}
/* the name starting with "_" will be ignored, but the attribute don't */ or;

[ViewXNameAlias("Example_Name_","Click")]
void  Abracadbra_Name(object sender, RoutedEventArgs e){}
```

Examples of binding the View Model property to the View property

Binding goal is to bind the View element like:
`<Label x:Name="Example_Name" ..>` and property "`Content`" to a property in the View Model.

The View Model wiring C# definition variants:

Without any attributes

```
string  Example_Name_Content {get;set;}  or;

string  ExampleName_Content {get;set;}
```

With attribute [`ViewTarget` (...)]

```
[ViewTarget("Content")]
string  Example_Name {get;set;}  or;

[ViewTarget("Content")]
string  ExampleName_BadTag {get;set;}
```

With attribute [`ViewXNameAlias` (...)]

```
[ViewXNameAlias("ExampleName","Content")]
string  AbracadbraName{get;set;}    or;

[ViewXNameAlias("Example_Name","Content")]
string  _AbracadbraName{get;set;}
/* the name starting with "_" will be ignored, but the attribute don't */ or;
```

```
[ViewXNameAlias("Example_Name_","Content")]
string  Abracadbra_Name{get;set;}
```

Examples of wiring the View Model method to the View "Command" property

Wiring goal is to wire the View element event like:
`<Button x:Name="Example_Name_" …>` and property `"Conmmad"` to methods in the View Model.

The View Model wiring C# definition variants:

Without any attributes

```
ICommand  Example_Name_Command {get;set;}  or;

string  ExampleName_Command {get;set;}
```

With attribute [`ViewTarget` (...)]

```
[ViewTarget("Command")]
ICommand  Example_Name {get;set;}  or;

[ViewTarget("Command")]
ICommand  ExampleName_BadTag {get;set;}
```

With attribute [`ViewXNameAlias` (...)]

```
[ViewXNameAlias("ExampleName","Command")]
ICommand  AbracadbraName{get;set;}    or;

[ViewXNameAlias("Example_Name","Command")]
ICommand  _AbracadbraName{get;set;}
/* the name starting with "_" will be ignored, but the attribute don't */ or;

[ViewXNameAlias("Example_Name_","Command")]
ICommand  Abracadbra_Name{get;set;}

Separate "Execute" and "CanExecute" wiring.
[ViewXNameAlias("Example_Name", "Command.Execute")]
void NameVM2MExecute(object obj){...}

[ViewXNameAlias("Example_Name", "Command.CanExecute")]
bool Method_NameVM2MCanExecute(object obj){....} or;

[ViewXNameAlias("Example_Name", "Command.CanExecute")]
bool Prop_NameVM2MCanExecute{get;set;}
```

Examples of wiring(just copy) the View Model fields to the View property

Wiring goal is to wire the View element event like:
`<Grid x:Name="Example_Name_" …>` and property `"Width"` to fields (just copy from) in the View Model.

The View Model wiring C# definition variants:

With attribute [`ViewXNameAlias` (...)]

```
[ViewXNameAlias("ExampleName","Width")]
string _textAndMsgLabelTxtC = "Content was copied from the field";
/*the field name will always be ignored.*/
```

Examples of wiring/referencing the View fields into the View Model

Sometimes, very often there is a vital case to have a link from View Model to a View element or property or event.

Get a reference/link to the element type like:

```
<Label x:Name="LabelXNameVM2M" ..>
```

The View Model wiring C# definition variants:

```
[ViewXNameSourceObjectMapping("LabelXNameVM2M")]
private object _LabelXNameVM2M; // can be used the 'Label' type  instead of the 'Object' type.
```

Get a reference/link to the property "Content" of the element type like

```
<Label  x:Name="LabelXNameVM2M"  ..>
```

The View Model wiring C# definition variants:

```
[ViewXNameSourceTargetMapping("LabelXNameVM2M", "Content")]
private ViewXNameSourceTarget _LabelXNameVM2MContent;
```

[ViewTarget]

The mapping attribute that marks a method or property name (or x:Name candidate) with set "**targets**" for a **View** XAML x:Name element.

C# "View Model" fragment example:

```
[ViewTarget("Click")]
void  ExampleName_Clk(object sender, RoutedEventArgs e){}  or;

[ViewTarget("Click")]
void  ExampleName_Other(object sender, RoutedEventArgs e){}


[ViewTarget("Content")]
string  Example_Name {get;set;}  or;

[ViewTarget("Content")]
string  ExampleName_BadTag {get;set;}
```

[ViewXNameAlias]

The mapping attribute that marks a filed, method or property name (or x:Name candidate) with set alias "**names**" + "**targets**" for **View** XAML x:Name element.

ViewXNameAliasAttribute extra binding parameters:

- BindingMode - Gets or sets a value that indicates the direction of the data flow in the binding.
- HandledEventsToo - If it is true to register the handler such that it is invoked even when the routed event is marked handled in its event data.
- ValidatesOnDataErrors - The DataErrorValidationRule is a built-in validation rule that checks for errors that are raised by the IDataErrorInfo implementation of the source object.

- ValidatesOnExceptions - The ExceptionValidationRule is a built-in validation rule that checks for exceptions that are thrown during the update of the source property.
- ValidatesOnNotifyDataErrors - When ValidatesOnNotifyDataErrors is true, the binding checks for and reports errors that are raised by a data source that implements INotifyDataErrorInfo.

C# "View Model" fragment example:

```
[ViewXNameAlias("ExampleName","Content")]
string  AbracadbraName{get;set;}    or;

[ViewXNameAlias("Example_Name","Content")]
string  _AbracadbraName{get;set;}
/* the name starting with "_" will be ignored, but the attribute don't */ or;

[ViewXNameAlias("Example_Name_","Content")]
string  Abracadbra_Name{get;set;}

[ViewXNameAlias("LabelXNameC", "Content")]
string _textAndMsgLabelTxtC = "Content was binded - C";

[ViewXNameAlias("ExampleName","Command")]
ICommand  AbracadbraName{get;set;}    or;

[ViewXNameAlias("Example_Name","Command")]
ICommand  _AbracadbraName{get;set;}
/* the name starting with "_" will be ignored, but the attribute don't */ or;

[ViewXNameAlias("Example_Name_","Command")]
ICommand  Abracadbra_Name{get;set;}

Separate "Execute" and "CanExecute" wiring.
[ViewXNameAlias("Example_Name", "Command.Execute")]
void NameVM2MExecute(object obj){...}

[ViewXNameAlias("Example_Name", "Command.CanExecute")]
bool Method_NameVM2MCanExecute(object obj){....} or;

[ViewXNameAlias("Example_Name", "Command.CanExecute")]
bool Prop_NameVM2MCanExecute{get;set;}

[ViewXNameAlias("LabelXName", "Content", ValidatesOnNotifyDataErrors = true)]
public string KadLabelXNameD1
{
    get { return _textAndMsgLabelTxtF; }
    set { _textAndMsgLabelTxtF = value; NotifyPropertyChanged(); }
}
```

[ViewXNameSourceTargetMapping]

The mapping attribute that marks a field reference to `ViewXNameSourceTarget` type for a View XAML x:Name element. This class is used to access to properties or events of the View XAML element.

C# "View Model" fragment example:

```
[ViewXNameSourceTargetMapping("LabelXNameVM2M", "Content")]
private ViewXNameSourceTarget _LabelXNameVM2MContent;

[ViewXNameSourceTargetMapping("ButtonXNameVM2M", "Click")]
private ViewXNameSourceTarget _ButtonXNameVM2MClick;
```

[ViewXNameSourceObjectMapping]

The mapping attribute that marks the field of the any type where the reference to XAML x:Named element will be set to.

C# “View Model” fragment example:

```
[ViewXNameSourceObjectMapping("LabelXNameVM2M")]
private Label _LabelXNameVM2M;

[ViewXNameSourceObjectMapping("ButtonXNameVM2M")]
private Button _ButtonXNameVM2M;
```

[AppendViewModel]

The mapping attribute that appends(extends) the bindings list of wiring candidates with another reference type object members. Value type, "boxed value type" and types started with "**System**" .. "**MicroSoft**" will be ignored. The members are appended to a list of wiring candidates. They have a low priority.Recursive view model appending is not supported.

- [**AppendViewModel] -** the mapping attribute that appends(extends) the binding list now is supported with:
  - BindEventHandler
  - BindCommand

C# “View Model” fragment example:

```
namespace WpfDemoAutoWire.ViewModels
{

    public class WindowAutoBind : NotifyChangesBase
    {
       [AppendViewModel]
       private AppendedViewModel1 _appendedViewModel1;


       [AppendViewModel]
       public AppendedViewModel2  AppendedViewModel2 {get; set;};
    }

}
```

BindXAML.ProcessMvvmExtensions

XAML attached property, fake collection, that used for processing extensions: AutoWireVmDataContext, AutoWireViewConrols.

XAML “View” fragment example:

```
<vm:BindXAML.ProcessMvvmExtensions>
    <vm:AutoWireVmDataContext/>
    <vm:AutoWireViewConrols/>
</vm:BindXAML.ProcessMvvmExtensions
```

Event Binding

BindEventHandler

XAML mark-up, BindXAML.AddEvents and BindXAML.AddPropertyChangeEvents extensions; it binds a control event to a method with a compatible signature of the object which is located in DataContext referenced object.

- Source (default key) – It is a "back-door" feature which allows to setup the source object. If it is not set on, by default, the markup extension will use the defined DataContext property value. It is referring to the source object

which has the method or property used by the markup extension.There may be used {locBinding ...} or other "agnostic" mark up extension(not {Binding ...}) which provides by the independent way to a source object.

- MethodName – The method name of the source object that has …EventHandler delegate signature (can be static).It's mutually exclusive versus PropertyName.
- PropertyName – The property name of the source object that contains …EventHandler delegate (can be static). It's mutually exclusive versus MethodName.
- TargetEventName (external key, used by BindXAML.AddEvents) – The key is used to pass a target event name to the BindXAML.AddEvents attached property collection.
- TargetPropertyName (external key, used for BindXAML.AddPropertyChangeEvents) – The key is used to pass a target property name to the BindXAML.AddPropertyChangeEvent.
- DeepScanAllTrees- If it is set to "true", all DataContext properties in the logical tree will be scanned until the math to a property or method name (PropertyName, MethodName). Smart feature allows to ignore the current DataContext property value and traverse to other parent DataContext value.If set on true, it will cause to scan for the DataContext property objects over the trees and get the first one that contains the binding property or method. It used in case when there is need to ignore the binding ItemsSource DataContext for the ItemsControl item, just bind a Button to a View Model for the item of the ListView or so on.
- [**AppendViewModel] -** the mapping attribute that appends(extends) the binding list now is supported.

XAML(WPF) "View" fragment example of using with MethodName**:**

```
<Button Content="Button Click Method"
        Click="{vm:BindEventHandler MethodName=ButtonClickMethod}"/>
```

XAML(WinRt) "View" fragment example of using with MethodName**:**

```
<Button Content="Button Click Method" >
      <vm:BindXAML.AddEvents>
          <vm:BindEventHandler MethodName="ButtonClickMethod" TargetEventName="Click"/>
      </vm:BindXAML.AddEvents>
</Button>
```

XAML(WPF) "View" fragment example of using with PropertyName**:**

```
<Button Content="Button Click Method via PropertyName"
        Click="{vm:BindEventHandler PropertyName=ButtonClickProperty}"/>
```

XAML(WinRt) "View" fragment example of using with PropertyName**:**

```
<Button Content="Button Click Method via PropertyName" >
      <vm:BindXAML.AddEvents>
          <vm:BindEventHandler PropertyName="ButtonClickProperty" TargetEventName="Click"/>
      </vm:BindXAML.AddEvents>
</Button>
```

C# "View Model" fragment example:

```
public ViewModelNew()
{
    _buttonClickPropDelegate = new RoutedEventHandler(ButtonClickMethod);
}

private RoutedEventHandler _buttonClickPropDelegate;

public RoutedEventHandler ButtonClickProperty
{
    get { return _buttonClickPropDelegate; }
}

public void ButtonClickMethod(object sender, RoutedEventArgs e)
{

}
```

XAML(WPF) "View" fragment example of using with TargetPropertyName;
it subscribes to the Dependency property change events:

```
<Label Content="Button Click Method via PropertyName" >
      <vm:BindXAML.AddPropertyChangeEvents>
          <vm:BindEventHandler MethodName="DataContextChanged"
```

```
            TargetPropertyName="DataContext"/>
      </vm:BindXAML.AddPropertyChangeEvents>
</Label>
```

C# "View Model" fragment example of using with TargetPropertyName; it subscribes to the Dependency property change events:

```
public void DataContextChanged(object sender, EventArgs e)
{

}
```

BindEventHandlerIoc

XAML mark-up, BindXAML.AddEvents and BindXAML.AddPropertyChangeEvents extensions; it binds a control event to a method with a compatible signature of the object which is located in the IoC container.

- ServiceType (default key) –The type (System.Type) or the type name (System.String) of the requested object.
- ServiceKey – The key of the requested DI (IoC) object.
- MethodName – The method name of the source object that has …EventHandler delegate signature (it can be static).It's mutually exclusive versus PropertyName.
- PropertyName – The property name of the source object that contains …EventHandler delegate (it can be static). It's mutually exclusive versus MethodName.
- TargetEventName (external key, used by BindXAML.AddEvents) – The key is used to pass a target event name to the BindXAML.AddEvents.
- TargetPropertyName (external key, used for BindXAML.AddPropertyChangeEvents) – The key is used to pass a target property name to the BindXAML.AddPropertyChangeEvent.

XAML(WPF) "View" fragment example of using with MethodName:

```
<Button Content="Button Click Method"
        Click="{vm:BindEventHandlerIoc  ServiceType=ViewModels.ViewModelNew,
                MethodName=ButtonClickMethod}"/>
```

XAML(WinRt) "View" fragment example of using with MethodName:

```
<Button Content="Button Click Method" >
      <vm:BindXAML.AddEvents>
          <vm:BindEventHandlerIoc  ServiceType=ViewModels.ViewModelNew,
              MethodName="ButtonClickMethod" TargetEventName="Click"/>
      </vm:BindXAML.AddEvents>
</Button>
```

XAML(WPF) "View" fragment example of using with PropertyName:

```
<Button Content="Button Click Method via PropertyName"
        Click="{vm:BindEventHandlerIoc  ServiceType=ViewModels.ViewModelNew,
                PropertyName=ButtonClickProperty}"/>
```

XAML(WinRt) "View" fragment example of using with PropertyName:

```
<Button Content="Button Click Method via PropertyName" >
      <vm:BindXAML.AddEvents>
          <vm:BindEventHandlerIoc  ServiceType="ViewModels.ViewModelNew"
               PropertyName="ButtonClickProperty" TargetEventName="Click"/>
      </vm:BindXAML.AddEvents>
</Button>
```

C# setup "View Model" for Unity DI container example in App.xaml.cs:

```
    public partial class App : Application
    {
        private UnityContainer _unityContainer;
        private UnityServiceLocator _servicelocator;
        public App()
        {
            _unityContainer = new UnityContainer();
            _servicelocator = new UnityServiceLocator(_unityContainer);
```

```
            ServiceLocator.SetLocatorProvider(() => _servicelocator);
            var vmMw = new ViewModelNew();
            // instance that will be resolved when it's used ServiceType
            _unityContainer.RegisterInstance(typeof(ViewModelNew),
                vmMw,new ContainerControlledLifetimeManager());
        }
    }
```

C# "View Model" fragment example:

```
public ViewModelNew()
{
    _buttonClickPropDelegate = new RoutedEventHandler(ButtonClickMethod);
}

private RoutedEventHandler _buttonClickPropDelegate;

public RoutedEventHandler ButtonClickProperty
{
    get { return _buttonClickPropDelegate; }
}

public void ButtonClickMethod(object sender, RoutedEventArgs e)
{

}
```

XAML(WPF) "View" fragment example of using with TargetPropertyName;
it subscribes to the Dependency property change events:

```
<Label Content="Button Click Method via PropertyName" >
      <vm:BindXAML.AddPropertyChangeEvents>
          <vm:BindEventHandlerIoc  ServiceType="ViewModels.ViewModelNew"
                MethodName="DataContextChanged" TargetPropertyName="DataContext"/>
      </vm:BindXAML.AddPropertyChangeEvents>
</Label>
```

C# "View Model" fragment example of using with TargetPropertyName;
it subscribes to the Dependency property change events:

```
public void DataContextChanged(object sender, EventArgs e)
{

}
```

BindEventHandlerResource

XAML mark-up, BindXAML.AddEvents and BindXAML.AddPropertyChangeEvents extensions; it binds control events to a method with a compatible signature of object which is located in Resources.

- ResourceKey (default key) – Sets the key value to a static resource. The key is used to return the object matching that key in the resource dictionaries.
- MethodName – The method name of the source object that has …EventHandler delegate signature (it can be static).It's mutually exclusive versus PropertyName.
- PropertyName – The property name of the source object that contains …EventHandler delegate (it can be static). It's mutually exclusive versus MethodName.
- TargetEventName (external key, used by BindXAML.AddEvents) – The key is used to pass a target event name to the BindXAML.AddEvents.
- TargetPropertyName (external key, used for BindXAML.AddPropertyChangeEvents) – The key is used to pass a target property name to the BindXAML.AddPropertyChangeEvent.

XAML(WPF) "View" fragment example of using with MethodName**:**

```
<Button Content="Button Click Method"
        Click="{vm:BindEventHandlerResource ResourceKey=ViewModelNewKey,
                MethodName=ButtonClickMethod}"/>
```

XAML(WinRt) "View" fragment example of using with MethodName:

```
<Button Content="Button Click Method" >
      <vm:BindXAML.AddEvents>
          <vm:BindEventHandlerResource ResourceKey=ViewModelNewKey,
              MethodName="ButtonClickMethod" TargetEventName="Click"/>
      </vm:BindXAML.AddEvents>
</Button>
```

XAML(WPF) "View" fragment example of using with PropertyName:

```
<Button Content="Button Click Method via PropertyName"
        Click="{vm:BindEventHandlerResource ResourceKey=ViewModelNewKey,
                PropertyName=ButtonClickProperty}"/>
```

XAML(WinRt) "View" fragment example of using with PropertyName:

```
<Button Content="Button Click Method via PropertyName" >
      <vm:BindXAML.AddEvents>
          <vm:BindEventHandlerResource ResourceKey="ViewModelNewKey"
               PropertyName="ButtonClickProperty" TargetEventName="Click"/>
      </vm:BindXAML.AddEvents>
</Button>
```

C# "View Model" fragment example:

```
public ViewModelNew()
{
    _buttonClickPropDelegate = new RoutedEventHandler(ButtonClickMethod);
}

private RoutedEventHandler _buttonClickPropDelegate;

public RoutedEventHandler ButtonClickProperty
{
    get { return _buttonClickPropDelegate; }
}

public void ButtonClickMethod(object sender, RoutedEventArgs e)
{

}
```

XAML(WPF) "View" fragment example of using with TargetPropertyName;
it subscribes to the Dependency property change events:

```
<Label Content="Button Click Method via PropertyName" >
      <vm:BindXAML.AddPropertyChangeEvents>
          <vm:BindEventHandlerResource ResourceKey="ViewModelNewKey"
               MethodName="DataContextChanged" TargetPropertyName="DataContext"/>
      </vm:BindXAML.AddPropertyChangeEvents>
</Label>
```

C# "View Model" fragment example of using with TargetPropertyName;
it subscribes to the Dependency property change events:

```
public void DataContextChanged(object sender, EventArgs e)
{

}
```

BindXAML.AddEvents

XAML attached property, fake collection, that used for processing extensions: BindEventHandler, BindEventHandlerIoc, BindEventHandlerResource.
It is used for compatibility between WinRT, Win Store App and WPF.

XAML "View" fragment example:

```
<Button Content="Button Click Method" HorizontalAlignment="Left" >
```

```
    <vm:BindEventHandler MethodName="LoadedMethod" TargetEventName="Loaded"/>
    <vm:BindXAML.AddEvents>
        <vm:BindEventHandler MethodName="UnloadedMethod" TargetEventName="Unloaded"/>
        <vm:BindEventHandler MethodName="ButtonClickMethod" TargetEventName="Click"/>
    </vm:BindXAML.AddEvents>
</Button>
```

BindXAML.AssignProperties

XAML attached extension that used for processing IocBinding and LocateDataContext extensions. It is used for compatibility between WinRT, Win Store App and WPF.

XAML "View" fragment example:

```
<Label Content="{Binding ButtonClickMethodMsg}" HorizontalAlignment="Left">
    <vm:BindXAML.AssignProperties>
        <vm:IocBinding ServiceType="MainWindowVm"
             ServiceKey="MainWindowVm" TargetPropertyName="DataContext"/>
    </vm:BindXAML.AssignProperties>
</Label>
```

C# setup "View Model" for Unity DI container example in App.xaml.cs:

```
    public partial class App : Application
    {
        private UnityContainer _unityContainer;
        private UnityServiceLocator _servicelocator;
        public App()
        {
            _unityContainer = new UnityContainer();
            _servicelocator = new UnityServiceLocator(_unityContainer);
            ServiceLocator.SetLocatorProvider(() => _servicelocator);
            var vmMw = new ViewModelNew();
            // instance that will be resolved when it's used ServiceType
            _unityContainer.RegisterInstance(typeof(ViewModelNew),
                vmMw,new ContainerControlledLifetimeManager());
        }
    }
```

Command Binding

BindCommand

XAML mark-up and BindXAML.BindToCommand extensions; it binds binds a control command type dependency property to methods with using the **CommandHadlerProxy** wrapper class. It binds to the source object members defined by a **DataContext** dependency property.

- Source (default key) – It is a "back-door" feature which allows to setup the source object. If it is not set on, by default,the markup extension will use the defined DataContext dependency property value. It may be used with {IocBinding ...} or other "agnostic" mark up extension(not {Binding ...}) which provides by the independent way a source object reference.
- ExecuteMethodName – The method name of the source object that performs as "void ICommand:Execute(object parameter)". It's mutually exclusive versus ExecutePropertyName. It can be static.
- CanExecuteBooleanPropertyName - The property name of the source object that refers to Boolean property that would be return by method "bool ICommand:CanExecute(object parameter)". INotifyPropertyChanged interface will be subscribed to trigger event "event EventHandler ICommand:CanExecuteChanged". It's mutually exclusive versus CanExecuteMethodName, CanExecutePropertyName, EventToInvokeCanExecuteChanged and PropertyActionCanExecuteChanged.
- CanExecuteMethodName – The method name of the source object that performs as "bool ICommand:CanExecute(object parameter)". It can be static and optional. It's mutually exclusive versus CanExecutePropertyName.

- ExecutePropertyName – The property name of the source object that has a type of Action<object> delegate that performs as "void ICommand:Execute(object parameter)". It's mutually exclusive versus ExecutePropertyName. It can be static.
- CanExecutePropertyName –The property name of the source object that has a type of **Func**<object, bool> delegate that performs as "bool ICommand:CanExecute(object parameter)". It can be static and optional. It's mutually exclusive versus CanExecutePropertyName.
- EventToInvokeCanExecuteChanged - Name of an event class member to which will be added a delegate for rising an event in the proxy class "event EventHandler ICommand:CanExecuteChanged". Notification delegate of types **Action**<> or **EventHandler**<> will be added or removed synchronously when the event handler will be add or removed in "event EventHandler ICommand:CanExecuteChanged". It's mutually exclusive versus PropertyActionCanExecuteChanged. It can be static.
- PropertyActionCanExecuteChanged - Name of a property that will accept a delegate of **Action**<> delegate that can be used for rising an event in the proxy class "event EventHandler ICommand:CanExecuteChanged".Notification delegate of types **Action**<> or **EventHandler**<> will be set or cleared synchronously when the event handler will be add or removed in "event EventHandler ICommand:CanExecuteChanged". It's mutually exclusive versus EventToInvokeCanExecuteChanged. It can be static.
- DeepScanAllTrees- If it is set to "true", all DataContext properties in the logical tree will be scanned until the math to a property or method name (ExecutePropertyName, ExecuteMethodName). Smart feature allows to ignore the current DataContext property value and traverse to other parent DataContext value.If set on true, it will cause to scan for the DataContext property objects over the trees and get the first one that contains the binding property or method. It used in case when there is need to ignore the binding ItemsSource DataContext for the ItemsControl item, just bind a Button to a View Model for the item of the ListView or so on.
- [**AppendViewModel] -** the mapping attribute that appends(extends) the binding list now is supported.

XAML(WPF) "View" fragment example:

```
<Button Content="Click here!"
   Command="{vm:BindCommand ExecuteMethodName=CommandToExcute,
                CanExecuteBooleanPropertyName=Flag}"/>

<Button Content="Property Cmd-ExCe"
   Command="{vm:BindCommand ExecutePropertyName=ButtonExecuteProperty,
                CanExecutePropertyName=CanExecuteProperty,
                DeepScanAllTrees=True}"/>

<Button Content="Button Cmd-ExCeEv"
   Command="{vm:BindCommand  ExecuteMethodName=ExecuteMethod,
                CanExecuteMethodName=CanExecuteMethod,
                EventToInvokeCanExecuteChanged=ActionNotifyCanExecuteChanged,
                DeepScanAllTrees=True}"/>

<Button Content="Button Cmd-ExProp"
   Command="{vm:BindCommand  ExecuteMethodName=ExecuteMethod,
              CanExecuteBooleanPropertyName=CanExecuteFlag,
              DeepScanAllTrees=True}"/>
```

XAML(WinRt) "View" fragment example:

```
<Button Content="Click here!">
    <vm:BindXAML.BindToCommand>
        <vm:BindCommand ExecuteMethodName="CommandToExcute"
            CanExecuteBooleanPropertyName="Flag"/>
    </vm:BindXAML.BindToCommand>
</Button>
```

C# "View Model" fragment example:

```
bool _canExecuteFlag = true;

public bool CanExecuteFlag
 {
     get { return _canExecuteFlag; }
```

```
    set
    {
        _canExecuteFlag = value; NotifyPropertyChanged();
    }
}

public Action<object> ButtonExecuteProperty { get; set; }

public void ExecuteMethod(object sender)
{

}


public Func<object, bool> CanExecuteProperty { get; set; }

public bool CanExecuteMethod(object sender)
{
    return CanExecuteFlag;
}

public event Action ActionNotifyCanExecuteChanged;

public event EventHandler EventHandlerNotifyCanExecuteChanged;

private Action _propertyDelegateNotifyCanExecuteChanged;

public Action PropertyDelegateNotifyCanExecuteChanged
{
    get { return _propertyDelegateNotifyCanExecuteChanged; }
    set { _propertyDelegateNotifyCanExecuteChanged = value; }
}
```

BindCommandIoc

XAML mark-up and BindXAML.BindToCommand extensions; it binds binds a control command dependency property to methods with using the **CommandHadlerProxy** wrapper class. It binds to the source object members of the type resolved by the IoC container.

- ServiceType (default key) – The type of the requested object. The string of a type name of the requested object.
- ServiceKey – The key of the requested DI (IoC) object.
- ExecuteMethodName – The method name of the source object that performs as "void ICommand:Execute(object parameter)". It's mutually exclusive versus ExecutePropertyName. It can be static.
- CanExecuteBooleanPropertyName - The property name of the source object that refers to Boolean property that would be return by method "bool ICommand:CanExecute(object parameter)". INotifyPropertyChanged interface will be subscribed to trigger event "event EventHandler ICommand:CanExecuteChanged". It's mutually exclusive versus CanExecuteMethodName, CanExecutePropertyName, EventToInvokeCanExecuteChanged and PropertyActionCanExecuteChanged.
- CanExecuteMethodName – The method name of the source object that performs as "bool ICommand:CanExecute(object parameter)". It can be static and optional. It's mutually exclusive versus CanExecutePropertyName.
- ExecutePropertyName – The property name of the source object that has a type of Action<object> delegate that performs as "void ICommand:Execute(object parameter)". It's mutually exclusive versus ExecutePropertyName. It can be static.
- CanExecutePropertyName –The property name of the source object that has a type of **Func**<object, bool> delegate that performs as "bool ICommand:CanExecute(object parameter)". It can be static and optional. It's mutually exclusive versus CanExecutePropertyName.
- EventToInvokeCanExecuteChanged - Name of an event class member to which will be added a delegate for rising an event in the proxy class "event EventHandler ICommand:CanExecuteChanged". Notification delegate of types **Action**<> or **EventHandler**<> will be added or removed synchronously when the event handler will be add or removed in "event EventHandler ICommand:CanExecuteChanged". It's mutually exclusive versus PropertyActionCanExecuteChanged. It can be static.
- PropertyActionCanExecuteChanged - Name of a property that will accept a delegate of **Action**<> delegate that can be used for rising an event in the proxy class "event EventHandler

ICommand:CanExecuteChanged".Notification delegate of types **Action**<> or **EventHandler**<> will be set or cleared synchronously when the event handler will be add or removed in "event EventHandler ICommand:CanExecuteChanged". It's mutually exclusive versus EventToInvokeCanExecuteChanged. It can be static.

- DeepScanAllTrees- If it is set to “true”, all DataContext properties in the logical tree will be scanned until first math to a property or method name (ExecutePropertyName, ExecuteMethodName). Smart feature which allows ignore the current DataContext property value and traverse to other parent DataContext value.If set on true, it will cause to scan for the DataContext property objects over the trees and get the first one that contains the binding property or method. It used in case when there is need to ignore the binding ItemsSource DataContext for the ItemsControl item, just bind a Button to a View Model for the item of the ListView or so on.

C# setup “View Model” for Unity DI container example in App.xaml.cs:

```
public partial class App : Application
{
    private UnityContainer _unityContainer;
    private UnityServiceLocator _servicelocator;
    public App()
    {
        _unityContainer = new UnityContainer();
        _servicelocator = new UnityServiceLocator(_unityContainer);
        ServiceLocator.SetLocatorProvider(() => _servicelocator);
        var vmMw = new ViewModelNew();
        // instance that will be resolved when it's used ServiceType
        _unityContainer.RegisterInstance(typeof(ViewModelNew),
            vmMw,new ContainerControlledLifetimeManager());
    }
}
```

XAML(WPF) “View” fragment example:

```
<Button Content="Click here!"
   Command="{vm:BindCommandIoc ServiceType=ViewModels.ViewModelNew,
                ExecuteMethodName=CommandToExcute,
                CanExecuteBooleanPropertyName=Flag}"/>

<Button Content="Property Cmd-ExCe"
   Command="{vm:BindCommandIoc ServiceType=ViewModels.ViewModelNew,
                ExecutePropertyName=ButtonExecuteProperty,
                CanExecutePropertyName=CanExecuteProperty,
                DeepScanAllTrees=True}"/>

<Button Content="Button Cmd-ExCeEv"
   Command="{vm:BindCommandIoc ServiceType=ViewModels.ViewModelNew,
                ExecuteMethodName=ExecuteMethod,
                CanExecuteMethodName=CanExecuteMethod,
                EventToInvokeCanExecuteChanged=ActionNotifyCanExecuteChanged,
                DeepScanAllTrees=True}"/>

<Button Content="Button Cmd-ExProp"
   Command="{vm:BindCommandIoc ServiceType=ViewModels.ViewModelNew,
               ExecuteMethodName=ExecuteMethod,
               CanExecuteBooleanPropertyName=CanExecuteFlag,
               DeepScanAllTrees=True}"/>
```

XAML(WinRt) “View” fragment example:

```
<Button Content="Click here!">
    <vm:BindXAML.BindToCommand>
        <vm:BindCommandIoc ServiceType="ViewModels.ViewModelNew"
            ExecuteMethodName="CommandToExcute"
            CanExecuteBooleanPropertyName="Flag"/>
    </vm:BindXAML.BindToCommand>
</Button>
```

C# “View Model” fragment example:

```
bool _canExecuteFlag = true;

public bool CanExecuteFlag
 {
     get { return _canExecuteFlag; }
     set
     {
         _canExecuteFlag = value; NotifyPropertyChanged();
     }
 }

 public Action<object> ButtonExecuteProperty { get; set; }

 public void ExecuteMethod(object sender)
 {

 }


 public Func<object, bool> CanExecuteProperty { get; set; }
 public bool CanExecuteMethod(object sender)
 {
     return CanExecuteFlag;
 }

 public event Action ActionNotifyCanExecuteChanged;

 public event EventHandler EventHandlerNotifyCanExecuteChanged;

 private Action _propertyDelegateNotifyCanExecuteChanged;

 public Action PropertyDelegateNotifyCanExecuteChanged
 {
     get { return _propertyDelegateNotifyCanExecuteChanged; }
     set { _propertyDelegateNotifyCanExecuteChanged = value; }
 }
```

BindCommandResource

XAML mark-up and BindXAML.BindToCommand extensions; it binds binds a control command dependency property to methods with using the **CommandHadlerProxy** wrapper class. It binds to the source object members located in Resources.

- ResourceKey (default key) – Gets or sets the key value passed by a static resource reference. The key is used to return the object matching that key in resource dictionaries.
- ExecuteMethodName – The method name of the source object that performs as "void ICommand:Execute(object parameter)". It's mutually exclusive versus ExecutePropertyName. It can be static.
- CanExecuteBooleanPropertyName - The property name of the source object that refers to Boolean property that would be return by method "bool ICommand:CanExecute(object parameter)". INotifyPropertyChanged interface will be subscribed to trigger event "event EventHandler ICommand:CanExecuteChanged". It's mutually exclusive versus CanExecuteMethodName, CanExecutePropertyName, EventToInvokeCanExecuteChanged and PropertyActionCanExecuteChanged.
- CanExecuteMethodName – The method name of the source object that performs as "bool ICommand:CanExecute(object parameter)". It can be static and optional. It's mutually exclusive versus CanExecutePropertyName.
- ExecutePropertyName – The property name of the source object that has a type of Action<object> delegate that performs as "void ICommand:Execute(object parameter)". It's mutually exclusive versus ExecutePropertyName. It can be static.
- CanExecutePropertyName –The property name of the source object that has a type of **Func**<object, bool> delegate that performs as "bool ICommand:CanExecute(object parameter)". It can be static and optional. It's mutually exclusive versus CanExecutePropertyName.

- EventToInvokeCanExecuteChanged - Name of an event class member to which will be added a delegate for rising an event in the proxy class "event EventHandler ICommand:CanExecuteChanged". Notification delegate of types **Action**<> or **EventHandler**<> will be added or removed synchronously when the event handler will be add or removed in "event EventHandler ICommand:CanExecuteChanged". It's mutually exclusive versus PropertyActionCanExecuteChanged. It can be static.
- PropertyActionCanExecuteChanged - Name of a property that will accept a delegate of **Action**<> delegate that can be used for rising an event in the proxy class "event EventHandler ICommand:CanExecuteChanged".Notification delegate of types **Action**<> or **EventHandler**<> will be set or cleared synchronously when the event handler will be add or removed in "event EventHandler ICommand:CanExecuteChanged". It's mutually exclusive versus EventToInvokeCanExecuteChanged. It can be static.
- DeepScanAllTrees- If it is set to "true", all DataContext properties in the logical tree will be scanned until first math to a property or method name (ExecutePropertyName, ExecuteMethodName). Smart feature which allows ignore the current DataContext property value and traverse to other parent DataContext value.If set on true, it will cause to scan for the DataContext property objects over the trees and get the first one that contains the binding property or method. It used in case when there is need to ignore the binding ItemsSource DataContext for the ItemsControl item, just bind a Button to a View Model for the item of the ListView or so on.

XAML(WPF) "View" fragment example:

```
<Button Content="Click here!"
   Command="{vm:BindCommandResource ResourceKey=ViewModelNewKey,
                ExecuteMethodName=CommandToExcute,
                CanExecuteBooleanPropertyName=Flag}"/>

<Button Content="Property Cmd-ExCe"
   Command="{vm:BindCommandResource ResourceKey=ViewModelNewKey,
                ExecutePropertyName=ButtonExecuteProperty,
                CanExecutePropertyName=CanExecuteProperty,
                DeepScanAllTrees=True}"/>

<Button Content="Button Cmd-ExCeEv"
   Command="{vm:BindCommandResource ResourceKey=ViewModelNewKey,
                ExecuteMethodName=ExecuteMethod,
                CanExecuteMethodName=CanExecuteMethod,
                EventToInvokeCanExecuteChanged=ActionNotifyCanExecuteChanged,
                DeepScanAllTrees=True}"/>

<Button Content="Button Cmd-ExProp"
   Command="{vm:BindCommandResource ResourceKey=ViewModelNewKey,
                ExecuteMethodName=ExecuteMethod,
                CanExecuteBooleanPropertyName=CanExecuteFlag,
                DeepScanAllTrees=True}"/>
```

XAML(WinRt) "View" fragment example:

```
<Button Content="Click here!">
    <vm:BindXAML.BindToCommand>
        <vm:BindCommandResource ResourceKey="ViewModelNewKey"
            ExecuteMethodName="CommandToExcute" CanExecuteBooleanPropertyName="Flag"/>
    </vm:BindXAML.BindToCommand>
</Button>
```

C# "View Model" fragment example:

```
bool _canExecuteFlag = true;

public bool CanExecuteFlag
 {
     get { return _canExecuteFlag; }
     set
     {
         _canExecuteFlag = value; NotifyPropertyChanged();
     }
 }
```

```
public Action<object> ButtonExecuteProperty { get; set; }

public void ExecuteMethod(object sender)
{

}


public Func<object, bool> CanExecuteProperty { get; set; }
public bool CanExecuteMethod(object sender)
{
    return CanExecuteFlag;
}

public event Action ActionNotifyCanExecuteChanged;

public event EventHandler EventHandlerNotifyCanExecuteChanged;

private Action _propertyDelegateNotifyCanExecuteChanged;

public Action PropertyDelegateNotifyCanExecuteChanged
{
    get { return _propertyDelegateNotifyCanExecuteChanged; }
    set { _propertyDelegateNotifyCanExecuteChanged = value; }
}
```

BindXAML.BindToCommand

XAML attached property, fake collection, is used for processing extensions: BindCommand, BindEventHandlerIoc, BindEventHandlerResource.
It is used for compatibility between WinRT, Win Store App and WPF.

XAML "View" fragment example:

```
<Button Content="Button Cmd-Ex " HorizontalAlignment="Left"">
    <vm:BindXAML.BindToCommand>
        <vm:BindCommand  ExecuteMethodName="ExecuteMethod" DeepScanAllTrees="True"/>
    </vm:BindXAML.BindToCommand>
</Button>
```

C# "View Model" fragment example:

```
bool _canExecuteFlag = true;

public bool CanExecuteFlag
{
    get { return _canExecuteFlag; }
    set
    {
        _canExecuteFlag = value; NotifyPropertyChanged();
    }
}

public Action<object> ButtonExecuteProperty { get; set; }

public void ExecuteMethod(object sender)
{

}
```

Other Elements

LocateDataContext

XAML mark-up and BindXAML.AssignProperties extensions; it finds in the chain of **DataContext** objects, the first, which contains the exact method or property. It comes through parent elements of logical and visual trees.

- DataContextType (default key,optional) – The type (System.Type) or the type name (System.String) of the required **DataContext** object. If it is not set, a method or property name will be only used to locate.
- MethodName – The method name is used to search in DataContext object methods. It's priority versus PropertyName.
- PropertyName – The property name is used to search in DataContext object properties.
- TargetPropertyName (external key, used for BindXAML.AssignProperties) – The target dependency property name; it will be to set to the located **DataContext** object.

BindXAML.AddPropertyChangeEvents

XAML attached property, fake collection, is used for processing extensions: BindEventHandler, BindEventHandlerIoc, BindEventHandlerResource.
It binds a View dependency property change event handler to a event handler in a View Model. It is only applicable to WPF.

XAML "View" fragment example of binding property "Content" change event to View Model:

```
<Label Content="{Binding FluentLabel}">
  <vm:BindXAML.AddPropertyChangeEvents>
    <vm:BindEventHandler MethodName="ContentChanged" TargetPropertyName="Content"/>
  </vm:BindXAML.AddPropertyChangeEvents>
</Label>
```

C# "View Model" fragment example:

```
public void ContentChanged(object sender, EventArgs e)
{

}
```

IocBinding

XAML mark-up and BindXAML.AssignProperties extensions; it binds to IoC container elements.

- ServiceType (default key) – The type (System.Type) or the type name (System.String) of the requested object.
- ServiceKey – The key of the requested DI (IoC) object.
- TargetPropertyName (external key, used for BindXAML.AssignProperties) – The target dependency property name; it will be set to the located object by the IoC container.

XAML "View" fragment example:

```
<Label Content="{Binding ButtonClickMethodMsg}" HorizontalAlignment="Left">
    <vm:BindXAML.AssignProperties>
        <vm:IocBinding ServiceType="MainWindowVm"
             ServiceKey="MainWindowVm" TargetPropertyName="DataContext"/>
    </vm:BindXAML.AssignProperties>
</Label>
```

C# setup "View Model" for Unity DI container example in App.xaml.cs:

```
    public partial class App : Application
    {
        private UnityContainer _unityContainer;
        private UnityServiceLocator _servicelocator;
        public App()
        {
            _unityContainer = new UnityContainer();
            _servicelocator = new UnityServiceLocator(_unityContainer);
            ServiceLocator.SetLocatorProvider(() => _servicelocator);
            var vmMw = new ViewModelNew();
            // instance that will be resolved when it's used ServiceType
```

```
            _unityContainer.RegisterInstance(typeof(ViewModelNew),
                vmMw,new ContainerControlledLifetimeManager());
        }
    }
```

MvvmBindingPack BindEventHandler vs EventTrigger

RelayCommand, **DelegateCommand** and **ActionCommand** classes are well known as solutions for creating instant **ICommand** interface implementations. It's quite a bulky job to wrap the every method of the **View Model** with using **RelayCommand** or **DelegateCommand** classes. Refactoring of such View Models is a real nightmare. The description of **DelegateCommand** class you can find here. It is the class that implements an ICommand interface and its delegates provides Execute() and CanExecute() method functionality.
It is hard to describe the inefficiency of the way of implementing the event invocation shown in the classic **View Model** example.
For binding the CommandToExcute method you have to write lots line of code(see marked in red) and make a "magic dance" around the "**event trigger**".

XAML fragment:

```
<Button Content="Click here!"   Margin="5">
        <i:Interaction.Triggers>
            <i:EventTrigger EventName="Click" >
                <i:InvokeCommandAction Command="{Binding ClickCommand}" />
             </i:EventTrigger>
          </i:Interaction.Triggers>
</Button>
```

View Model fragment:

```
class ViewModelOld
    {
        private readonly DelegateCommand<string> _command;
        private bool _flag;
        public ViewModelOld()
        {
            _command = new DelegateCommand<string>(
                   (s) => { CommandToExcute(s); }, //Execute
                (s) => { return _flag; } //CanExecute
                );
        }
        public DelegateCommand<string> ClickCommand
        {
            get { return _command; }
        }

        public void CommandToExcute(object parameter)
        {

        }

    }
```

There will be absolutely simple solution if you are using **MvvmBindingPack.**
XAML fragment:

```
<Button Content="Click here!"
        Click="{vm:BindEventHandler  MethodName=Button_Click}"  Margin="5"/>
```

or XAML fragment (WinRt):

```
<Button Content="Click here!">
   <vm:BindXAML.AddEvents>
       <vm:BindEventHandler MethodName="Button_Click" TargetEventName="Click"/>
   </vm:BindXAML.AddEvents>
</Button>
```

View Model fragment:

```
namespace ViewModels
{
 class ViewModelNew : NotifyChangesBase
    {

       public void Button_Click(object sender, RoutedEventArgs e)
       {

       }

    }
}
```

There is another variant to bind to a **View Model with using Resources.**

XAML fragment (WPF):

```
<Button Content="Click here!"
        Click="{vm:BindEventHandlerResource
        ResourceKey=ViewModelNewKey, MethodName=Button_Click}"
        Margin="5"/>
```

or XAML fragment (WinRt):

```
<Button Content="Click here!">
    <vm:BindXAML.AddEvents>
        <vm:BindEventHandlerResource ResourceKey="ViewModelNewKey"
         MethodName="Button_Click" TargetEventName="Click"/>
    </vm:BindXAML.AddEvents>
</Button>
```

There is another variant to bind to a **View Model with IoC containers.**

XAML fragment (WPF):

```
<Button Content="Click here!"
        Click="{vm:BindEventHandlerIoc
        ServiceType=ViewModels.ViewModelNew, MethodName=Button_Click}"
        Margin="5"/>
```

or XAML fragment (WinRt):

```
<Button Content="Click here!">
    <vm:BindXAML.AddEvents>
       <vm:BindEventHandlerIoc  ServiceType="ViewModels.ViewModelNew"
                 MethodName="Button_Click" TargetEventName="Click"/>
    </vm:BindXAML.AddEvents>
</Button>
```

Compare to well-known MVVM binding techniques

MvvmBindingPack BindCommand vs DelegateCommand

RelayCommand, **DelegateCommand** and **ActionCommand** classes are well known as solutions for creating instant **ICommand** interface implementations. It's quite a bulky job to wrap the every method of the **View Model** with using **RelayCommand** or **DelegateCommand** classes. Refactoring of such View Models is a real nightmare. The description of **DelegateCommand** class you can find here. It is the class that implements an ICommand interface and its delegates provides Execute() and CanExecute() method functionality. Inefficiency of using these class wrappers cost time and a budget surplus.
In the **View Model** example, for calling the CommandToExcute method you have to write lots lines of code (see marked in red).

XAML fragment:

```
<Button Content="Click here!" Command="{Binding ButtonClickCommand}" Margin="5"/>
```

View Model fragment:

```
class ViewModelOld
    {
        private readonly DelegateCommand<string> _command;
        private bool _flag;
        public ViewModelOld()
        {
            _command = new DelegateCommand<string>(
                   (s) => { CommandToExcute(s); }, //Execute
                   (s) => { return _flag;       }  //CanExecute
               );
        }
        public DelegateCommand<string> ButtonClickCommand
        {
            get { return _command; }
        }
        public void SetFlag(bool flag)
        {
              _flag=flag;
              _command.RaiseCanExecuteChanged();
        }


        public void CommandToExcute(object parameter)
        {

        }
    }
```

So, it is a very bulky and looking weird. In order to call (bind) one method you have to implement the redundant code lines. It is introducing additional complexity that are not appropriate or useful. It makes harder to understand the code and you cannot immediately change the code of the **View Model after Agile scrum meeting**. There are also the practical difficulties: code browsing; try to find out what is a code about; and the test coverage.

There is an absolutely different picture if you are using **MvvmBindingPack.**

XAML fragment:

```
<Button Content="Click here!"
        Command="{vm:BindCommand ExecuteMethodName=CommandToExcute,
        CanExecuteBooleanPropertyName=Flag}"  Margin="5"/>
```

or XAML fragment (WinRt):

```
<Button Content="Click here!"   Margin="5">
   <vm:BindXAML.BindToCommand>
     <vm:BindCommand  ExecuteMethodName="CommandToExcute"
                      CanExecuteBooleanPropertyName="Flag"/>
   </vm:BindXAML.BindToCommand>
```

```
</Button>
```

View Model fragment:

```
namespace ViewModels
{
 class ViewModelNew : NotifyChangesBase
    {

        private bool _flag;
        public bool Flag
        {
            get { return _flag; }
            set { _flag = value; NotifyPropertyChanged(); }
        }

        public void CommandToExcute(object parameter)
        {

        }

    }
}
```

Nothing redundant that you will not want to have in the code.

There is another variant to bind to a **View Model with using Resources.**

XAML fragment (WPF):

```
 <Button Content="Click here!"
         Command="{vm:BindCommandResource ResourceKey=ViewModelNewKey,
         ExecuteMethodName=CommandToExcute,CanExecuteBooleanPropertyName=Flag}"
         Margin="5"/>
```

or XAML fragment (WinRt):

```
 <Button Content="Click here!"   Margin="5">
    <vm:BindXAML.BindToCommand>
        <vm:BindCommandResource ResourceKey="ViewModelNewKey"
                                ExecuteMethodName="CommandToExcute"
                                CanExecuteBooleanPropertyName="Flag"/>
    </vm:BindXAML.BindToCommand>
 </Button>
```

There is another variant to bind to a **View Model with IoC containers.**

XAML fragment (WPF):

```
<Button Content="Click here!"
        Command="{vm:BindCommandIoc ServiceType=ViewModels.ViewModelNew,
        ExecuteMethodName=CommandToExcute,CanExecuteBooleanPropertyName=Flag}"
        Margin="5"/>


or with container key:
<Button Content="Click here!" Command="{vm:BindCommandIoc
        ServiceKey=NewModel,ServiceType=ViewModels.ViewModelNew,
        ExecuteMethodName=CommandToExcute,CanExecuteBooleanPropertyName=Flag}"
        Margin="5"/>
```

or XAML fragment (WinRt):

```
 <Button Content="Click here!"   Margin="5">
    <vm:BindXAML.BindToCommand>
       <vm:BindCommandIoc ServiceKey="NewModel"
                          ServiceType="ViewModels.ViewModelNew"
                          ExecuteMethodName="CommandToExcute"
                          CanExecuteBooleanPropertyName="Flag"/>
```

```
    </vm:BindXAML.BindToCommand>
</Button>
```